FUJITSU

Fujitsu Software Compiler Package V1.0L21

C言語使用手引書

J2UL-2582-03Z0(04)
2023年10月

まえがき

本書の目的

本書は、PRIMEHPC FX700システム(以降、FXシステムと呼びます)向けのC言語処理系(以降、本処理系と呼びます)の使用方法を記述しています。

また、本書は、以下のC言語規格書およびOpenMP仕様に基づいて本処理系で拡張した言語仕様についても、記述しています。

- 日本工業規格 プログラム言語 C JIS X 3010-1993(ISO/IEC 9899:1990)

  本書では本JIS規格並びにその原規格であるISO規格およびANSI規格を総じて、一般的な表現としてC89規格と呼んでいます。また、これらの規格で定義された仕様を、C89仕様と呼びます。

- 日本工業規格 プログラム言語 C JIS X 3010-2003(ISO/IEC 9899:1999)

  本書では本JIS規格並びにその原規格であるISO規格を総じて、一般的な表現としてC99規格と呼んでいます。

  また、この規格で定義された仕様をC99仕様と呼びます。

- Programming languages -- C(ISO/IEC 9899:2011)

  本書では本ISO規格を、一般的な表現としてC11規格と呼んでいます。

  また、この規格で定義された仕様をC11仕様と呼びます。

- OpenMP Application Program Interface Version 3.1 July 2011

  本書ではこの規格で定義された仕様をOpenMP 3.1と呼びます。

- OpenMP Application Program Interface Version 4.0 July 2013

  本書ではこの規格で定義された仕様をOpenMP 4.0と呼びます。

- OpenMP Application Programming Interface Version 4.5 November 2015

  本書ではこの規格で定義された仕様をOpenMP 4.5と呼びます。

- OpenMP Application Programming Interface Version 5.0 November 2018

  本書ではこの規格で定義された仕様をOpenMP 5.0と呼びます。

本書の読者

本書は、C言語規格書で規定された言語仕様を前提に記述しています。規格仕様については、C言語規格書を参照するか、または規格準拠の市販本をお読みください。

本書は、本処理系を使用して、Cプログラムを処理する人およびCプログラムを作成する人を対象に記述しています。

本書を読むにあたっては、Linux(R)コマンド、ファイル操作、およびシェルプログラミングの基本的な操作知識が必要です。

本書の構成

本書は、以下の構成になっています。

第1章 概要

本処理系を使用して、Cプログラムを処理する場合の概要について記述しています。

第2章 翻訳から実行まで

Cプログラムの翻訳から実行までの手続きについて記述しています。

第3章 最適化機能

最適化機能を使用する場合に、利用者が注意しなければならない事項について記述しています。

第4章 並列化機能

自動並列化機能やOpenMP仕様を利用したプログラムの高速化について、利用者が注意しなければならない事項について記述しています。

第5章 出力情報

本処理系が出力する情報について記述しています。

第6章 言語仕様

C言語規格の言語仕様に基づいて本処理系で拡張した言語仕様、および言語仕様のサポート状況について記述しています。

第7章 言語およびtrad/clang間結合における注意事項

プログラム言語やモード(trad/clang)が異なるオブジェクトプログラムを結合する場合の注意事項について記述しています。

第8章 プログラムのデバッグ

本処理系が用意している各種のデバッグ機能について記述しています。

第9章 clangモード

オープンソースソフトウェアであるClang/LLVMコンパイラのユーザインタフェースをベースにしたコンパイラを使用するモードについて記述しています。

付録A 処理系依存の仕様

C言語規格で処理系依存としている項目に対する、本処理系での動作について記述しています。

付録B 翻訳限界

本処理系における翻訳限界について記述しています。

付録C 制限事項および注意事項

本処理系の制限および利用する上での注意事項を記述しています。

付録D GNU C互換機能

GNU C互換機能について記述しています。

付録E データおよびメモリ領域

本処理系でのデータおよびメモリ領域の扱いについて記述しています。

付録F コードカバレッジ機能

コードカバレッジ機能について記述しています。

付録G 富士通拡張関数

富士通拡張関数について記述しています。

付録H 実行時情報出力機能

実行時情報出力機能について記述しています。

付録I 高速化機能の利用

FXシステム向けの高速化機能を利用するためのプログラム翻訳および実行について記述しています。

付録J マルチプロセスによる実行

マルチプロセスでプログラムを実行する方法および注意事項について記述しています。

付録K 富士通OpenMPライブラリ

富士通独自のOpenMPライブラリを利用して、C言語プログラムを並列処理する方法について記述しています。

付録L ラージページライブラリ

標準のLinuxに含まれる機能をFXシステム用に拡張したHPC (High Performance Computing) 拡張機能の、ラージページライブラリについて記述しています。

本書の注意事項

本書における最適化のプログラム例は、機能の理解を助けるための概念ソースです。したがって、例のプログラムをそのまま抜き出して実行した場合、翻訳時オプションやプログラム中の変数の宣言文などの条件によっては、期待した機能が動作しない場合があります。

本書の表記について

構文表記記号

本書では、構文表記記号の形式に従って説明します。

構文表記記号とは、構文を記述する上で、特別な意味で定められた記号であり、以下のものがあります。

記号名	記号	説明
選択記号	{ }	この記号で囲まれた項目の中から、どれか1つを選択することを表します。
	|	この記号を区切りとして、複数の項目を列挙することを表します。
省略可能記号	[ ]	この記号で囲まれた項目を省略して良いことを表します。また、この記号は選択記号“{ }”の意味を含みます。
反復記号	...	この記号の直前の項目を繰り返して指定できることを表します。

並列

本書における並列に関する記述は、並列方式の明示的な記述がない場合、スレッド並列を意味します。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

商標

- Arm is trademark or registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
- OpenMPは、OpenMP Architecture Review Boardの商標です。
- Linux(R)は米国及びその他の国におけるLinus Torvaldsの登録商標です。
- そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。
- 本資料に掲載されているシステム名、製品名などには、必ずしも商標表示(TM、(R))を付記しておりません。

引用文献

本書は、プログラム言語C(JIS X 3010-1993: 日本規格協会発行)の内容を部分的に引用しています。

出版年月および版数

版数	マニュアルコード
2023年10月 第3.4版	J2UL-2582-03Z0(04)
2023年 4月 第3.3版	J2UL-2582-03Z0(03)
2022年10月 第3.2版	J2UL-2582-03Z0(02)
2022年 4月 第3.1版	J2UL-2582-03Z0(01)
2022年 1月 第3版	J2UL-2582-03Z0(00)
2021年 8月 第2.6版	J2UL-2582-02Z0(06)
2021年 7月 第2.5版	J2UL-2582-02Z0(05)
2021年 3月 第2.4版	J2UL-2582-02Z0(04)
2021年 1月 第2.3版	J2UL-2582-02Z0(03)
2020年11月 第2.2版	J2UL-2582-02Z0(02)
2020年 9月 第2.1版	J2UL-2582-02Z0(01)
2020年 7月 第2版	J2UL-2582-02Z0(00)
2020年 2月 初版	J2UL-2582-01Z0(00)

変更履歴

変更内容	変更箇所	版数
以下の翻訳時オプションの説明を見直しました。 ・ -K{fp_relaxed|nofp_relaxed} ・ -K{preex|nopreex} ・ -N{Rtrap|Rnotrap}	2.2.2.5	第3.4版
以下の最適化指示子の説明を見直しました。 ・ fission_point ・ fp_relaxed ・ nopreex	3.4.1.2	
以下の翻訳時オプションの説明を見直しました。 ・ -O[n] ・ -f{fj-fast-matmul|fj-no-fast-matmul} ・ -f{fj-optlib-string|fj-no-optlib-string} ・ -f{lto|no-lto}	9.1.2.2.3	
説明を見直しました。	9.1.2.3.2	
サポートするOpenMP仕様の説明を修正しました。	1.1 4.3 4.3.2.2 J.3	第3.3版
注意を追加しました。	1.1	
表「翻訳コマンドが設定する復帰値」を修正しました。	2.1.3	
説明文を修正しました。	8.1.2	
以下の翻訳時オプションの説明を見直しました。 ・ -f{fj-swp|fj-no-swp} ・ -f{fj-zfill[=N]|fj-no-zfill} ・ -f{lto|no-lto}	9.1.2.2.3	
以下の翻訳時オプションを削除しました。 ・ -f{fj-loop-interchange|fj-no-loop-interchange}	9.1.2.2.3 9.1.2.3.1	
以下の最適化指示子を追加しました。 ・ #pragma fj loop swp ・ #pragma fj loop noswp	9.2.2.1.3	
以下の翻訳時オプションの説明を修正しました。 ・ -fvisibility=internal	9.8.2	
tradモードの以下のGNU C互換オプションを削除しました。 ・ -fvisibility	D.2	

変更内容	変更箇所	版数
図のデザインを改善しました。 用語を見直しました。	―	
説明文を修正しました。	6.1.6 9.1.5	第3.2版
以下の翻訳時オプションを追加しました。 ・ -f{debug-info-for-profiling|no-debug-info-for-profiling}	9.1.2.2.1	
以下の翻訳時オプションの説明を修正しました。 ・ -f{optimize-sibling-calls|no-optimize-sibling-calls}	9.8.2	
「制限事項および注意事項」を追加しました。	9.10	
説明文を修正しました。	I.1	
C++標準ライブラリの表記を統一しました。	―	
説明文を見直しました。	―	
以下の翻訳時オプションの説明を見直しました。 ・ -O[n] ・ -msve-vector-bits={512|scalable} ・ -f{lto|no-lto} ・ -f{omit-frame-pointer|no-omit-frame-pointer} ・ -f{slp-vectorize|no-slp-vectorize}	9.1.2.2.3	第3.1版
以下の翻訳時オプションを追加しました。 ・ -m{omit-leaf-frame-pointer|no-omit-leaf-frame-pointer}	9.1.2.1 9.1.2.2.3	
説明を追加しました。	9.1.2.3.1	
「SIMD化」を追加しました。	9.2.3	
「asmレジスタ宣言のレジスタ名およびインラインアセンブラの破壊(clobber)リストに対するレジスタの指定について」を追加しました。	C.2.11	
説明文を見直しました。	―	
以下の翻訳時オプションの説明を見直しました。 ・ -K{region_extension|noregion_extension} ・ -K{striping[=N]|nostriping}	2.2.2.5	第3版
翻訳時オプション-Klibの最適化対象とする標準ライブラリ関数を追加しました。	2.2.2.5	
「SIMD化可能な組込み関数」を追加しました。	3.2.7.5	
以下の最適化指示子の説明を見直しました。 ・ fission_point ・ striping	3.4.1.2	
「表 自動並列化用の最適化指示子一覧」を修正しました。	4.2.6.1	
環境変数XOS_MMM_L_ARENA_LOCK_TYPEの説明を変更しました。	L.3.3	
説明文を見直しました。	―	
「あらかじめ定義されたマクロ名」の記述を見直しました。	6.1.6 9.1.5	第2.6版
SIMD組込み関数のドキュメントについての注意事項を追加しました。	9.7	
注意を追加しました。	1.2.1	第2.5版

変更内容	変更箇所	版数
以下の翻訳時オプションを追加しました。 ・ -Kopenmp_loop_variable={private|standard}	2.2.2.3 2.2.2.5	
翻訳時オプション-K{array_declaration_opt|noarray_declaration_opt}の省略値を-Knoarray_declaration_optに変更しました。	2.2.2.5	
以下の環境変数を追加しました。 ・ FCOMP_LINK_FJOBJ	2.3	
説明を追加しました。	3.2.7.4	
説明を修正しました。	3.3.9	
タイトルを変更しました。	第7章 9.6	
「結合時の翻訳コマンドと必須オプション」を追加しました。	7.1	
以下の翻訳時オプションを追加しました。 ・ -f{fj-regalloc-using-latency|fj-no-regalloc-using-latency} ・ -f{fj-promote-licm-addressing|fj-no-promote-licm-addressing} ・ -f{fj-sched-insn-contiguous|fj-no-sched-insn-contiguous}	9.1.2.2.3	
注意事項を追加しました。	9.4.1.2.4	
「リンクエラー(undefined reference to)について」を追加しました。	C.2.10	
注意を削除しました。	2.1.2	第2.4版
zfillの最適化の説明文を見直しました。	2.2.2.5 3.3.5 3.4.1.2	
以下の翻訳時オプションの説明を見直しました。 ・ -K{fp_relaxed|nofp_relaxed} ・ -K{ilfunc[={loop|procedure}]|noilfunc}	2.2.2.5	
以下の環境変数を追加しました。 ・ FLIB_L1_SCCR_CNTL	3.5.2.2 I.2 K.4.3	
説明を追加しました。	3.6.1	
サポートする仕様の説明を見直しました。	4.3	
「clangモードにおける共有ライブラリの作成」を追加しました。	4.3.4.4	
「翻訳時の注意事項」を追加しました。	9.1.1.4	
以下の翻訳時オプションを追加しました。 ・ -msve-vector-bits={512|scalable}	9.1.2.1 9.1.2.2.3	
以下の翻訳時オプションを追加しました。 ・ -f{fj-interleave-loop-insns[=*N*]|fj-no-interleave-loop-insns} ・ -f{fj-loop-fission|fj-no-loop-fission} ・ -ffj-loop-fission-threshold=*N* ・ -f{fj-swp|fj-no-swp} ・ -f{fj-zfill[=*N*]|fj-no-zfill}	9.1.2.2.3 9.1.2.3.1	
記事を追加しました。	9.1.2.3.4	

変更内容	変更箇所	版数
「SVEのベクトルレジスタサイズを指定する場合の注意事項」を追加しました。	9.1.2.3.5	
以下の最適化指示子を追加しました。 ・ #pragma fj loop clone *var*==*n* ・ #pragma fj loop loop_fission_target [cl] ・ #pragma fj loop loop_fission_threshold *n* ・ #pragma fj loop zfill [*N*] ・ #pragma fj loop nozfill	9.2.2.1.2 9.2.2.1.3	
以下の最適化指示子を追加しました。 ・ #pragma fj loop swp ・ #pragma fj loop noswp ・ #pragma clang loop vectorize_width(*n*, scalable)	9.2.2.1.2	
最適化情報に、以下の情報を追加しました。 ・ ループ分割 ・ ソフトウェアパイプライニング ・ clone最適化	9.4.1.2.2	
説明文を見直しました。	―	
注意を追加しました。	4.2.2.4 4.3.2.1 K.2.2 K.3.2	第2.3版
言語間結合で使用するコマンドと必須オプションの表を追加しました。	第7章 9.6	
-O2オプション以上で誘導する最適化に、ループ交換を追加しました。	9.1.2.2.3	
「マルチプロセスによる実行」を追加しました。	付録J	
「マルチプロセスによる実行」を追加しました。	K.5	
説明文を見直しました。	―	
注意を追加しました。	2.1.2	第2.2版
以下の翻訳時オプションを追加しました。 ・ --linkcoarray ・ --linkfortran	2.2.1 2.2.2.4 9.1.2.1 9.1.2.2.4	
-N{reordered_variable_stack|noreordered_variable_stack}オプションに関する記事の誤りを修正しました。	2.2.2.6 E.4	
-N{Rtrap|Rnotrap}オプションの記事を修正しました。	2.2.2.6	
ループアンローリングの記事を見直しました。	3.2.4	
翻訳例を追加しました。	7.2 9.6.2	
補足シグナルに関する記事の誤りを修正しました。	8.2.1	
以下のオプションに関する記事の誤りを修正しました。 ・ -ffj-lst[={p|t}] ・ -ffj-lst-out=*file*	9.1.2.2.1 9.4.1.2	

変更内容	変更箇所	版数
「除数が0の場合の整数除算例外について」を追加しました。	C.2.9	
説明文を見直しました。	―	
プログラム例を改善しました。	3.3.6 3.4.1.2 3.5.2.1 3.5.2.2	第2.1版
simd指示子のalignedおよびunalignedパラメタの記事を見直しました。	3.4.1.2 3.4.1.3	
「誤りがあるプログラムに対する注意事項」を追加しました。	3.6.4	
以下の最適化指示子を追加しました。 ・ #pragma clang loop vectorize(assume_safety)	9.2.2.1.2	
「SVEのベクトルレジスタのサイズの変更について」を追加しました。	C.2.8	
「ラージページライブラリ」を追加しました。	付録K	
説明文を見直しました。	―	
サポートするOpenMP仕様を変更しました。	1.1 4.3 J.3	第2版
以下の翻訳時オプションを追加しました。 ・ {-help|--help}	2.2.1 2.2.2.2	
-O3オプションで誘導する最適化に、clone最適化を追加しました。	2.2.2.3	
以下の翻訳時オプションを追加/改善しました。 ・ -K{array_declaration_opt|noarray_declaration_opt} ・ -K{extract_stride_store|noextract_stride_store} ・ -K{fp_precision|nofp_precision} ・ -K{loop_fission_stripmining[={*N*|L1|L2}]|loop_nofission_stripmining} ・ -Kloop_fission_threshold=*N* ・ -K{loop_perfect_nest|loop_noperfect_nest} ・ -K{prefetch_stride[={soft|hard_auto|hard_always}]|prefetch_nostride} ・ -K{preload|nopreload} ・ -K{rdconv[={1|2}]|nordconv} ・ -K{simd_uncounted_loop|simd_nouncounted_loop} ・ -K{simd_use_multiple_structures|simd_nouse_multiple_structures} ・ -Kswp_policy={auto|small|large} ・ -Ktls_size={12|24|32|48}	2.2.2.3 2.2.2.5	
以下の翻訳時オプションを削除しました。 ・ -K{lto|nolto}	2.2.2.3 2.2.2.5	
翻訳時オプション-Kilfunc[={loop|procedure}]のパラメタ省略値を変更しました。	2.2.2.5	
翻訳時オプション-Kloop_fissionの最適化対象ループを変更しました	2.2.2.5	
以下の環境変数を追加しました。 ・ fccpx_trad_ENV	2.3	

変更内容	変更箇所	版数
・ fcc_trad_ENV ・ FCOMP_UNRECOGNIZED_OPTION		
以下の翻訳時プロフィルファイルを追加しました。 ・ fccpx_trad_PROF ・ fcc_trad_PROF	2.4	
以下の環境変数を追加しました。 ・ FLIB_TRACEBACK_MEM_SIZE	2.5.1	
「リンク時最適化」を削除しました。	3.3	
「ストリップマイニング」を追加しました。	3.3.10.1	
各最適化制御行の書式として"#pragma fj"を指定可能としました。	3.4.1.1	
以下の最適化指示子を追加しました。 ・ [no]array_declaration_opt ・ [no]extract_stride_store ・ [no]fullunroll_pre_simd [*n*] ・ loop_[no]fission_stripmining [*n*|"L1"|"L2"] ・ loop_fission_target [cl|ls] ・ loop_fission_threshold *n* ・ loop_[no]perfect_nest ・ prefetch_cache_level *c-level* ・ prefetch_[no]conditional ・ prefetch_[no]indirect ・ prefetch_[no]infer ・ prefetch_iteration *n* ・ prefetch_iteration_L2 *n* ・ prefetch_line *n* ・ prefetch_line_L2 *n* ・ prefetch_nosequential ・ prefetch_[no]stride [soft|hard_auto|hard_always] ・ [no]preload ・ scache_isolate_assign *array1*[,*array2*]... end_scache_isolate_assign ・ scache_isolate_way L2=*n1* [L1=*n2*] end_scache_isolate_way ・ simd_[no]use_multiple_structures ・ swp_policy {auto|small|large}	3.4.1.2	
以下の最適化指示子を削除しました。 ・ loop_[no]fission	3.4.1.2	
「ハードウェアストライドプリフェッチャーを利用する最適化」を追加しました。	3.6.3	

変更内容	変更箇所	版数
LLVM OpenMPライブラリの環境変数OMP_PROC_BINDの省略値を変更しました。	4.2.2.4 4.3.2.1 4.3.3	
エラー発生時に関する記事を追加しました。	4.3.4.2	
既定義マクロを追加/変更しました。	6.1.6 9.1.5	
以下の翻訳時オプションを追加しました。 ・ --coverage	9.1.2.1 9.1.2.2.1	
以下の翻訳時オプションを追加しました。 ・ -mfj-tls-size={12|24|32|48}	9.1.2.1 9.1.2.2.6	
以下の翻訳時オプションを追加しました。 ・ -f{fj-line|fj-no-line} ・ -fprofile-dir=*dir_name*	9.1.2.2.1	
以下の翻訳時オプションを追加しました。 ・ -f{fj-fast-matmul|fj-no-fast-matmul} ・ -f{fj-fp-precision|fj-no-fp-precision} ・ -f{fj-fp-relaxed|fj-no-fp-relaxed} ・ -f{fj-hpctag|fj-no-hpctag} ・ -f{fj-loop-interchange|fj-no-loop-interchange} ・ -f{fj-ocl|fj-no-ocl} ・ -f{fj-optlib-string|fj-no-optlib-string} ・ -f{fj-prefetch-strong|fj-no-prefetch-strong} ・ -f{fj-prefetch-strong-L2|fj-no-prefetch-strong-L2}	9.1.2.2.3	
翻訳時オプション-ffj-ilfunc[={loop|procedure}]のパラメタ省略値を変更しました。	9.1.2.2.3	
以下の翻訳時オプションを改善しました。 ・ -march=*arch*[+*features*]... ・ -mcpu=*cpuname*[+*features*]...	9.1.2.2.5	
以下の翻訳時オプションを削除しました。 ・ -mtune=*cpu*	9.1.2.1 9.1.2.2.5	
以下の翻訳時オプションを削除しました。 ・ -f{plt|no-plt}	9.1.2.2.6	
翻訳時オプションの注意事項を追加しました。	9.1.2.3	
以下の環境変数を追加しました。 ・ fccpx_clang_ENV ・ fcc_clang_ENV	9.1.3	
以下の翻訳時プロフィルファイルを追加しました。 ・ fccpx_clang_PROF ・ fcc_clang_PROF	9.1.4	
「実行の手続き」を追加しました。	9.1.6	

変更内容	変更箇所	版数
以下の最適化制御行を追加しました。 ・ #pragma fj *最適化指示子*	9.2.2.1	
以下の最適化指示子を追加しました。 ・ #pragma clang fp contract(fast)	9.2.2.1.2	
最適化情報に「レジスタに関する情報」を追加しました。	9.4.1.2.2	
「SIMD組込み関数」を追加しました。	9.7	
「コードカバレッジ機能」を追加しました。	9.9	
「デバッガを用いたデバッグについて」を追加しました。	C.2.7	
以下の属性を追加しました。 ・ aarch64_vector_pcs	D.1	
以下のGNU C互換オプションを追加しました。 ・ -ffp-contract=fast ・ -floop-parallelize-all ・ -fprofile-dir=*path* ・ -funroll-loops ・ -f{unsafe-math-optimizations|no-unsafe-math-optimizations} ・ -mcmodel={small|large}	D.2	
以下のGNU C互換オプションを削除しました。 ・ -frename-registers	D.2	
「高速化機能の利用」を追加しました。	付録I	
説明文を見直しました。	―	
製品のレベルアップに伴い、体裁を変更しました。	―	

目　次

第1章 概要

本章では、本処理系を利用してCプログラムを処理する場合の概要について説明します。

1.1 本処理系の構成

本処理系は、言語処理プログラムの1つです。本処理系は以下で構成されます。

翻訳コマンド

翻訳コマンドは、Cコンパイラ、プリプロセッサ、アセンブラ、およびリンカを呼び出すプログラムです。C言語で記述されたプログラムを翻訳およびリンクして実行可能ファイル形式に変換します。

本処理系では、2種類の翻訳コマンドがあります。

表1.1 翻訳コマンド

種別	コマンド名	説明
クロスコンパイラ	fccpx	ログインノード上で使用するコマンド
ネイティブコンパイラ	fcc	計算ノード上で使用するコマンド

以降、翻訳コマンド名としてfccpxを用いて説明します。ネイティブコンパイラを使用する場合は、適宜fccpxコマンドをfccコマンドに読み替えてください。

Cコンパイラ

Cコンパイラは、翻訳コマンドから呼び出されます。Cコンパイラは、C言語で記述されたプログラムを翻訳して、オブジェクトプログラムを生成するプログラムです。

Cコンパイラは、ユーザインタフェースの異なる2種類のモードを持ちます。使用するモードは、翻訳コマンドのオプションで選択します。

表1.2 Cコンパイラの2種類のモード

モード名	説明
tradモード	京およびPRIMEHPC FX100以前のシステム向け富士通コンパイラをベースとしています。 tradモードは、従来の富士通コンパイラとの互換を重視する場合に適しています。サポートしている仕様は、C89/C99/C11、OpenMP 3.1/OpenMP 4.0(一部)/OpenMP 4.5(一部)です。 tradモードの仕様については、“第2章 翻訳から実行まで”から“第8章 プログラムのデバッグ”、および付録をお読みください。
clangモード	オープンソースソフトウェアであるClang/LLVMコンパイラをベースとしています。 clangモードは、最新言語仕様を使用したプログラムや、オープンソースソフトウェアを翻訳する場合に適しています。サポートしている仕様は、C89/C99/C11、OpenMP 4.5/OpenMP 5.0(一部)です。 clangモードの仕様については、“第9章 clangモード”をお読みください。

注意

tradモードとclangモードでは、ベースとするコンパイラが異なるため、処理系依存の仕様が異なる場合があります。

並列処理用C言語ライブラリ

本処理系では、並列処理用に2つのライブラリを提供します。

表1.3 2つの並列処理用ライブラリ

ライブラリ名	説明
LLVM OpenMPライブラリ	オープンソースソフトウェアであるLLVM OpenMP Runtime Libraryをベースにした並列化機能用のライブラリです。サポートしている仕様は、OpenMP 4.5/OpenMP 5.0(一部)です。tradモードおよびclangモードで利用できます。 LLVM OpenMPライブラリの仕様については、“第4章 並列化機能”をお読みください。
富士通OpenMPライブラリ	京およびPRIMEHPC FX100以前のシステム向けの富士通OpenMPライブラリをベースとした並列化機能用のライブラリです。従来の富士通OpenMPライブラリとの互換を重視する場合に適しています。サポートしている仕様は、OpenMP 3.1/OpenMP 4.0(一部)/OpenMP 4.5(一部)です。tradモードでのみ利用できます。 富士通OpenMPライブラリの仕様については、“付録K 富士通OpenMPライブラリ”をお読みください。

オンラインマニュアル

manコマンドを使用して本処理系の使用方法を検索することができます。

本処理系向けにfccpx(1)、fcc(1)、fccpx_trad_mode(7)、fcc_trad_mode(7)、fccpx_clang_mode(7)、およびfcc_clang_mode(7)を提供しています。

1.2 使用方法

ここでは、本処理系の使用方法を簡単に説明します。

1.2.1 準備

本処理系を使用する前に、利用者の環境変数に以下の値を追加してください。“*製品インストールパス*”は、システム管理者にお問い合わせください。

環境変数	設定値
PATH	/*製品インストールパス*/bin
LD_LIBRARY_PATH	/*製品インストールパス*/lib64
MANPATH	/*製品インストールパス*/man

設定例:

```
$ PATH=/製品インストールパス/bin:$PATH
$ export PATH
$ LD_LIBRARY_PATH=/製品インストールパス/lib64:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH
$ MANPATH=/製品インストールパス/man:$MANPATH
$ export MANPATH
```

注意

環境変数LANGにロケールを指定するときは、翻訳を行うノードにそのロケールがインストールされていることを確認してください。

インストールされているロケールの一覧は“locale -a”コマンドで確認できます。

例:

```
$ locale -a
C
  :
```

インストールされていない場合、警告メッセージが出力される、または翻訳エラーが発生することがあります。その場合、以下のどちらかの対応をしてください。

- 環境変数LANGにCを指定
- 指定したいロケールのインストールをシステム管理者に依頼

現在設定されているロケールは、“locale”コマンドで確認できます。

例:

```
$ locale
LANG=C
  :
```

1.2.2 翻訳とリンク

利用者は、翻訳コマンドを使用して、C言語で記述されたソースプログラムを翻訳します。翻訳時には、オブジェクトプログラムの最適化などの豊富な機能が使用できます。これらの機能は、翻訳時オプションとして翻訳コマンドのオペランドに指定することにより、使用できます。

翻訳コマンドは、リンクのためにリンカ(ldコマンド)を呼び出す機能を提供します。利用者は、翻訳コマンドを使用して、翻訳の過程で生成されたオブジェクトプログラムとC言語ライブラリなどを結合させて実行可能プログラムを生成します。

本処理系は、GNU Cコンパイラの言語仕様(GNU C拡張仕様)に基づいて翻訳します。ソースプログラムおよびコマンドラインにGNU C拡張仕様の一部が使用できます。

使用できる言語仕様については、tradモードは“D.1 GNU C拡張仕様”を、clangモードは“9.8.1 GNU C拡張仕様”をお読みください。

使用できるオプションについては、tradモードは“D.2 GNU C互換オプション”を、clangモードは“9.8.2 GNU C互換オプション”をお読みください。

以下に、クロスコンパイラを用いた例題プログラムsample.cの翻訳例を示します。

例題プログラムsample.c

```
#include <stdio.h>
int main(void)
{
    (void)printf("Hello, world.¥n");
    return 0;
}
```

翻訳とリンク

```
$ fccpx sample.c
```

1.2.3 デバッグ

Cソースプログラムのデバッグの方法として、ソースレベルによるデバッグ機能があります。

ソースレベルのデバッグは、gdbを利用してください。

この機能を使用する場合は、Cソースプログラムを翻訳コマンドの-gオプションを指定して翻訳しておく必要があります。

1.2.4 チューニング

Cソースプログラムのチューニングの方法として、プロファイラで提供される実行時間のサンプリング機能があります。プロファイラについては、“プロファイラ使用手引書”をお読みください。

第2章 翻訳から実行まで

本章では、ソースプログラムを翻訳し実行するための手続きについて説明します。

なお、本説明はtradモード向けのものです。clangモードについては、“第9章 clangモード”をお読みください。

2.1 翻訳コマンド

利用者は、翻訳コマンドを使用することにより、ソースプログラムの翻訳から実行可能プログラムおよび共有オブジェクトの作成まで行うことができます。翻訳コマンドは、翻訳コマンド行に指定された各種のオペランドを解析し、必要に応じてプリプロセッサ、コンパイラ、アセンブラ(asコマンド)およびリンカ(ldコマンド)を呼び出します。

2.1.1 翻訳コマンドの形式

利用者は、翻訳コマンドのオペランドとして、プリプロセッサ、コンパイラ、アセンブラおよびリンカに対するオプションの並びとファイル名を指定することができます。ここで、プリプロセッサおよびコンパイラに対するオプションを翻訳時オプションといいます。コンパイラ(翻訳コマンド)、アセンブラ(asコマンド)およびリンカ(ldコマンド)に対するオプションについては、manコマンドを使用して参照することができます。

“表2.1 翻訳コマンドの形式”に、翻訳コマンドの形式を示します。

表2.1 翻訳コマンドの形式

コマンド名		オペランド
クロスコンパイラ	fccpx	[□オプションの並び]□ファイル名の並び
ネイティブコンパイラ	fcc	[□オプションの並び]□ファイル名の並び

□:1個以上の空白が必要であることを意味します。

参考

“オプションの並び”と“ファイル名の並び”の指定順序に制約はありません。

“オプションの並び”と“ファイル名の並び”を混在して指定してもかまいません。

2.1.2 翻訳コマンドの入力ファイル

“表2.2 入力ファイルの形式”に、翻訳コマンドに入力ファイルとして指定できるファイルを示します。

表2.2 入力ファイルの形式

ファイルの種別	ファイルのサフィックス	渡し先
ヘッダ	.h	プリプロセッサ (注)
	.H	
Cソースファイル	.c	プリプロセッサおよび コンパイラ
	.i	
アセンブラソースファイル	.s	アセンブラ
前処理を要するアセンブラソースファイル	.S	プリプロセッサおよび アセンブラ
オブジェクトファイル	.o	リンカ

注) -Eオプションが有効な場合

2.1.3 翻訳コマンドの復帰値

“表2.3 翻訳コマンドが設定する復帰値”に、翻訳コマンドが設定する復帰値を示します。

表2.3 翻訳コマンドが設定する復帰値

復帰値	意味
0	正常に終了しました。
0以外	翻訳時またはリンク時にエラーが発生しました。

2.2 翻訳時オプション

翻訳コマンドに指定できる翻訳時オプションの形式と意味について説明します。

翻訳時オプションは、以下の方法で指定することができます。

- 翻訳コマンドのオペランド
- 環境変数 (“2.3 翻訳コマンドの環境変数”参照)
- 翻訳時プロフィルファイル (“2.4 翻訳時プロフィルファイル”参照)

翻訳時オプションの指定の優先順位は、“2.4 翻訳時プロフィルファイル”を参照してください。

通常、翻訳コマンドが識別不可能なオプションは、警告メッセージが出力されて無視されます。環境変数FCOMP_UNRECOGNIZED_OPTIONを設定することで、識別不可能なオプションに対する動作を変更することができます。環境変数FCOMP_UNRECOGNIZED_OPTIONについては、“2.3 翻訳コマンドの環境変数”を参照してください。

2.2.1 翻訳時オプションの形式

翻訳コマンドのオペランドで翻訳時オプションを指定することができます。

以下に、翻訳時オプションの形式を示します。

- コンパイラ全般に関連するオプション

```
[ -# ] [ -### ] [ -A- ] [ -Aname[(tokens)] ] [ -B{dynamic|static} ] [ -C ] [ -Dname[=tokens] ] [ -E ] [ -H ] [ -Idir ]
[ -Ldir ] [ -M ] [ -MD ] [ -MF filename ] [ -MM ] [ -MMD ] [ -MP ] [ -MT target ] [ -Nopt ] [ -P ] [ -Qc ] [ -S ] [ -SSL2 ]
[ -SSL2BLAMP ] [ -Uname ] [ -Wtool,arg1[,arg2]... ] [ -Yitem,dir ] [ -c ] [ {-g|-g0} ] [ -lname ] [ -mt ] [ -o
pathname ] [ -shared ]
```

- メッセージ関連オプション

```
[ -V ] [ {-help|--help} ] [ -j ] [ -w ]
```

- 最適化関連オプション

```
[ -Kopt ] [ -O[n] ] [ -x- ] [ -xfunc1[,func2]... ] [ -xn ]
```

- 言語仕様関連オプション

```
[ -ansi ] [ -std=/eve/ ] [ --linkcoarray ] [ --linkfortran ]
```

2.2.2 翻訳時オプションの意味

以下に、翻訳時オプションの意味を示します。

2.2.2.1 コンパイラ全般に関連するオプション

ここでは、翻訳で使用される一般的なオプションについて説明します。

-#

翻訳コマンドによって実行されるコマンドを出力します。ただし、プログラムの翻訳処理は行いません。

-###

翻訳コマンドによって実行されるコマンドを出力します。

-A-

あらかじめ定義されたマクロ(_で始まるものは除く)と、あらかじめ定義されたアサーションをすべて無効にします。

-A*name*[(*tokens*)]

#assert前処理指令の機能と同様、*name*を述語とし、指定された*tokens*と関連付けます。

あらかじめ定義されたアサーションを、以下に示します。

- system(unix)

-B{dynamic|static}

リンク時に結合するライブラリの種類を指定します。省略時は、-Bdynamicオプションが適用されます。

本オプションは、リンカに渡されます。

本オプションは-l*name*オプションと組み合わせて使用することで、ライブラリごとに結合するライブラリの種類を指定できます。

-Bdynamic

動的ライブラリまたは静的ライブラリと結合します。

-Bdynamicオプションに続いて-l*name*オプションを指定すると、リンカは、最初にlib*name*.soという名前のライブラリを検索し、次にlib*name*.aという名前のライブラリを検索します。

-Bstatic

静的ライブラリのみ結合します。

-Bstaticオプションに続いて-l*name*オプションを指定すると、リンカは、lib*name*.aという名前のライブラリのみ検索します。

例

libxxx.aとlibyyy.soをリンクする場合

```
$ fccpx a.c -Bstatic -lxxx -Bdynamic -lyyy
```

注意

-Bstaticオプションを指定した場合、それ以降の-l*name*オプションで指定した静的ライブラリに加えて、標準ライブラリ(例えば-lc)や機能によって必要となるライブラリも静的ライブラリだけを探すため、必要なライブラリが見つからず、リンクエラーとなることがあります。

-Bstaticオプションを指定した時は、コマンド行の最後に-Bdynamicオプションを指定してください。

例1: 最後に-Bdynamicオプションを指定しなかった場合

```
$ fccpx a.c -Bstatic -lxxx
ld: cannot find -lc
```

例2: 最後に-Bdynamicオプションを指定した場合

```
$ fccpx a.c -Bstatic -lxxx -Bdynamic
```

-C

前処理の段階で通常削除される注釈を、前処理指令の行にあるもの以外、この段階では削除されないようにします。

-D*name*[=*tokens*]

#define前処理指令と同様に、マクロを定義します。*name*にはマクロ名を指定します。*tokens*には*name*に関連付ける値を指定します。

-Dオプションと-Uオプションに同一の*name*を指定すると、後に指定した方が有効となります。

あらかじめ定義されたマクロについては、“6.1.6 あらかじめ定義されたマクロ名”をお読みください。

-E

指定されたCソースファイルについて前処理だけを実行し、結果を標準出力に出力します。

出力は、処理系の次の過程で使用される前処理指令を含みます。

-H

現在の翻訳中にインクルードされている各ファイルのパス名を、1行ごとに標準エラーに出力します。

-I*dir*

名前が/以外で始まるヘッダの検索を、*dir*で指定されたディレクトリを先に検索し、その後、通常のディレクトリを検索するように変更します。
複数の-Iオプションでディレクトリが複数指定された場合、指定された順に検索します。

ヘッダの検索は以下の順序で行われます。

ー 二重引用符(")で囲まれたファイルは、以下の順に検索します。

1. #include前処理指令を含む現ディレクトリ
2. -Iオプションで指定されたディレクトリ
3. 環境変数CPATHで指定されたディレクトリ
4. -isystemオプションで指定されたディレクトリ
5. 環境変数C_INCLUDE_PATHで指定されたディレクトリ
6. 本処理系が提供するヘッダを格納するディレクトリ
7. 標準のディレクトリ
8. -idirafterオプションで指定されたディレクトリ

ー 角括弧(<>)で囲まれたファイルは、以下の順に検索します。

1. -Iオプションで指定されたディレクトリ
2. 環境変数CPATHで指定されたディレクトリ
3. -isystemオプションで指定されたディレクトリ
4. 環境変数C_INCLUDE_PATHで指定されたディレクトリ
5. 本処理系が提供するヘッダを格納するディレクトリ
6. 標準のディレクトリ
7. -idirafterオプションで指定されたディレクトリ

ヘッダが絶対パス名で指定された場合、指定された絶対パス名だけ検索します。

*dir*で指定されたディレクトリが存在しない場合、本オプションは無効となります。

-L*dir*

リンカがライブラリを検索するディレクトリのリストに、*dir*を加えます。

本オプションは、リンカに渡されます。

ライブラリの検索は以下の順序で行われます。

1. -Lオプションの引数に指定されたディレクトリ
2. 本処理系が提供するライブラリを格納するディレクトリ
3. 標準ライブラリのディレクトリ
4. 環境変数LIBRARY_PATHに指定されたディレクトリ

-M

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。

本オプションを指定すると、前処理だけが実施されます。

-MD

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。

結果をサフィックス.dの付いたファイルに格納します。

-Eオプションおよび-oオプションと同時に指定された場合、結果を-oオプションで指定されたファイルに格納します。

-MFオプションと同時に指定された場合、結果を-MFオプションで指定されたファイルに格納します。

-MF *filename*

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係を*filename*に出力することを指示します。

-MM

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。ただし、標準ヘッダは出力対象に含まれません。

本オプションを指定すると、前処理だけが実施されます。

-MMD

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。ただし、標準ヘッダは出力対象に含まれません。

結果をサフィックス.dの付いたファイルに格納します。

-Eオプションおよび-oオプションと同時に指定された場合、結果を-oオプションで指定されたファイルに格納します。

-MFオプションと同時に指定された場合、結果を-MFオプションで指定されたファイルに格納します。

-MP

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係に疑似ターゲットを追加することを指示します。

-MT *target*

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係のターゲットを*target*に変更することを指示します。

-N*opt*

*opt*には、以下のいずれかを指定します。

{ {Rtrap|Rnotrap} | {cancel_overtime_compilation|nocancel_overtime_compilation} | check_cache_arraysize | {clang|noclang} | {coverage|nocoverage} | {exceptions|noexceptions} | {fjcex|nofjcex} | {fjprof|nofjprof} | {hook_func|nohook_func} | {hook_time|nohook_time} | {libomp|fjomplib} | {line|noline} | lst[={p|t}] | lst_out=*file* | profile_dir=*dir_name* | quickdbg[={subchk|nosubchk|heapchk|noheapchk|inf_detail|inf_simple}] | {reordered_variable_stack|noreordered_variable_stack} | {rt_tune|rt_notune} | rt_tune_func | rt_tune_loop[={all|innermost}] | {setvalue[=*setarg*]|nosetvalue} | src | sta }

-Nオプションには、コンマ(,)を区切りとして、複数の*opt*を指定することができます。例えば、

```
-Nsrc -Nsta
```

の代わりに、

```
-Nsrc,sta
```

のように指定することができます。

-Nオプションの詳細については、“2.2.2.6 -Nオプション”をお読みください。

-P

-Eオプションによる出力結果に、linemarker(行情報)を含めないことを指示します。

-Q*c*

*c*には、yまたはnを指定します。-Qyオプションが指定された場合、呼び出された各翻訳ツールについての識別情報が、出力ファイルに追加されます。-Qnオプションが指定された場合、この情報を出力しません。省略時は、-Qyオプションが適用されます。

-S

指定されたCソースファイルを翻訳し、アセンブリ言語出力を、サフィックス.sの付いた対応するファイルに残します。

本オプションと同時に-Eオプションを指定した場合、本オプションは無効となります。

-SSL2

C-SSLII、C-SSLIIスレッド並列機能およびBLAS/LAPACKを結合することを指示します。

-SSL2BLAMP

C-SSLII、C-SSLIIスレッド並列機能およびBLAS/LAPACKスレッド並列版を結合することを指示します。

-SSL2オプションとの違いは、BLAS/LAPACKにおいてスレッド並列版が結合されることです。

-U*name*

#undef前処理指令と同様に、マクロ定義を無効にします。*name*には無効にするマクロ名を指定します。

-Dオプションと-Uオプションに同一の*name*を指定すると、後に指定した方が有効となります。

-W*tool*,*arg1*[,*arg2*]...

指定された*arg1*[,*arg2*]...をそれぞれ別の引数として*tool*に渡します。各引数は、直前の引数とコンマだけで区切られていなければなりません。

*tool*には、以下のいずれかの文字を指定します。

p	プリプロセッサ
0	コンパイラ
a	アセンブラ
l	リンカ

-Y*item*,*dir*

新しいディレクトリ*dir*を*item*の位置として指定します。

*item*には、以下のいずれかの文字を指定します。

0	コンパイラのパス名を*dir*/ccpcompxに変更します(クロスコンパイラ)。 コンパイラのパス名を*dir*/ccpcomに変更します(ネイティブコンパイラ)。
a	アセンブラのパス名を*dir*/asに変更します。
l	リンカのパス名を*dir*/ldに変更します。
M	メッセージファイルのディレクトリを*dir*に指定します。

1つの*item*に対して複数の-Yオプションが指定された場合、最後の指定が有効となります。

-c

翻訳処理の最後の段階であるリンクを行わないようにします。

それまでに作成されたオブジェクトファイルが削除されることはありません。

本オプションと同時に-Eオプションまたは-Sオプションを指定した場合、本オプションは無効となります。

{-g|-g0}

-gオプションは実行可能ファイルにデバッガで利用可能な追加情報を含めることを指示します。

-g0オプションは実行可能ファイルにデバッガで利用可能な追加情報を含めないことを指示します。

省略時は、-g0オプションが適用されます。

tradモードにおけるデバッガの注意事項については、“C.2.7 デバッガを用いたデバッグについて”を参照してください。

-l*name*

lib*name*.soまたはlib*name*.aというファイル名のライブラリを結合します。

シンボルは指定された順序にライブラリで解決されるため、コマンド行でのライブラリの指定順序が重要になります。本オプションは、ソースファイル名の後に指定してください。

本オプションは、リンカに渡されます。

-mt

マルチスレッドセーフオブジェクトを生成します。

ファイル名の並びに-mtオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-mtオプションを指定する必要があります。

ユーザー固有のマルチスレッドプログラムの場合、-mtオプションを指定する必要があります。

-o *pathname*

*pathname*で指定された名前のファイルを作成します。

- -cオプションを同時に指定した場合、*pathname*の名前でオブジェクトファイルが作成されます。
- -sharedオプションを同時に指定した場合、*pathname*の名前で共有オブジェクトファイルが作成されます。
- -Sオプションを同時に指定した場合、*pathname*の名前でアセンブリ言語出力ファイルが作成されます。
- そのほかの場合、省略時設定のa.outの代わりに、*pathname*の名前で実行可能プログラムが作成されます。

-shared

リンカに対して、共有オブジェクトを作成することを指示します。

本オプションは、リンカに渡されます。

2.2.2.2 メッセージ関連オプション

ここでは、翻訳時メッセージに関するオプションについて説明します。

-V

起動された各コマンドについて、そのバージョン情報を標準エラーに出力します。

{-help|--help}

ヘルプ情報を出力します。

-j

値が設定される前に使用される自動変数に関する警告を抑止します。

-w

警告メッセージの出力を抑止します。

2.2.2.3 最適化関連オプション

ここでは、最適化機能を使用する際に指定するオプションについて説明します。

-K*opt*

*opt*には、以下のいずれかを指定します。

{ {PIC|pic} | {SVE|NOSVE} | {alias_const|noalias_const} | {align_loops[=*N*]|noalign_loops} | *archi* | {array_declaration_opt|noarray_declaration_opt} | {array_private|noarray_private} | assume={shortloop|noshortloop| memory_bandwidth|nomemory_bandwidth|time_saving_compilation|notime_saving_compilation} | cmodel={small|large} | {const|noconst} | *cpu* | {dynamic_iteration|nodynamic_iteration} | {eval|noeval} | {eval_concurrent|eval_noconcurrent} | {extract_stride_store|noextract_stride_store} | fast | {fast_matmul|nofast_matmul} | {fconst|nofconst} | {fenv_access| nofenv_access} | {fp_contract|nofp_contract} | {fp_precision|nofp_precision} | {fp_relaxed|nofp_relaxed} | {fsimple| nofsimple} | {fz|nofz} | {hpctag|nohpctag} | {ilfunc[={loop|procedure}]|noilfunc} | instance=*N*| {largepage|nolargepage} | {lib|nolib} | {loop_blocking[=*N*]|loop_noblocking} | {loop_fission|loop_nofission} | {loop_fission_stripmining[={*N*|L1| L2}]|loop_nofission_stripmining} | loop_fission_threshold=*N*| {loop_fusion|loop_nofusion} | {loop_interchange| loop_nointerchange} | {loop_part_parallel|loop_nopart_parallel} | {loop_part_simd|loop_nopart_simd} | {loop_perfect_nest|loop_noperfect_nest} | {loop_versioning|loop_noversioning} | looptype={f|n|s} | {memalias| nomemalias} | {mfunc[={1|2|3}]|nomfunc} | noprefetch | {ocl|noocl} | {omitfp|noomitfp} | {openmp|noopenmp} | {openmp_assume_norecurrence|openmp_noassume_norecurrence} | {openmp_collapse_except_innermost|

openmp_nocollapse_except_innermost} | openmp_loop_variable={private|standard} | {openmp_ordered_reduction| openmp_noordered_reduction} | {openmp_simd|noopenmp_simd} | {optlib_string|nooptlib_string} | {optmsg[={1|2}]| nooptmsg} | {parallel|noparallel} | {parallel_fp_precision|parallel_nofp_precision} | parallel_iteration=*N*| parallel_strong | {pc_relative_literal_loads|nopc_relative_literal_loads} | {plt|noplt} | {preex|nopreex} | prefetch_cache_level={1|2|all} | {prefetch_conditional|prefetch_noconditional} | {prefetch_indirect|prefetch_noindirect} | {prefetch_infer|prefetch_noinfer} | prefetch_iteration=*N*| prefetch_iteration_L2=*N*| prefetch_line=*N*| prefetch_line_L2=*N*| {prefetch_sequential[={auto| soft}]|prefetch_nosequential} | {prefetch_stride[={soft|hard_auto|hard_always}]|prefetch_nostride} | {prefetch_strong| prefetch_nostrong} | {prefetch_strong_L2|prefetch_nostrong_L2} | {preload|nopreload} | {rdconv[={1|2}]|nordconv} | {reduction|noreduction} | {region_extension|noregion_extension} | {restp[={all|arg|restrict}]|norestp} | {sch_post_ra| nosch_post_ra} | {sch_pre_ra|nosch_pre_ra} | {sibling_calls|nosibling_calls} | {simd[={1|2|auto}]|nosimd} | {simd_packed_promotion|simd_nopacked_promotion} | {simd_reduction_product|simd_noreduction_product} | simd_reg_size={128|256|512|agnostic} | {simd_uncounted_loop|simd_nouncounted_loop} | {simd_use_multiple_structures|simd_nouse_multiple_structures} | {strict_aliasing|nostrict_aliasing} | {striping[=*N*]| nostriping} | {swp|noswp} | {swp_freg_rate=*N*|swp_ireg_rate=*N*|swp_preg_rate=*N*} | swp_policy={auto|small|large} | swp_strong | swp_weak | tls_size={12|24|32|48} | {unroll[=*N*]|nounroll} | {unroll_and_jam[=*N*]|nounroll_and_jam} | visimpact | {zfill[=*N*]|nozfill} }

-Kオプションには、コンマ(,)を区切りとして、複数の*opt*を指定することができます。例えば、

```
-Kfast -Kparallel
```

の代わりに、

```
-Kfast,parallel
```

のように指定することができます。

-Kオプションの詳細については、“2.2.2.5 -Kオプション”をお読みください。

-O[*n*]

*n*には、最適化レベル0、1、2、または3を指定します。最適化レベル*n*が指定されていない場合、-O2オプションが適用されます。最適化レベルが高いほど、実行時間が短縮されますが、翻訳時間は増加します。高位の最適化レベルは、低位の最適化レベルを機能的に包含します。

-O[*n*]オプションを指定しない場合、-O0オプションが適用されます。

本オプションは、Cソースファイル以外のファイルには無効です。Cソースファイルについては、“2.1.2 翻訳コマンドの入力ファイル”をお読みください。

ー 最適化レベル0

最適化されません。

ー 最適化レベル1

基本的な最適化が行われます。また、-Knoalias_constが有効となります。

最適化レベル0に比べてオブジェクトプログラムの大きさが縮小され、実行時間も大幅に短縮されます。

ー 最適化レベル2

最適化レベル1に加えて、以下の最適化が行われます(括弧内のオプションが有効となります)。

- ループアンローリング (-Kunroll)
- ソフトウェアパイプライニング (-Kswp)
- ループブロッキング (-Kloop_blocking)
- ループ融合 (-Kloop_fusion)
- ループ分割 (-Kloop_fission)
- ループ交換 (-Kloop_interchange)
- prefetch命令の生成 (-Kprefetch_sequential,prefetch_cache_level=all)
- SIMD化 (-Ksimd=auto)

- ループの先頭アライメント調整 (-Kalign_loops)
- 末尾呼出しの最適化 (-Ksibling_calls)
- 最適化機能の繰返し実施

  最適化機能の繰返し実施は、最適化レベル1で行われる最適化機能を、最適化の余地がなくなるまで繰り返して実施します。
- ループの繰返しに伴って一定値ずつ増加または減少する4バイト以下の符号付き整数型の式がオーバフローしないと仮定した最適化 (-Krdconv=1)

オブジェクトプログラムの大きさが増加する場合があります。

オブジェクトプログラムの実行性能向上を追求したいときに利用してください。

ー 最適化レベル3

最適化レベル2に加えて、以下の最適化が行われます(括弧内のオプションが有効となります)。

- 多重ループのアンローリング
- ループ交換などの最適化を促進するための完全多重ループ化 (-Kloop_perfect_nest)
- ループアンスイッチング
- clone最適化
- インライン展開 (-x-)

翻訳時間は大幅に増加しますが、オブジェクトプログラムの実行性能向上を追求したいときに利用してください。

-x-

ソースプログラム上で定義された関数に対して、マクロ展開後の初期化のある宣言と実行文の数がインライン展開の許容値(*n*)を超えない関数を、インライン展開の対象にします。

*n*の値はコンパイラが自動的に適切な値を決定します。

本オプションは、-O1オプション以上と同時に指定した場合に有効となります。

本オプションと-x*n*オプションを同時に指定した場合、後に指定した方が有効となります。

本オプションは、*n*の値を用いてインライン展開するか否かを決定するため、インライン展開すべきでないと判断した場合は、インライン関数においても、インライン展開の対象にはなりません。

インライン展開については、“3.2.9 インライン展開”をお読みください。

-x*func1*[,*func2*]...

ソースプログラム上で定義された関数*func1*[,*func2*]...に対して、関数呼出しのインライン展開を行います。

本オプションは、-O1オプション以上と同時に指定した場合に有効となります。

-x*n*

インライン展開の許容値を指定して、インライン展開を行います。

関数内の初期化のある宣言と、実行文の数が*n*で指定された数以下の関数が、インライン展開の対象になります。

*n*には、0から2147483647の整数値を指定します。

本オプションと-x-オプションを同時に指定した場合、後に指定した方が有効となります。

-x0オプションが指定された場合、-x-、-x*n*および-x*func1*[,*func2*]...オプションは無効となり、インライン展開を行いません。ただし、後に指定した方が有効となります。

本オプションは、-O1オプション以上と同時に指定した場合に有効となります。

インライン関数は*n*の値に関係なくインライン展開の対象になります。

2.2.2.4 言語仕様関連オプション

ここでは、言語仕様に関係するオプションについて説明します。

-ansi

本オプションは、-std=c89と等価です。

詳細は、-std=*level*オプションを参照してください。

-std=*level*

コンパイラ(プリプロセッサを含む)が解釈する言語仕様のレベルを指定します。*level*には、c89、c99、c11、gnu89、gnu99、またはgnu11のいずれかを指定します。省略時は、-std=gnu11オプションが適用されます。

-std=c89
-std=gnu89

C89仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

-std=c99
-std=gnu99

C99仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

C99仕様のサポート状況については、“6.2.1 C99規格の言語仕様”をお読みください。

-std=c11
-std=gnu11

C11仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

C11仕様のサポート状況については、“6.2.2 C11規格の言語仕様”をお読みください。

あらかじめ定義されたマクロ(以降、既定義マクロと呼びます)のいくつかは、-std=*level*オプションの指定に応じて値が変わります。既定義マクロについては、“6.1.6 あらかじめ定義されたマクロ名”をお読みください。

--linkcoarray

COARRAY仕様を利用しているFortranとの言語間結合で必要なライブラリの検索を行うよう指示します。

本オプションを指定しない場合、この検索処理を行わないため翻訳時間を短縮することができます。

--linkfortran

COARRAY仕様を利用していないFortranとの言語間結合で必要なライブラリの検索を行うよう指示します。

本オプションを指定しない場合、この検索処理を行わないため翻訳時間を短縮することができます。

2.2.2.5 -Kオプション

-K{PIC|pic}

位置独立コード(PIC)を生成することを指示します。-KPICオプションと-Kpicオプションは等価です。

本オプションと-Kcmodel=largeオプションを同時に指定することはできません。

本オプションは、翻訳時の指定だけが有効となります。

-K{SVE|NOSVE}

Armv8-Aアーキテクチャの拡張であるSVEを利用するか否かを指示します。-KSVEオプションが指定された場合、SVEを利用したオブジェクトファイルを出力します。省略時は、-KSVEオプションが適用されます。

SVEをサポートしないプロセッサ向けのオブジェクトファイルを作成したい場合は、-KNOSVEオプションを指定してください。

-K{alias_const|noalias_const}

const修飾されたポインタは、ほかのポインタによって定義されないと限定解釈して、ポインタにより指されるデータの最適化を行うか否かを指示します。-Knoalias_constオプションが指定された場合、最適化を行います。省略時は、-Knoalias_constオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Knoalias_constオプションは、-Knoconstオプションが有効な場合、無効となります。

-K{align_loops[=*N*]|noalign_loops}

ループの先頭アライメントを2の累乗バイトの境界に合わせるか否かを指示します。-Kalign_loopsオプションが指定された場合、最適化を行います。*N*はループの先頭アライメントのバイト境界の値です。*N*は、1から32768までの2の累乗の整数値または0です。*N*の指定を省略した場合または*N*に0を指定した場合、コンパイラが自動的に値を決定します。-O1オプションが有効な場合、省略時は、-Knoalign_loopsオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kalign_loopsオプションが適用されます。

-Kalign_loops=1オプションは、-Knoalign_loopsオプションと等価です。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K*archi*

アーキテクチャを指定します。*archi*には、ARMV8_A、ARMV8_1_A、ARMV8_2_A、またはARMV8_3_Aを指定します。省略時は、-KARMV8_3_A オプションが適用されます。

-KARMV8_A

Armv8-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-KARMV8_1_A

Armv8-AおよびArmv8.1-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-KARMV8_2_A

Armv8-A、Armv8.1-A、およびArmv8.2-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-KARMV8_3_A

Armv8-A、Armv8.1-A、Armv8.2-A、およびArmv8.3-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-K{array_declaration_opt|noarray_declaration_opt}

最適化を行うときに、配列の添字が配列宣言の範囲を超えないことを前提にするか否かを指示します。-Karray_declaration_optオプションが指定された場合、前提にします。SIMD化など最適化が促進されます。省略時は、-Knoarray_declaration_optオプションが適用されます。

前提条件を満足していない場合に-Karray_declaration_optオプションを指定すると、実行結果は保証されません。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{array_private|noarray_private}

ループ内のプライベート化可能な配列に対して、プライベート化を行うか否かを指示します。-Karray_privateオプションが指定された場合、プライベート化します。省略時は、-Knoarray_privateオプションが適用されます。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

-Kassume={shortloop|noshortloop|memory_bandwidth|nomemory_bandwidth|time_saving_compilation|notime_saving_compilation}

-Kassumeオプションは、プログラムの特徴に合わせて、最適化を調整するか否かを指示するためのオプションです。複数の-Kassumeオプションを同時に指定することもできます。

省略時は、-Kassume=noshortloop,assume=nomemory_bandwidth,assume=notime_saving_compilationオプションが適用され、コンパイラは以下の方針で最適化を実施します。

- 最内ループの繰返し数が翻訳時に不明な場合は、繰返し数が大きいとみなす
- 最内ループをメモリバンド幅がボトルネックではないとみなし、CPU演算のボトルネックを優先的に解消する
- 翻訳時間を消費するプログラムでも実行可能プログラムの高速化を優先する

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kassume={shortloop|noshortloop}

プログラム中の最内ループにおいて、ループの繰返し数が翻訳時に不明な場合は繰返し数が小さいとみなすか否かを指示します。-Kassume=shortloopオプションが指定された場合、最内ループの繰返し数が翻訳時に不明なときは繰返し数が小さいとみなします。省略時は、-Kassume=noshortloopオプションが適用されます。

-Kassume=shortloopオプションを指定した場合は、自動並列化、ループアンローリング、ソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

-Kassume={memory_bandwidth|nomemory_bandwidth}

プログラム中の最内ループにおいて、メモリバンド幅がボトルネックになるとみなすか否かを指示します。-Kassume=memory_bandwidthオプションが指定された場合、メモリバンド幅がボトルネックになるとみなして最適化を実施します。省略時は、-Kassume=nomemory_bandwidthオプションが適用されます。

-Kassume=memory_bandwidthオプションを指定した場合は、zfillの最適化の促進およびソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

-Kassume={time_saving_compilation|notime_saving_compilation}

プログラムの翻訳時間が短くなるように最適化を調整するか否かを指示します。-Kassume=time_saving_compilationオプションが指定された場合、プログラムの翻訳時間が短くなるように最適化を調整します。省略時は、-Kassume=notime_saving_compilationオプションが適用されます。

-Kcmodel={small|large}

実行可能プログラムおよび共有オブジェクトの、コード領域と静的データ領域の最大値を指示します。省略時は、-Kcmodel=smallオプションが適用されます。

-Kcmodel=largeオプションと-K{PIC|pic}オプションを同時に指定することはできません。

-Kcmodel=small

リンク後のコード領域と静的データ領域の合計が4GB以内になると仮定し、効率の良いオブジェクトプログラムを生成します。

-Kcmodel=large

リンク後のコード領域が4GB以内になると仮定します。静的データ領域の大きさに制約はありません。静的データ領域が大きくリンク時にエラーが発生する場合は、本オプションを指定します。

-K{const|noconst}

const型修飾子が指定されたデータを、定数として取り扱うか否かを指示します。-Kconstオプションが指定された場合、定数として取り扱います。省略時は、-Kconstオプションが適用されます。

-K*cpu*

ターゲットのプロセッサを指定します。*cpu*には、A64FXまたはGENERIC_CPUを指定します。省略時は、-KA64FXオプションが適用されます。

-KA64FX

A64FXプロセッサ向けのオブジェクトファイルを出力するように指示します。-KA64FXオプションが有効な場合、-Khpctagオプションが有効になります。

-KGENERIC_CPU

Armプロセッサ向けのオブジェクトファイルを出力するように指示します。

-K{dynamic_iteration|nodynamic_iteration}

多重ループにおいて、外側ループと内側ループの両方で並列化可能な場合、並列化するループを動的に選択することを指示します。-Kdynamic_iterationオプションが指定された場合、動的に選択します。-Knodynamic_iterationオプションが指定された場合、外側のループが並列化されます。対象となるループのネストレベルは3までです。省略時は、-Knodynamic_iterationオプションが適用されます。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

-K{eval|noeval}

演算の評価方法を変更する最適化を行うか否かを指示します。-Kevalオプションが指定された場合、最適化を行います。省略時は、-Knoevalオプションが適用されます。

-Kevalオプションを指定した場合、実行結果に副作用(計算誤差および実行時の例外発生など)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

-Kevalオプションが有効な場合、-Kfsimpleオプションが有効になります。

-Kevalオプションが有効、かつ、-Kparallelオプションが指定されている場合、-Kreductionオプションも有効となります。

-Kevalオプションが有効、かつ、-Ksimd[={1|2|auto}]オプションが指定されている場合、-Ksimd_reduction_productオプションも有効となります。

-Knoevalオプションが指定された場合、-Knofsimpleオプション、-Knoreductionオプション、および-Ksimd_noreduction_productオプションも有効となります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kevalオプションを使用する際は、“3.3.1.2 演算評価方法の変更”をお読みください。

-K{eval_concurrent|eval_noconcurrent}

tree-height-reduction最適化において、命令の並列性を優先するか否かを指示します。

-Keval_concurrentオプションが指定された場合、tree-height-reduction最適化において、命令の並列性を優先することを指示します。-Keval_noconcurrentオプションが指定された場合、tree-height-reduction最適化において、命令の並列性を抑え、FMA命令の利用を優先することを指示します。省略時は、-Keval_noconcurrentオプションが適用されます。

本オプションは、-O1オプション以上が有効、かつ、-Kevalオプションが有効な場合に意味があります。

ループの繰返し数が小さく、ソフトウェアパイプライニングが動作しない場合に効果が期待できます。

tree-height-reduction最適化については、“3.3.9 tree-height-reduction最適化”をお読みください。

-K{extract_stride_store|noextract_stride_store}

SIMD化対象ループにあるストライドアクセスのストア命令を、スカラ命令に展開するか否かを指示します。-Kextract_stride_storeオプションが指示された場合、スカラ命令に展開します。省略時は、-Knoextract_stride_storeオプションが適用されます。

本機能が適用されたループでは、“3.2.7.4 SIMD長の倍数冗長実行によるSIMD化機能”が抑止されます。

-Kextract_stride_storeオプションを指定した場合、オブジェクトプログラムの大きさおよび翻訳時間が増加することがあります。

本オプションは、-O2オプション以上が有効な場合に意味があります。

-Kfast

本オプションは、ターゲットマシン上で高速に実行するオブジェクトプログラムを作成することを指示します。

本オプションは、

-O3 -Keval,fast_matmul,fp_contract,fp_relaxed,fz,ilfunc,mfunc,omitfp,simd_packed_promotion

を指定した場合と等価です。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

本オプションは、-Kevalオプション、-Kfast_matmulオプション、-Kfp_contractオプション、-Kfp_relaxedオプション、-Kilfuncオプションおよび-Kmfuncオプションの影響により、計算結果に副作用を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

-K{fast_matmul|nofast_matmul}

行列積のループを高速なライブラリ呼び出しに変換します。-Kfast_matmulオプションが指定された場合、最適化を行います。省略時は、-Knofast_matmulオプションが適用されます。

-Kfast_matmulオプションを指定した場合、実行結果に副作用(計算誤差)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O2オプション以上が有効な場合に意味があります。

ファイル名の並びに-Kfast_matmulオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-Kfast_matmulオプションを指定する必要があります。

-K{fconst|nofconst}

接尾語が指定されていない浮動小数点定数を、float型として扱うか否かを指示します。-Kfconstオプションが指定された場合、float型として扱います。省略時は、-Knofconstオプションが適用されます。

-K{fenv_access|nofenv_access}

プログラムが以下の動作をする可能性があるか否かを指示します。

ー 浮動小数点状態フラグを判定するために浮動小数点環境へアクセスする、または

ー 既定状態以外の浮動小数点制御モードの下で走行する

-Kfenv_accessオプションが指定された場合、これらの動作をする可能性があることを指示します。省略時は、-Knofenv_accessオプションが適用されます。

-Kfenv_accessオプションを指定した場合、いくつかの浮動小数点最適化が抑止されます。

-K{fp_contract|nofp_contract}

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うか否かを指示します。

-Kfp_contractオプションが指定された場合、最適化を行います。省略時は、-Knofp_contractオプションが適用されます。

-Kfp_contractオプションを指定した場合、実行結果に副作用(丸め誤差程度の違い)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{fp_precision|nofp_precision}

浮動小数点演算の計算誤差が生じないようなオプションの組合せを誘導するか否かを指示します。

-Kfp_precisionオプションが指定された場合、浮動小数点演算の計算誤差が生じないようなオプションの組合せを誘導します。省略時は、-Knofp_precisionオプションが適用されます。

-Kfp_precisionオプションは、-Kfp_precisionオプションを

-Knoeval,nofast_matmul,nofp_contract,nofp_relaxed,nofz,noilfunc,nomfunc,parallel_fp_precision

に置き換えた場合と等価です。そのため、-Kfp_precisionオプションが有効な場合、一部の最適化機能が制限されるので、実行性能が低下する可能性があります。

-Knofp_precisionオプションは、-Kfp_precisionオプションの指定を無効にしますが、-Kfp_precisionオプションが誘導する個々のオプションを同時に指定していても、その指定には影響を与えません。

-K{fp_relaxed|nofp_relaxed}

単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数で、逆数近似演算命令とFloating-Point Multiply-Add/Subtract命令を利用した逆数近似演算を行うか否かを指示します。-Kfp_relaxedオプションが指定された場合、逆数近似演算を行います。省略時は、-Knofp_relaxedオプションが適用されます。

-Kfp_relaxedオプションを指定した場合、実行結果に副作用を生じることがあります。また、-NRtrapオプションの指定に関わらず、プログラムの論理上は発生しない浮動小数点例外が発生することがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kfp_relaxedオプションを指定しても、-NRtrapオプションが有効、かつ-Knosimdオプションまたは-KNOSVEオプションのいずれかが有効である場合、sqrt関数を逆数近似演算に変換する最適化が抑止されます。したがって、-NRnotrapオプションが有効な場合と比べ、実行性能が低下することがあります。なお、-Knopreexオプションが有効な場合でも、-Kfp_relaxedオプションによって生成された逆数近似演算命令が先行評価されることがあります。また、-Knopreex、-Kfp_relaxed、および-NRtrapオプションがすべて有効な場合、浮動小数点例外が発生することがあります。

-K{fsimple|nofsimple}

ソースプログラムに対して浮動小数点演算の単純化を行うか否かを指示します。-Kfsimpleオプションが指定された場合、浮動小数点演算の単純化を行います。省略時は、-Knofsimpleオプションが適用されます。

-Kfsimpleオプションを指定した場合、実行結果に副作用を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{fz|nofz}

flush-to-zeroモードを使用するか否かを指示します。-Kfzオプションが指定された場合、flush-to-zeroモードを使用します。省略時は、-Knofzオプションが適用されます。

-Kfzオプションを指定した場合、演算結果またはソースオペランドが非正規化数のときにはそれらを同符号の0で置き換えます。また、実行結果に副作用を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。プログラムによっては実行性能が向上する場合があります。

本オプションは、プログラムのリンク時に指定する必要があります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{hpctag|nohpctag}

A64FXプロセッサのHPCタグアドレスオーバライド機能を利用するか否かを指示します。-Khpctagオプションが指定された場合、HPCタグアドレスオーバライド機能を利用します。省略時は-Khpctagオプションが適用されます。

HPCタグアドレスオーバライド機能を利用することで、セクタキャッシュ機能やハードウェアプリフェッチアシスト機能を有効にできます。

本オプションは、-KA64FXオプションが有効な場合に意味があります。

本オプションはプログラムの翻訳時およびリンク時に指定する必要があります。

-K{ilfunc[={loop|procedure}]|noilfunc}

数学関数をインライン展開するか否かを指示します。以下に、対象となる関数を示します。

- 単精度実数型および倍精度実数型の数学関数atan、atan2、cos、exp、log、log10、pow、sin、およびtan
- 単精度複素数型および倍精度複素数型の数学関数absおよびexp

-Klibオプションが有効、かつ-Kilfuncオプションが指定された場合、インライン展開します。

-Kilfuncオプションの={loop|procedure}が省略された場合、-Kilfunc=procedureオプションが適用されます。

省略時は、-Knoilfuncオプションが適用されます。

コンパイラは、性能低下が生じる可能性があると判断した場合、インライン展開を行いません。

本最適化によってインライン展開が行われた場合、実行結果に副作用を生じることがあります。また、-NRtrapオプションが無効な場合でも、プログラムの論理上は発生しない浮動小数点例外を発生させることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kilfunc=loop

ループに含まれる数学関数をインライン展開します。ループに含まれない数学関数はインライン展開されません。

-Kilfunc=procedure

関数内の数学関数をインライン展開します。

-Kinstance=*N*

実行時のスレッド数*N*を指定します。*N*は、2から512までの整数値です。

本オプションが有効な場合、*N*の値はプログラム実行時のスレッド数と同じでなければなりません。実行時のスレッド数が*N*の値と異なる場合、以下の実行時メッセージを出力し、実行を打ち切ります。

```
jwe1040i-s This program cannot be executed, because the number of threads specified by -Kinstance=N option and the
actual number of threads are not equal.
```

本オプションは、-Kparallelオプションが有効な場合に意味があります。

本オプションを使用する際は、“4.2.7.1 並列処理の入れ子での注意”をお読みください。

-K{largepage|nolargepage}

作成する実行可能プログラムがラージページ機能を使用するか否かを指示します。-Klargepageオプションが指定された場合、ラージページ機能を使用します。省略時は、-Klargepageオプションが適用されます。

本オプションは、リンク時に指定する必要があります。

ラージページについては、“付録L ラージページライブラリ”を参照してください。

-K{lib|nolib}

標準ライブラリ関数の動作を認識して、最適化を促進させるか否かを指示します。-Klibオプションが指定された場合、最適化を促進させます。-O0オプションが有効な場合、省略時は、-Knolibオプションが適用されます。-O1オプション以上が有効な場合、省略時は、-Klibオプションが適用されます。

利用者が標準ライブラリ関数と同名の関数を定義した場合、利用者の意図した結果にならない場合があります。

対象となる標準ライブラリ関数は、以下のとおりです。

```
abort, abs, acos, acosf, acosh, acoshf, asin, asinf, asinh, asinhf, atan, atan2, atan2f, atanf, atanh, atanhf, calloc,
cbrt, cbrtf, ceil, ceilf, clearerr, copysign, copysignf, cos, cosf, cosh, coshf, csqrt, csqrtf, erf, erfc, erfcf, erff,
exit, exp, exp2, exp2f, expf, expm1, expm1f, fabs, fabsf, fclose, fdim, fdimf, feof, ferror, fflush, fgetc, fgetpos,
fgets, floor, floorf, fma, fmaf, fmax, fmaxf, fmin, fminf, fmod, fmodf, fputc, fputs, free, frexp, frexpf, fseek,
fwrite, getenv, hypot, hypotf, ilogb, ilogbf, isalnum, isalpha, iscntrl, isdigit, isgraph, islower, ispunct, isspace,
isupper, isxdigit, ldexp, ldexpf, lgamma, lgammaf, llrint, llrintf, llround, llroundf, log, log10, log10f, log1p,
log1pf, log2, log2f, logb, logbf, logf, lrint, lrintf, lround, lroundf, malloc, memchr, memcmp, memcpy, memmove,
memset, modf, nearbyint, nearbyintf, nextafter, nextafterf, nexttoward, nexttowardf, perror, pow, powf, printf,
putchar, rand, realloc, remainder, remainderf, remove, remquo, remquof, rename, rint, rintf, round, roundf, scalbln,
scalblnf, scalbn, scalbnf, scanf, setvbuf, sin, sinf, sinh, sinhf, sprintf, sqrt, sqrtf, srand, sscanf, strcat, strchr,
strcmp, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtod, strtok,
strtol, strtoul, system, tan, tanf, tanh, tanhf, tgamma, tgammaf, tolower, toupper, trunc, truncf, vprintf, vsprintf
```

-K{loop_blocking[=*N*]|loop_noblocking}

ループブロッキングの最適化を行うか否かを指示します。-Kloop_blockingオプションが指定された場合、最適化を行います。-O1オプションが有効な場合、常に-Kloop_noblockingオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kloop_blockingオプションが適用されます。

-Kloop_blockingオプションが有効な場合、*N*にはブロックの大きさを指定します。*N*は、2から10000までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に最適な値を決定します。

本オプションは、-O1オプション以上が有効な場合に意味があります。

ループブロッキングについては、“3.2.5 ループブロッキング”をお読みください。

-K{loop_fission|loop_nofission}

ソフトウェアパイプライニングの促進、キャッシュメモリ利用効率の改善、およびレジスタ不足の解消のために、ループを複数のループに分割する最適化を行うか否かを指示します。-Kloop_fissionオプションが指定された場合、ループを複数のループに分割します。-O1オプションが有効な場合、常に-Kloop_nofissionオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kloop_fissionオプションが適用されます。

ループ分割の対象ループおよび分割位置は、以下の最適化指示子を用いて指定します。

- loop_fission_target指示子

  指定されたループを自動で分割します。

- fission_point指示子

  指定された実行文の位置でループを分割します。

loop_fission_target指示子およびfission_point指示子については、“3.4.1 最適化制御行(OCL)の利用”をお読みください。なお、最適化指示子を有効にするためには、-Koclオプションを指定する必要があります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

本オプションは、上記の最適化指示子と組み合わせて使用した場合にのみ、動作します。

本機能については、“3.3.10 ループ分割”をお読みください。

-K{loop_fission_stripmining[={*N*|L1|L2}]|loop_nofission_stripmining}

自動ループ分割時にストリップマイニングの最適化を行うか否かを指示します。-Kloop_fission_stripminingオプションが指定された場合、ストリップマイニングの最適化を行います。

-Kloop_fission_stripminingオプションの={*N*|L1|L2}が省略された場合、ストリップの長さはコンパイラが自動的に決定します。省略時は、-Kloop_nofission_stripminingオプションが適用されます。

本最適化により、ループ分割したループ間でアクセスされるデータに対して、キャッシュメモリの利用効率の向上が期待できます。

最適化制御行にloop_fission_target指示子が指定されており、-Kloop_fissionオプション、-Koclオプション、および-O2オプション以上が有効な場合に、本オプションは有効になります。

本機能については、“3.3.10.1 ストリップマイニング”をお読みください。

-Kloop_fission_stripmining=*N*

ストリップの長さを*N*にします。*N*は、2から100000000までの整数値です。

-Kloop_fission_stripmining=L1

キャッシュメモリの利用効率を考慮し、ストリップの長さを1次キャッシュのサイズに合わせます。

-Kloop_fission_stripmining=L2

キャッシュメモリの利用効率を考慮し、ストリップの長さを2次キャッシュのサイズに合わせます。

-Kloop_fission_threshold=*N*

自動ループ分割における分割後のループの粒度(ループ内の命令数やレジスタ数など)を決める閾値*N*を指示します。*N*は、1から100までの整数値です。*N*の値を小さくした場合、分割後のループが小さくなり、ループの分割数が増える傾向があります。省略時は、-Kloop_fission_threshold=50が適用されます。

最適化制御行にloop_fission_target指示子が指定されており、-Kloop_fissionオプション、-Koclオプション、および-O2オプション以上が有効な場合に、本オプションは有効になります。

-K{loop_fusion|loop_nofusion}

隣接するループを融合する最適化を行うか否かを指示します。-Kloop_fusionオプションが指定された場合、融合します。-O1オプションが有効な場合、常に-Kloop_nofusionオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kloop_fusionオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{loop_interchange|loop_nointerchange}

ループ交換する最適化を行うか否かを指示します。-Kloop_interchangeが指定された場合、ループ交換します。-O1オプションが有効な場合、常に-Kloop_nointerchangeオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kloop_interchangeオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{loop_part_parallel||loop_nopart_parallel}

ループ中に並列化できる部分と並列化できない部分が存在した場合、そのループを分割して自動並列化を行うか否かを指示します。-Kloop_part_parallelオプションが指定された場合、部分的に自動並列化します。省略時は-Kloop_nopart_parallelオプションが適用されます。

対象ループは最内ループです。ループを分割することにより、翻訳時間または実行時間が増加する場合があるため注意が必要です。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

-K{loop_part_simd|loop_nopart_simd}

ループ中にSIMD化できる部分とSIMD化できない部分が存在した場合、そのループを分割して部分的にSIMD化を行うか否かを指示します。-Kloop_part_simdオプションが指定された場合、部分的にSIMD化します。省略時は-Kloop_nopart_simdオプションが適用されます。

対象ループは最内ループです。ループを分割することにより、翻訳時間または実行時間が増加する場合があるため注意が必要です。

本オプションは、-Ksimd[={1|2|auto}]オプションが有効な場合に意味があります。

-K{loop_perfect_nest|loop_noperfect_nest}

不完全多重ループを分割して完全多重ループにするか否かを指示します。-Kloop_perfect_nestオプションが指定された場合、不完全多重ループを完全多重ループにします。-O2オプションが有効な場合、省略時は-Kloop_noperfect_nestオプションが適用されます。-O3オプションが有効な場合、省略時は-Kloop_perfect_nestオプションが適用されます。

本機能により、ループ交換、ループ一重化などの最適化が促進されます。

本オプションは、-O2オプション以上が有効な場合に意味があります。

-K{loop_versioning|loop_noversioning}

ループバージョニングの最適化を行うか否かを指示します。-Kloop_versioningオプションが指定された場合、最適化を行います。省略時は、-Kloop_noversioningオプションが適用されます。

本機能により、SIMD化、ソフトウェアパイプライニング、または自動並列化などの最適化が促進されます。

ループバージョニングは、ループを複数生成するため、オブジェクトプログラムの大きさおよび翻訳時間が増加する場合があります。また、生成したループを選択するための判定処理がオーバヘッドとなり、実行性能が低下する場合があります。

本オプションは、-O2オプション以上が有効な場合に意味があります。

ループバージョニングについては、“3.3.6 ループバージョニング”をお読みください。

-Klooptype={f|n|s}

ループの継続条件および方向を指定します。本オプションを指定することで、最適化が促進される可能性があります。省略時は、-Klooptype=fオプションが適用されます。

-Klooptype=f

継続条件が常に真となるループはないが、逆向きループが含まれる可能性があることを指示します。

-Klooptype=n

継続条件が常に真となるループおよび逆向きループが含まれる可能性があることを指示します。

-Klooptype=s

継続条件が常に真となるループおよび逆向きループがないことを指示します。

-K{memalias|nomemalias}

ポインタからの間接参照を行う場合に、型が一致しない変数同士は違う領域をアクセスするものとみなすか否かを指示します。本オプションを指定することで、最適化が促進される可能性があります。-Kmemaliasオプションが指定された場合、違う領域をアクセスするものとみなします。省略時は、-Knomemaliasオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

本オプションは非推奨です。今後、廃止される予定です。

-K{mfunc[={1|2|3}]|nomfunc}

関数をマルチ演算関数に変換する最適化を行うか否かを指示します。マルチ演算関数とは、1回の呼出しで複数の引数に対する同種の関数計算を行うことにより、実行性能を向上させた関数です。-Kmfuncオプションの={1|2|3}が省略された場合、-Kmfunc=1が指定されたものとみなします。-Klibオプションが有効、かつ、-Kmfuncオプションが指定された場合、マルチ演算関数の候補となります。省略時は、-Knomfuncオプションが適用されます。

-Kmfuncオプションを指定した場合、実行結果に副作用を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-O2オプション以上が有効な場合に意味があります。

以下に対象となる関数を示します。

関数群	float型	double型
acos	○(注1)	○(注1)
asin	○(注1)	○(注1)
atan	○	○
atan2	○	○
cos	○	○
erf	○(注1)	○(注1)
erfc	○(注1)	○(注1)
exp	○	○
log	○	○
log10	○	○
sin	○	○
pow	○	○

○: 対象、×: 対象外

注1) マルチ演算関数内で演算を逐次実行しているため、通常のマルチ演算関数と比べて実行性能向上率は低いですが、分岐の最適化などで実行性能が向上します。

-Kmfunc=1

引数の多重度をSIMD長と同じ値としたマルチ演算関数を使用します。

-Kmfunc=2

-Kmfunc=1の機能に加え、多重度をコンパイラが自動的に決定する値としたマルチ演算関数も使用します。

本機能は、スタック領域を多く必要とします。

-Kmfunc=3

-Kmfunc=2の機能に加え、if文などを含むループに対しても多重度をコンパイラが自動的に決定する値としたマルチ演算関数を使用します。

if文の真率が低い場合、-Kmfunc=1または-Kmfunc=2より実行性能が低下することがあるので注意が必要です。また、この最適化を行うことで実行時異常終了となる副作用を生じることがあるため注意が必要です。副作用については"3.3.3.2 -Kmfunc=3オプションの影響"をお読みください。

本機能は、スタック領域を多く必要とします。

-Knoprefetch

prefetch命令を使用したオブジェクトを生成しないことを指示します。

-K{ocl|noocl}

最適化制御行(OCL)を有効にするか否かを指示します。-Koclオプションが指定された場合、有効にします。省略時は、-Knooclオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{omitfp|noomitfp}

関数呼出しにおける、フレームポインタレジスタを保証しない最適化を行うか否かを指示します。-Komitfpオプションが指定された場合、最適化を行います。省略時は、-Knoomitfpオプションが適用されます。

-Komitfpオプションを指定した場合、トレースバック情報は保証されません。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{openmp|noopenmp}

OpenMP仕様の指示文(以降、OpenMP指示文と呼びます)を有効にするか否かを指示します。-Kopenmpオプションが指定された場合、有効にします。省略時は、-Knoopenmpオプションが適用されます。

-Kopenmpオプションが指定された場合、-mtオプションが指定されたものとみなされます。

ファイル名の並びに-Kopenmpオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-Kopenmpオプションを指定する必要があります。

-O1オプション以下が有効な場合、simd構文またはdeclare simd構文が有効であったとしても、SIMD化は行われません。

-K{openmp_assume_norecurrence|openmp_noassume_norecurrence}

OpenMPのfor指示文が指定されたループに対して、チャンク内の配列要素が回転を跨いで定義引用されないと解釈し、最適化を促進するか否かを指示します。-Kopenmp_assume_norecurrenceオプションが指定された場合、最適化を行います。省略時は、-Kopenmp_noassume_norecurrenceオプションが適用されます。

本機能により、SIMD化、ソフトウェアパイプライニングなどの最適化が促進されます。

本オプションは、最内ループにfor指示文が指定されている場合のみ対象とします。

本オプションは、-Kopenmpオプションおよび-O2オプション以上が有効な場合に意味があります。

-K{openmp_collapse_except_innermost|openmp_nocollapse_except_innermost}

OpenMPのループ構文において、以下の条件をすべて満たしている場合に、最内ループをcollapseの対象から外すか否かを指示します。

ー 最内ループまで含めたcollapse指示節が指定されている。

ー 最内ループまで含めたcollapseにより実行性能が低下する可能性があると、コンパイラが翻訳時に判断できる。

-Kopenmp_collapse_except_innermostオプションが指定された場合、最内ループをcollapseの対象から外します。省略時は、-Kopenmp_nocollapse_except_innermostオプションが適用されます。

-Kopenmp_collapse_except_innermostオプションにより、collapseによる実行性能の低下を防止できる場合があります。

本オプションは、-Kopenmpオプションが有効な場合に意味があります。

また、-Koptmsg=2オプションを有効にすることにより、collapseの対象から外した最内ループを翻訳時の診断メッセージで知ることができます。

-Kopenmp_loop_variable={private|standard}

OpenMPのparallel構文内にある逐次ループの制御変数を、privateとして扱うかsharedとして扱うかを指示します。省略時は-Kopenmp_loop_variable=privateオプションが適用されます。

本オプションは、-Kopenmpオプションが有効な場合に意味があります。

-Kopenmp_loop_variable=private

OpenMPのparallel構文内にある逐次ループの制御変数をprivateとして扱います。

ループの制御変数の扱いは規格非準拠となります。

-Kopenmp_loop_variable=standard

OpenMPのparallel構文内にある逐次ループの制御変数をsharedとして扱います。

ループの制御変数の扱いは規格準拠となります。

-K{openmp_ordered_reduction|openmp_noordered_reduction}

OpenMPのreduction指示節が指定されたリージョンの終わりにおけるリダクション演算の演算順序を、スレッド番号順に固定するか否かを指示します。-Kopenmp_ordered_reductionオプションが指定された場合、スレッド番号順に固定します。省略時は、-Kopenmp_noordered_reductionオプションが適用されます。

リダクション演算の演算順序をスレッド番号順に固定することにより、使用されるスレッド数が同じであれば、常に同じ結果を得ることができます。ただし、スケジュール種別がdynamicまたはguidedであるループ構文、またはsections構文のスケジューリングの影響によって演算順序が変わる場合、丸め誤差を生じることがあります。また、演算順序を固定しない場合に比べて、実行性能が低下する場合があります。

本オプションは、-Kopenmpオプションが有効な場合に意味があります。

-K{openmp_simd|noopenmp_simd}

OpenMP仕様のsimd構文およびdeclare simd構文のみを有効にするか否かを指示します。省略時は、-Knoopenmp_simdオプションが適用されます。

-Kopenmp_simdオプションが指定された場合、OpenMP仕様のsimd構文およびdeclare simd構文のみを有効にします。OpenMP仕様によるSIMD化だけを行い、並列化は行いません。

-Knoopenmp_simdオプションが指定された場合、-Kopenmp_simdオプションを無効にします。

-O1オプション以下が有効な場合、simd構文またはdeclare simd構文が有効であったとしても、SIMD化は行われません。

-K{optlib_string|nooptlib_string}

文字列操作関数において、最適化版のライブラリをリンクするか否かを指示します。-Koptlib_stringオプションが指定された場合、最適化版のライブラリを静的にリンクします。省略時は、-Knooptlib_stringオプションが適用されます。

-Koptlib_stringオプションは、-KSVEオプションおよび-KA64FXオプションと同時に指定した場合に有効になります。

本オプションはプログラムのリンク時に指定する必要があります。

以下に、対象となる文字列操作関数を示します。

```
bcopy, bzero, memchr, memcmp, memccpy, memcpy, memmove, memset, strcat, strcmp, strcpy, strlen, strncmp, strncpy,
strncat
```

-K{optmsg[={1|2}]|nooptmsg}

最適化状況をメッセージ出力するか否かを指示します。-Koptmsgオプションが指定された場合、出力します。-Koptmsgオプションの=｛1|2｝が省略された場合、-Koptmsg=1が指定されたものとみなします。省略時は、-Knooptmsgオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Koptmsg=1

実行結果に副作用を生じる可能性がある最適化をしたことをメッセージ出力します。

-Koptmsg=2

-Koptmsg=1オプションのメッセージに加えて、自動並列化、SIMD化、ループアンローリングなどの最適化機能が動作したことをメッセージ出力します。

-K{parallel|noparallel}

自動並列化を行うか否かを指示します。-Kparallelオプションが指定された場合、自動並列化を行います。ただし、並列化の効果が期待できないものに対しては、自動並列化を行いません。省略時は、-Knoparallelオプションが適用されます。

-Kparallelオプションを指定した場合、-O2オプション、-Kregion_extensionオプション、-Kloop_part_parallelオプション、-Kloop_perfect_nestオプション、および-mtオプションも有効になります。ただし、-O3オプションを同時に指定した場合は、-O3オプションが有効になります。

本オプションは、-O0または-O1オプションが有効な場合、無効となります。

ファイル名の並びに-Kparallelオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-Kparallelオプションを指定する必要があります。

-K{parallel_fp_precision|parallel_nofp_precision}

スレッド並列数の変化による浮動小数点型または複素数型の演算結果に計算誤差を考慮し、最適化の適用をコンパイラが制御するか否かを指示します。-Kparallel_fp_precisionオプションが指定された場合、計算誤差が発生しない範囲の最適化を適用します。省略時は、-Kparallel_nofp_precisionオプションが適用されます。

本オプションは、-Kparallelまたは-Kopenmpオプションが有効な場合に意味があります。

-Kparallel_fp_precisionオプションおよび-Kopenmpオプションが有効な場合、-Kopenmp_ordered_reductionオプションが有効となります。

-Kparallel_fp_precisionオプションを指定した場合、一部の最適化機能が制限されるため、実行性能が低下する可能性があります。

OpenMPのreduction指示節が指定された場合には、-Kparallel_fp_precisionオプションが有効であったとしても、スレッド並列数の変化による浮動小数点型または複素数型の演算結果に計算誤差が発生する可能性があるため注意が必要です。

-Kparallel_iteration=*N*

翻訳時に繰返し数が*N*以上と判明しているループのみを並列化の対象とすることを指示します。*N*は、1から2147483647までの整数値です。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

-Kparallel_strong

並列化の効果に関係なく、自動並列化が可能なループをすべて並列化することを指示します。

-Kevalオプションおよび-Kpreexオプションも同時に有効になります。

上記を除き、機能や注意事項は、-Kparallelオプションと同じです。

-K{pc_relative_literal_loads|nopc_relative_literal_loads}

関数内のコード領域が1MB以内であるとして扱い、リテラルプールに1命令でアクセスします。省略時は-Knopc_relative_litesral_loadsオプションが適用されます。

本オプションは、翻訳時の指定だけが有効になります。

-K{plt|noplt}

位置独立コード(PIC)での外部シンボルへのアクセスにProcedure Linkage Table(PLT)を使用するか否かを指示します。省略時は、-Kpltオプションが適用されます。

本オプションは、-KPICオプションまたは-Kpicオプションが有効な場合に意味があります。

-K{preex|nopreex}

ソースプログラムに対して不変式の先行評価の最適化を行うか否かを指示します。-Kpreexオプションが指定された場合、最適化を行います。省略時は、-Knopreexオプションが適用されます。

-Kpreexオプションを指定した場合、プログラムの論理上、実行されないはずの命令が実行され、実行結果に副作用(実行時の例外発生など)を生じることがあります。本機能による副作用については、“3.3.1.1 不変式の先行評価”をお読みください。

本オプションは、-O1オプション以上が有効な場合に意味があります。なお、-Knopreex、-Kfp_relaxed、および-NRtrapオプションがすべて有効な場合、浮動小数点例外が発生することがあります。

-Kprefetch_cache_level={1|2|all}

どのキャッシュレベルにデータをプリフェッチするかを指示します。省略時は、-Kprefetch_cache_level=allオプションが適用されます。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が有効な場合に意味があります。

-Kprefetch_cache_level=1

データを1次キャッシュにプリフェッチすることを指示します。1次キャッシュにのみプリフェッチするprefetch命令を使用します。

-Kprefetch_cache_level=2

データを2次キャッシュにのみプリフェッチすることを指示します。2次キャッシュにのみプリフェッチするprefetch命令を使用します。

-Kprefetch_cache_level=all

-Kprefetch_cache_level=1オプションおよび-Kprefetch_cache_level=2オプションの機能を同時に有効にします。2種類のprefetch命令を組み合わせることにより、より効果的なプリフェッチを実現することができます。

-K{prefetch_conditional|prefetch_noconditional}

if文やswitch文に含まれるブロックの中で使用される配列データに対してprefetch命令を生成するか否かを指示します。-Kprefetch_conditionalオプションが指定された場合、生成します。省略時は、-Kprefetch_noconditionalオプションが適用されます。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が有効な場合に意味があります。

-K{prefetch_indirect|prefetch_noindirect}

ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を使用したオブジェクトを生成するか否かを指示します。-Kprefetch_indirectオプションが指定された場合、生成します。省略時は、-Kprefetch_noindirectオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{prefetch_infer|prefetch_noinfer}

プリフェッチの距離が不明な場合でも連続アクセスのプリフェッチを出力するか否かを指示します。-Kprefetch_inferオプションが指定された場合、プリフェッチを出力します。省略時は、-Kprefetch_noinferオプションが適用されます。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が有効な場合に意味があります。

-Kprefetch_iteration=*N*

prefetch命令を生成する際、ループの*N*回転後に引用されるデータを対象とすることを指示します。*N*は、1から10000までの整数値です。SIMD化が適用されるループの場合、*N*にはSIMD化されたあとのループの繰返し数を指定してください。

本オプションは、1次キャッシュにのみプリフェッチするprefetch命令を対象とします。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=1または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

本オプションは、-Kprefetch_lineオプションと同時には指定できません。

-Kprefetch_iteration_L2=*N*

prefetch命令を生成する際、ループの*N*回転後に引用されるデータを対象とすることを指示します。*N*は、1から10000までの整数値です。SIMD化が適用されるループの場合、*N*にはSIMD化されたあとのループの繰返し数を指定してください。

本オプションは、2次キャッシュにのみプリフェッチするprefetch命令を対象とします。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=2または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

本オプションは、-Kprefetch_line_L2オプションと同時には指定できません。

-Kprefetch_line=*N*

prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示します。*N*は、1から100までの整数値です。

本オプションは、1次キャッシュにプリフェッチするprefetch命令を対象とします。

本オプションは、-Kprefetch_indirectまたは-Kprefetch_sequentialオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=1または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

本オプションは、-Kprefetch_iterationオプションと同時には指定できません。

-Kprefetch_line_L2=*N*

prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示します。*N*は、1から100までの整数値です。

本オプションは、2次キャッシュにのみプリフェッチするprefetch命令を対象とします。

本オプションは、-Kprefetch_indirectまたは-Kprefetch_sequentialオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=2または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

本オプションは、-Kprefetch_iteration_L2オプションと同時には指定できません。

-K{prefetch_sequential[={auto|soft}]|prefetch_nosequential}

ループ内で使用される連続的にアクセスされる配列データに対して、prefetch命令を使用したオブジェクトを生成するか否かを指示します。-Kprefetch_sequentialオプションが指定された場合、生成します。-O1オプションが有効な場合、省略時は、-Kprefetch_nosequentialオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kprefetch_sequentialオプションが適用されます。

-Kprefetch_sequentialオプションの={auto|soft}が省略された場合、-Kprefetch_sequential=autoオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kprefetch_sequential=auto

ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用するか、prefetch命令を出力するかをコンパイラが自動的に選択します。

-Kprefetch_sequential=soft

ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用せずに、prefetch命令を出力します。

-K{prefetch_stride[={soft|hard_auto|hard_always}]|prefetch_nostride}

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、プリフェッチを実施するか否かを指示します。プリフェッチするアドレスが翻訳時に確定しないループを含みます。-Kprefetch_strideオプションが指定された場合、実施します。省略時は、-Kprefetch_nostrideオプションが適用されます。

-Kprefetch_strideオプションの={soft|hard_auto|hard_always}が省略された場合、-Kprefetch_stride=softオプションが適用されます。

-Kprefetch_stride=softオプションおよび-Kprefetch_nostrideオプションは、-O1オプション以上が有効な場合に意味があります。

-Kprefetch_stride=hard_autoオプションおよび-Kprefetch_stride=hard_alwaysオプションは、-Khpctagオプションおよび-O1オプション以上が有効な場合に意味があります。

-Kprefetch_stride=soft

prefetch命令を生成して、プリフェッチを実施します。

-Kprefetch_stride=hard_auto

ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施します。

本オプションを指定した場合、キャッシュ上にないデータのみプリフェッチするようハードウェアストライドプリフェッチャーを設定します。

ハードウェアストライドプリフェッチャーの利用については、“3.6.3 ハードウェアストライドプリフェッチャーを利用する最適化”をお読みください。

-Kprefetch_stride=hard_always

ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施します。

本オプションを指定した場合、-Kprefetch_stride=hard_autoオプションと異なり、常にプリフェッチを行うようハードウェアストライドプリフェッチャーを設定します。

ハードウェアストライドプリフェッチャーの利用については、“3.6.3 ハードウェアストライドプリフェッチャーを利用する最適化”をお読みください。

-K{prefetch_strong|prefetch_nostrong}

1次キャッシュに対するprefetch命令を生成する際、strong prefetch命令を生成するか否かを指示します。-Kprefetch_strongが指定された場合、生成します。省略時は、-Kprefetch_strongが適用されます。

本オプションは、1次キャッシュにプリフェッチするprefetch命令のみを対象とします。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=1または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

-Kprefetch_nostrongオプションは、-Khpctagオプションが有効な場合に意味があります。

-K{prefetch_strong_L2|prefetch_nostrong_L2}

2次キャッシュに対するprefetch命令を生成する際、strong prefetch命令を生成するか否かを指示します。-Kprefetch_strong_L2が指定された場合、生成します。省略時は、-Kprefetch_strong_L2が適用されます。

本オプションは、2次キャッシュにプリフェッチするprefetch命令のみを対象とします。

本オプションは、-Kprefetch_indirect、-Kprefetch_sequential、または-Kprefetch_strideオプションのうちの1つ以上が指定されている、かつ、-Kprefetch_cache_level=2または-Kprefetch_cache_level=allオプションが指定されている場合に有効になります。

-Kprefetch_nostrong_L2オプションは、-Khpctagオプションが有効な場合に意味があります。

-K{preload|nopreload}

ロード命令を投機実行する最適化を行うか否かを指示します。本最適化を行うことで、命令スケジューリングの最適化が促進され、実行性能の向上が期待できます。

-Kpreloadオプションが指定された場合、最適化を行います。省略時は、-Knopreloadオプションが適用されます。

-Kpreloadオプションを指定した場合、プログラムの論理上、実行されないはずのロード命令が実行され、不当な領域を参照してプログラムの実行が中断することがあります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{rdconv[={1|2}]|nordconv}

ループの繰返しに伴って一定値ずつ増加または減少する4バイト以下の整数型の式が、オーバフローまたはラップアラウンドしないと仮定した最適化を行うか否かを指示します。

-Krdconvオプションが指定された場合、最適化を行います。-Krdconvオプションの={1|2}が省略された場合、-Krdconv=1オプションが指定されたものとみなします。

-O2オプション以上が有効な場合、省略時は、-Krdconv=1オプションが適用されます。

該当する式がオーバフローまたはラップアラウンドする場合、-Krdconvオプションの有無により結果が異なることがあります。

本オプションは、-O2オプション以上が有効な場合に意味があります。

-Krdconv=1

ループの繰返しに伴って一定値ずつ増加または減少する4バイト以下の符号付き整数型の式がオーバフローしないと仮定した最適化を行います。

-Krdconv=2

-Krdconv=1オプションの機能に加え、ループの繰返しに伴って一定値ずつ増加または減少する4バイト以下の符号無し整数型の式がラップアラウンドしないと仮定した最適化を行います。

-K{reduction|noreduction}

リダクションの最適化を行うか否かを指示します。-Kreductionオプションが指定された場合、最適化を行います。省略時は、-Knoreductionオプションが適用されます。

-Kreductionオプションを指定した場合、実行結果に副作用(計算誤差)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

-Kreductionオプションを使用する際は、“4.2.1.1 自動並列化のための翻訳時オプション”をお読みください。

-K{region_extension|noregion_extension}

自動並列化によるオーバヘッドを削減するために、パラレルリージョンを拡大する最適化を行うか否かを指示します。-Kregion_extensionオプションが指定された場合、最適化を行います。省略時は、-Knoregion_extensionオプションが適用されます。

-Kregion_extensionオプションを指定した場合、並列化効果の小さいループでは、実行性能が低下する場合があります。

本オプションは、-Kparallelオプションが有効な場合に意味があります。

本機能の詳細については、“4.2.5.10 パラレルリージョンの拡大”をお読み下さい。

-K{restp[={all|arg|restrict}]|norestp}

ポインタの最適化を行うか否かを指示します。-Krestpオプションが指定された場合、最適化を行います。省略時は、-Krestp=restrictオプションが適用されます。

-Krestpオプションを指定した場合、実行結果に副作用(計算誤りまたは実行時の例外発生など)を生じることがあります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Krestpオプションを使用する際は、“3.3.2 ポインタの最適化”をお読みください。

-Krestp=all

すべてのポインタがrestrict修飾されているものとみなして、最適化を行います。

-Krestp=arg

引数として現れたポインタがrestrict修飾されているものとみなして、最適化を行います。

-Krestp=restrict

ソースプログラムで明にrestrict修飾子の指定されたポインタだけがrestrict修飾されているものとみなして、最適化を行います。

-Krestp

引数として現れたポインタおよびソースプログラムで明にrestrict修飾子の指定されたポインタがrestrict修飾されているものとみなして、最適化を行います。

-K{sch_post_ra|nosch_post_ra}

レジスタ割付け後に、命令スケジューリングの最適化を行うか否かを指示します。実行する命令の順序を入れ替えることで、実行性能を向上させます。-Ksch_post_raオプションが指定された場合、最適化を行います。-O0オプションが有効な場合、常に-Knosch_post_raオプションが適用されます。-O1オプション以上が有効な場合、省略時は、-Ksch_post_raオプションが適用されます。

-Ksch_post_raオプションを指定しても、レジスタのメモリへの退避・復元命令は増加しません。

-K{sch_pre_ra|nosch_pre_ra}

レジスタ割付け前に、命令スケジューリングの最適化を行うか否かを指示します。実行する命令の順序を入れ替えることで、実行性能を向上させます。-Ksch_pre_raオプションが指定された場合、最適化を行います。-O0オプションが有効な場合、常に-Knosch_pre_raオプションが適用されます。-O1オプション以上が有効な場合、省略時は、-Ksch_pre_raオプションが適用されます。

-Ksch_pre_raオプションを指定した場合、レジスタのメモリへの退避・復元命令が増加し、実行性能が低下することがあります。

-K{sibling_calls|nosibling_calls}

末尾呼出しの最適化を行うか否かを指示します。-Ksibling_callsオプションが指定された場合、最適化を行います。-O1オプションが有効な場合、省略時は、-Knosibling_callsオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Ksibling_callsオプションが適用されます。

-Ksibling_callsオプションを指定した場合、トレースバック情報は保証されません。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-K{simd[={1|2|auto}]|nosimd}

SIMD拡張命令を利用したオブジェクトを生成するか否かを指示します。-Ksimd[={1|2|auto}]オプションが指定された場合、ループ内の命令に対してSIMD拡張命令を利用したオブジェクトを生成します。

-Ksimdオプションの={1|2|auto}が省略された場合、-Ksimd=autoオプションが指定されたものとみなします。

-O2オプション以上が有効な場合、省略時は、-Ksimd=autoオプションが適用されます。

-Ksimd[={1|2|auto}]オプションが有効な場合、-Kloop_part_simdオプションも有効になります。

本オプションは、-O2オプション以上が有効な場合に意味があります。

SIMD化については、“3.2.7 SIMD化”をお読みください。

-Ksimd=1

SIMD拡張命令を利用したオブジェクトを生成することを指示します。

-Ksimd=2

-Ksimd=1オプションの機能に加え、if文などを含むループに対して、SIMD拡張命令を利用したオブジェクトを生成することを指示します。

if文内の命令を冗長実行するため、if文の真率によっては実行性能が低下する場合があります。

また、if文内の式を投機実行するため、プログラムの論理上、実行されないはずの命令が実行され、実行結果に副作用(実行時の例外発生など)を生じることがあります。

-Ksimd=auto

ループをSIMD化するか否かをコンパイラが自動的に判定することを指示します。if文を含むループのSIMD化が促進されます。

-K{simd_packed_promotion|simd_nopacked_promotion}

単精度浮動小数点型ならびに4バイト整数型の配列要素のインデックス計算が4バイトの範囲を超えないと仮定して、packed-SIMD化を促進するか否かを指示します。-Ksimd_packed_promotionが指定された場合、packed-SIMD化を促進します。省略時は-Ksimd_nopacked_promotionが適用されます。

配列要素のインデックス計算が4バイトの範囲を超える場合、不当な領域をアクセスし、実行時異常終了または実行結果に誤りが生じることがあります。

本オプションは、-Ksimd[={1|2|auto}]オプションが有効な場合に意味があります。

-K{simd_reduction_product|simd_noreduction_product}

乗算のリダクション演算に対して、SIMD化を行うか否かを指示します。-Ksimd_reduction_productオプションが指定された場合、乗算のリダクション演算に対して、SIMD化を行います。

省略時は、-Ksimd_noreduction_productオプションが適用されます。

-Ksimd_reduction_productオプションを指定した場合、実行結果に副作用(計算誤差)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

本オプションは、-Ksimd[={1|2|auto}]オプションが有効な場合に意味があります。

本オプションは、-Ksimd_reg_size=agnosticオプションと同時には指定できません。

-Ksimd_reg_size={128|256|512|agnostic}

SVEのベクトルレジスタサイズを指定します。単位はビットです。

-Ksimd_reg_size={128|256|512}オプションが指定された場合、翻訳時にベクトルレジスタのサイズを指定された固定値とみなして最適化を実施します。そのため、最適化が促進され、実行性能の向上が期待されます。ただし、生成した実行可能プログラムは、翻訳時に指定したサイズのベクトルレジスタのSVEを実装しているCPUアーキテクチャにおいてのみ正常に動作します。詳細は、“3.6.2 SVEのベクトルレジスタのサイズを指定する場合の注意事項”をお読みください。

-Ksimd_reg_size=agnosticオプションが指定された場合、SVEのベクトルレジスタを特定のサイズとみなさず翻訳を行い、実行時にベクトルレジスタサイズを決定する実行可能プログラムを作成します。この実行可能プログラムは、CPUアーキテクチャに実装されたSVEのベクトルレジスタサイズによらず実行可能です。ただし、-Ksimd_reg_size={128|256|512}オプションを指定した場合と比べて、実行性能が低下する場合があります。

省略時は、-Ksimd_reg_size=512オプションが適用されます。

本オプションは、-KSVEオプションが有効な場合に意味があります。

-K{simd_uncounted_loop|simd_nouncounted_loop}

SIMD拡張命令を利用したオブジェクトを生成するか否かを指示します。-Ksimd_uncounted_loopオプションが指定された場合、whileループ、do-whileループ、if-gotoループ、およびループ外への飛出しがあるforループ内の命令に対してSIMD拡張命令を利用したオブジェクトを生成します。省略時は-Ksimd_nouncounted_loopオプションが適用されます。

本オプションは、-Ksimd[={1|2|auto}]オプションおよび-KSVEオプションが有効な場合に意味があります。

-K{simd_use_multiple_structures|simd_nouse_multiple_structures}

SIMD化の際、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用するか否かを指示します。-Ksimd_use_multiple_structuresオプションが指定された場合、上記の命令を利用します。SIMD化対象のロードおよびストアに上記の命令を利用することで、実行性能を向上させます。ただし、データのアライメントによっては実行性能が劣化する場合があります。

省略時は、-Ksimd_use_multiple_structuresオプションが適用されます。

本オプションは、-Ksimd[={1|2|auto}]オプションおよび-KSVEオプションが有効な場合に意味があります。

-K{strict_aliasing|nostrict_aliasing}

言語規格で規定された厳密なaliasing ruleに従って、メモリ領域の重なりを考慮した最適化を許すか否かを指示します。対象となる最適化は、不変式の移動や共通式の除去などです。省略時は-Knostrict_aliasingオプションが適用されます。

本オプションは、-O2オプション以上が有効な場合に意味があります。

-Kstrict_aliasingオプションを使用する際は、“3.3.11 Strict Aliasing”をお読みください。

-K{striping[=*N*]|nostriping}

ループストライピングの最適化を行うか否かを指示します。-Kstripingオプションが指定された場合、最適化を行います。*N*はストライピング展開数です。*N*は、2から100までの整数値です。-Kstripingオプションの[=*N*]が省略された場合、-Kstriping=2が指定されたものとみなします。-O1オプションが有効な場合、常に-Knostripingオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Knostripingオプションが適用されます。-Kstripingオプションは、-O2オプション以上と同時に指定した場合に有効となります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

ループストライピングについては、“3.3.4 ループストライピング”をお読みください。

-K{swp|noswp}

ソフトウェアパイプライニングの最適化を行うか否かを指示します。-Kswpオプションが指定された場合、最適化を行います。ただし、最適化の効果が期待できないものに対しては、最適化を行いません。-O1オプションが有効な場合、常に-Knoswpオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kswpオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-Kswpオプションを、-Kswp_weakオプションまたは-Kswp_strongオプションと同時に指定した場合、後に指定した方が有効となります。

ソフトウェアパイプライニングについては、“3.2.6 ソフトウェアパイプライニング”をお読みください。

-K{swp_freg_rate=*N*|swp_ireg_rate=*N*|swp_preg_rate=*N*}

以下のレジスタについて、ソフトウェアパイプライニングで使用可能な割合(百分率)を指示します。

- 浮動小数点レジスタおよびSVEのベクトルレジスタ
- 整数レジスタ
- SVEのプレディケートレジスタ

*N*は、1から1000までの整数値です。省略時は、-Kswp_freg_rate=100,swp_ireg_rate=100,swp_preg_rate=100オプションが適用されます。

レジスタ数に関する条件を変更することで、ソフトウェアパイプライニングの適用を調整できます。レジスタが不足するためソフトウェアパイプライニングが適用されない場合、100より大きな整数値を本オプションに指定することで、ソフトウェアパイプライニングが適用できることがあります。

本オプションを指定することで、レジスタのメモリへの退避・復元命令が増加し、実行性能が低下する場合があります。

本オプションは、-O2オプション以上が有効な場合に意味があります。

-Kswp_freg_rate=*N*

ソフトウェアパイプライニングを適用する際、浮動小数点レジスタならびにSVEのベクトルレジスタの*N*%が使用可能であると指示します。

-Kswp_ireg_rate=*N*

ソフトウェアパイプライニングを適用する際、整数レジスタの*N*%が使用可能であると指示します。

-Kswp_preg_rate=*N*

ソフトウェアパイプライニングを適用する際、SVEのプレディケートレジスタの*N*%が使用可能であると指示します。

-Kswp_policy={auto|small|large}

ソフトウェアパイプライニングで使用する命令スケジューリングアルゴリズムの選択基準を指示します。

ソフトウェアパイプライニングは、-Kswpオプション、-Kswp_weakオプション、-Kswp_strongオプション、またはそれぞれのオプションに対応した最適化制御行が有効な場合に行われます。

省略時は、-Kswp_policy=autoオプションが適用されます。

-Kswp_policy=auto

ループ毎に命令スケジューリングアルゴリズムを自動で選択します。

-Kswp_policy=small

小さなループ(例えば、必要レジスタ数が少ないループ)に適した命令スケジューリングアルゴリズムを使用します。

-Kswp_policy=large

大きなループ(例えば、必要レジスタ数が多いループ)に適した命令スケジューリングアルゴリズムを使用します。

-Kswp_strong

対象のループに対するソフトウェアパイプライニング適用の条件を緩和し、ソフトウェアパイプライニングを促進することを指示します。

本オプションを指定することで、翻訳メモリや翻訳時間が大幅に増加する場合があります。

-Kswp_strongオプションを、-Kswpオプションまたは-Kswp_weakオプションと同時に指定した場合、後に指定した方が有効となります。

上記を除き、機能や注意事項は、-Kswpオプションと同じです。

-Kswp_weak

対象のループに対するソフトウェアパイプライニングを調整し、ループ内の実行文の重なりを小さくすることを指示します。実行時にソフトウェアパイプライニングが適用されたループを実行するために必要なループの繰返し数が小さくなるため、ループの繰返し数が翻訳時に不明であり、かつループの繰返し数が小さい場合に最適化の効果が期待できます。

本オプションを指定した場合、実行文の重なりが小さくなるため、実行性能が低下する場合があります。

-Kswp_weakオプションを、-Kswpオプションまたは-Kswp_strongオプションと同時に指定した場合、後に指定した方が有効となります。

上記を除き、機能や注意事項は、-Kswpオプションと同じです。

-Ktls_size={12|24|32|48}

スレッド・ローカル・ストレージ(Thread-Local Storage)へのアクセスに必要なオフセットのサイズを指定します。単位はビットです。

スレッド・ローカル・ストレージのサイズに合わせ、-Ktls_size=12(4Kバイト)、-Ktls_size=24(16Mバイト)、-Ktls_size=32(4Gバイト)、-Ktls_size=48(256Tバイト)を指定することができます。

スレッド・ローカル・ストレージのサイズがオフセットの範囲を超えた場合、リンク時にエラーが生じます。

本オプションは、-KPICオプションまたは-Kpicオプションと同時に指定した場合、無効となります。

-K{unroll[=*N*]|nounroll}

ループアンローリングの最適化を行うか否かを指示します。-Kunrollオプションが指定された場合、最適化を行います。*N*はループ展開数の上限です。*N*は、2から100までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に最適な値を決定します。-O1オプションが有効な場合、省略時は、-Knounrollオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Kunrollオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

ループアンローリングについては、“3.2.4 ループアンローリング”をお読みください。

-K{unroll_and_jam[=*N*]|nounroll_and_jam}

アンロールアンドジャムの最適化を行うか否かを指示します。-Kunroll_and_jamオプションが指定された場合、最適化を行います。ただし、以下の場合には最適化を行いません。

- 最適化の効果が期待できないと判断した場合
- ループの繰返しを跨いだデータ依存があると判断した場合

*N*はループ展開数の上限です。*N*は、2から100までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に最適な値を決定します。省略時は-Knounroll_and_jamオプションが適用されます。

本オプションは-O2オプション以上が有効な場合に意味があります。

アンロールアンドジャムを適用することで、共通式の除去が促進され、実行性能が向上する場合があります。ただし、データストリーム数の増加やデータのアクセス順序の変化は、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。

また、アンロールアンドジャムの最適化による実行性能への影響はループ単位で異なります。このため、本最適化は、-Kunroll_and_jam[=*N*]オプションでプログラム全体へ適用せず、unroll_and_jam指示子やunroll_and_jam_force指示子でループ単位に適用することを推奨します。

アンロールアンドジャムについては、“3.3.8 アンロールアンドジャム”を参照してください。

-Kvisimpact

VISIMPACT(Virtual Single Processor by Integrated Multicore Parallel Architecture)用の最適なオブジェクトを生成することを指示します。

本オプションは、

-Kfast,parallel

を指定した場合と等価です。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

-K{zfill[=*N*]|nozfill}

zfillの最適化を行うか否かを指示します。zfillの最適化は、ループ内で書き込みのみを行う配列データについて、データをメモリからロードすることなく、キャッシュ上に書き込み用の領域を確保する命令(DC ZVA)を使って、書き込み動作を高速化します。zfillの最適化は対象となるストア命令の指すアドレスを起点として、256バイトを1ブロックとする単位で*N*ブロック分先のデータを最適化の対象とします。*N*は、1から100までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に値を決定します。-O1オプションが有効な場合、常に-Knozfillオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-Knozfillオプションが適用されます。

-Kzfillオプションは、-KA64FXオプションおよび-O2オプション以上と同時に指定した場合に有効となります。

なお、-Kzfillオプションを指定して翻訳したオブジェクトプログラムをDC ZVA命令の1回のキャッシュ書き込み動作が256バイトであるCPU以外で実行した場合、実行時異常終了または実行結果に誤りが生じることがあります。A64FXプロセッサでのDC ZVA命令の1回のキャッシュ書き込み動作は256バイトです。

また、以下の場合に本最適化を適用すると実行性能が低下することがあります。

- メモリバンド幅のボトルネック影響を受けていないプログラム
- 繰返し数が小さいループ
- -Kzfill=*N*オプションでブロック数を指定している、かつ、ループによって書き込まれるメモリ領域のサイズが*N*ブロックよりも小さい場合

本最適化による実行性能への影響はループ単位で異なります。このため、本最適化は、-Kzfill[=*N*]オプションでプログラム全体へ適用せず、zfill指示子でループ単位に適用することを推奨します。

zfillについては、“3.3.5 zfill”をお読みください。

2.2.2.6 -Nオプション

-N{Rtrap|Rnotrap}

実行時の割込み事象を検出するか否かを指示します。-NRtrapオプションが指定された場合、次の割込み事象を検出します。

- SIGILL、SIGBUS、SIGSEGV (jwe0019i-u)
- SIGXCPU (jwe0017i-u)
- SIGFPE (jwe1017i-u)

アンダフローによる浮動小数点演算の割込み事象は、環境変数FLIB_EXCEPT=uが指定された場合に検出されます。

なお、Armアーキテクチャである富士通製CPU A64FXでは、除数が0の場合の整数除算例外は検出されません。詳細は、“C.2.9 除数が0の場合の整数除算例外について”をお読みください。

各割込み事象については、“8.2.1 異常終了の原因”をお読みください。

省略時は、-NRnotrapオプションが適用されます。

-NRtrapオプションは、-Kpreexオプションまたは-Ksimd=2オプションと組み合わせて指定すると、命令の投機実行の影響により、実行結果に副作用(実行時の例外発生など)を生じることがあります。

また、-Kfp_relaxedおよび-NRtrapオプションが有効、かつ-Knosimdオプションまたは-KNOSVEオプションのいずれかが有効である場合、浮動小数点例外が発生することがあります。ただし、sqrt関数に対してはsqrt(0.0)での浮動小数点ゼロ除算による例外発生を回避するため、逆数近似演算に変換する最適化が抑止されます。したがって、-NRnotrapオプションが有効な場合と比べ、実行性能が低下する場合があります。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

-N{cancel_overtime_compilation|nocancel_overtime_compilation}

プログラムの翻訳が長時間(24時間以上を目安とします)になると予測される場合に、翻訳を中止するか否かを指示します。-Ncancel_overtime_compilationオプションが指定された場合、翻訳を中止します。省略時は、-Ncancel_overtime_compilationオプションが適用されます。

-Ncheck_cache_arraysize

翻訳時に配列の大きさを検査して、実行性能に影響を与える可能性がある場合に警告メッセージを出力します。

本機能は、配列の大きさが2次キャッシュの大きさの倍数である場合にキャッシュの競合を起こすとみなします。メッセージが出力された配列については、2次キャッシュの大きさの倍数とならないように配列の大きさを変更することで影響を避けられることがあります。

なお、配列が可変長配列である場合、警告メッセージは出力されません。また、セクタキャッシュの利用時には正確な診断ができない場合があります。

-N{clang|noclang}

clangモードを使用するか否かを指示します。-Nclangオプションが指定された場合、clangモードを使用してオブジェクトを生成します。-Nnoclangオプションが指定された場合、tradモードを使用してオブジェクトを生成します。省略時は、-Nnoclangオプションが適用されます。

clangモードについては、“第9章 clangモード”をお読みください。

-N{coverage|nocoverage}

コードカバレッジ機能を利用するための情報を生成するか否かを指示します。-Ncoverageオプションが指定された場合、情報を生成します。-Ncoverageオプションは、プログラムの翻訳時およびリンク時に指定する必要があります。省略時は、-Nnocoverageオプションが適用されます。

-Ncoverageオプションを指定した場合、実行回数を計測する命令がオブジェクトファイル内に追加されるため、実行性能が低下する場合があります。

コードカバレッジ機能については、“付録F コードカバレッジ機能”をお読みください。

-N{exceptions|noexceptions}

-Nexceptionsが指定された場合、既定義マクロ__EXCEPTIONSを定義します。省略時は、-Nnoexceptionsオプションが適用されます。

-N{fjcex|nofjcex}

本処理系が提供する富士通拡張関数を利用するか否かを指示します。-Nfjcexオプションが有効な場合、利用します。省略時は、-Nnofjcexオプションが適用されます。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

富士通拡張関数については、“付録G 富士通拡張関数”をお読みください。

-N{fjprof|nofjprof}

プロファイラ機能を有効にするか否かを指示します。-Nfjprofオプションが指定された場合、有効にします。省略時は、-Nfjprofオプションが適用されます。本オプションはリンク時に指定する必要があります。

プロファイラ機能については、“プロファイラ使用手引書”をお読みください。

-N{hook_func|nohook_func}

特定箇所からの呼出しによるフック機能を利用するか否かを指示します。-Nhook_funcオプションが指定された場合、フック機能を利用します。省略時は、-Nnohook_funcオプションが適用されます。

-Nhook_funcオプションが指定された場合、次の箇所からユーザー定義関数を呼び出します。

- プログラムの入口/出口
- 関数の入口/出口
- パラレルリージョン(OpenMP/自動並列化)の入口/出口

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

フック機能については、“8.3 フック機能”をお読みください。

-N{hook_time|nohook_time}

一定時間間隔での呼出しによるフック機能を利用するか否かを指示します。-Nhook_timeオプションが指定された場合、フック機能を利用します。省略時は、-Nnohook_timeオプションが適用されます。

-Nhook_timeオプションが指定された場合、一定時間間隔でユーザー定義関数を呼び出します。

ユーザー定義関数を呼び出す時間間隔は、環境変数FLIB_HOOK_TIMEで指定可能です。

環境変数FLIB_HOOK_TIMEを指定しない場合は、1分ごとにユーザー定義関数が呼び出されます。環境変数FLIB_HOOK_TIMEについては、“2.5.1 実行時環境変数”をお読みください。

本オプションは、リンク時に指定する必要があります。

フック機能については、“8.3 フック機能”をお読みください。

-N{libomp|fjomplib}

並列処理に使用するライブラリを指定します。省略時は-Nlibompオプションが適用されます。

-Nclangオプションが有効な場合、-Nfjomplibオプションは無効になり、-Nlibompオプションが有効になります。

本オプションは、リンク時に指定する必要があります。

-Nlibomp

並列処理にLLVM OpenMPライブラリを使用します。

LLVM OpenMPライブラリについては、“第4章 並列化機能”をお読みください。

-Nfjomplib

並列処理に富士通OpenMPライブラリを使用します。

富士通OpenMPライブラリについては、“付録K 富士通OpenMPライブラリ”をお読みください。

-N{line|noline}

プロファイラで提供される実行時間のサンプリング機能に必要な追加情報を生成するか否かを指示します。-Nlineオプションが指定された場合、生成します。

省略時は、-Nlineオプションが適用されます。

-Nlst[={p|t}]

-Nlstオプションは、翻訳情報をファイルに出力することを指示します。翻訳情報は、サフィックス.lstの付いたファイルへ出力します。Cソースファイルを複数指定した場合には、各ファイル名.lstファイルに出力します。

-Nlstオプションが引数を省略して指定された場合、-Nlst=pが適用されます。

OpenMP仕様の構文によって並列化などを指示された文の出力については、“第4章 並列化機能”をお読みください。

-Nlst=p

翻訳情報として、ソースリストと統計情報を出力することを指示します。

ソースリストに含まれる最適化情報により、自動並列化、インライン展開、ループアンローリング等の状況がわかります。

-Nlst=t

-Nlst=pでの出力に加えて、より詳細な最適化情報を出力することを指示します。

-Nlst_out=*file*

指定されたファイル名*file*に翻訳情報を出力することを指示します。

本オプションを指定した場合、-Nlst=pオプションも有効になります。

-Nprofile_dir=*dir_name*

コードカバレッジ機能使用時に必要な.gcdaファイルの格納先ディレクトリを指示します。*dir_name*には、格納先ディレクトリ名を相対パスまたは絶対パスで指定します。

.gcdaファイルは、コードカバレッジ用の情報を含むオブジェクトファイルを結合した実行可能プログラムを実行すると、生成されます。実行時に、*dir_name*で指定されたディレクトリが存在しない場合は、新規に作成されます。

本オプションは、-Ncoverageおよび、-Sまたは-cオプションが有効な場合に意味があります。

コードカバレッジ機能および省略時の格納先ディレクトリについては、“付録F コードカバレッジ機能”をお読みください。

-Nquickdbg[={subchk|nosubchk|heapchk|noheapchk|inf_detail|inf_simple}]

-NquickdbgオプションはCプログラムをデバッグするための機能です。本デバッグ機能は、Cプログラムのオブジェクトプログラム内にデバッグするために必要な情報を組み込み、実行時に自動検査します。検査は、-Nquickdbg=subchkまたは-Nquickdbg=heapchkオプションが有効な場合に行われます。

複数の-Nquickdbgオプションを指定することで、複数のデバッグ機能を組み合わせて利用することができます。-Nquickdbg=inf_detailおよび-Nquickdbg=inf_simpleオプションは、-Nquickdbg、-Nquickdbg=subchk、-Nquickdbg=heapchkオプションのいずれかと組み合わせて指定することで有効となります。

-Nquickdbgオプションが引数を省略して指定された場合、-Nquickdbg=subchk,quickdbg=heapchkが適用されます。

-Nquickdbg=subchkまたは-Nquickdbg=heapchkオプションが有効な場合、-Nquickdbg=inf_detailオプションも有効になります。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

本オプションが提供するデバッグ機能については、“8.1 デバッグのための検査機能”をお読みください。

-Nquickdbg={subchk|nosubchk}

配列の使用時において、宣言した配列の大きさと使用時の添字範囲の正当性を検査するか否かを指示します。本検査機能については、“8.1.1 配列範囲の検査(subchk機能)”をお読みください。

-Nquickdbg=subchk

添字範囲の検査を行います。誤りを検出した場合は、診断メッセージを出力します。

-Nquickdbg=nosubchk

添字範囲の検査を行いません。

-Nquickdbg={heapchk|noheapchk}

ヒープメモリに対して、メモリの不当な解放、領域外書き込み、およびメモリリークを検査するか否かを指示します。本検査機能については、“8.1.2 ヒープメモリの検査(heapchk機能)”をお読みください。

-Nquickdbg=heapchk

ヒープメモリの検査を行います。誤りを検出した場合は、診断メッセージを出力します。

ヒープメモリの獲得と解放が異なる翻訳単位に存在する場合、本オプションは双方の翻訳単位に対して指定する必要があります。

-Nquickdbg=noheapchk

ヒープメモリの検査を行いません。

-Nquickdbg={inf_detail|inf_simple}

本検査機能で検出された誤りにおいて、診断メッセージに出力する情報を指示します。本機能については、“8.1 デバッグのための検査機能”をお読みください。

-Nquickdbg=inf_detailおよび-Nquickdbg=inf_simpleオプションは、-Nquickdbg、-Nquickdbg=subchk、-Nquickdbg=heapchkオプションのいずれかと組み合わせて指定することで有効となります。

-Nquickdbg=inf_detail

エラーの原因を識別するためのメッセージとエラーが発生した行番号に加えて、エラー発生の原因となった変数名、要素位置などの情報が、診断メッセージとして出力されます。

-Nquickdbg=inf_simple

エラーが発生したことを通知するメッセージとエラーが発生した行番号が、診断メッセージとして出力されます。

-N{reordered_variable_stack|noreordered_variable_stack}

自動変数をスタック領域に割り付ける順序をコンパイラに指示します。

-Nreordered_variable_stackオプションが指定された場合、以下の順序で自動変数の割付け順を決めます。

1. アライメントの降順
2. アライメントが等しい場合は、データサイズの降順
3. アライメントおよびデータサイズが等しい場合は、ソースプログラム内の宣言文の記載順

アライメントの降順に自動変数を割り付けると、プログラム全体のスタック領域を減らすことができます。

-Nnoreordered_variable_stackオプションが指定された場合、ソースプログラム内の宣言文の記載順に自動変数を割り付けます。省略時は、-Nnoreordered_variable_stackオプションが適用されます。

-Nnolineオプション、-g0オプション、または-Koclオプションが有効な場合、割付け順序は保証されません。

本機能の詳細については、“E.4 スタック領域へのデータ割付け”を参照してください。

-N{rt_tune|rt_notune}

実行時情報を出力するか否かを指示します。-Nrt_tuneオプションが指定された場合、出力します。省略時は、-Nrt_notuneオプションが適用されます。

実行時情報はプログラムのチューニングに利用することができます。詳細については、“付録H 実行時情報出力機能”をお読みください。

-Nrt_tune_func

-Nrt_tuneオプションの出力に加えて、利用者定義の関数ごとの実行時情報を出力します。

本オプションを指定した場合、-Nrt_tuneオプションも有効になります。本オプションより後に-Nrt_notuneオプションが指定された場合、実行時情報は出力されません

実行時情報出力機能については、“付録H 実行時情報出力機能”をお読みください。

-Nrt_tune_loop[={all|innermost}]

-Nrt_tuneオプションの出力に加えて、ループごとの実行時情報を出力します。-Nrt_tune_loopオプションが引数を省略して指定された場合、-Nrt_tune_loop=allが適用されます。

本オプションを指定した場合、-Nrt_tuneオプションも有効になります。本オプションより後に-Nrt_notuneオプションが指定された場合、実行時情報は出力されません

実行時情報出力機能については、“付録H 実行時情報出力機能”をお読みください。

-Nrt_tune_loop=all

すべてのループの実行時情報を出力します。

-Nrt_tune_loop=innermost

最内ループの実行時情報を出力します。

-N{setvalue[=*setarg*]|nosetvalue}

ヒープ領域またはスタック領域にゼロ値を自動的に設定するか否かを指示します。-Nsetvalueオプションが指定された場合、ゼロ値を設定します。省略時は、-Nnosetvalueオプションが適用されます。

ゼロ値を設定することにより、実行時間が増加する場合があるため注意が必要です。

*setarg*には、ゼロ値を設定する対象の種別として、以下のいずれかを指定します。

{ {heap|noheap} | {stack|nostack} | {scalar|noscalar} | {array|noarray} | {struct|nostruct} }

-Nsetvalueオプションの=*setarg*が省略された場合、-Nsetvalue=heap,setvalue=stackが指定されたものとみなします。

-Nsetvalueオプションと同時に-Nquickdbgオプションを指定した場合、-Nsetvalueオプションは無効となります。

-Nsetvalue={heap|noheap}

ヒープ領域の獲得時にゼロ値を設定するか否かを指示します。-Nsetvalue=heapオプションが指定された場合、ゼロ値を設定します。

対象とするヒープ領域は、以下のとおりです。

- malloc関数で獲得された領域

-Nsetvalue={stack|nostack}

スタック領域に割り付けられた未初期化変数にゼロ値を設定するか否かを指示します。-Nsetvalue=stackオプションが指定された場合、ゼロ値を設定します。

対象とする未初期化変数は、以下のとおりです。

- 自動変数

ただし、OpenMPのprivate指示節またはlastprivate指示節で指定した変数には、ゼロ値は設定されません。

-Nsetvalue=stackオプションは、

-Nsetvalue=stack,setvalue=scalar,setvalue=array,setvalue=struct

を指定した場合と等価です。

-Nsetvalue={scalar|noscalar}

スタック領域に割り付けられた未初期化のスカラ型の変数にゼロ値を設定するか否かを指示します。-Nsetvalue=scalarオプションが指定された場合、ゼロ値を設定します。

-Nsetvalue=scalarオプションは、-Nsetvalue=stackオプションが有効な場合に意味があります。

-Nsetvalue={array|noarray}

スタック領域に割り付けられた未初期化の配列型の変数にゼロ値を設定するか否かを指示します。-Nsetvalue=arrayオプションが指定された場合、ゼロ値を設定します。

なお、C99の可変長配列については、本オプションの対象外です。

-Nsetvalue=arrayオプションは、-Nsetvalue=stackオプションが有効な場合に意味があります。

-Nsetvalue={struct|nostruct}

スタック領域に割り付けられた未初期化の構造体型および共用体型の変数にゼロ値を設定するか否かを指示します。-Nsetvalue=structオプションが指定された場合、ゼロ値を設定します。

-Nsetvalue=structオプションは、-Nsetvalue=stackオプションが有効な場合に意味があります。

例

-Nsetvalueオプションの指定方法

スタック領域に割り付けられたスカラ型の変数だけをゼロ値設定する場合は、以下のように指定します。

```
-Nsetvalue=stack,setvalue=noarray,setvalue=nostruct
```

-Nsrc

ソースリストを標準出力に出力することを指示します。

ソースリストに含まれる最適化情報により、自動並列化、インライン展開、ループアンローリング等の状況がわかります。

OpenMP仕様の構文によって並列化などを指示された文の出力については、“第4章 並列化機能”をお読みください。

-Nlstオプションまたは-Nlst_out=*file*オプションが同時に指定された場合、ソースリストはファイルに出力されます。

-Nsta

統計情報を標準出力に出力することを指示します。

-Nlstオプションまたは-Nlst_out=*file*オプションが同時に指定された場合、統計情報はファイルに出力されます。

2.2.3 翻訳時オプションの注意事項

翻訳時オプションで注意すべき点について説明します。

-Kオプションおよび-Nオプションの複数指定

-Kオプションおよび-Nオプションは、その後ろにコンマ(,)で区切って複数のオプションを続けて指定することができます。また、複数の-Kオプションおよび-Nオプションに分けて指定することもできます。

以下の例では、2つのコマンドは同じオプションが指定されたものとみなします。

例

```
$ fccpx -o pgm.out -Kfast -Kparallel -Nsrc -Nsta pgm.c
```

```
$ fccpx -o pgm.out -Kfast,parallel -Nsrc,sta pgm.c
```

空白の禁止

-Kオプションおよび-Nオプションに続いて指定するオプションには、空白を含むことはできません。

以下の例では、"parallel"の前に空白があるため、"parallel"は無効となります。

例

誤った指定

```
$ fccpx -Kfast, parallel pgm.c
```

正しい指定

```
$ fccpx -Kfast,parallel pgm.c
```

2.3 翻訳コマンドの環境変数

翻訳コマンドが認識する環境変数について説明します。

fccpx_ENV
fcc_ENV

翻訳時オプション設定用の環境変数です。trad/clangモード共通で有効です。

環境変数fccpx_ENVはクロスコンパイラ用、環境変数fcc_ENVはネイティブコンパイラ用です。

利用者は、これらの環境変数の値としてtrad/clangモード共通の翻訳時オプションを設定することができます。trad/clangモード共通の翻訳時オプションについては、“9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応”を参照してください。

これらの環境変数に設定された翻訳時オプションは、tradモードおよびclangモードのどちらでも有効となります。常に指定する翻訳時オプションを、あらかじめこれらの環境変数に設定しておくと便利です。

翻訳時オプションの優先順位については、“2.4 翻訳時プロフィルファイル”を参照してください。

例

以下の2つの例は、等価です。

例1: 翻訳時オプションを環境変数fccpx_ENVに指定(shの場合)

```
$ fccpx_ENV="-Kfast -I/usr/prv/include"
$ export fccpx_ENV
$ fccpx a.c
```

例2: 翻訳時オプションを翻訳コマンドのオペランドに指定

```
$ fccpx -Kfast -I/usr/prv/include a.c
```

fccpx_trad_ENV
fcc_trad_ENV

翻訳時オプション設定用の環境変数です。tradモードでのみ有効です。

環境変数fccpx_trad_ENVはクロスコンパイラ用、環境変数fcc_trad_ENVはネイティブコンパイラ用です。

利用者は、これらの環境変数の値として、tradモード固有の翻訳時オプションおよびtrad/clangモード共通の翻訳時オプションを設定することができます。trad/clangモード共通の翻訳時オプションについては、“9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応”を参照してください。

これらの環境変数に設定された翻訳時オプションは、tradモードでのみ有効となります。

翻訳時オプションの優先順位については、“2.4 翻訳時プロフィルファイル”を参照してください。

FCOMP_LINK_FJOBJ

本処理系では、通常、リンク時に本処理系独自オブジェクトを結合しますが、以下の条件のときは結合しません。

ー 本処理系がリンカに渡している-Lオプションを、ユーザーが本処理系の翻訳コマンドに直接指定してリンクを行う。

これにより、リンクエラー(undefined reference to)になる場合がありますが、環境変数FCOMP_LINK_FJOBJを設定することで回避できます。

trad/clangモード共通で有効です。環境変数FCOMP_LINK_FJOBJに設定する値は任意です。環境変数FCOMP_LINK_FJOBJが設定されていると、リンク時に本処理系独自オブジェクトを結合します。

リンクエラーの例は、“C.2.10 リンクエラー(undefined reference to)について”を参照してください。

例

環境変数FCOMP_LINK_FJOBJの設定例

```
$ export FCOMP_LINK_FJOBJ=true
```

FCOMP_UNRECOGNIZED_OPTION

環境変数FCOMP_UNRECOGNIZED_OPTIONを設定することにより、翻訳コマンドが識別不可能な翻訳時オプションに対する動作を変更することができます。tradモードでのみ有効です。

環境変数FCOMP_UNRECOGNIZED_OPTIONには、warningまたはerrorのいずれかを設定します。環境変数FCOMP_UNRECOGNIZED_OPTIONを設定しない場合、または無効な値が設定された場合は、warningが適用されます。

設定値	説明
warning	識別不可能な翻訳時オプションに対して警告メッセージを出力し、翻訳を継続します。
error	識別不可能な翻訳時オプションに対してエラーメッセージを出力し、翻訳を中断します。

例

例1: 環境変数FCOMP_UNRECOGNIZED_OPTIONにwarningを設定(shの場合)

```
$ FCOMP_UNRECOGNIZED_OPTION=warning
$ export FCOMP_UNRECOGNIZED_OPTION
$ fccpx -unrecognized_option a.c
fccpx: 警告: -unrecognized_optionは認識できないオプションです。
$ echo $?
0
$
```

例2: 環境変数FCOMP_UNRECOGNIZED_OPTIONにerrorを設定(shの場合)

```
$ FCOMP_UNRECOGNIZED_OPTION=error
$ export FCOMP_UNRECOGNIZED_OPTION
$ fccpx -unrecognized_option a.c
fccpx: エラー: -unrecognized_optionは認識できないオプションです。
$ echo $?
1
$
```

CPATH
C_INCLUDE_PATH

環境変数CPATHまたはC_INCLUDE_PATHを設定することにより、名前が/以外で始まるヘッダを検索するディレクトリを追加することができます。trad/clangモード共通で有効です。

コロン(:)を区切りとして、複数のディレクトリを設定することができます。

ヘッダの検索順の詳細については、“2.2.2.1 コンパイラ全般に関連するオプション”の-Iオプションの説明をお読みください。

例

環境変数CPATHおよびC_INCLUDE_PATHの使用例(shの場合)

```
$ CPATH="/cpath_inc:/cpath_inc_2"
$ export CPATH
$ C_INCLUDE_PATH="/c_include_path:/c_include_path_2"
$ export C_INCLUDE_PATH
```

LIBRARY_PATH

環境変数LIBRARY_PATHを設定することにより、リンク時にライブラリを検索するディレクトリを追加することができます。trad/clangモード共通で有効です。

コロン(:)を区切りとして、複数のディレクトリを設定することができます。

ライブラリの検索は以下の順序で行われます。

1. -Lオプションの引数に指定されたディレクトリ
2. 本処理系が提供するライブラリを格納するディレクトリ
3. 標準ライブラリのディレクトリ
4. 環境変数LIBRARY_PATHに指定されたディレクトリ

例

環境変数LIBRARY_PATHの使用例(shの場合)

```
$ LIBRARY_PATH="/usr/local/lib64:/usr/local_2/lib64"
$ export LIBRARY_PATH
```

TMPDIR

環境変数TMPDIRを設定することにより、翻訳コマンドが使用するテンポラリディレクトリを変更することができます。trad/clangモード共通で有効です。

環境変数TMPDIRを設定しない場合、/tmpが使用されます。共通領域へ出力したくない場合は、テンポラリディレクトリを環境変数TMPDIRで変更してください。

例

環境変数TMPDIRの使用例(shの場合)

```
$ TMPDIR=/usr/local/tmp
$ export TMPDIR
```

2.4 翻訳時プロフィルファイル

翻訳時プロフィルファイルを設定することにより、翻訳時オプションのデフォルトを変更することができます。

使用可能な翻訳時プロフィルファイルを以下に示します。

表2.4 翻訳時プロフィルファイル(tradモード)

翻訳コマンドの種別	モード	ファイル名
クロスコンパイラ	trad/clangモード共通	/etc/opt/FJSVstclanga/fccpx_PROF
	tradモード固有	/etc/opt/FJSVstclanga/fccpx_trad_PROF
ネイティブコンパイラ	trad/clangモード共通	/etc/opt/FJSVstclanga/fcc_PROF
	tradモード固有	/etc/opt/FJSVstclanga/fcc_trad_PROF

trad/clangモード共通の翻訳時プロフィルファイルには、trad/clangモード共通の翻訳時オプションを設定することができます。trad/clangモード共通の翻訳時オプションについては、“9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応”を参照してください。

tradモード固有の翻訳時プロフィルファイルには、tradモード固有の翻訳時オプションおよびtrad/clangモード共通の翻訳時オプションを設定することができます。

注意

翻訳時プロフィルファイルの設定内容については、システム管理者にお問い合わせください。

翻訳時プロフィルファイルの形式は、以下のとおりです。

- 二重引用符(")または一重引用符(')で囲まれていない空白文字は、意味を持ちません。
- 文字列の開始および終了を表す二重引用符および一重引用符は、すべて等価です。ただし、一連の文字列は、必ず開始を表す二重引用符および一重引用符と同一の文字または行末によって終了します。
- 文字列に含まれていない“#”から行末までは、コメントと解釈します。

例

翻訳時プロフィルファイルの記述例

```
#Default options
-w -O
```

翻訳時オプションは、以下の優先順位に従って決まります。

1. 翻訳コマンドのオペランド
2. 翻訳時オプション設定用の環境変数 (モード固有)
3. 翻訳時オプション設定用の環境変数 (モード共通)
4. 翻訳時プロフィルファイル (モード固有)
5. 翻訳時プロフィルファイル (モード共通)
6. 省略値

2.5 実行の手続き

ここでは、プログラムを実行するための手続きについて説明します。

2.5.1 実行時環境変数

実行時の制御は環境変数を指定することで変更できます。“表2.5 実行時環境変数”に、実行時に指定できる環境変数を示します。

なお、OpenMP仕様の環境変数については、“4.3.2.2 OpenMP仕様の環境変数”をお読みください。

表2.5 実行時環境変数

環境変数名	オペランド	意味
FLIB_C_MESSAGE	NO_MESSAGE	ユーザープログラムの実行時にエラーが発生した場合、エラー情報が標準エラーへ出力されますが、この環境変数を設定した場合はエラー出力が抑止されます。 発生するエラーの詳細については、“Fortran/C/C++実行時メッセージ”をお読みください。-NRtrapオプションが指定された場合、本環境変数の指定は無効となります。 注意 本環境変数は、LLVM OpenMPライブラリのエラー出力に対しては無効です。LLVM OpenMPライブラリのエラー出力については“4.4 実行時メッセージ”をお読みください。
FLIB_EXCEPT	u	浮動小数点アンダフローによる割込み事象を検出します。 翻訳時およびリンク時に-NRtrapオプションを指定し、実行時に本環境変数を指定した場合、浮動小数点アンダフローによるプログラム割込み(jwe0012i-u)が検出されます。
FLIB_HEAPCHK_VALUE	*hex*	ヒープメモリの領域外書き込み検査において、検査に使用する検査値を変更します。 詳細については、“8.1.2.2 ヒープメモリの領域外書き込み検査”をお読みください。
FLIB_HOOK_TIME	*time*	一定時間間隔呼出しによるフック機能において、*time*ミリ秒ごとにユーザー定義関数が呼び出されます。 *time*は0～2147483647の値です。*time*に0が指定された場合、一

環境変数名	オペランド	意味
		定時間間隔での呼出しは無効となります。 詳細については、“8.3 フック機能”をお読みください。
FLIB_RTINFO	なし	実行時情報を出力します。 詳細については、“付録H 実行時情報出力機能”をお読みください。
FLIB_RTINFO_CSV	*file*	実行時情報出力機能において、取得した情報を、CSV形式でファイルに出力します。*file*には任意のファイル名が指定可能です。*file*を省略した場合は、flib_rtinfo.csvというファイルに出力します。 詳細については、“付録H 実行時情報出力機能”をお読みください。
FLIB_RTINFO_NOFUNC	なし	実行時情報出力機能において、翻訳時オプション-Nrt_tune_funcを指定した場合でも、利用者定義の関数ごとの情報出力を抑止します。 詳細については、“付録H 実行時情報出力機能”をお読みください。
FLIB_RTINFO_NOLOOP	なし	実行時情報出力機能において、翻訳時オプション-Nrt_tune_loopを指定した場合でも、ループごとの情報出力を抑止します。 詳細については、“付録H 実行時情報出力機能”をお読みください。
FLIB_TRACEBACK_MEM_SIZE	*size*	トレースバックマップの情報に出力されるデバッグ情報を保持するヒープ領域の大きさを変更します。 *size*には1～128の整数値を指定します。*size*の単位はMiBです。 環境変数が指定されていない場合、または、*size*の値が無効である場合、ヒープ領域の大きさは本処理系の省略値となります。省略値は16MiBです。 トレースバックマップについては、“5.2.2トレースバック情報”をお読みください。

2.5.2 実行時の注意事項

本処理系で作成されたプログラムを実行する場合の注意事項を、以下に示します。

2.5.2.1 実行時の変数割付け

本処理系で作成されたプログラムは、関数内でローカルな変数およびプライベート変数をスタック領域に割り付けます。関数内でローカルな変数およびプライベート変数に必要な領域が大きい場合には、スタック領域を十分な大きさに拡張する必要があります。

プロセスのスタック領域は、ulimitコマンド(bash組み込みコマンド)などで設定できます。

第3章 最適化機能

本章では、最適化機能の概要および有効に活用する方法について説明します。

3.1 最適化の概要

最適化の目的は、プログラムを可能な限り高速に実行できるようなオブジェクトプログラム(命令列およびデータ域)を生成することです。最適化機能を使用することにより、プログラムの実行時間を短縮することができます。コンパイラの最適化機能は、Cソースプログラムからオブジェクトプログラムを生成する過程で、以下に示す作業を行います。

削除

同じ結果をもたらす文や式がある場合、無駄な部分を削除します。また、実行しても意味がない文や式があれば、それらも削除します。

移動

繰り返し実行する部分(ループと呼ばれる)に、何回繰り返しても同じ結果になる文や式があれば、それらをループの外側に移動します。また、繰り返し実行しなくても最終的な結果が分かる場合は、必要な部分だけをループの中に残し、他の部分はループの外側に移動します。

変更

演算式の演算子を、計算誤差が生じない範囲で変更することがあります。また、演算の対象になるデータも変更することがあります。例えば、一度だけ定数が代入されて、それ以後は値が変わらない変数の参照がある場合、その変数の参照は定数の参照に変更します。

展開

標準ライブラリ関数や利用者定義の関数を呼び出している場合、可能であれば関数の本体が呼び出しているところに展開します。引数や関数値の受け渡しおよび呼び出しや復帰のための命令列が不要になるので、実行時間が短縮されます。

実行

文の実行結果や式の演算結果が翻訳時に分かる場合、文の実行や式の計算を行う命令列は出力せずに、翻訳時に実行または計算した結果を使用します。

その他

ハードウェアの機能・特徴(命令の種類、レジスタの種類と個数、アドレッシング方法など)を最大限に利用して高速に実行できる命令列およびデータ域を出力します。

最適化機能を使用することによって翻訳時間と翻訳作業域は増加します。また、最適化の内容によっては、オブジェクトプログラムの大きさが大幅に増加したり、実行結果に副作用を及ぼしたりする可能性があります。このため、最適化機能を使用するか否かは、翻訳時オプションにより選択できるようになっています。

翻訳時オプションには、最適化レベルを指定する標準最適化と、それを補う拡張最適化があります。

ポイント

実行結果に副作用を及ぼす可能性がある最適化機能項目は、利用者が翻訳時オプションで指定しない限り実施されないので、通常の使用で副作用が発生することはありません。

3.2 標準最適化

最適化レベル1(-O1)から最適化レベル3(-O3)が指定されると、標準最適化が実施されます。標準最適化の代表的な機能について説明します。

3.2.1 共通式の除去

等しい演算結果をもたらす2つの式(共通式)が存在する場合、後の式では演算せずに前の式の演算結果を利用します。

共通式の除去の対象となるのは、算術演算、関係演算、論理演算および一部の標準ライブラリ関数の引用です。

共通式の除去は、最適化レベル1(-O1)以上の場合に行われます。

以下に、共通式の除去の例を示します。

例:共通式の除去

```
a = x * y + c;
...
b = x * y + d;
```

→

```
t = x * y;
a = t + c;
...
b = t + d;
```

[説明]

代入式の右辺の一部であるx*yが共通式です。2度目の計算を除去して最初の計算結果tを使用するように変更します。tはコンパイラが生成した変数です。

3.2.2 不変式の移動

ループ内で値が変化しない式(不変式)を、ループの外側に移動します。不変式の移動の対象となるのは、算術演算、関係演算、論理演算および一部の標準ライブラリ関数の引用です。

不変式の移動は、最適化レベル1(-O1)以上の場合に行われます。

以下に、不変式の移動の例を示します。

例:不変式の移動

```
for(i = 0; i < n; i++) {
    y = a[j] * 2;
    x = x + y * z;
}
```

→

```
y = a[j] * 2;
t = y * z;
for(i = 0; i < n; i++) {

    x = x + t;
}
```

[説明]

文全体のy = a[j]*2;および式の一部のy*zをループ外に移動します。tはコンパイラが生成した変数です。

この最適化により移動するのは、通常はループ内で必ず実行される文の一部または全体です。ただし、-Kpreexオプションを指定した場合は、判定文などにより、ループ内で選択的に実行される文に含まれる不変式も移動します。この最適化を、通常の不変式の移動とは区別して、不変式の先行評価と呼びます。不変式の先行評価を実施することにより、より実行時間を短縮することができます。しかし、移動による副作用が生じる場合があります。

不変式の先行評価による副作用については、“3.3.1.1 不変式の先行評価”をお読みください。

3.2.3 演算子の強さの縮小

演算子の強さとは、演算に要する実行時間の相対的な大小関係をいいます。強い演算子ほど実行時間を多く必要とします。一般に、加算と減算の強さは同じです。乗算は加減算よりも強く、除算は乗算よりも強くなります。また、整数型と浮動小数点型の間の型変換は、加減算よりも強く乗算より弱くなります。演算子の強さの縮小とは、“乗算は加減算に”というように、演算子の強さをより弱い方向に変更することにより、実行時間を短縮する最適化のことです。

この最適化は、ループ内で値が階段状に規則的に変化する整数型の変数(誘導変数)と、その変数を引用している式を対象に、加減算への縮小を行います。

演算子の強さの縮小は、最適化レベル1(-O1)以上の場合に行われます。

参考

誘導変数

ループの繰返しに伴って一定値ずつ増加または減少する、ループ変数以外の変数を誘導変数と呼びます。

このため、演算子の強さの縮小は、誘導変数の最適化とも呼びます。

以下に、強さの縮小の例を示します。

例1:演算子の強さの縮小(乗算から加算への縮小)

```
for(i = 1; i < 10; i++) {
    ...
    j = k + i * 100;
    ...
}
```

→

```
t = 100;
for(i = 1; i < 10; i++) {
    ...
    j = k + t;
    ...
    t = t + 100;
}
```

[説明]

誘導変数iに対する演算i*100に対して最適化を行い、ループ内の演算i*100を加算t = t+100に変換します。tはコンパイラが生成した変数です。

例2:演算子の強さの縮小(型変換から加算への縮小)

```
for(i = 1; i < 10; i++) {
    ...
    x = (float)i;
    ...
}
```

→

```
t = 1.0;
for(i = 1; i < 10; i++) {
    ...
    x = t;
    ...
    t = t + 1.0;
}
```

[説明]

誘導変数iに対する演算(float)iに対して最適化を行い、整数型から浮動小数点型への型変換を、浮動小数点型の加算t = t+1.0に変換します。tはコンパイラが生成した浮動小数点型の変数です。

3.2.4 ループアンローリング

ループアンローリングとは、ループ内のすべての実行文をループ内に*N*重に展開し、その代わりにループの繰返し数を*N*分の1に縮小する最適化です。ただし、-Ksimd[={1|2|auto}]オプションが有効でループ内がSIMD化された場合、実行文はSIMD長×*N*重に展開され、ループの繰返し数はSIMD長×*N*分の1に縮小されます。

ループアンローリングは、ループ外からの飛込みもループ外への飛出しもないループに対して行われます。

展開数*N*は、ループの繰返し数、ループ内の実行文の数と種類および使われているデータの型などから、コンパイラにより最適な値が決められます。また、最適化制御行(OCL)により、ソースプログラム上に展開数を指定することもできます。この機能については、“3.4.1 最適化制御行(OCL)の利用”をお読みください。

繰返し数が縮小され、かつ多重に展開された実行文がループ内で一体になって最適化されるので、高速なオブジェクトプログラムが得られます。ただし、ループ内の実行文が多重に展開されるので、オブジェクトプログラムの大きさが増加します。

ループアンローリングはSIMD化前後で動作します。

- SIMD化前

  外側ループでSIMD化やソフトウェアパイプライニングを適用するために、内側ループの繰返し数が小さい場合に内側ループをフルアンローリングします。これをSIMD化前のフルアンローリングと呼びます。

- SIMD化後

  共通式などの最適化の促進のために、内側ループをアンローリングまたはフルアンローリングします。

ループアンローリングは、最適化レベル2(-O2)以上の場合に行われます。

-Kunroll[=*n*]オプションまたはunroll指示子が有効なループでは、ループアンローリングはSIMD化前後で動作します。

-Knounrollオプションまたはnounroll指示子が有効なループでは、ループアンローリングは抑止されます。

SIMD化前のフルアンローリングは、fullunroll_pre_simd指示子またはnofullunroll_pre_simd指示子を使って動作を制御できます。-Kunroll[=*n*]オプション、-Knounrollオプション、unroll指示子、またはnounroll指示子が有効なループに、fullunroll_pre_simd指示子またはnofullunroll_pre_simd指示子が指定された場合、SIMD化前のフルアンローリングではfullunroll_pre_simd指示子またはnofullunroll_pre_simd指示子が優先されます。

また、-Koptmsg=2オプションを指定することにより、最適化されたループとその展開数を示すメッセージを出力することができます。

以下に、ループアンローリングの例を示します。

例:ループアンローリング

```
int i, j, k;
float a[400][400], b[400][400], c[400][400];
for(i = 0; i < 400; i++) {
    for(j = 0; j < 400; j++) {
        c[i][j] = 0.0;
        for(k = 0; k < 400; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

例えば、展開数が4の場合、オブジェクトプログラムを生成する過程で、最も内側のループが以下のように変形されます。

```
for(k = 0; k < 400; k += 4) {
    c[i][j] += a[i][k  ] * b[k  ][j];
    c[i][j] += a[i][k+1] * b[k+1][j];
    c[i][j] += a[i][k+2] * b[k+2][j];
    c[i][j] += a[i][k+3] * b[k+3][j];
}
```

3.2.5 ループブロッキング

ループブロッキングとは、ループをブロックサイズ単位で多重化して、配列へのアクセスを細分化させる最適化です。これによってメモリアクセスの局所性を高められるため、キャッシュの効率的な利用が期待できます。

ループブロッキングは、ループ外からの飛込みもループ外への飛出しもないループに対して行われます。

ループブロッキングは、最適化レベル2(-O2)以上の場合に行われます。また、-Kloop_noblockingオプションを指定することにより、抑止することができます。

-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージを出力することができます。

例:ループブロッキング

```
#define N 200
int a_array[N][N];
int b_array[N][N];

void sub() {
    int i,j;
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            a_array[i][j] = b_array[j][i];
        }
    }
}
```

→

```
#define MIN(x,y) ((x<y)?x:y)
#define BLOCK 2
#define N 200
int a_array[N][N];
int b_array[N][N];

void sub() {
    int i,j,ii,jj;
    for(ii = 0; ii < N; ii += BLOCK) {
        for(jj = 0; jj < N; jj += BLOCK) {
            for(i = ii; i < MIN(N,(ii+BLOCK)); i++) {
                for(j = jj; j < MIN(N,(jj+BLOCK)); j++) {
                    a_array[i][j] = b_array[j][i];
                }
            }
        }
    }
}
```

3.2.6 ソフトウェアパイプライニング

ソフトウェアパイプライニングとは、命令レベルの並列化が可能なマシンにおいて、ループ内の命令ができるだけ並列実行されるように命令を並べ替える最適化です。

ソフトウェアパイプライニングは、翻訳時オプション-Knoswpを指定することにより、抑止することができます。

ソフトウェアパイプライニングは、ループの任意の繰返しにおける実行文と、それ以降の繰返しにおける実行文を重ね合わせた命令スケジューリングを行い、ループの形状を変更します。

そのため、ループには十分な繰返し数が必要になります。必要な繰返し数は、ソフトウェアパイプライニングを適用するループ内の命令列によって決まるため、他の最適化の影響を受けます。また、元のループの繰返し数が小さい場合、ループ形状を変更するために必要な繰返し数を小さくするような命令スケジューリングを行うため、ソフトウェアパイプライニングによる効果が小さくなる場合があります。

繰返し数が変数のループに対してソフトウェアパイプライニングが適用された場合、ループの繰返し数が必要な繰返し数に満たない場合を考慮し、下記の例に示すように、ソフトウェアパイプライニングを適用したループまたは元のループを選択する分岐構造を生成します。どちらのループを選択するかは実行時に判断されます。また、オブジェクトプログラムの大きさは増加します。

例:

```
void sub (int N) {
    (元のループ: 繰返し数N)
}
```

[最適化後(ソースイメージ)]

```
void sub (int N) {
    if (Nは必要な繰返し数以上か?) {
        (ソフトウェアパイプライニングを適用したループ)
    } else {
        (元のループ: 繰返し数N)
    }
}
```

また、翻訳時オプション-Koptmsg=2を指定することにより、ループに対するソフトウェアパイプライニングの適用情報が出力されます。ソフトウェアパイプライニングが適用された場合、翻訳時メッセージjwd8204o-iおよびjwd8205o-iが同時に出力されます。ソフトウェアパイプライニングが適用されなかった場合、その理由を表示する翻訳時メッセージjwd8662o-i～jwd8674o-iのいずれかが出力されます。翻訳時オプション-Knoswpを指定している場合、およびソフトウェアパイプライニングを適用する前にフルアンローリングなどの最適化によりループが無くなった場合は、ソフトウェアパイプライニングに関するメッセージは出力されません。

翻訳時メッセージjwd8205o-iは、ソフトウェアパイプライニングを適用したループが実行時に選択されるために必要な、ソースプログラム上のループの繰返し数を表示します。また、ソフトウェアパイプライニングは最内ループを対象とするため、いくつかの最適化が適用された後では、-Koptmsg=2で出力される最適化の適用状況を加味して判断する必要があります。具体的には、ループ一重化、ループ交換、スレッド並列化、インライン展開が動作した場合には、ソフトウェアパイプライニングが対象とする最内ループの繰返し数(翻訳時メッセージjwd8205o-iで表示される繰返し数と比較すべき繰返し数)は以下のように変更されるため注意が必要です。

同時に出力される 最適化メッセージ	メッセージ番号	ソフトウェアパイプライニングが対象とする最内ループの繰返し数
ループ一重化	jwd8330o-i	一重化された複数のループの繰返し数の積
ループ交換	jwd8211o-i	ループ交換された結果、最内となったループの繰返し数
スレッド並列化	jwd5001p-iなど	ループの繰返し数を並列化スレッド数で割った数
インライン展開	jwd8101o-i	インライン展開された結果、最内となったループの繰返し数

翻訳時メッセージjwd8205o-iで表示されるループの繰返し数は、ループのSIMD化により同時に実行される繰返し数を加味した値が出力されます。ただし、-Ksimd_reg_size=agnosticオプションが有効な場合、SVEのベクトルレジスタサイズが翻訳時には不明であるため、SVEのベクトルレジスタサイズを128ビットとみなした値を出力します。

3.2.7 SIMD化

3.2.7.1 一般的なSIMD化

SIMD化とは、複数の同種の演算を同時実行するSIMD命令への変換を行う最適化です。

SIMD化は、ループ外からの飛込みもループ外への飛出しもないループに対して行われます。

SIMD化は、最適化レベル2(-O2)以上の場合に行われます。また、-Knosimdオプションを指定することにより、抑止することができます。

また、-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージが出力されます。

3.2.7.2 if文を含むループのSIMD化

-Ksimd={2|auto}オプションを指定することで、if文を含むループをSIMD化することができます。特に、if文の条件の真率が高い場合には性能向上が期待できます。性能向上がif文の条件の真率が高い場合に限定される理由は、CPUアーキテクチャで用意されている条件付き命令が条件付きストア命令と条件付き転送命令のみであり、それ以外の命令は投機実行(if文内の命令を条件が偽の時も実行すること)されるからです。

また、命令を投機実行することで、実行時の例外発生など実行結果に副作用を生じる場合があります。

3.2.7.3 リストベクトル変換

if文の真率が低い場合でも、if文内の命令数が多いループに対しては、リストベクトル変換を適用することで性能向上できる可能性があります。リストベクトル変換とは、if文の条件を満たすループ制御変数の値を配列に登録するループと、その登録した配列のサイズ分だけ繰返して元のif文内の命令を実行するループへの変換です(下例参照)。

変換した後半のループはリストアクセスのループとなりますが、SIMD化およびソフトウェアパイプライニングの対象となります。

以下にリストベクトル変換の変換例を示します。リストベクトル変換は最適化制御行で動作します。詳細は"3.4.1 最適化制御行(OCL)の利用"のsimd_listvをお読みください。

例: リストベクトル変換

```
for (i=0; i<n; ++i) {
#pragma statement simd_listv
  if (m[i]) {
    a[i] = b[i] + c[i];
  }
}
```

→

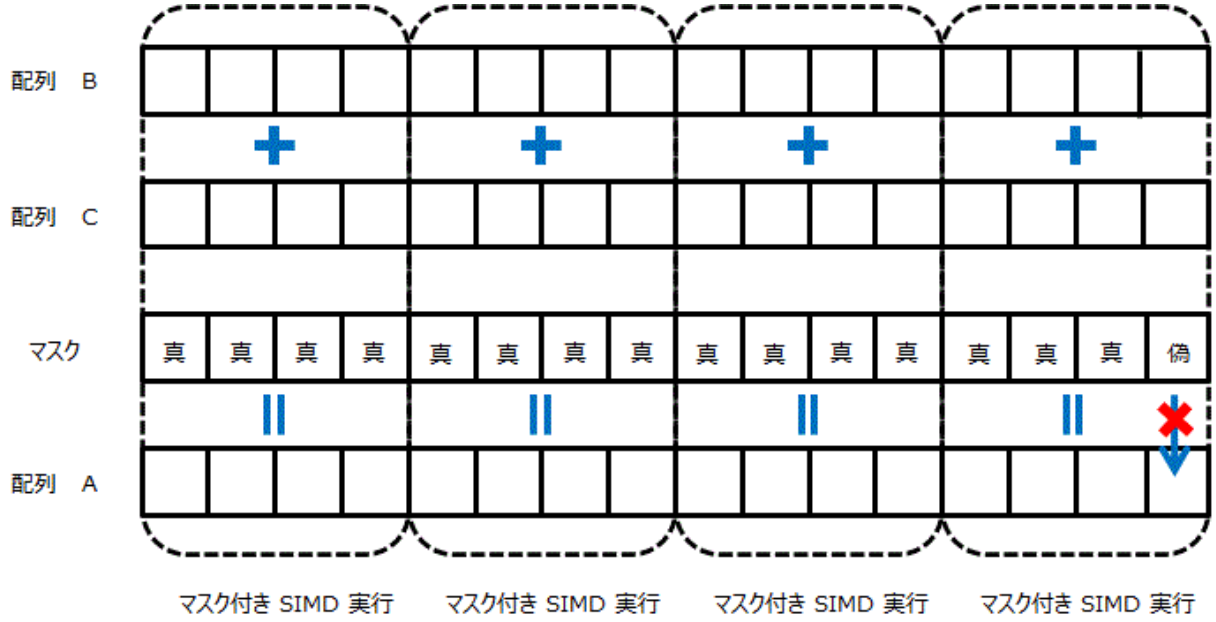
```
j = 0;
for (i=0; i<n; ++i) {
  if (m[i]) {
    idx[j] = i;
    j = j + 1;
  }
}
for (k=0; k<j-1; ++k) {
  i = idx[k];
  a[i] = b[i] + c[i];
}
```

3.2.7.4 SIMD長の倍数冗長実行によるSIMD化機能

本機能は、ループの繰返し数がSIMD長で割り切れない場合でも、マスク付きSIMD命令を利用することで、その端数部分を含めてSIMD実行するループを生成します。

本機能の動作イメージを下図に示します。繰返しすべてがマスク付きでSIMD実行されます。

図3.1 本機能有効時の動作(4SIMDイメージ)



なお、繰返し数をSIMD長で割った余りが1や2となることが多いループでは、最適化の効果が得られない可能性があります。

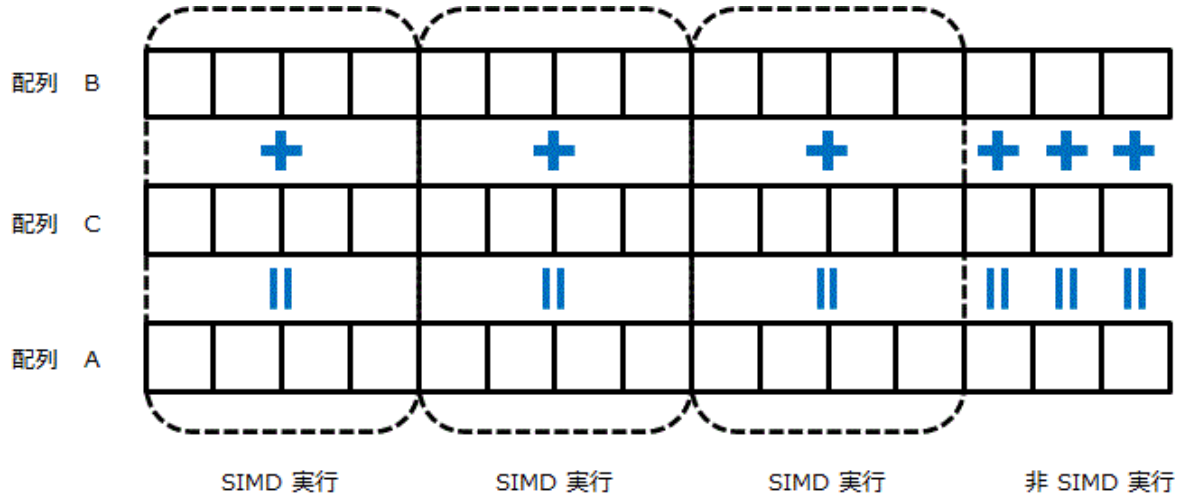
本機能を抑止したい場合は、最適化制御行にsimd_noredundant_vl指示子を指定してください。

例: 本機能を抑止する最適化制御行(OCL)の指定

```
#pragma loop simd_noredundant_vl
    for(i = 0; i < m; i++) {
        a[i] = b[i] + c[i];
    }
```

このプログラム例の動作イメージを下図に示します。SIMD長で割り切れない端数部分i=13～15は非SIMD実行されます。

図3.2 本機能抑止時の動作(m=15の4SIMDイメージ)



また、-Kextract_stride_storeオプションを指定した場合にも本機能が抑止されます。

3.2.7.5 SIMD化可能な数学関数

-Klibオプションが有効な場合にSIMD化可能な数学関数を示します。

```
abs, acos, acosf, acosh, acoshf, asin, asinf, asinh, asinhf, atan, atan2, atan2f, atanf, atanh, atanhf, cbrt, cbrtf, ceil,
ceilf, cimag, cimagf, conj, conjf, copysign, copysignf, cos, cosf, cosh, coshf, creal, crealf, erf, erfc, erfcf, erff, exp,
```

```
exp2, exp2f, expf, fabs, fabsf, floor, floorf, fmax, fmaxf, fmin, fminf, lgamma, lgammaf, log, log10, log10f, log2, log2f, logf, nearbyint, nearbyintf, pow, powf, rint, rintf, round, roundf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, tgamma, tgammaf, trunc, truncf
```

SIMD化可能な数学関数については、“2.2 2.2 翻訳時オプション”の-Kilfuncオプションおよび-Kmfuncオプションの説明も参照してください。

3.2.8 ループアンスイッチング

ループアンスイッチングとは、ループ内に不変な条件で分岐するif文が存在する場合に、そのif文をループの外に出し、if文の条件が成立した場合のループと成立しない場合のループを作成する最適化です。ループ内にあるif文がループの外に追い出されることで、命令スケジューリングなどの最適化が促進されます。

ループアンスイッチングは、最適化レベル3(-O3)、または翻訳時オプション-Kparallelが有効な場合に行われます。また、最適化レベル2(-O2)以下を指定する、かつ翻訳時オプション-Kparallelを無効にすることにより、抑止することができます。

また、-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージを出力することができます。

以下に、ループアンスイッチングの例を示します。

例: ループアンスイッチング

```
void foo(double a[], int b, int c, int n) {
    int i;
    for(i = 0; i < n; i++) {
        if(b == c) {
            a[i] = 0;
        } else {
            a[i] = 1;
        }
    }
}
```

→

```
void foo(double a[], int b, int c, int n) {
    int i;
    if(b == c) {
        for(i = 0; i < n; i++) {
            a[i] = 0;
        }
    } else {
        for(i = 0; i < n; i++) {
            a[i] = 1;
        }
    }
}
```

3.2.9 インライン展開

インライン展開とは、関数の引用(呼出し)がある場合に、その引用箇所に関数の本体を直接展開する最適化です。展開することによって、引数や関数値の受渡しおよび呼出しと復帰のためのオーバヘッド(レジスタの退避・復元と分岐)が削除されます。また、展開された部分が呼出し元と一体化されるので、他の最適化が促進されます。したがって、実行時間を大幅に短縮することができますが、オブジェクトプログラムの大きさが増加します。

インライン展開は、最適化レベル3(-O3)の場合に行われます。また、-x0オプションを指定することにより、抑止することができます。

-Koptmsg=2オプションを指定することにより、展開された場所と関数名を示すメッセージを出力することができます。

3.3 拡張最適化

拡張最適化機能のオプションが指定されると、標準最適化に加えて、拡張最適化が実施されます。

ここでは、拡張最適化の代表的な機能について説明します。

拡張最適化の機能項目によっては、コンパイラが出力するオブジェクトプログラム(命令列およびデータ域)の大きさが大幅に増加したり、実行結果に副作用が生じたりする場合があります。

3.3.1 評価方法を変更する最適化

不変式の先行評価および演算評価方法の変更による最適化について説明します。

3.3.1.1 不変式の先行評価

if文などにより、ループ内で選択的に実行される文に含まれる不変式を、ループの外側に移動します。不変式の先行評価を実施することにより、実行時間を短縮できますが、移動により副作用が生じることがあるので注意が必要です。

不変式の先行評価は、-K{preex|nopreex}オプションで制御することができます。

不変式の先行評価により、プログラムの論理上、実行されないはずの命令が実行され、実行結果に副作用(実行時の例外発生など)を生じることがあります。ただし、計算結果およびその精度に影響を与えることはありません。

実行場所の移動による影響は、標準ライブラリ関数の引用、配列要素の引用および除算で発生する可能性があります。以下に、不変式の先行評価による影響の例を示します。

例1:配列要素の引用

```
int a[10], b[10];
...

for(i = 0; i < 10; i++) {
    if(j < 10) {
        a[i] = b[j] * f;
    }
}
```

→

```
int a[10], b[10];
...
t = b[j] * f;
for(i = 0; i < 10; i++) {
    if(j < 10) {
        a[i] = t;
    }
}
```

[説明]

左側のプログラムでは、変数jが10未満のときだけ配列要素b[j]を引用しているので、配列bの宣言範囲を超えることはありません。

不変式の先行評価により、配列要素b[j]がループ外に移動されて、jの値に関係なく引用されます。その結果、例えばjの値が10以上であれば、配列の宣言範囲を超えて引用されます。

jの値が非常に大きい場合は、不当な領域を参照して、プログラムの実行が中断することがあります。tはコンパイラが生成したint型の変数です。

例2:除算

```
int a[100], b, f;
...

for(i = 0; i < n; i++) {
    if(f != 0) {
        a[i] = b / f;
    } else {
        a[i] = 0;
    }
}
```

→

```
int a[100], b, f;
...
t = b / f;
for(i = 0; i < n; i++) {
    if(f != 0) {
        a[i] = t;
    } else {
        a[i] = 0;
    }
}
```

[説明]

左側のプログラムでは、変数fが0でないときだけ除算を行っているので、除算例外のエラーが発生することはありません。

不変式の先行評価により、除算b/fがループ外に移動されて、fの値に関係なく実行されます。その結果、例えば変数fの値が0であれば、プログラムの実行が中断します。tはコンパイラが生成したint型の変数です。

3.3.1.2 演算評価方法の変更

演算の評価方法を変更して実行時間を短縮します。変更により副作用が生じる場合があります。

演算順序の変更

定数同士の翻訳時計算、不変式の移動、命令スケジューリングなどの最適化を促進するため、演算順序を変更します。

演算例外(オーバフローやアンダフロー)が発生しなかった計算が、演算順序を変更することによって、演算例外が発生する場合があります。また、浮動小数点演算では、各演算項の有効桁によっては、演算結果の精度が異なる場合があります。演算順序の変更は、-K{eval|noeval}オプションで制御することができます。

以下に、演算順序の変更の例を示します。

例:演算順序の変更

```
double a[100], b[100], c[100];
double x, y;
...

for(i = 0; i < 100; i++) {
    a[i] = b[i] * x * c[i] * y;
}
```

→

```
double a[100], b[100], c[100];
double x, y;
...
t = x * y;
for(i = 0; i < 100; i++) {
    a[i] = b[i] * c[i] * t;
}
```

[説明]

不変項同士の演算が集められた後、x*yが不変式としてループの外側に移動されます。

配列b、cおよび変数x、yの値によっては、左側の計算では演算例外が発生せず意図したとおり計算できたものが、右側の計算では演算例外が発生し意図した結果が得られない場合があります。tはコンパイラが生成したdouble型の変数です。

なお、演算順序の変更により演算例外が発生したり演算結果の精度が異なるようなプログラムは、もともと限界値に近い値を処理したり精度に敏感な計算をしているプログラムであるともいえます。したがって、本最適化を実施して副作用が発生したとしても、本最適化が根本的な原因ではない場合もあります。

除算の乗算化

ループ内で不変な除算を乗算に変更します。除算の乗算化により、実行速度を短縮することができますが、実行結果の精度に影響を与える場合があります。除算の乗算化は、-K{eval|noeval}オプションで制御することができます。

以下に、除算の乗算化の例およびその影響を示します。

例:除算の乗算化

```
float a[10], b[10], v;
...

for(i = 0; i < 10; i++) {
    a[i] = b[i] / v;
}
```

→

```
float a[10], b[10], v;
...
t = 1.0 / v;
for(i = 0; i < 10; i++) {
    a[i] = b[i] * t;
}
```

[説明]

ループ内の演算b[i]/vを乗算b[i]*tに変換します。tはコンパイラが生成したfloat型の変数です。

右側のプログラムでは、最適化により1.0/vがループの外に移動されてtの乗算に変更され、計算誤差が生じることがあります。

3.3.2 ポインタの最適化

ポインタの最適化を行わない場合(通常)は、ポインタ変数がどこを指しているかわからないという前提のもとに、安全側に立って最適化を行います。これに対して、ポインタの最適化は、以下の前提条件を付けて最適化を行います。

- 前提条件1

  同じ領域を異なるポインタ変数が指すことはない。

- 前提条件2

  同じ領域に対して、ポインタ変数でアクセスする場合とポインタ変数を使わずに直接アクセスする場合が混在することはない。

例えば、ポインタ変数でアクセスする領域が、malloc関数などで動的に確保した領域に限られていることが明らかな場合は、前提条件2を満たします。

これらの前提条件を付けることにより、共通式の除去や不変式の移動などの最適化が、ポインタ変数を使わずにアクセスする場合と同様に行われるので、高速なオブジェクトプログラムが得られます。

ただし、上記の前提条件を満たさないプログラムに対して、ポインタの最適化を行った場合は、使用者が意図した結果が得られないことがあります。

ポインタの最適化は、-K{restp[={all|arg|restrict}]|norestp}オプションで制御することができます。

例1:ポインタの最適化(前提条件1の説明)

```
a1 = *p + 100;
*q = 0;
a2 = *p + 100;
```

→

```
a1 = *p + 100;
*q = 0;
a2 = a1;      /* 共通式の除去により変更 */
```

[説明]

ポインタ変数pとポインタ変数qは、同じ領域を指していないという前提で最適化します。

その結果、*p+100は共通式とみなされます。もし、ポインタ変数pとポインタ変数qが同じ領域を指している場合は、利用者が意図した結果になりません。

例2:ポインタの最適化(前提条件2の説明)

```
a1 = *p + b;
x = 0;
a2 = *p + b;
```

→

```
a1 = *p + b;
x = 0;
a2 = a1; /* 共通式の除去により変更 */
```

[説明]

ポインタ変数pは、変数a1、a2、b、xを指していないという前提で最適化します。

その結果、*p+bは共通式とみなされます。もし、ポインタ変数pが変数a1または変数xを指している場合は、利用者が意図した結果になりません。

これらの前提条件は、ヘッダについても適用されることに注意してください。

3.3.3 マルチ演算関数

マルチ演算関数とは、1回の呼出しで複数の引数に対する同種の関数計算および演算を行うことにより、実行性能を向上させた関数です。本処理系では、-Kmfuncオプションまたは最適化制御行にmfunc指示子を指定することにより、コンパイラがループを解析し、可能ならばマルチ演算関数に置き換えます。

-Kmfuncオプションおよびマルチ演算関数の対象関数については、“2.2.2.5 -Kオプション”をお読みください。

mfunc指示子については、“3.4.1 最適化制御行(OCL)の利用”をお読みください。

3.3.3.1 マルチ演算関数の直接呼出しについて

複雑なループなどコンパイラが解析できない場合、ユーザープログラムからマルチ演算関数を直接呼び出して利用することができます。

直接呼出しを行う場合、以下の注意が必要です。

- 標準ヘッダfjmfunc.hをインクルードしてください。
- 演算結果の返却に使う引数とそのほかの引数の記憶領域は異なる記憶領域としてください。記憶領域が重なると、結果が保証されないことがあります。
- プログラムのリンク時に-Kmfuncオプションを指定してください。
- マルチ演算関数では高速化のために、引数チェックを省略する場合があります。このため、IEEE 754で規定されている特別な値(NaN、Infなど)が入力された場合は、異常終了することがあります。

“表3.1 ユーザープログラムから直接呼び出すことができるマルチ演算関数一覧”に、ユーザープログラムから直接呼び出すことができるマルチ演算関数を示します。

表3.1 ユーザープログラムから直接呼び出すことができるマルチ演算関数一覧

関数	引数の型	呼出し形式	計算内容
acos	long n; float x[n],y[n];	void v_acos(x,&n,y);	for(i=0; i<n; i++) y[i] = acosf(x[i]);
	long n; double x[n],y[n];	void v_dacos(x,&n,y);	for(i=0; i<n; i++) y[i] = acos(x[i]);

関数	引数の型	呼出し形式	計算内容
asin	long n; float x[n],y[n];	void v_asin(x,&n,y);	for(i=0; i<n; i++) y[i] = asinf(x[i]);
	long n; double x[n],y[n];	void v_dasin(x,&n,y);	for(i=0; i<n; i++) y[i] = asin(x[i]);
atan	long n; float x[n],y[n];	void v_atan(x,&n,y);	for(i=0; i<n; i++) y[i] = atanf(x[i]);
	long n; double x[n],y[n];	void v_datan(x,&n,y);	for(i=0; i<n; i++) y[i] = atan(x[i]);
atan2	long n; float x1[n],x2[n],y[n];	void v_atan2(x1,x2,&n,y);	for(i=0; i<n; i++) y[i] = atan2f(x1[i],x2[i]);
	long n; double x1[n],x2[n],y[n];	void v_datan2(x1,x2,&n,y);	for(i=0; i<n; i++) y[i] = atan2(x1[i],x2[i]);
erf	long n; float x[n],y[n];	void v_erf(x,&n,y);	for(i=0; i<n; i++) y[i] = erff(x[i]);
	long n; double x[n],y[n];	void v_derf(x,&n,y);	for(i=0; i<n; i++) y[i] = erf(x[i]);
erfc	long n; float x[n],y[n];	void v_erfc(x,&n,y);	for(i=0; i<n; i++) y[i] = erfcf(x[i]);
	long n; double x[n],y[n];	void v_derfc(x,&n,y);	for(i=0; i<n; i++) y[i] = erfc(x[i]);
exp	long n; float x[n],y[n];	void v_exp(x,&n,y);	for(i=0; i<n; i++) y[i] = expf(x[i]);
	long n; double x[n],y[n];	void v_dexp(x,&n,y);	for(i=0; i<n; i++) y[i] = exp(x[i]);
exp10	long n; float x[n],y[n];	void v_exp10(x,&n,y);	for(i=0; i<n; i++) y[i] = exp10f(x[i]);
	long n; double x[n],y[n];	void v_dexp10(x,&n,y);	for(i=0; i<n; i++) y[i] = exp10(x[i]);
log	long n; float x[n],y[n];	void v_log(x,&n,y);	for(i=0; i<n; i++) y[i] = logf(x[i]);
	long n; double x[n],y[n];	void v_dlog(x,&n,y);	for(i=0; i<n; i++) y[i] = log(x[i]);
log10	long n; float x[n],y[n];	void v_log10(x,&n,y);	for(i=0; i<n; i++) y[i] = log10f(x[i]);
	long n; double x[n],y[n];	void v_dlog10(x,&n,y);	for(i=0; i<n; i++) y[i] = log10(x[i]);
sin	long n; float x[n],y[n];	void v_sin(x,&n,y);	for(i=0; i<n; i++) y[i] = sinf(x[i]);
	long n; double x[n],y[n];	void v_dsin(x,&n,y);	for(i=0; i<n; i++) y[i] = sin(x[i]);

関数	引数の型	呼出し形式	計算内容
cos	long n; float x[n],y[n];	void v_cos(x,&n,y);	for(i=0; i<n; i++) y[i] = cosf(x[i]);
	long n; double x[n],y[n];	void v_dcos(x,&n,y);	for(i=0; i<n; i++) y[i] = cos(x[i]);
sincos	long n; float x[n],y1[n],y2[n];	void v_scn(x,&n,y1,y2);	for(i=0; i<n; i++) sincosf(x[i], &y1[i], &y2[i]);
	long n; double x[n],y1[n],y2[n];	void v_dscn(x,&n,y1,y2);	for(i=0; i<n; i++) sincos(x[i], &y1[i], &y2[i]);
pow	long n; float x1[n],x2[n],y[n];	void v_pow(x1,x2,&n,y);	for(i=0; i<n; i++) y[i] = powf(x1[i], x2[i]);
	long n; float x1[n],x2[n],y[n],a;	void v_pow1(x,&a,&n,y);	for(i=0; i<n; i++) y[i] = powf(x[i], a);
	long n; double x1[n],x2[n],y[n];	void v_dpow(x1,x2,&n,y);	for(i=0; i<n; i++) y[i] = pow(x1[i], x2[i]);
	long n; double x1[n],x2[n],y[n],a;	void v_dpow1(x,&a,&n,y);	for(i=0; i<n; i++) y[i] = pow(x[i], a);

以下にマルチ演算関数の直接呼出しの例を示します。

例1:

```
#include <math.h>
#define N 1000
...
    double a;
    double x[N], y[N], z[N];
    int i;
...
    for(i = 0; i < N; i++) {
        y[i] = exp(x[i]);
        z[i] = pow(x[i], a);
    }
```

マルチ演算関数の直接呼出し

```
#include <fjmfunc.h>
#define N 1000
...
    long n = N;
    double a;
    double x[N], y[N], z[N];
...
    v_dexp(x, &n, y);
    v_dpow1(x, &a, &n, z);
```

例2:

if文の中で関数が呼ばれている場合、計算すべき要素だけを配列に格納することにより、マルチ演算関数を使用することができます。

```
#include <math.h>
#define N 1000
...
    double a, x, y;
    int i;
...
```

```
    for(i = 0; i < N; i++) {
        x = ...
        if(x > a) {
            y = y + sin(x);
        }
    }
```

マルチ演算関数の直接呼出し

```
#include <fjmfunc.h>
#define N 1000
...
    double a, x, y;
    double wx[N], wy[N];
    int i;
    long j;
...
    j = 0;
    for(i = 0; i < N; i++) {
        x = ...
        if(x > a) {
            wx[j] = x;
            j++;
        }
    }
    v_dsin(wx, &j, wy);
    for(i = 0; i < j; i++) {
        y = y + wy[i];
    }
```

3.3.3.2 -Kmfunc=3オプションの影響

-Kmfunc=3オプションが指定された場合、if文を含むループのマルチ演算関数化が行われます。if文を含むループがマルチ演算関数化された場合、if文の判定結果に関係なく、ループの繰返し数分の配列要素の参照を行うため、実行時異常終了となる場合があります。

以下に、-Kmfunc=3による影響の例を示します。

例: if文を含むループの例

```
    double x[1000], y[1000];
    int i;
...
    for(i = 0; i < 2000; i++) {
        if(i < 1000) {
             y[i] = exp(x[i]);
        }
    }
```

上記の例では、条件i<1000を満たす場合にif文の判定が真となります。exp関数の引数値は1000未満となるため、配列x、yの添字の値は宣言された上下限の範囲を超えることはありません。

マルチ演算関数化では、if文の条件に関係なく、for文の繰返し数分の配列要素を、マルチ演算関数化された関数の引数として使用するため、配列xは宣言範囲を超えて引用されます。そのため、記憶保護例外(読み出し)が発生してプログラムの実行が中断することがあります。

このような場合、-Kmfunc=3オプションを指定しないでください。

3.3.4 ループストライピング

ループストライピングとは、ループ内の実行文を、ストライプ長と呼ぶ一定数だけ展開する最適化です。ループストライピングを適用することにより、ループの繰返しによるオーバヘッドが少なくなる、またはソフトウェアパイプライニングが促進されるという効果が期待できます。

ループストライピングは、ループ外からの飛込みもループ外への飛出しもないループに対して行われます。

例:ループストライピング

ストライプ長4(-Kstriping=4)で展開することが指定された場合、以下のように実行文がループ内で展開されます。

```
for(i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

→

```
for(i = 0; i < N; i += 4) {
    tmp_b1 = b[i];
    tmp_b2 = b[i+1];
    tmp_b3 = b[i+2];
    tmp_b4 = b[i+3];
    tmp_c1 = c[i];
    tmp_c2 = c[i+1];
    tmp_c3 = c[i+2];
    tmp_c4 = c[i+3];
    tmp_a1 = tmp_b1 + tmp_c1;
    tmp_a2 = tmp_b2 + tmp_c2;
    tmp_a3 = tmp_b3 + tmp_c3;
    tmp_a4 = tmp_b4 + tmp_c4;
    a[i] = tmp_a1;
    a[i+1] = tmp_a2;
    a[i+2] = tmp_a3;
    a[i+3] = tmp_a4;
}
```

ループストライピングは、ループ内の文が多重に展開されるので、ループアンローリング同様、オブジェクトプログラムの大きさが増加します。また、多くの翻訳時間を必要とする場合があります。実行性能に関しても、使用するレジスタが増加するため実行性能が低下する場合があるので注意が必要です。

ループストライピングは、-Knostripingオプションを指定することにより、抑止することができます。

また、-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージが出力されます。

3.3.5 zfill

zfillとは、データをメモリからロードすることなく、キャッシュ上に書き込み用の領域を確保する命令(DC ZVA)を使い、データの書き込み動作を高速にする最適化です。zfillは、ループ内でストアされる配列データに対して適用されます。

ただし、同一ループ内に参照がある配列、非連続アクセスされる配列、またはif文配下でストアされる配列に対しては適用されません。また、zfillが適用された場合、2次キャッシュへのプリフェッチ命令は出力されません。

zfillの最適化は対象となるストア命令の指すアドレスを起点として、256バイトを1ブロックとする単位で*N*ブロック分先のデータを最適化の対象とします。*N*はオプションまたは最適化制御行で指定する1から100までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に値を決定します。

zfillの最適化は確保した領域は必ずストアするようなループ変形を行うため、以下の最適化が適用できなくなります。

- ループアンローリング
- ループストライピング

また、以下の場合に本最適化を適用すると実行性能が低下することがあります。

- メモリバンド幅のボトルネック影響を受けていないプログラム
- 繰返し数が小さいループ
- -Kzfill=*N*オプションでブロック数を指定している、かつ、ループによって書き込まれるメモリ領域のサイズが*N*ブロックよりも小さい場合

-Kzfillオプションの指定によって実行性能の低下が起きる場合は、-Kzfillオプションを指定しないでください。zfillの制御は本来ループ単位で行うことが望ましいです。したがって、プログラム全体に影響のあるオプション指定より、最適化指示子zfill指示子を指定することを推奨します。

3.3.6 ループバージョニング

翻訳時オプション-Kloop_versioningを指定した場合、実行時に配列のデータ依存を判断します。コンパイラは、配列のデータ依存がないものとして最適化を適用したループおよび配列のデータ依存があるものとして最適化を適用しないループを生成します。実行時に配列のデータ依存を判断し、どちらかのループを選択します。

本機能により、SIMD化、ソフトウェアパイプライニング、または自動並列化などの最適化が促進されます。

ループバージョニングは、ループを複数生成するため、オブジェクトプログラムの大きさおよび翻訳時間が増加する場合があります。

また、生成したループを選択するための判定処理がオーバヘッドとなり、実行性能が低下する場合があります。

このオーバヘッドを軽減するため、本機能は、最内ループのみに適用されます。また、ループ中にデータ依存が不明な配列が1つだけの場合に適用されます。

-Koptmsg=2オプションを指定することにより、最適化された箇所を示すメッセージが出力されます。

以下に、実行時に配列のデータ依存が判断できる例を示します。

例:

```
void f(double *a, double *b, int n) {
  for (int i = 0; i < n; i++) {
    a[i] += b[i];
  }
}
```

[最適化後(ソースイメージ)]

```
void f(double *a, double *b, int n) {
  if ((&a[n-1] < &b[0]) || (&b[n-1] < &a[0])) {
#pragma loop norecurrence
    for (int i = 0; i < n; i++) {  /* 最適化を適用したループ */
      a[i] += b[i];
    }
  } else {
    for (int i = 0; i < n; i++) {  /* 最適化を適用しないループ */
      a[i] += b[i];
    }
  }
}
```

翻訳時は、配列aと配列bに重なりがある可能性があるため依存関係が解析できません。翻訳時オプション-Kloop_versioningが有効な場合、実行時に配列aおよび配列bの重なり具合を調べて、最適化を適用したループまたは最適化を適用しないループのどちらかを選択します。

3.3.7 clone最適化

ループに対して変数の値による条件分岐を生成し、フルアンローリングなどのほかの最適化を促進する最適化です。clone最適化は最適化制御行で動作します。誤った最適化制御行の使い方をした場合、結果が保証されません。詳細は"3.4.1.2 最適化指示子の種類"および"3.4.1.3 最適化指示子の注意事項"をお読みください。

例:

```
#pragma loop clone n==10
for (i=0;i<n; ++i) {
    a[i] = i;
}
```

→

```
if (n==10) {
    for (i=0;i<10;++i) {
        a[i] = i;
    }
} else {
    for (i=0;i<n;++i) {
        a[i] = i;
    }
}
```

3.3.8 アンロールアンドジャム

アンロールアンドジャムとは、多重ループの外側のループをアンローリングにより*N*重に展開し、さらに展開された内側のループを融合する最適化です。アンロールアンドジャムを適用することで、共通式の除去が促進され、実行性能が向上する場合があります。ただし、データストリーム数の増加やデータのアクセス順序の変化は、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。なお、本最適化は最内ループには適用されません。

また、本最適化による実行性能への影響はループ単位で異なります。このため、本最適化は、-Kunroll_and_jam[=*N*]オプションでプログラム全体へ適用せず、unroll_and_jam指示子やunroll_and_jam_force指示子でループ単位に適用することを推奨します。詳細は、“3.4.1 最適化制御行(OCL)の利用”のunroll_and_jamまたはunroll_and_jam_forceをお読みください。

-Koptmsg=2オプションを指定することにより、最適化されたループとその展開数を示すメッセージを出力することができます。

例: アンロールアンドジャム

```
#pragma loop unroll_and_jam_force 2
for(i = 0; i < 128; i++) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

→

```
for(i = 0; i < 128; i += 2) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        a[i+1][j] = b[i+1][j] + b[i+2][j];
        ...
    }
}
```

3.3.9 tree-height-reduction最適化

tree-height-reduction最適化とは、ループ中の演算に対して演算木をなるべく低くなるように演算の並び替えを行い、命令の並列性を高める最適化です。

演算木とは、演算子の優先順位に従って演算式を木構造で表現したものです。葉の部分には、値を置きます。それ以外の節には、演算子を置きます。優先順位が高い演算子ほど、上位の階層に配置します。

-Keval_concurrentオプションが有効な場合に、本最適化が浮動小数点演算に対して適用されます。この場合、実行結果に副作用(計算誤差)を生じることがあります。最適化の副作用については、“3.6.1 浮動小数点演算に対する最適化の副作用”を参照してください。

以下に最適化の例を示します。

例:

```
for(i = 0; i < n; i++) {
    x[i] = a[i] * b[i] + c[i] * d[i] + e[i] * f[i] + g[i] * h[i];
}
```

図3.3 tree-height-reduction最適化前の演算木

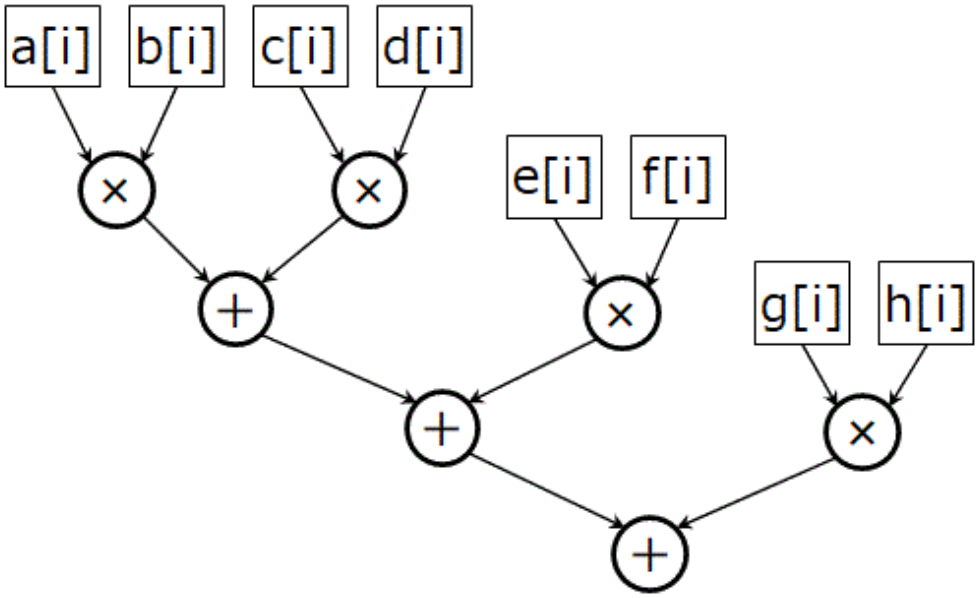
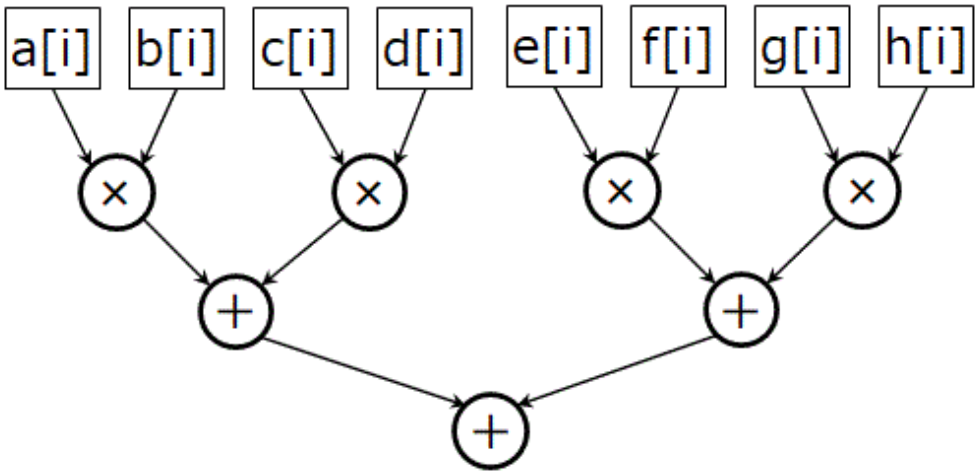


図3.4 tree-height-reduction最適化後の演算木



3.3.10 ループ分割

ループ分割は、以下を目的としてループを複数の小さなループに分割する機能です。

- ソフトウェアパイプライニングの促進
- キャッシュメモリ利用効率の改善
- レジスタ不足の解消

ループ分割は、ループ外からの飛込みもループ外への飛出しもないループに対して行われます。

自動ループ分割は、-Kloop_fissionオプションおよび-Koclオプションが有効な場合に、loop_fission_target指示子が指定されたループに対して行われます。-Kloop_nofissionオプションを指定することにより、ループ分割を抑止することができます。

-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージを出力することができます。

以下に、自動ループ分割の例を示します。

例: 自動ループ分割

```
#pragma loop loop_fission_target
for(i = 0; i < N; i++) {
    e[i] = a1[i]*b1[i] + a2[i]*b2[i] + a3[i]*b3[i] +
           a4[i]*b4[i] + a5[i]*b5[i];
    f[i] = c1[i]*d1[i] + c2[i]*d2[i] + c3[i]*d3[i] +
           c4[i]*d4[i] + c5[i]*d5[i];
}
```

→

```
for(i = 0; i < N; i++) {
    e[i] = a1[i]*b1[i] + a2[i]*b2[i] + a3[i]*b3[i] +
           a4[i]*b4[i] + a5[i]*b5[i];
}

for(i = 0; i < N; i++) {
    f[i] = c1[i]*d1[i] + c2[i]*d2[i] + c3[i]*d3[i] +
           c4[i]*d4[i] + c5[i]*d5[i];
}
```

3.3.10.1 ストリップマイニング

ストリップマイニングとは、ループを小さい繰返し数で断片化する最適化です。自動ループ分割後のループにこの最適化を適用することにより、ループ分割されたループ間でアクセスされるデータに対して、キャッシュメモリの利用効率の向上が期待できます。

ストリップマイニングは、-Kloop_fission_stripminingオプションを指定した場合に行われます。-Koptmsg=2オプションを指定することにより、最適化されたループを示すメッセージを出力することができます。

以下に、自動ループ分割およびストリップマイニング(-Kloop_fission_stripmining=256オプション指定時)の例を示します。

例: 自動ループ分割およびストリップマイニング

[ソースプログラム]

```
#pragma loop loop_fission_target
float a[N], b[N], p[N], q;
```

```
for(i = 0; i < N; i++) {
    q = a[i] + b[i];
    ...
    p[i] = p[i] + q;
    ...
}
```

[自動ループ分割後のソースイメージ]

```
float a[N], b[N], p[N], q;
float temparray_q[N];
for(i = 0; i < N; i++) {
    temparray_q[i] = a[i] + b[i];
    ...
}
for(i = 0; i < N; i++) {
    p[i] = p[i] + temparray_q[i];
    ...
}
```

このソースプログラムでは、ループ分割したループ間でデータを受け渡す必要があるため、ループ分割に伴い、コンパイラが一時的な配列temparray_qを生成します。この配列の要素数はループの繰返し数と同じ値になります。

[自動ループ分割およびストリップマイニング後のソースイメージ]

```
#define MIN(x,y) ((x<y)?x:y)
float a[N], b[N], p[N], q;
float temparray_q[256];
for(ii = 0; ii < N; ii = ii+256) {
    for(i = ii; i < MIN(N, ii+256); i++) {
        temparray_q[i-ii] = a[i] + b[i];
        ...
    }
    for(i = ii; i < min(N, ii+256); i++) {
        p[i] = p[i] + temparray_q[i-ii];
        ...
    }
}
```

分割したループの外側にループを生成し、元のループの繰返し数をストリップ長の256で断片化します。一時的な配列temparray_qの要素数は256となります。

3.3.11 Strict Aliasing

Strict Aliasingは、言語規格で規定された厳密なaliasing ruleに従ってメモリ領域の重なりを考慮した最適化を許すか否かを指示します。対象となる最適化は、不変式の移動や共通式の除去などです。

Strict Aliasingは、-K{strict_aliasing|nostrict_aliasing}オプションで制御できます。

以下に、aliasの例を示します。

例1:

-Kstrict_aliasingオプションが指定された場合、int型とfloat型はaliasがないと判断し、最適化を促進します。

```
void sub(int *a, float *b) {
    long i;
    for(i = 0; i < 256; i++) {
        a[i] = *b;
    }
}
```

例2:

-Kstrict_aliasingオプションが指定された場合でも、int型とunsigned int型はaliasがあると判断し、最適化を促進しません。

```
void sub(int *a, unsigned int *b) {
    long i;
    for(i = 0; i < 256; i++) {
        a[i] = *b;
    }
}
```

例3:

-Kstrict_aliasingオプションが指定された場合でも、文字型と他の型は常にaliasがあると判断し、最適化を促進しません。

```
void sub(int *a, char *b) {
    long i;
    for(i = 0; i < 256; i++) {
        a[i] = *b;
    }
}
```

例4:

規格に準拠していないプログラムに-Kstrict_aliasingオプションが指定された場合、誤ってaliasがないと判断し、最適化を促進してしまう可能性があります。このため、実行結果は保証されません。

```
void sub(int *i_pointer, long *l_pointer) {
    *i_pointer = 10;
    *l_pointer = 15;
}
int main() {
    long double l_double = 0.0;
    int        *i_pointer = &l_double;
    long       *l_pointer = &l_double;
    sub(i_pointer, l_pointer);
    if(*i_pointer == *l_pointer) {
        // OK
    } else {
        // NG
    }
    return 0;
}
```

3.4 最適化機能の活用方法

最適化機能を有効に活用するための機能について説明します。

3.4.1 最適化制御行(OCL)の利用

最適化にとって有効な情報をソースプログラム上に記述することにより、最適化の効果をより高めることができます。

#pragma行に以下の記述がある場合、それを最適化制御行と呼びます。

最適化制御行は、-Koclまたは-Knooclオプションで制御することができます。

3.4.1.1 最適化制御行(OCL)の種類

最適化制御行の種類とその有効範囲を“表3.2 最適化制御行と有効範囲”に示します。

表3.2 最適化制御行と有効範囲

前処理字句列	最適化制御行の名称	最適化の有効範囲
global	global行	翻訳単位内
procedure	procedure行	関数内
loop	loop行	直後のループ

前処理字句列	最適化制御行の名称	最適化の有効範囲
statement	statement行	直後の文

global行

指定された最適化指示子を翻訳単位内すべてに有効にします。

書式

```
#pragma[ fj] global 最適化指示子
```

挿入位置

対象にしたい翻訳単位の先頭に記述します。

procedure行

指定された最適化指示子を関数内で有効にします。

書式

```
#pragma[ fj] procedure 最適化指示子
```

挿入位置

対象にしたい関数の宣言文(宣言と同時に初期値を設定している文も含みます)と最初の実行文の間に記述します。

loop行

指定された最適化指示子を直後のループで有効にします。

書式

```
#pragma[ fj] loop 最適化指示子
```

挿入位置

対象にしたいループの直前に記述します。

statement行

指定された最適化指示子を直後の文で有効にします。

書式

```
#pragma[ fj] statement 最適化指示子
```

挿入位置

対象にしたい文の直前に記述します。

注意

最適化制御行に指定できる最適化指示子は1つです。global行、procedure行、およびloop行では、複数の最適化制御行を連続して記述することにより、複数の最適化指示子が有効となります。

statement行を連続して記述した場合、最後のstatement行が有効となります。

3.4.1.2 最適化指示子の種類

“表3.3 最適化制御行に指定できる最適化指示子”に、最適化に対する最適化指示子を以下に示します。

表3.3 最適化制御行に指定できる最適化指示子

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
array_declaration_opt	最適化を行うときに、配列の添字が配列宣言の範囲を超えないことを前提にします。	○	○	○	×
noarray_declaration_opt	最適化を行うときに、配列の添字が配列宣言の範囲を超えないことを前提にしません。	○	○	○	×
assume {shortloop|noshortloop|memory_bandwidth| nomemory_bandwidth|time_saving_compilation| notime_saving_compilation}	プログラムの特徴に合わせて、最適化を調整することを指示します。	○	○	×	×
clone *var*==*n1*[,*n2*]...(注2)	ループ内で変数*var*が不変とみなして、引数で指示をした条件分岐を生成し、条件節内にループを複製する最適化を行うことを指示します。 条件式は第一引数の変数*var*と、第二引数以降で指定した値*n1*[,*n2*]...の等式とします。 *var*は整数型の変数です。 *n1*[,*n2*]...は-9223372036854775808から9223372036854775807までの整数値です。	×	×	○	×
eval	演算の評価方法を変更する最適化を行うことを指示します。	○	○	○	×
noeval	演算の評価方法を変更する最適化を行わないことを指示します。	○	○	○	×
eval_concurrent	tree-height-reduction最適化において、命令の並列性を優先することを指示します。	○	○	○	×
eval_noconcurrent	tree-height-reduction最適化において、命令の並列性を抑え、FMA命令の利用を優先することを指示します。	○	○	○	×
extract_stride_store	SIMD化対象ループにあるストライドアクセスのストア命令を、スカラ命令に展開することを指示します。	○	○	○	×
noextract_stride_store	SIMD化対象ループにあるストライドアクセスのストア命令を、スカラ命令に展開しないことを指示します。	○	○	○	×
fission_point [*n*]	ループ内の指定された位置でループを分割することを指示します。最内ループから数えて*n*重ネストされた多重ループを対象に分割します。*n*は1～6の整数値です。 コンパイラがループ分割可能と判断し、ループ分割の際にループ間でデータを受け渡す必要がある場合は、コンパイラが自動的に一時的な領域を生成します。そのため実行結果には影響ありません。	×	×	×	○
fp_contract	Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うことを指示します。	○	○	○	×
nofp_contract	Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行わないことを指示します。	○	○	○	×
fp_relaxed	浮動小数点除算、sqrt関数について、高速な演算を行うことを指示します。	○	○	○	×

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
nofp_relaxed	浮動小数点除算、sqrt関数について、通常の演算を行うことを指示します。	○	○	○	×
fullunroll_pre_simd [*n*]	SIMD化前のフルアンローリングを促進することを指示します。*n*は、対象とするループの回転数の上限を表す2～100の整数値です。	×	×	○	×
nofullunroll_pre_simd	SIMD化前のフルアンローリングを抑止することを指示します。	×	×	○	×
iterations max=*n1* iterations avg=*n2* iterations min=*n3*	ループの繰返し数の最大値を*n1*、平均値を*n2*、最小値を*n3*とみなして最適化することを指示します。*n1*、*n2*、および*n3*は、0から2147483647までの整数値です。max、avg、およびminは個別または空白を区切りとして順不同の複数指定が可能です。	○	○	○	×
loop_blocking *n*	ブロッキング機能を有効にします。 *n*はブロックサイズを表す2～10000の整数値です。	○	○	○	×
loop_noblocking	ブロッキング機能を無効にします。	○	○	○	×
loop_fission_stripmining [*n*|"*c-level*"]	自動ループ分割したループに対して、ストリップマイニングの最適化を行うことを指示します。*n*は、2から100000000までの整数値です。*c-level*は、L1またはL2です。	○	○	○	×
loop_nofission_stripmining	自動ループ分割したループに対して、ストリップマイニングの最適化を抑止します。	○	○	○	×
loop_fission_target [cl|ls]	コンパイラによる自動ループ分割を行うことを指示します。	×	×	○	×
loop_fission_threshold *n*	自動ループ分割における分割後のループの粒度を決める閾値*n*を指示します。*n*は、1から100までの整数値です。	○	○	○	×
loop_interchange *var1*,*var2*[,*var3*]...(注2)	指定された順序*var1*、*var2*、...で多重forループの入換えを指示します。	×	×	○	×
loop_nointerchange	多重forループの入換えを実施しません。	×	×	○	×
loop_nofusion	ループを融合する機能を無効にすることを指示します。	○	○	○	×
loop_part_simd	ループを分割して部分的にSIMD化する機能を有効にします。	○	○	○	×
loop_nopart_simd	ループを分割して部分的にSIMD化する機能を無効にします。	○	○	○	×
loop_perfect_nest	不完全多重ループを分割して完全多重ループにすることを指示します。	○	○	○	×
loop_noperfect_nest	不完全多重ループを完全多重ループにしないことを指示します。	○	○	○	×
loop_versioning	ループバージョニングの最適化を行うことを指示します。	○	○	○	×
loop_noversioning	ループバージョニングの最適化を行わないことを指示します。	○	○	○	×
mfunc [*level*]	マルチ演算関数化を行うことを指示します。	○	○	○	×

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
	*level*は機能レベルを表す整数値で、1、2、または3です。				
nomfunc	マルチ演算関数化を行わないことを指示します。	○	○	○	×
noalias	ポインタ変数が他の変数と記憶域を共有しないことを指示します。	○	○	○	×
norecurrence [*array1*[,*array2*]...](注2)	ループの繰返しを跨いで定義引用されない配列を指示します。 *array1*、*array2*、…は配列名です。	○	○	○	×
novrec [*array1*[,*array2*]...](注2)	回帰演算がないループ(SIMD命令が利用可能なループ)であることを指示します。 *array1*、*array2*、...は配列名です。	○	○	○	×
preex	不変式の先行評価を行うことを指示します。	○	○	○	×
nopreex	不変式の先行評価を行わないことを指示します。	○	○	○	×
prefetch	コンパイラの自動prefetch機能を有効にすることを指示します。	○	○	○	×
noprefetch	コンパイラの自動prefetch機能を無効にすることを指示します。	○	○	○	×
prefetch_cache_level *c-level*	キャッシュレベル*c-level*にデータをプリフェッチすることを指示します。 *c-level*は1、2、またはallです。	○	○	○	×
prefetch_conditional	if文やswitch文に含まれるブロックの中で使用される配列データに対してprefetch命令を生成することを指示します。	○	○	○	×
prefetch_noconditional	if文やswitch文に含まれるブロックの中で使用される配列データに対してprefetch命令を生成しないことを指示します。	○	○	○	×
prefetch_indirect	ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成することを指示します。	○	○	○	×
prefetch_noindirect	ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成しないことを指示します。	○	○	○	×
prefetch_infer	プリフェッチの距離が不明な場合でも連続アクセスのプリフェッチを出力することを指示します。	○	○	○	×
prefetch_noinfer	プリフェッチの距離が不明な場合は連続アクセスのプリフェッチを出力しないことを指示します。	○	○	○	×
prefetch_iteration *n*	prefetch命令を生成する際、ループの*n*回転後に引用されるデータを対象とすることを指示します。本指示子では1次キャッシュにのみプリフェッチするprefetch命令を対象とします。*n*は1～10000の整数値です。SIMD化が適用されるループの場合、*n*にはSIMD化されたあとのループの繰返し数を指定してください。	○	○	○	×

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
prefetch_iteration_L2 *n*	prefetch命令を生成する際、ループの*n*回転後に引用されるデータを対象とすることを指示します。本指示子では2次キャッシュにのみプリフェッチするprefetch命令を対象とします。*n*は1～10000の整数値です。SIMD化が適用されるループの場合、*n*にはSIMD化されたあとのループの繰返し数を指定してください。	○	○	○	×
prefetch_line *n*	prefetch命令を生成する際、*n*キャッシュライン先に該当するデータを対象とすることを指示します。本指示子では1次キャッシュにプリフェッチするprefetch命令を対象とします。 *n*は1～100の整数値です。	○	○	○	×
prefetch_line_L2 *n*	prefetch命令を生成する際、*n*キャッシュライン先に該当するデータを対象とすることを指示します。本指示子では2次キャッシュにプリフェッチするprefetch命令を対象とします。 *n*は1～100の整数値です。	○	○	○	×
prefetch_sequential [auto|soft]	連続的にアクセスされる配列データに対してprefetch命令を生成することを指示します。	○	○	○	×
prefetch_nosequential	連続的にアクセスされる配列データに対してprefetch命令を生成しないことを指示します。	○	○	○	×
prefetch_stride [soft|hard_auto|hard_always]	ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、プリフェッチを実施することを指示します。	○	○	○	×
prefetch_nostride	ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成しないことを指示します。	○	○	○	×
prefetch_strong	1次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。	○	○	○	×
prefetch_nostrong	1次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。	○	○	○	×
prefetch_strong_L2	2次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。	○	○	○	×
prefetch_nostrong_L2	2次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。	○	○	○	×
preload	ロード命令を投機実行する最適化を行うことを指示します。	○	○	○	×
nopreload	ロード命令を投機実行する最適化を行わないことを指示します。	○	○	○	×
scache_isolate_way L2=*n1* [L1=*n2*] および end_scache_isolate_way	1次キャッシュと2次キャッシュのセクタ1の最大ウェイ数を指示します。	×	○	×	○

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
scache_isolate_assign *array1*[,*array2*]...(注2) および end_scache_isolate_assign	キャッシュのセクタ1に載せる配列を指定します。 *array1*、*array2*、...は配列名です。	×	○	×	○
simd [aligned|unaligned]	SIMD化を有効にします。	○	○	○	×
nosimd	SIMD化を無効にします。	○	○	○	×
simd_listv [all|then|else]	if文のブロック内の実行文をリストベクトル変換することを指示します。	×	×	×	○
simd_noredundant_vl	SIMD長の倍数冗長実行によるSIMD化機能を無効にします。	○	○	○	×
simd_use_multiple_structures	SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用します。	○	○	○	×
simd_nouse_multiple_structures	SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用しません。	○	○	○	×
striping [*n*]	ループストライピングの最適化を有効にします。 *n*はストライプ長(展開数)を表す2～100の整数値です。	○	○	○	×
nostriping	ループストライピングの最適化を無効にします。	○	○	○	×
swp	ソフトウェアパイプライニング機能を有効にすることを指示します。	○	○	○	×
noswp	ソフトウェアパイプライニング機能を無効にすることを指示します。	○	○	○	×
swp_freg_rate *n* swp_ireg_rate *n* swp_preg_rate *n*	ソフトウェアパイプライニング機能における、レジスタ数の条件を変更することを指示します。 *n*はソフトウェアパイプライニングで使用可能なレジスタ数の割合(百分率)を指定します。*n*は1～1000までの整数値です。	○	○	○	×
swp_policy {auto|small|large}	ソフトウェアパイプライニングで使用する命令スケジューリングアルゴリズムの選択基準を指示します。	○	○	○	×
swp_weak	ソフトウェアパイプライニング機能を調整し、実行文の重なりを小さくします。	○	○	○	×
unroll [*n*|"full"]	対応するループをアンローリングすることを指示します。*n*はループアンローリングの展開数(多重度)を指定します。*n*は2～100の整数値です。	×	×	○	×
nounroll	対応するループをアンローリングしないことを指示します。	×	×	○	×
unroll_and_jam [*n*]	最適化の効果があると判断したループに対してアンロールアンドジャムの最適化を有効にします。 *n*は展開数(多重度)を表す2～100の整数値です。	○	○	○	×
unroll_and_jam_force [*n*]	アンロールアンドジャムの最適化を有効にします。 *n*は展開数(多重度)を表す2～100の整数値です。	×	×	○	×
nounroll_and_jam	アンロールアンドジャムの最適化を無効にします。	○	○	○	×

最適化指示子	機能説明	指定できる最適化制御行(注1)			
		global行	procedure行	loop行	statement行
unswitching	指定されたif文をループアンスイッチングの対象とすることを指示します。	×	×	×	○
zfill [*N*]	zfillの最適化を有効にします。 *N*はDC ZVA命令の書き込みブロック数を表す1～100の整数値です。	×	×	○	×
nozfill	zfillの最適化を無効にします。	×	×	○	×

(注1)
○:最適化指示子を最適化制御行に指定できます。
×:最適化指示子を最適化制御行に指定できません。

(注2) *var*、*var1*、*var2*、*var3*、*array1*、*array2*は使用する前に宣言してください。

array_declaration_opt指示子

最適化を行うときに、配列の添字が配列宣言の範囲を超えないことを前提にします。

以下に例を示します。

例:

```
double a[8];
#pragma loop array_declaration_opt
for(int i = 0; i < n; i++) {
  a[i] = 0;
}
```

添字の動作範囲をSIMD長以下と仮定して最適化を行います。

noarray_declaration_opt指示子

最適化を行うときに、配列の添字が配列宣言の範囲を超えないことを前提にしません。

以下に例を示します。

例:

```
double a[8];
#pragma loop noarray_declaration_opt
for(int i = 0; i < n; i++) {
  a[i] = 0;
}
```

添字の動作範囲を不明として最適化を行います。

assume指示子

プログラムの特徴に合わせて、最適化の調整をすることを指示します。複数のassume指示子を同時に指定することもできます。

assume指示子に続けて以下のいずれかを指定します。

shortloop

プログラム中の最内ループにおいて、ループの繰返し数が翻訳時に不明な場合は繰返し数が小さいとみなして最適化を行います。
自動並列化、ループアンローリング、ソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

noshortloop

プログラム中の最内ループにおいて、ループの繰返し数が翻訳時に不明な場合は繰返し数が大きいとみなして最適化を行います。

memory_bandwidth

プログラム中の最内ループにおいて、メモリバンド幅がボトルネックになるとみなして最適化を行います。zfillの最適化の促進およびソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

nomemory_bandwidth

プログラム中の最内ループにおいて、メモリバンド幅がボトルネックにならないとみなして最適化を行います。

time_saving_compilation

プログラムの翻訳時間が短くなるように最適化を調整します。

notime_saving_compilation

プログラムの翻訳時間の短縮より、実行可能プログラムの高速化を優先します。

以下に例を示します。

例:

```
#pragma global assume shortloop
#pragma global assume memory_bandwidth
#pragma global assume time_saving_compilation
void sub() {
    for(int i = 0; i < N; i++) {
        a[i] = b[i];
    }
}
```

対象となるループにおいて、指示された特徴に合わせて、最適化の調整を行います。

clone指示子

ループ内で変数*var*の値が不変とみなして、指定された変数*var*と値*n1*[,*n2*]...の等式を条件式とする分岐を生成し、ループを複製することを指示します。分岐は指定した値の順に生成されます。本機能により、フルアンローリングなどのほかの最適化を促進します。そのため、対象となるループ内で変数*var*が更新される場合、結果が保証されません。詳細については、“3.4.1.3 最適化指示子の注意事項”をお読みください。cloneはループを複製するため、オブジェクトプログラムの大きさおよび翻訳時間が増加する場合があります。本最適化指示子は-O3オプションが有効な場合に意味があります。

*var*は整数型の変数です。

整数型であっても、以下の場合は指定できません。

- 構造体メンバ変数
- 共用体メンバ変数
- 配列要素
- threadprivate対象変数

n1[,*n2*],...は-9223372036854775808から9223372036854775807までの整数値です。値の指定方法として、コンマ“,”による列挙、およびコロン“:”による値の範囲指定ができます。コンマ“,”とコロン“:”を組み合わせて指定することが可能です。下記の2つの指定は同等です。

```
#pragma loop clone var==1,3:5,7
```

```
#pragma loop clone var==1,3,4,5,7
```

また、1つのclone指示子で指定できる値の個数の上限は20個です。21個以上の値を指定した場合、個数の上限を超えた値の指定は無効とされます。

下記の指定は、30個の値を指定しているため、個数の制約を受けます。

```
#pragma loop clone var==11:40
```

個数の上限を超えた値の指定は無効とされるため、上記のclone指示子は、下記のclone指示子と同等に扱います。

```
#pragma loop clone var==11:30
```

以下に例を示します。

例1:

コンマで指示した順番に分岐を生成し、ループが複製されます。

```
#pragma loop clone N==10,20
for (i=0;i<N; ++i) {
    a[i] = i;
}
```

→

```
if (N==10) {
    for (i=0;i<10;++i) {
        a[i] = i;
    }
} else if (N==20) {
    for (i=0;i<20;++i) {
        a[i] = i;
    }
} else {
    for (i=0;i<N;++i) {
        a[i] = i;
    }
}
```

例2:

同一ループに対して複数のclone指示子を指定した場合、指定した順番にループが複製されます。

```
#pragma loop clone N==10
#pragma loop clone M==20
for (i=0;i<N;++i) {
    a[i] = M;
}
```

→

```
if (N==10) {
    for (i=0;i<10;++i) {
        a[i] = M;
    }
} else if (M==20) {
    for (i=0;i<N;++i) {
        a[i] = 20;
    }
} else {
    for (i=0;i<N;++i) {
        a[i] = M;
    }
}
```

例3:

多重ループにclone指示子を指定した場合、多重に分岐を生成し、ループが複製されます。

```
#pragma loop clone M==10
for (i=0;i<M;++i) {
#pragma loop clone N==20
    for (j=0;j<N;++j) {
        a[i][j] = 0;
    }
}
```

→

```
if (M==10) {
    for (i=0;i<10;++i) {
        if (N==20) {
            for (j=0;j<20;++j) {
                a[i][j] = 0;
            }
        } else {
            for (j=0;j<N;++j) {
                a[i][j] = 0;
            }
        }
    }
} else {
    for (i=0;i<M;++i) {
        if (N==20) {
            for (j=0;j<20;++j) {
                a[i][j] = 0;
            }
        } else {
            for (j=0;j<N;++j) {
                a[i][j] = 0;
```

```
            }
        }
    }
}
```

eval指示子

eval指示子は、演算評価方法を変更する最適化を行うことを指示します。

以下に例を示します。

例:

```
#pragma loop eval
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i] + c[i] + d[i];
}
```

→

```
for(i = 0; i < n; i++) {
    a[i] = (a[i] + b[i]) + (c[i] + d[i]);
}
```

対象となるループ中で、演算評価方法を変更する最適化を行います。

noeval指示子

noeval指示子は、演算評価方法を変更する最適化を抑止することを指示します。

以下に例を示します。

例:

```
#pragma loop noeval
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i] + c[i] + d[i];
}
```

対象となるループ中で、演算評価方法を変更する最適化を行いません。

eval_concurrent指示子

eval_concurrent指示子は、tree-height-reduction最適化において、命令の並列性を優先することを指示します。

以下に例を示します。

例:

```
#pragma loop eval_concurrent
for(i = 0; i < n; i++) {
    x[i] = a[i] * b[i] + c[i] * d[i] + e[i] * f[i] + g[i] * h[i];
}
```

tree-height-reduction最適化については、“3.3.9 tree-height-reduction最適化”をお読みください。

eval_noconcurrent指示子

eval_noconcurrent指示子は、tree-height-reduction最適化において、命令の並列性を抑え、FMA命令の利用を優先することを指示します。

以下に例を示します。

例:

```
#pragma loop eval_noconcurrent
for(i = 0; i < n; i++) {
    x[i] = a[i] * b[i] + c[i] * d[i] + e[i] * f[i] + g[i] * h[i];
}
```

tree-height-reduction最適化については、“3.3.9 tree-height-reduction最適化”をお読みください。

extract_stride_store指示子

SIMD化対象ループにあるストライドアクセスのストア命令を、スカラ命令に展開します。

以下に例を示します。

例:

```
#pragma loop extract_stride_store
for(int i = 0; i < n; i+=2) {
  a[i] = b[i] + c[i];
}
```

SIMD化可能な配列aに対し、スカラ命令を利用する最適化を行います。

noextract_stride_store指示子

SIMD化対象ループにあるストライドアクセスのストア命令を、スカラ命令に展開しません。

以下に例を示します。

例:

```
#pragma loop noextract_stride_store
for(int i = 0; i < n; i+=2) {
  a[i] = b[i] + c[i];
}
```

SIMD化可能な配列aに対し、スカラ命令を利用せずSIMD化命令を利用します。

fission_point指示子

fission_point指示子は、ループ分割の最適化を行うことを指示します。fission_point指示子に続いて指定する1～6の整数値*n*によって、最内ループから数えたネスト数の多重ループを分割することを指示します。

*n*の指定を省略した場合、最内(1次元)ループのみを分割します。

fission_point指示子は、最内ループに記述した場合のみ有効となります。

以下に例を示します。

例:

```
for(j = 1; j < n; j++) {
    for(i = 1; i < n; i++) {
        a[i][j] = a[i-1][j-1];
#pragma statement fission_point 1
        a[i][j] = a[i][j] + a[i-1][j];
    }
}
```

→

```
for(j = 1; j < n; j++) {
    for(i = 1; i < n; i++) {
        a[i][j] = a[i-1][j-1];
    }
    for(i = 1; i < n; i++) {
        a[i][j] = a[i][j] + a[i-1][j];
    }
}
```

注意

プログラムの論理上は実行されないループをループ分割した場合、実行時に例外が発生するなど、副作用を生じることがあります。

例えば、ループ分割によって分割したループ間でデータの受け渡しが必要な場合、コンパイラが一時的な配列を生成します。その際、一時的な配列の生成に必要な命令が投機実行されることがあります。

以下の例では、配列要素IDX[M][M]のロード命令が投機実行されます。Nが0以下、かつ配列要素IDX[M][M]のMが宣言のサイズより大きい場合、実行時に例外が発生することがあります。

例:

```
for(j = 0; j < N; j++) {
    for(i = 0; i < IDX[M][M]; i++) {
        x = a[i] + b[i];
#pragma statement fission_point 1
        c[i] = x;
    }
}
```

→

```
double *tmp;
tmp =(double *)malloc(IDX[M][M]*sizeof(double)); // 投機実行
for(j = 0; j < N; j++) {
    for(i = 0; i < IDX[M][M]; i++) {
        tmp[i] = a[i] + b[i];
    }
    for(i = 0; i < IDX[M][M]; i++) {
        c[i] = tmp[i];
    }
```

```
}
free(tmp);
```

fp_contract指示子

fp_contract指示子は、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うことを指示します。

なお、Floating-Point Multiply-Add/Subtract演算命令を使用した場合、演算結果に丸め誤差程度の違いが生じることがあります。

以下に例を示します。

例:

```
#pragma loop fp_contract
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i] * c[i];
}
```

対象となるループ中で、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行います。

nofp_contract指示子

nofp_contract指示子は、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行わないことを指示します。

以下に例を示します。

例:

```
#pragma loop nofp_contract
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i] * c[i];
}
```

対象となるループ中で、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行いません。

fp_relaxed指示子

fp_relaxed指示子は、単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数について、逆数近似演算命令を使用した高速な演算を行うことを指示します。

なお、高速な演算を使用した場合、通常の除算、sqrt関数演算に比べて、丸め誤差程度の違いが生じることがあります。また、-NRtrapオプションの指定に関わらず、プログラムの論理上は発生しない浮動小数点例外が発生することがあります。

また、-NRtrapオプションが有効、かつ-KNOSVEオプションまたは-Knosimdオプションのいずれかが有効である場合、sqrt関数に対する逆数近似命令への変換が抑止されます。したがって、-NRnotrapオプションが有効な場合と比べ、実行性能が低下する場合があります。なお、-Knopreexオプションが有効な場合でも、本最適化指示子によって生成された逆数近似演算命令が先行評価されることがあります。また、-Knopreexおよび-NRtrapオプションと本最適化指示子が有効な場合、浮動小数点例外が発生することがあります。

以下に例を示します。

例:

```
#pragma loop fp_relaxed
for(i = 0; i < n; i++) {
    a[i] = sqrt(b[i] / c[i]);
}
```

対象となるループ中で、逆数近似命令を使用します。

nofp_relaxed指示子

nofp_relaxed指示子は、単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数について、逆数近似命令を使用せず通常の除算およびsqrt関数演算を利用することを指示します。

以下に例を示します。

例:

```
#pragma loop nofp_relaxed
for(i = 0; i < n; i++) {
    a[i] = sqrt(b[i] / c[i]);
}
```

対象となるループ中で、通常の除算およびsqrt関数演算を使用します。

fullunroll_pre_simd指示子

SIMD化前のフルアンローリングを促進することを指示します。

指示子に続いて指定する2から100までの整数値で、対象とするループの繰返し数の上限を指定します。上限の指定を省略した場合、コンパイラが自動的に最適な値を決定します。

なお、この指示子は指定した直後のループのみが対象となります。

繰返し数が不明な場合は、最適化を行いません。

以下に例を示します。

例:

```
for(i = 0; i < n; i++) {
#pragma loop fullunroll_pre_simd
  for(j = 0; j < 16; j++) {
    a[i][j] = b[j][i] + c[j][i];
  }
}
```

内側ループに対してSIMD化前のフルアンローリングを適用します。

nofullunroll_pre_simd指示子

SIMD化前のフルアンローリングを抑止することを指示します。

なお、この指示子は指定した直後のループのみが対象となります。

以下に例を示します。

例:

```
for(i = 0; i < n; i++) {
#pragma loop nofullunroll_pre_simd
  for(j = 0; j < 5; j++) {
    a[i][j] = b[j][i] + c[j][i];
  }
}
```

内側ループに対してSIMD化前のフルアンローリングを適用しません。

iterations max=*n1*指示子
iterations avg=*n2*指示子
iterations min=*n3*指示子

指定された値をループの繰返し数とみなして最適化することを指示します。

iterations指示子に続いて、max=*n1*、avg=*n2*、およびmin=*n3*を指定できます。*n1*、*n2*、および*n3*は、0から2147483647の整数値です。max=*n1*、avg=*n2*、およびmin=*n3*は、空白を区切りとして、順不同で複数指定が可能です。ただし、指定した値の大小関係は、*n1*≧*n2*≧*n3*とする必要があります。

指示子に続いてmax=*n1*を指定した場合は、ループの繰返し数の最大値を*n1*とみなして最適化を行います。

指示子に続いてavg=*n2*を指定した場合は、ループの繰返し数の平均値を*n2*とみなして最適化を行います。

指示子に続いてmin=*n3*を指定した場合は、ループの繰返し数の最小値を*n3*とみなして最適化を行います。

本指示子は、翻訳時にループの繰返し数が不明な場合に有効です。

指定した平均値*n2*は、コンパイラが最適化の参考情報として利用しますが、実際のループの繰返し数の平均値と異なる場合でも実行結果は保証されます。

注意

以下の場合は、実行結果は保証されません。

- 実際のループの繰返し数が、指定した最大値*n1*より大きくなる場合
- 実際のループの繰返し数が、指定した最小値*n3*より小さくなる場合

詳細は"3.4.1.3 最適化指示子の注意事項"のiterations max=*n1*指示子およびiterations min=*n3*指示子の項目をお読みください。

本指示子は、global行、procedure行、およびloop行に指定できますが、複数ループがあるプログラムに対して、一律同じ指定が有効なケースばかりではないため、loop行への指定を推奨します。

以下に例を示します。

例1: 繰返し数の平均値を指定

```
#pragma loop iterations avg=32
for(i = 0; i < m; i++) {
    a[i] = b[i] + c[i];
}
```

ループの繰返し数の平均値を32として最適化します。

例2: 繰返し数の最小値および平均値を指定

```
#pragma loop iterations min=1 avg=8
for(i = 0; i < m; i++) {
    a[i] = b[i] + c[i];
}
```

ループの繰返し数の最小値を1、平均値を8として最適化します。

例3: 繰返し数の最大値、平均値、および最小値を指定

```
#pragma loop iterations max=128 avg=16 min=16
for(i = 0; i < m; i++) {
    a[i] = b[i] + c[i];
}
```

ループの繰返し数の最大値を128、平均値を16、最小値を16として最適化します。

loop_blocking指示子

loop_blocking指示子はブロッキングの最適化を行うことを指示します。loop_blocking指示子に続いて指定する2から10000の整数値*n*によって、ブロックサイズを指定します。*n*の指定を省略した場合、コンパイラが自動的に最適な値を決定します。

以下に例を示します。

例:

```
#define MIN(x,y) ((x<y)?x:y)
#pragma loop loop_blocking 80
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        a[i][j] = a[i][j] + b[i][j];
    }
}
```

→

```
for(ii = 0; ii < n; ii += 80) {
    for(jj = 0; jj < n; jj += 80) {
        for(i = ii; i < MIN(n,(ii+80)); i++) {
            for(j = jj; j < MIN(n,(jj+80)); j++) {
                a[i][j] = a[i][j] + b[i][j];
            }
        }
    }
}
```

対象となるループ中で、ブロックサイズ80でブロッキングの最適化を行います。

loop_noblocking指示子

loop_noblocking指示子はブロッキングの最適化を抑止することを指示します。

以下に例を示します。

例:

```
#pragma loop loop_noblocking
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        a[i][j] = a[i][j] + b[i][j];
    }
}
```

対象となるループ中で、ブロッキングの最適化を行いません。

loop_fission_stripmining指示子

自動ループ分割時にストリップマイニングの最適化を行うことを指示します。

指示子に続いて、*n*、"L1"、または"L2"でストリップの長さを指定できます。ストリップの長さの指定を省略した場合、コンパイラが自動的に値を決定します。

指示子に続いて*n*を指定した場合は、ストリップの長さを*n*にします。*n*は、2から100000000までの整数値です。

指示子に続いて"L1"を指定した場合は、キャッシュメモリの利用効率を考慮し、ストリップの長さを1次キャッシュのサイズに合わせます。

指示子に続いて"L2"を指定した場合は、キャッシュメモリの利用効率を考慮し、ストリップの長さを2次キャッシュのサイズに合わせます。

本最適化指示子は、最適化制御行にloop_fission_target指示子が指定されている、かつ-Kloop_fissionオプションおよび-O2オプション以上が有効な場合に意味があります。

ストリップマイニングついては、“3.3.10.1 ストリップマイニング”をお読みください。

以下に例を示します。

例:

```
float a[N], b[N], p[N], q;
#pragma loop loop_fission_target
#pragma loop loop_fission_stripmining 256
for(i = 0; i < N; i++) {
    q = a[i] + b[i];
    ...
    p[i] = p[i] + q;
    ...
}
```

→

```
#define MIN(x,y) ((x<y)?x:y)
float a[N], b[N], p[N], q;
float temparray_q[256];
for(ii = 0; ii < N; ii = ii+256) {
    for(i = ii; i < MIN(N, ii+256); i++) {
        temparray_q[i-ii] = a[i] + b[i];
        ...
    }
    for(i = ii; i < min(N, ii+256); i++) {
        p[i] = p[i] + temparray_q[i-ii];
        ...
    }
}
```

loop_nofission_stripmining指示子

自動ループ分割したループに対して、ストリップマイニングの最適化を抑止することを指示します。

以下に例を示します。

例:

```
float a[N], b[N], p[N], q;
#pragma loop loop_fission_target
#pragma loop loop_nofission_stripmining
for(i = 0; i < N; i++) {
    q = a[i] + b[i];
    ...
    p[i] = p[i] + q;
```

```
    ...
}
```

loop_fission_target指示子

loop_fission_target指示子は、指定されたループに対して、自動ループ分割の最適化を行うことを指示します。

loop_fission_target指示子に続けてclを指定した場合、クラスタリングアルゴリズムでループ分割を行うことを指示します。本機能では、ループ分割に伴う一時的なデータ転送のための作業配列の削減を優先したループ分割を行います。本機能により翻訳時間が増加します。

loop_fission_target指示子に続けてlsを指定した場合、局所探索アルゴリズムでのループ分割を行うことを指示します。本機能では、ソフトウェアパイプライニングの促進を優先したループ分割を行います。本機能により、クラスタリングアルゴリズムに比べてさらに翻訳時間が増加します。

loop_fission_target指示子に続くclおよびlsを省略した場合は、clを指定したものとみなします。

本最適化指示子は、-Kloop_fissionオプションが有効な場合に意味があります。

ループ分割ついては、“3.3.10 ループ分割”をお読みください。

以下に、指定の例を示します。

例1: cl指定

```
#pragma loop loop_fission_target cl
for(i = 0; i < n; i++) {
    s1 = a[i] + b[i];
    s2 = c[i] + d[i];
    ...
    p[i] = s1 + q[i];
    x[i] = s2 + y[i];
    ...
}
```

ループ分割に伴う一時的なデータ転送のための作業配列の削減を優先して、ループを分割します。

例2: ls指定

```
#pragma loop loop_fission_target ls
for(i = 0; i < n; i++) {
    s1 = a[i] + b[i];
    s2 = c[i] + d[i];
    ...
    p[i] = s1 + q[i];
    x[i] = s2 + y[i];
    ...
}
```

ソフトウェアパイプライニングの促進を優先して、ループを分割します。

loop_fission_threshold指示子

loop_fission_threshold指示子は、自動ループ分割における分割後のループの粒度を決める閾値を指示します。指示子に続いて指定する1から100の整数値*n*によって、分割後のループの粒度の閾値を指示します。

本最適化指示子は、最適化制御行にloop_fission_target指示子が指定されている、かつ-Kloop_fissionオプションおよび-O2オプション以上が有効な場合に意味があります。

以下に例を示します。例1に比べて、loop_fission_threshold指示子に小さい値を指定した例2の方が細かい粒度でループが分割され、分割ループ数が増えます。

例1:

```
#pragma loop loop_fission_target
#pragma loop loop_fission_threshold 50
for(i = 0; i < N; i++) {
    a1[i] = a1[i] + b1[i];
    ...
```

→

```
for(i = 0; i < N; i++) {
    a1[i] = a1[i] + b1[i];
    ...
    a2[i] = a2[i] + b2[i];
    ...
```

```
    a2[i] = a2[i] + b2[i];
    ...
    a3[i] = a3[i] + b3[i];
    ...
    a4[i] = a4[i] + b4[i];
    ...
}
```

```
}
for(i = 0; i < N; i++) {
    a3[i] = a3[i] + b3[i];
    ...
    a4[i] = a4[i] + b4[i];
    ...
}
```

例2:

```
#pragma loop loop_fission_target
#pragma loop loop_fission_threshold 20
for(i = 0; i < N; i++) {
    a1[i] = a1[i] + b1[i];
    ...
    a2[i] = a2[i] + b2[i];
    ...
    a3[i] = a3[i] + b3[i];
    ...
    a4[i] = a4[i] + b4[i];
    ...
}
```

→

```
for(i = 0; i < N; i++) {
    a1[i] = a1[i] + b1[i];
    ...
}
for(i = 0; i < N; i++) {
    a2[i] = a2[i] + b2[i];
    ...
}
for(i = 0; i < N; i++) {
    a3[i] = a3[i] + b3[i];
    ...
}
for(i = 0; i < N; i++) {
    a4[i] = a4[i] + b4[i];
    ...
}
```

loop_interchange指示子

loop_interchange指示子は、指定された変数名をループ変数とするループで構成される多重ループを、指示子に続く変数名の並びで指定された順序で内側から入れ換えます。これにより、最適なループの並びとなり実行性能を向上させる可能性が上がります。ただし、入換えが不可能な場合は最適化を適用しません。

以下に入換えの例を示します。

例:

```
#pragma loop loop_interchange j, i
for(j = 0; j < n; j++) {
    for(i = 0; i < m; i++) {
        a[i][j] = b[i][j] * c[j][i];
    }
}
```

→

```
for(i = 0; i < m; i++) {
    for(j = 0; j < n; j++) {
        a[i][j] = b[i][j] * c[j][i];
    }
}
```

左側の例では、多重ループの並びを、内側から、i、jをループ変数とするfor文に入れ換えます。このため入れ換えた結果は、右側の例と等価です。

loop_nointerchange指示子

loop_nointerchange指示子は、多重ループの入換えを実施しないことを指示します。

以下に例を示します。

例:

```
#pragma loop loop_nointerchange
for(j = 0; j < n; j++) {
    for(i = 0; i < m; i++) {
        a[i][j] = b[i][j] * c[j][i];
    }
}
```

上記で、ループの入換えを行いません。

loop_nofusion指示子

loop_nofusion指示子は、指定されたループと隣接したループの融合を抑止することを指示します。

以下に例を示します。

例:

```
#pragma loop loop_nofusion
for(i = 0; i < n;) {
  i++;
}
for(j = 0; j < n;) {
  j++;
}
for(k = 0; k < n;) {
  k++;
}
```

iとjのループ融合は抑止されますが、jとkのループの融合は抑止されません。

loop_part_simd指示子

loop_part_simd指示子は、ループを分割して部分的にSIMD化することを指示します。

以下に例を示します。

例:

```
#pragma loop loop_part_simd
for(i = 1; i < n; i++) {
   a[i] = a[i] - b[i] + log(c[i]);  /* SIMD化可能 */
   d[i] = d[i-1] + a[i];        /* SIMD化不可 */
}
```

→

```
#pragma loop loop_part_simd
for(i = 1; i < n; i++) {
   a[i] = a[i] - b[i] + log(c[i]);  /* SIMD実行 */
}
for(i= 1; i < n; i++) {
   d[i] = d[i-1] + a[i];        /* 非SIMD実行 */
}
```

分割されたiのループのみSIMD化されます。

loop_nopart_simd指示子

loop_nopart_simd指示子は、部分的にSIMD化しないことを指示します。

以下に例を示します。

例:

```
#pragma loop loop_nopart_simd
for(i = 1; i < n; i++) {
   a[i] = a[i] - b[i] + log(c[i]);
   d[i] = d[i-1] + a[i];
}
```

対象となるループは、部分的にSIMD化されません。

loop_perfect_nest指示子

loop_perfect_nest指示子は、不完全多重ループを分割して完全多重ループにすることを指示します。

以下に例を示します。

例:

```
#pragma loop loop_perfect_nest
  for(i = 0; i < n; i++) {    /* 不完全多重ループ */
    a[i] = b[i] + 1;
    for(j = 0; j < n; j++) {
      c[j][i] = d[j][i] + a[i];
```

→

```
  for(i = 0; i < n; i++) {
    a[i] = b[i]+1;
  }
  for(i = 0; i < n; i++) {    /* 完全多重ループ */
    for(j = 0; j < n; j++) {
      c[j][i] = d[j][i]+a[i];
```

```
    }
  }
```

```
    }
  }
```

i,jの不完全多重ループを分割して、完全多重ループにします。

loop_noperfect_nest指示子

loop_noperfect_nest指示子は、不完全多重ループを完全多重ループにしないことを指示します。

以下に例を示します。

例:

```
#pragma loop loop_noperfect_nest
  for(i = 0; i < n; i++) {    /* 不完全多重ループ */
    a[i] = b[i] + 1;
    for(j = 0; j < n; j++) {
      c[j][i] = d[j][i] + a[i];
    }
  }
```

i,jの不完全多重ループを完全多重ループにしません。

loop_versioning指示子

ループバージョニングの最適化を行うことを指示します。

loop_versioning指示子は、最内ループのみ有効となります。

以下に、指定の例を示します。

例:

```
    for (i = 0; i < n; i++) {
#pragma loop loop_versioning
        for (j = 0; j < n; j++) {
            a[j] = a[j+m] + b[i][j];
        }
    }
```

翻訳時は、変数nおよび変数mの値が不明なため、配列aのデータ依存も不明になります。loop_versioning指示子が有効な場合、実行時に変数nおよび変数mの値を用いて、最適化を適用したループまたは最適化を適用しないループのどちらかを選択します。

loop_noversioning指示子

ループバージョニングの最適化を行わないことを指示します。

loop_noversioning指示子は、最内ループのみ有効となります。

以下に、指定の例を示します。

例:

```
    for (i = 0; i < n; i++) {
#pragma loop loop_noversioning
        for (j = 0; j < n; j++) {
            a[j] = a[j+m] + b[i][j];
        }
    }
```

mfunc指示子

mfunc指示子は、マルチ演算関数化を行うことを指示します。

指示子に続いて、1、2または3で、マルチ演算関数化のレベルを指定します。マルチ演算関数化のレベルを省略した場合、1が指定されたものとみなします。マルチ演算関数化のレベルの詳細については、“2.2.2.5 -Kオプション”の-Kmfuncオプションをお読みください。

以下に例を示します。

例:

```
#pragma loop mfunc 1
for(i = 0; i < n; i++) {
    a[i] = log(b[i]);
}
```

対象となるループ中の関数log(b[i])は、マルチ演算関数に変換されます。

nomfunc指示子

nomfunc指示子は、マルチ演算関数化を行わないことを指示します。

以下に例を示します。

例:

```
#pragma loop nomfunc
for(i = 0; i < n; i++) {
    a[i] = log(b[i]);
}
```

対象となるループ中の関数は、マルチ演算関数に変換されません。

noalias指示子

noalias指示子は、異なるポインタ変数が同一の記憶領域を指さないことを指示します。

ポインタ変数が記憶領域のどこを占めるかは実行時に決まりますが、noalias指示子を指定することにより、異なるポインタ変数が同一の記憶領域を指さないことを翻訳時に判断することができます。これにより、ポインタ変数に関する最適化が促進されます。ただし、ポインタ変数の値がループ中で変更される場合、本最適化指示子を指定しても最適化が促進されない場合があります。

以下に例を示します。

例:

```
float *a;
float *b;

#pragma loop noalias
for (i=0; i<10; i++) {
   b[i] = a[i] + 1.0;
}
```

注意

noalias指示子が指定されたループ中に同一の記憶領域を指すポインタ変数が複数含まれる場合、実行結果は保証されません。

norecurrence指示子

norecurrence指示子は、ループ内の演算対象となる配列の要素が、繰返しを跨いで定義引用されないことを本処理系に指示します。

これにより、配列の定義引用順序が不明で最適化できなかったループを最適化の対象にします。

以下の最適化などが該当します。

- ループスライス(自動並列化)
- SIMD化
- ソフトウェアパイプライニング

“norecurrence指示子のないループの例”の場合、配列aの添字式が別の配列要素l[i]であるため、本処理系は配列aがループスライスしても問題がないか判断できません。したがって、この外側のループはループスライスされません。

例1: norecurrence指示子のないループ

```
for(j = 0; j < 1000; j++) {
    for(i = 0; i < 1000; i++) {
```

```
        a[j][l[i]] = a[j][l[i]] + b[j][i];
    }
}
```

もし配列aがループスライスしても問題がないと分かっているのであれば、“norecurrence指示子の使用例”のようにnorecurrenceを使用することにより、この外側のループはループスライスされます。

例2: norecurrence指示子の使用例

```
#pragma loop norecurrence a

for(j = 0; j < 1000; j++) {
    for(i = 0; i < 1000; i++) {
        a[j][l[i]] = a[j][l[i]] + b[j][i];
    }
}
```

並列動作

注意

norecurrence指示子がループの繰返し数に依存する配列に対して誤って指定された場合、実行結果は保証されません。

novrec指示子

novrec指示子は、ループ中に回帰演算となる配列がないことを指示します。この指定によりループ中の配列に対してSIMD命令を使用することになりますが、演算の種類やループ構造によってはSIMD命令が使用されない場合もあります。novrec指示子に続いて指定するオペランドに配列名を指定した場合は、指定された配列だけが回帰演算とならないことを指示します。オペランドを省略した場合は、すべての配列が回帰演算とならないことを指示します。

以下に例を示します。

例:

```
double a[20], b[20];
...
#pragma loop novrec a
for(i = 1; i < 10; i++) {
    a[i] = a[i+n] + 1;
    b[i] = b[i] + 2;
}
```

データ依存が不明な配列aに対して、回帰演算でないことを示します。配列aおよびbの演算についてSIMD命令が利用されます。

preex指示子

preex指示子は、不変式の先行評価の最適化を行うことを指示します。

以下に例を示します。

例:

```
#pragma loop preex
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] / b[k];
    }
}
```

→

```
t = 1 / b[k];
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] * t;
    }
}
```

対象となるループ中で、不変式の先行評価の最適化を行います。

nopreex指示子

nopreex指示子は、不変式の先行評価の最適化を抑止することを指示します。なお、-Kfp_relaxedおよび-NRtrapオプションと本最適化指示子が有効な場合、浮動小数点例外が発生することがあります。

以下に例を示します。

例:

```
#pragma loop nopreex
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] / b[k];
    }
}
```

対象となるループ中で、不変式の先行評価の最適化を行いません。

prefetch指示子

prefetch指示子は、コンパイラの自動プリフェッチ機能を有効にすることを指示します。自動プリフェッチ機能とは、コンパイラが自動的にprefetch命令を挿入するのに最適な位置を判断し、prefetch命令を生成する機能です。

以下に例を示します。

例:

```
#pragma loop prefetch
for(i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

forループに対して自動プリフェッチ機能が有効になります。

noprefetch指示子

noprefetch指示子は、コンパイラの自動プリフェッチ機能を無効にすることを指示します。

以下に例を示します。

例:

```
#pragma loop noprefetch
for(i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

forループに対して自動プリフェッチ機能が無効になります。

prefetch_cache_level指示子

prefetch_cache_level指示子は、どのキャッシュレベルにデータをプリフェッチするかを指示します。

prefetch_cache_level指示子に続けて1を指定した場合、データを1次キャッシュにプリフェッチすることを指示します。1次キャッシュにのみプリフェッチするprefetch命令を使用します。

prefetch_cache_level指示子に続けて2を指定した場合、データを2次キャッシュにプリフェッチすることを指示します。2次キャッシュにのみプリフェッチするprefetch命令を使用します。

prefetch_cache_level指示子に続けてallを指定した場合、prefetch_cache_level 1指示子およびprefetch_cache_level 2指示子の機能を同時に有効にします。2種類のprefetch命令を組み合わせることにより、より効果的なプリフェッチを実現することができます。

以下に例を示します。

例1:

```
#pragma loop prefetch_cache_level 1
for(i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

対象となるループにおいて、1次キャッシュにデータをプリフェッチするprefetch命令を生成します。

例2:

```
#pragma loop prefetch_cache_level 2
for(i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

対象となるループにおいて、2次キャッシュにデータをプリフェッチするprefetch命令を生成します。

例3:

```
#pragma loop prefetch_cache_level all
for(i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

対象となるループにおいて、1次キャッシュおよび2次キャッシュにデータをプリフェッチするprefetch命令を生成します。

prefetch_conditional指示子

if文やswitch文に含まれるブロックの中で使用される配列データに対して、prefetch命令を生成することを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_conditional
for(i = 0; i < n; i++) {
  if (x[i] = y[i]) {
    a[i] = b[i];
  }
}
```

if文に含まれるブロックの中で使用される配列データに対して、prefetch命令を生成します。

prefetch_noconditional指示子

if文やswitch文に含まれるブロックの中で使用される配列データに対して、prefetch命令を生成しないことを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_noconditional
for(i = 0; i < n; i++) {
  if (x[i] = y[i]) {
    a[i] = b[i];
  }
}
```

if文に含まれるブロックの中で使用される配列データに対して、prefetch命令を生成しません。

prefetch_indirect指示子

ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成することを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_indirect
for(i = 0; i < n; i++) {
    a[c[i]] = b[c[i]];
}
```

対象となるループ内の間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成します。

prefetch_noindirect指示子

ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成しないことを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_noindirect
for(i = 0; i < n; i++) {
    a[c[i]] = b[c[i]];
}
```

対象となるループ内の間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成しません。

prefetch_infer指示子

プリフェッチの距離が不明な場合でも連続アクセスとみなしてprefetch命令を生成することを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_infer
for(i = 0; i < n; i++) {
    a[c[i]] = b[c[i]];
}
```

対象となるループ内の配列データに対して連続アクセスとみなしてprefetch命令を生成します。

prefetch_noinfer指示子

プリフェッチの距離が不明な場合は連続アクセスとみなさずにprefetch命令を生成することを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_noinfer
for(i = 0; i < n; i++) {
    a[c[i]] = b[c[i]];
}
```

対象となるループ内の配列データに対して連続アクセスとみなさずにprefetch命令を生成します。

prefetch_iteration指示子

1次キャッシュに対してprefetch命令を生成する際、ループの*n*回転後に引用されるデータを対象とすることを指示します。*n*は1～10000の整数値です。

なお、SIMD化が適用されるループの場合、*n*にはSIMD化されたあとのループの繰返し数を指定してください。

以下に例を示します。

例:

```
#pragma loop prefetch_iteration 50
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループ内で生成される1次キャッシュのprefetch命令は、50回転先のデータをプリフェッチします。

prefetch_iteration_L2指示子

2次キャッシュに対してprefetch命令を生成する際、ループの*n*回転後に引用されるデータを対象とすることを指示します。*n*は1～10000の整数値です。

なお、SIMD化が適用されるループの場合、*n*にはSIMD化されたあとのループの繰返し数を指定してください。

以下に例を示します。

例:

```
#pragma loop prefetch_iteration_L2 50
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループ内で生成される2次キャッシュのprefetch命令は、50回転先のデータをプリフェッチします。

prefetch_line指示子

1次キャッシュに対してprefetch命令を生成する際、*n*キャッシュライン先に該当するデータを対象とすることを指示します。*n*は1～100の整数値です。

以下に例を示します。

例:

```
#pragma loop prefetch_line 5
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループ内で生成される1次キャッシュのprefetch命令は、5キャッシュライン先のデータをプリフェッチします。

prefetch_line_L2指示子

2次キャッシュに対してprefetch命令を生成する際、*n*キャッシュライン先に該当するデータを対象とすることを指示します。*n*は1～100の整数値です。

以下に例を示します。

例:

```
#pragma loop prefetch_line_L2 20
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループ内で生成される2次キャッシュのprefetch命令は、20キャッシュライン先のデータをプリフェッチします。

prefetch_sequential指示子

連続的にアクセスされる配列データに対してprefetch命令を生成することを指示します。

prefetch_sequential指示子に続けてautoを指定した場合、ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用するか、prefetch命令を生成するか否かを、コンパイラが自動的に選択することを指示します。

prefetch_sequential指示子に続けてsoftを指定した場合、ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用せずに、prefetch命令を生成することを指示します。

prefetch_sequential指示子に続くautoおよびsoftを省略した場合は、autoを指定したものとみなします。

以下に例を示します。

例1: auto指定

```
#pragma loop prefetch_sequential auto
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループにおいて、ハードウェアプリフェッチを利用するか、prefetch命令を生成するか否かを、コンパイラが自動的に選択します。

例2: soft指定

```
#pragma loop prefetch_sequential soft
for(i = 0; i < n; i++) {
```

```
    a[i] = b[i];
}
```

対象となるループにおいて、ハードウェアプリフェッチを利用せずにprefetch命令を生成します。

prefetch_nosequential指示子

連続的にアクセスされる配列データに対してprefetch命令を生成しないことを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_nosequential
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループ内の連続的にアクセスされる配列データに対してprefetch命令を生成しません。

prefetch_stride指示子

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、プリフェッチを実施することを指示します。

prefetch_stride指示子に続けてsoftを指定した場合、prefetch命令を生成して、プリフェッチを実施することを指示します。

prefetch_stride指示子に続けてhard_autoを指定した場合、ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施すること指示します。本最適化指示子を指定した場合、キャッシュ上にないデータのみプリフェッチするようストライドプリフェッチャーを設定します。

prefetch_stride指示子に続けてhard_alwaysを指定した場合、ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施すること指示します。本最適化指示子を指定した場合、hard_autoを指定した場合とは異なり、常にプリフェッチをおこなうようストライドプリフェッチャーを設定します。

なお、prefetch_stride指示子に続くsoft、hard_auto、およびhard_alwaysを省略した場合は、softを指定したものとみなします。

ハードウェアストライドプリフェッチャーの利用については、“3.6.3 ハードウェアストライドプリフェッチャーを利用する最適化”をお読みください。

以下に例を示します。

例1: soft指定

```
#pragma loop prefetch_stride soft
for(i = 0; i < n; i=i+k) {
    a[i] = b[i];
}
```

対象となるループにおいて、prefetch命令を生成します。

例2: hard_auto指定

```
#pragma loop prefetch_stride hard_auto
for(i = 0; i < n; i=i+k) {
    a[i] = b[i];
}
```

対象となるループにおいてprefetch命令を生成せず、ハードウェアストライドプリフェッチャーを利用します。キャッシュ上にないデータのみプリフェッチを行うよう、ストライドプリフェッチャーを設定します。

例3: hard_always指定

```
#pragma loop prefetch_stride hard_always
for(i = 0; i < n; i=i+k) {
    a[i] = b[i];
}
```

対象となるループにおいてprefetch命令を生成せず、ハードウェアストライドプリフェッチャーを利用します。常にプリフェッチを行うよう、ストライドプリフェッチャーを設定します。

prefetch_nostride指示子

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成しません。

以下に例を示します。

例:

```
#pragma loop prefetch_nostride
for(i = 0; i < n; i=i+k) {
    a[i] = b[i];
}
```

対象となるループにおいて、prefetch命令を生成しません。

prefetch_strong指示子

1次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_strong
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループで生成される1次キャッシュのprefetch命令はstrong prefetchとなります。

prefetch_nostrong指示子

1次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_nostrong
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループで生成される1次キャッシュのprefetch命令はstrong prefetchとなりません。

prefetch_strong_L2指示子

2次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_strong_L2
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループで生成される2次キャッシュのprefetch命令はstrong prefetchとなります。

prefetch_nostrong_L2指示子

2次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。

以下に例を示します。

例:

```
#pragma loop prefetch_nostrong_L2
for(i = 0; i < n; i++) {
    a[i] = b[i];
}
```

対象となるループで生成される2次キャッシュのprefetch命令はstrong prefetchとなりません。

preload指示子

preload指示子は、ロード命令を投機実行する最適化を行うことを指示します。

以下に例を示します。

例:

```
#pragma loop preload
for(i = 0; i < n; i++) {
    if(m[i] > 0) {
        c[i] = a[i] + b[i];
    }
}
```

対象となるループ中で、ロード命令を投機実行する最適化を行います。

nopreload指示子

nopreload指示子は、ロード命令を投機実行する最適化を行わないことを指示します。

以下に例を示します。

例:

```
#pragma loop nopreload
for(i = 0; i < n; i++) {
    if(m[i] > 0) {
        c[i] = a[i] + b[i];
    }
}
```

対象となるループ中で、ロード命令を投機実行する最適化を行いません。

scache_isolate_way指示子
end_scache_isolate_way指示子
scache_isolate_assign指示子
end_scache_isolate_assign指示子

これらの最適化指示については、“3.5 セクタキャッシュのソフトウェア制御”をお読みください。

simd指示子

SIMD化することを指示します。ただし、演算の種類やループ構造によりSIMD化しない場合もあります。

simd指示子として、simd、simd aligned、またはsimd unalignedを指定できますが、これらはすべて等価です。alignedおよびunaligned パラメタの利用は本製品では非推奨ですが、旧製品とのソースコード互換を保つためにサポートします。

simd指示子は、if-gotoループに対しては指定できません。

以下に例を示します。

例:

```
double a[10], b[10];
...
#pragma loop simd
for(i = 0; i < 10; i++) {
    a[i] = a[i] + b[i];
}
```

ループ内のデータに対してSIMD化を行います。

SIMD化を促進するためにアドレス境界を調整するためのループ変形を行います。

nosimd指示子

nosimd指示子は、SIMD化しないことを指示します。

以下に例を示します。

例:

```
#pragma loop nosimd
for(i = 0; i < 10; i++) {
    a[i] = a[i] + b[i];
}
```

ループ内のデータに対してSIMD化を行いません。

simd_listv指示子

simd_listv指示子は、if文のブロック内の実行文をリストベクトル変換することを指示します。

simd_listv指示子に続けてthenを指定した場合、if文に続く副文をリストベクトル変換します。

simd_listv指示子に続けてelseを指定した場合、else文に続く副文をリストベクトル変換します。

simd_listv指示子に続けてallを指定した場合、または省略した場合、if文およびelse文の両方に続く副文をリストベクトル変換します。

-O2オプション以上が有効、かつ、-Ksimd=2またはsimd指示子により-Ksimd=2相当の機能が動作する場合に意味があります。

リストベクトル変換については、“3.2.7.3 リストベクトル変換”をお読みください。

以下に、指定の例を示します。

例1:

```
#pragma loop simd
for(i = 0; i < n; i++ ) {
#pragma statement simd_listv then
 if(a[i] == 0.0) {
    a[i] = a[i] + b[i];
 } else {
    a[i] = a[i] + c[i];
 }
}
```

ifに続く副文をリストベクトル変換します。

例2:

```
#pragma loop simd
for(i = 0; i < n; i++ ) {
#pragma statement simd_listv else
 if(a[i] == 0.0) {
    a[i] = a[i] + b[i];
 } else {
    a[i] = a[i] + c[i];
 }
}
```

elseに続く副文をリストベクトル変換します。

例3:

```
#pragma loop simd
for(i = 0; i < n; i++ ) {
#pragma statement simd_listv all
 if(a[i] == 0.0) {
    a[i] = a[i] + b[i];
 } else {
    a[i] = a[i] + c[i];
 }
}
```

if文およびelse文の両方に続く副文をリストベクトル変換します。

simd_noredundant_vl指示子

simd_noredundant_vl指示子は、SIMD長の倍数冗長実行によるSIMD化機能を抑止することを指示します。

SIMD長の倍数冗長実行によるSIMD化機能の詳細については、“3.2.7.4 SIMD長の倍数冗長実行によるSIMD化機能”を参照してください。

以下に、指定の例を示します。本最適化指示子は、特定のループだけSIMD長の倍数冗長実行によるSIMD化機能を抑止することを指示するために利用します。

例:

```
void foo(double *a, double *b, double *c) {
  ...
#pragma loop simd_noredundant_vl
    for(i=0; i<m; i++) {
        a[i] = b[i]+c[i];
    }
  ...
}
```

simd_use_multiple_structures指示子

simd_use_multiple_structures指示子は、SIMD化の際、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用することを指示します。

本最適化指示子は、-Ksimd[={1|2|auto}]オプションまたはsimd指示子が有効、かつ、-KSVEオプションが有効である場合に意味があります。

以下に例を示します。

例:

```
#include <complex.h>
#define N 10000
float _Complex a[N];
float _Complex b[N];
float _Complex c[N];
double y[N];
double x[N][4];
...
#pragma loop simd_use_multiple_structures
  for(int i=0; i<N; ++i) {
    a[i] = b[i] * c[i];
  }
...
#pragma loop simd_use_multiple_structures
  for(int i=0; i<N; ++i) {
    y[i] = x[i][0] + x[i][1] + x[i][2] + x[i][3];
  }
```

対象となるループにおいて、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用します。

simd_nouse_multiple_structures指示子

simd_nouse_multiple_structures指示子は、SIMD化の際、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用しないことを指示します。

例:

```
#include <complex.h>
#define N 10000
float _Complex a[N];
float _Complex b[N];
float _Complex c[N];
double y[N];
double x[N][4];
...
```

```
#pragma loop simd_nouse_multiple_structures
  for(int i=0; i<N; ++i) {
    a[i] = b[i] * c[i];
  }
...
#pragma loop simd_nouse_multiple_structures
  for(int i=0; i<N; ++i) {
    y[i] = x[i][0] + x[i][1] + x[i][2] + x[i][3];
  }
```

対象となるループにおいて、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用しません。

striping指示子

ループストライピングの最適化を行うことを指示します。striping指示子に続けてストライプ長(展開数)を指定することができます。ストライプ長は2から100までの値を指定することができます。

ストライプ長の値を省略した場合、2が指定されたものとみなします。

ソースプログラムでループの繰返し数が既知の場合、ストライプ長に繰返し数を超える値を指定しても、コンパイラが自動的に決定した展開数が有効となります。

以下に、指定の例を示します。

例1:

```
#pragma loop striping
for(i = 0; i < n; i++) {
    statement;
}
```

コンパイラが決定したストライプ長で*statement*をループストライピングすることを指定します。

例2:

```
#pragma loop striping 4
for(i = 0; i < n; i++) {
    statement;
}
```

ストライプ長4で*statement*をループストライピングすることを指定します。

例3:

```
#pragma loop striping 8
for(i = 0; i < 4; i++) {
    statement;
}
```

ループの繰返し数4を超えるストライプ長8を指定しているので、ストライプ長はコンパイラが自動的に決定した展開数が有効となります。

例4:

```
void func() {
#pragma procedure striping 8
...
    for(i = 0; i < n; i++) {
        statement;
    }
...
    for(j = 0; j < m; j++) {
        statement;
    }
}
```

関数の先頭に指定した場合、プログラム内のすべてのforループが対象となります。

nostriping指示子

ループストライピングの最適化を抑止することを指示します。

以下に、指定の例を示します。

例:

```
#pragma loop nostriping
for(i = 0; i < n; i++) {
    statement;
}
```

ループストライピングの最適化を行いません。

swp指示子

swp指示子は、ソフトウェアパイプライニングの最適化を行うことを指示します。

本最適化指示子は、-O2オプション以上が有効な場合に意味があります。

以下に例を示します。

例:

```
#pragma loop swp
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループに対してソフトウェアパイプライニングの最適化を行います。

noswp指示子

noswp指示子は、ソフトウェアパイプライニングの最適化を抑止することを指示します。

例:

```
#pragma loop noswp
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループに対してソフトウェアパイプライニングの最適化を行いません。

swp_freg_rate指示子
swp_ireg_rate指示子
swp_preg_rate指示子

以下のレジスタについて、ソフトウェアパイプライニングで使用可能な割合(百分率)を指示します。

- 浮動小数点レジスタおよびSVEのベクトルレジスタ
- 整数レジスタ
- SVEのプレディケートレジスタ

swp_freg_rate指示子を指定した場合は、浮動小数点レジスタおよびSVEのベクトルレジスタについて指示します。

swp_ireg_rate指示子を指定した場合は、整数レジスタについて指示します。

swp_preg_rate指示子を指定した場合は、プレディケートレジスタについて指示します。

指示子に続いて指定する1から1000までの整数値で、ソフトウェアパイプライニングで使用可能とみなすレジスタ数の割合(百分率)を指定します。

本最適化指示子は、-O2オプション以上が有効な場合に意味があります。

以下に例を示します。

例:

```
#pragma loop swp_freg_rate 120
#pragma loop swp_ireg_rate 150
#pragma loop swp_preg_rate 80
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループに対して、レジスタ数に関する条件を調整したソフトウェアパイプライニングの最適化を行います。

swp_policy指示子

ソフトウェアパイプライニングで使用する命令スケジューリングアルゴリズムの選択基準を指示します。

swp_policy指示子に続けて以下のいずれかを指定します。

auto

ループ毎に命令スケジューリングアルゴリズムを自動で選択します。

small

小さなループ(例えば、必要レジスタ数が少ないループ)に適した命令スケジューリングアルゴリズムを使用します。

large

大きなループ(例えば、必要レジスタ数が多いループ)に適した命令スケジューリングアルゴリズムを使用します。

swp_weak指示子

swp_weak指示子は、実行文の重なりを小さくしたソフトウェアパイプライニングの最適化を行うことを指示します。

本最適化指示子は、-O2オプション以上が有効な場合に意味があります。

以下に例を示します。

例:

```
#pragma loop swp_weak
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループに対して、実行文の重なりを小さくしたソフトウェアパイプライニングの最適化を行います。

unroll指示子

unroll指示子は、ループアンローリングの最適化を行うことを指示します。

指示子に続いて2から100までの整数値で、ループ展開数の上限を指定します。上限の指定を省略した場合、コンパイラが自動的に最適な値を決定します。

指示子に続いて"full"を指定すると、指定した直後のfor文を対象として、ソースプログラムの繰返し数だけ実行文を展開します。繰返し数が不明な場合は、最適化を行いません。

なお、この指示子は指定した直後のループのみが対象となります。

以下に例を示します。

例:

```
#pragma loop unroll 8
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループ中の文を展開数8で展開します。

nounroll指示子

nounroll指示子は、ループアンローリングの最適化を抑止することを指示します。

なお、この指示子は指定した直後のループのみが対象となります。

以下に例を示します。

例:

```
#pragma loop nounroll
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループ中の文に対してループアンローリング最適化を行いません。

unroll_and_jam指示子

unroll_and_jam指示子は、アンロールアンドジャムの最適化を行うことを指示します。ただし、以下の場合には最適化を行いません。

－ 最適化の効果が期待できないと判断した場合

－ ループの繰返しを跨いだデータ依存があると判断した場合

指示子に続いて2から100までの整数値で、展開数の上限を指定します。上限の指定を省略した場合、コンパイラが自動的に値を決定します。

なお、本最適化は最内ループには適用されません。

本指示子は、-O2オプション以上が有効な場合に意味があります。

以下に例を示します。

例1:

```
#pragma loop unroll_and_jam 2
for(i = 0; i < 128; i++) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

→

```
for(i = 0; i < 128; i += 2) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        a[i+1][j] = b[i+1][j] + b[i+2][j];
        ...
    }
}
```

変数iのループを対象とし、アンロールアンドジャムの最適化を適用します。

例2:

```
for(i = 0; i < 128; i++) {
#pragma loop unroll_and_jam 2
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

指定したループが最内ループであるため、アンロールアンドジャムの最適化を適用しません。

例3:

```
#pragma loop unroll_and_jam 2
for(i = 0; i < 128; i++) {
    for(j = 0; j < 127; j++) {
        a[i][j] = a[i-1][j+1] + b[i][j];
        ...
    }
}
```

配列aにループの繰返しを跨いだデータ依存があるため、アンロールアンドジャムの最適化を適用しません。

unroll_and_jam_force指示子

unroll_and_jam_force指示子は、ループの繰返しを跨いだデータ依存がないものとみなしてアンロールアンドジャムの最適化を行うことを指示します。

本最適化指示子を誤って指定した場合、実行結果は保証されません。詳細は、“3.4.1.3 最適化指示子の注意事項”のunroll_and_jam_force指示子の項目をお読みください。指示子に続いて2から100までの整数値で、展開数の上限を指定します。上限の指定を省略した場合、コンパイラが自動的に値を決定します。

なお、この指示子は指定した直後のループのみが対象となります。また、ループが最内ループである場合は、本最適化の対象にはなりません。

本指示子は、-O2オプション以上が有効な場合に意味があります。

以下に例を示します。

例1:

```
#pragma loop unroll_and_jam_force 2
for(i = 0; i < 128; i++) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

→

```
for(i = 0; i < 128; i+=2) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        a[i+1][j] = b[i+1][j] + b[i+2][j];
        ...
    }
}
```

変数iのループを対象とし、アンロールアンドジャムの最適化を適用します。

例2:

```
for(i = 0; i < 128; i++) {
#pragma loop unroll_and_jam_force 2
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

指定したループが最内ループであるため、アンロールアンドジャムの最適化を適用しません。

nounroll_and_jam指示子

nounroll_and_jam指示子は、アンロールアンドジャムの最適化を行わないことを指示します。

以下に例を示します。

例:

```
#pragma loop nounroll_and_jam
for(i = 0; i < 128; i++) {
    for(j = 0; j < 128; j++) {
        a[i][j] = b[i][j] + b[i+1][j];
        ...
    }
}
```

アンロールアンドジャムの最適化を適用しません。

unswitching指示子

指定されたif文をループアンスイッチングの対象とすることを指示します。本最適化制御行はループ内で不変なif文の直前に指定してください。上記以外の箇所に記述した場合は指定が無効になります。

以下に指定の例を示します。

例1:

```
void foo(double *a, double *b, double *c, int x, int n) {
  int i;
  for (i=0; i<n; i++) {
#pragma statement unswitching
    if (x == 0) {
      a[i] = b[i];
```

```
    } else {
      a[i] = c[i];
    }
  }
}
```

if文をループアンスイッチングします。

なお、例2のように本最適化制御行が指定されたif文が含まれるループ内で、本最適化制御行が指定されていないif文はループアンスイッチングの対象となりません。

例2:

```
void foo(double *a, double *b, double *c, double *d, int x, int n) {
  int i;
  for (i=0; i<n; i++) {
    if (x == 0) {
      a[i] = b[i];
#pragma statement unswitching
    } else if (x == 1) {
      a[i] = c[i];
    } else {
      a[i] = d[i];
    }
  }
}
```

最適化制御行が指定されたif文のみをループアンスイッチングの対象とします。

なお、ループアンスイッチングの対象となるループに多くの実行文が含まれる場合、翻訳メモリや翻訳時間が大幅に増加する場合があります。

zfill指示子

ループ内で書き込みのみを行う配列データについて、データをメモリからロードすることなく、キャッシュ上に書き込み用の領域を確保する命令を使い、書き込み動作を高速にする最適化を行うことを指示します。zfill指示子に続けて指定する1から100までの整数値*N*によって、256バイトを1ブロックとする単位で*N*ブロック分先のデータをzfill最適化の対象とします。*N*の値が省略された場合、コンパイラが自動的に値を決定します。

zfillについては、“3.3.5 zfill”をお読みください。

以下に、指定の例を示します。

例1:

```
#pragma loop zfill
for(i = 0; i < n; i++) {
    ...
}
```

何ブロック先のデータをzfillの最適化の対象にするかはコンパイラが自動で判断します。

例2:

```
#pragma loop zfill 1
for(i = 0; i < n; i++) {
    ...
}
```

1ブロック先のデータをzfillの最適化の対象にすることを指示します。

nozfill指示子

zfillの最適化を実施しないことを指示します。

以下に、指定の例を示します。

例:

```
#pragma loop nozfill
for(i = 0; i < n; i++) {
    ...
}
```

データはzfillの最適化の対象となりません。

3.4.1.3 最適化指示子の注意事項

clone指示子

clone最適化は、生成された条件節内のループにおいて、指定した変数の値が不変であるものとした最適化を行います。そのため、対象となるループ内で変数が更新される場合、実行結果は保証されません。

例1:

ループ内で変数が更新される場合、結果が保証されないため指定しないでください。

```
int *M = &N;
#pragma loop clone N==10
for(i=0;i<32;++i) {
    *M = 5;    ← clone指示子に指定した変数Nの値を更新するため、実行結果は保証されません。
    a[i] = N;
}
```

例2:

ループ内で変数が更新される可能性がある場合、結果が保証されないため指定しないでください。

```
#pragma loop clone N==10
for(i=0;i<32;++i) {
     sub(&N); ←clone指示子に指定した変数Nの値を更新する可能性があるため、実行結果は保証されません。
}
```

novrec指示子

novrec指示子を指定すると、ループ内に回帰演算の対象となる配列がないとみなして、SIMD命令を利用します。

しかし、ループ内に回帰演算の対象となる配列がある場合、SIMD命令を利用しますが、実行結果は保証されません。

以下に例を示します。

例:

```
#pragma loop novrec
for(i = 0; i < 100; i++) {
    a[i] = a[i + m] + ...;    ← 配列aは回帰演算の対象になっているため、実行結果は保証されません。
}
```

ループ内に回帰演算の対象となる配列がある場合、SIMD命令を利用しますが、実行結果は保証されません。

unroll_and_jam_force指示子

unroll_and_jam_force指示子は、ループの繰返しを跨いだデータ依存がないものとみなしてアンロールアンドジャムを適用します。本最適化指示子を誤って指定した場合、実行結果は保証されません。

例:

```
#pragma loop unroll_and_jam_force 2
for (i=1; i<128; i++) {
    for (j=0; j<127; j++) {
        a[i][j] = a[i-1][j+1] + b[i][j];
        ...
    }
}
```

配列aにループの繰返しを跨いだデータ依存があるため、本最適化指示子を指定しないでください。

iterations max=*n1*指示子
iterations min=*n3*指示子

iterations max=*n1*指示子およびiterations min=*n3*指示子は、翻訳時にループの繰返し数が不明な場合に有効であり、ループの繰返し数の最大値を*n1*、最小値を*n3*とみなして最適化します。*n1*、*n3*を誤って指定した場合、実行結果は保証されません。

例:

```
#pragma loop iterations max=100 min=1
for(i = 1; i < m; i++) {  // 実際のループの繰返し数は最大値が1000、もしくは最小値が0
    a[i] = a[i] + b[i];
}
```

実際のループの繰返し数が最大値100を超える、もしくは最小値が0となる場合、実行結果は保証されません。

3.5 セクタキャッシュのソフトウェア制御

3.5.1 セクタキャッシュの利用について

セクタキャッシュは、再利用性のあるデータが再利用性のないデータによってキャッシュから追い出されることを防ぐことができるキャッシュ機構です。特に、スレッド並列時に複数コアが共有の2次キャッシュをアクセスする場合、キャッシュから追い出される現象が起こりやすくなります。セクタキャッシュは、再利用性のあるデータと再利用性のないデータをセクタごとに住み分けることができます。セクタキャッシュをソフトウェア制御する手段としては、環境変数または最適化制御行があります。環境変数では各セクタの最大ウェイ数が指定可能で、最適化指示行ではセクタ1の最大ウェイ数のみが指定可能です。

ただし、セクタキャッシュは、追い出しを防ぎたい配列のサイズを考慮せずに最大ウェイ数を指定しても性能は向上しません。キャッシュの利用効率が落ち、性能が低下する場合があります。また、LRU(Least Recently Used)によるキャッシュ制御は動作するため、セクタキャッシュのソフトウェア制御を指定しても必ず高速化できるわけではありません。セクタキャッシュで性能向上するための1つの有効な手段としては、ハードウェアモニタ情報を採取後、2次キャッシュミスが発生していることを確認した上で、再利用性のある配列サイズを把握し、その配列が載るだけのウェイ数をセクタ1のウェイ数に指定する方法があります。なお、セクタキャッシュを利用する際は、-Khpctagオプションが有効でなければなりません。

参考

最大ウェイ数

A64FXプロセッサでは、1次キャッシュの最大ウェイ数は4、2次キャッシュの最大ウェイ数は16です。

3.5.2 セクタキャッシュをソフトウェア制御する方法

本処理系には、セクタキャッシュをソフトウェア制御する手段として、以下の2つの方法があります。

- 最適化制御行による指示
- 環境変数および最適化制御行による指示

3.5.2.1 最適化制御行によるソフトウェア制御

セクタキャッシュをソフトウェア制御するための最適化指示子には、以下があります。

scache_isolate_way L2=*n1* [L1=*n2*]
および
end_scache_isolate_way

scache_isolate_way指示子は、キャッシュのセクタ1の最大ウェイ数を指示します。

scache_isolate_way指示子の引数には、キャッシュのセクタ1の最大ウェイ数を指示します。L2=*n1*で2次キャッシュにおけるセクタ1の最大ウェイ数を、L1=*n2*で1次キャッシュにおけるセクタ1の最大ウェイ数をそれぞれ指示します。L1=*n2*の指定は省略可能です。その場合、2次キャッシュのセクタ1の最大ウェイ数のみを制御します。

*n1*および*n2*に指定できる範囲は以下のとおりです。

－ 0 ≦ *n1* ≦ “2次キャッシュの最大ウェイ数”

－ 0 ≦ *n2* ≦ “1次キャッシュの最大ウェイ数”

関数内へ指示する場合は、procedure行としてscache_isolate_way指示子を記述します。指示子の有効範囲は関数内となります。

関数の一部に対して指示する場合は、指示する範囲をstatement行としてscache_isolate_way指示子とend_scache_isolate_way指示子で指定します。

なお、上記の範囲指定の指示を入れ子に記述することはできません。ただし、関数内への指示がある関数の一部に範囲指定の指示を記述することは可能です。

scache_isolate_assign *array1*[,*array2*]...
および
end_scache_isolate_assign

scache_isolate_assign指示子は、キャッシュのセクタ1に載せる配列または配列を指すポインタを指定します。ただし、算術型の配列を指定した場合のみ有効となります。

関数内に指示する場合は、procedure行としてscache_isolate_assign指示子を記述します。指示子の有効範囲は関数内となります。

関数の一部に対して指示する場合は、指示する範囲をstatement行としてscache_isolate_assign指示子とend_scache_isolate_assign指示子で指定します。

なお、上記の範囲指定の指示を入れ子に記述することはできません。ただし、関数内への指示がある関数の一部に範囲指定の指示を記述することは可能です。

 例

最適化制御行によるセクタキャッシュのソフトウェア制御

```
/* 10ウェイに収まるサイズの配列aをL2キャッシュで再利用 */
#pragma statement scache_isolate_way L2=10
#pragma statement scache_isolate_assign a
for(int j = 0; j < n; j++) {
#pragma omp parallel for
    for(int i = 0; i < m; i++) {
        a[i] = a[i] + b[j][i];
    }
}
#pragma statement end_scache_isolate_assign
#pragma statement end_scache_isolate_way
```

3.5.2.2 環境変数および最適化制御行によるソフトウェア制御

各セクタの最大ウェイ数の初期値は、以下の環境変数でも指定することができます。環境変数を指定することにより、オブジェクトプログラム起動時の最大ウェイ数を指定することができます。

FLIB_SCCR_CNTL

セクタキャッシュを利用するか否かを指定する環境変数です。環境変数に設定できる値とその意味は以下のとおりです。

値	説明
TRUE	セクタキャッシュを利用します。 デフォルトです。
FALSE	セクタキャッシュを利用しません。

FLIB_L1_SCCR_CNTL

NUMAノードを1プロセスで占有していない環境でプログラムを実行する場合、2次キャッシュでセクタキャッシュは利用できません。2次キャッシュでセクタキャッシュを利用できない時に、1次キャッシュでセクタキャッシュを利用するか否かを指定する環境変数です。FLIB_SCCR_CNTLがTRUEの時のみ有効です。

環境変数に指定できる値とその意味は以下のとおりです。

値	説明
TRUE	2次キャッシュでセクタキャッシュを利用できない場合、1次キャッシュでセクタキャッシュを利用します。 デフォルトです。
FALSE	2次キャッシュでセクタキャッシュを利用できない場合に、1次キャッシュでセクタキャッシュを利用しません。 以下の実行時メッセージを出力し、セクタキャッシュの制御を無効化して、プログラムの実行を継続します。 `jwe1047i-w A sector cache couldn't be used.`

FLIB_L2_SECTOR_NWAYS_INIT

2次キャッシュのセクタの最大ウェイ数の初期値を指定する環境変数です。FLIB_SCCR_CNTLがTRUEの時のみ有効です。

1つ目のセクタ(セクタ0)の最大ウェイ数*n0*と2つ目のセクタ(セクタ1)の最大ウェイ数*n1*を以下の形式で指定します。

```
n0,n1
```

*n0*と*n1*に指定できる範囲は以下のとおりです。

－ 0 ≦ *n0* ≦ “2次キャッシュの最大ウェイ数”

－ 0 ≦ *n1* ≦ “2次キャッシュの最大ウェイ数”

また、競合を防ぐため

－ *n0* + *n1* = “2次キャッシュの最大ウェイ数”

を満たすことを推奨します。

例

環境変数および最適化制御行によるセクタキャッシュのソフトウェア制御

以下の例の場合、オブジェクトプログラムの起動前に、環境変数FLIB_L2_SECTOR_NWAYS_INITの値として、2,10を設定することで、セクタ1の最大ウェイ数として10を採用し、配列aをセクタ1に載せるソフトウェア制御が可能となります。

1. オブジェクトプログラムの起動前に、環境変数を設定(bashの場合)

```
FLIB_SCCR_CNTL=TRUE
export FLIB_SCCR_CNTL
FLIB_L2_SECTOR_NWAYS_INIT=2,10
export FLIB_L2_SECTOR_NWAYS_INIT
```

2. セクタ1に載せる配列を、最適化制御行により指定

```
/* 配列aにセクタ1を割り当て */
#pragma statement scache_isolate_assign a
for(int j = 0; j < n; j++) {
#pragma omp parallel for
    for(int i = 0; i < m; i++) {
        a[i] = a[i] + b[j][i];
    }
}
#pragma statement end_scache_isolate_assign
```

3.5.2.3 例外的な指定に対する動作について

scache_isolate_way指示子と環境変数FLIB_L2_SECTOR_NWAYS_INITに上限を超えた値を指定した場合、上限値を指定したものとみなされます。0より小さい値を設定した場合、その最適化制御行を無視します。なお、2バイト符号付き整数型の範囲外の値を指定した場合、動作は保証されません。

3.6 注意事項

3.6.1 浮動小数点演算に対する最適化の副作用

最適化オプションや最適化指示子で浮動小数点演算を最適化した場合、副作用を生じる可能性があります。

ここでは、その副作用(主に計算誤差)について説明します。

本処理系は、基本的にはIEEE 754に準拠したオブジェクトを作成しますが、“表3.4 浮動小数点演算に対する最適化の副作用”の計算誤差を伴う最適化により、数値演算の取り扱い方法がIEEE 754に準拠しなくなる場合があります。

各オプションの詳細は“2.2.2.5 -Kオプション”を、各最適化指示子の詳細は“3.4.1.2 最適化指示子の種類”および“4.2.6.3 自動並列化用の最適化指示子”を参照ください。

表3.4 浮動小数点演算に対する最適化の副作用

最適化オプション	最適化指示子	副作用の内容
-Keval	eval	演算の評価順序を変更する最適化を行うので、計算誤差が生じることがあります。 例 `x*y + x*z → x*(y+z)` -Kevalオプションの指定に伴い、下記オプションが有効になります。それぞれの副作用も参照してください。 ・ -Kfsimpleオプション ・ -Kreductionオプション (-Kparallelオプション有効時) ・ -Ksimd_reduction_productオプション (-Ksimd[={1|2|auto}]オプション有効時)
-Kfsimple	—	浮動小数点演算の単純化の最適化を行うので、数値演算の取り扱い方法が、下例のように、厳密にはIEEE 754に準拠しなくなります。 例 `x*0.0 → 0.0` 上記の計算において、xがNaNである場合でも、最適化はx*0.0を0.0に置き換えます。
-Kfp_contract	fp_contract	Floating-Point Multiply-Add/Subtract演算命令を利用する最適化を行うので、丸め誤差程度の計算誤差が生じることがあります。
-Kfp_relaxed	fp_relaxed	単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数に対し、逆数近似演算命令を利用する最適化を行うので、以下のような副作用が生じる可能性があります。 ・ 実行結果に丸め誤差程度の違いが生じる。 ・ -Kfzオプションの指定に関係なく、引数および戻り値に現れる非正規化数を0で置き換える。 ・ 引数および戻り値に現れる負の0を正の0で置き換える。 ・ 引数または戻り値がNaN、±Inf、正規化数の最大値に近い値、正規化数の最小値に近い値のいずれかの場合、IEEE 754に準拠しない動作となる。
-Kfz	—	flush-to-zeroモードを使用するため、アンダーフローによって非正規化数となる演算結果やソースオペランドが同符号の0.0に変わります。
-Kfast_matmul	—	行列積のループを高速なライブラリ呼び出しに変換する最適化を行うので、計算誤差が生じることがあります。

最適化オプション	最適化指示子	副作用の内容
-Kilfunc	―	数学関数のインライン展開で、逆数近似演算命令や三角関数補助命令などを利用したアルゴリズムを用いるため、-Kfp_relaxedと同様の副作用が生じる可能性があります。さらに、-Knofp_contractオプションや-Knofp_relaxedオプションの指定のある・なし、指定順序に関係なく、逆数近似演算命令およびFloating-Point Multiply-Add/Subtract命令を使用するようになります。
-Kmfunc[={1|2|3}]	mfunc [*level*]	関数のマルチ演算関数への変換において、異なったアルゴリズムや逆数近似演算命令、三角関数補助命令などを利用するため、-Kfp_relaxedや-Kilfuncと同様の副作用が生じる可能性があります。 その他の副作用については、“3.3.3 マルチ演算関数”も参照してください。
-Kreduction	reduction	自動並列化のリダクション最適化を行うため、演算の評価順序が変更されることで、計算誤差が生じることがあります。
-Ksimd_reduction_product	―	乗算のリダクション演算をSIMD化するため、演算の評価順序が変更されることで、計算誤差が生じることがあります。
-Kfast	―	-Kevalオプションおよび-Kilfuncオプションなどを誘導するため、計算誤差が生じることがあります。
-Kvisimpact	―	-Kfastオプションを誘導するため、計算誤差が生じることがあります。

また、本処理系は、-O1オプション以上が有効な場合に、プログラムの論理上は実行されないはずの浮動小数点演算命令を実行する最適化を行い、プログラムの論理上は発生しない浮動小数点例外を発生させることがあります。その場合では、例えば以下の現象が発生することがあります。

- 標準ライブラリ関数に含まれるfetestexcept関数を用いて浮動小数点状態フラグにアクセスしたときに、プログラムの論理上はセットされないフラグがセットされている。
- GNU Cライブラリに含まれるfeenableexcept関数を用いて浮動小数点例外のトラップを有効にしたときに、プログラムの論理上は発生しないSIGFPEシグナルが生成される。

ただし、翻訳時オプション-NRtrapを指定することで回避できる場合があります。

参考

自動並列/OpenMPによるスレッド並列時に浮動小数点演算に対する計算誤差を抑えたい場合

以下に示す、スレッド数の変化による計算誤差が発生しない実行可能プログラムを作成することを指示する翻訳時オプションを指定してください。ただし、実行性能が低下する可能性があります。

― -Kparallel_fp_precisionオプション (-Kparallelオプションまたは-Kopenmpオプション有効時)

― -Kopenmp_ordered_reductionオプション(-Kopenmpオプション有効時)

各オプションの詳細は、“2.2.2.5 -Kオプション”を参照してください。

3.6.2 SVEのベクトルレジスタのサイズを指定する場合の注意事項

-Ksimd_reg_size={128|256|512}オプションは、SVEのベクトルレジスタのサイズをビット単位で指定します。本オプションを指定した場合、翻訳時にベクトルレジスタのサイズを指定された固定値とみなして最適化を実施します。そのため、生成した実行可能プログラムは、指定したサイズのベクトルレジスタのSVEを実装しているCPUアーキテクチャにおいてのみ正常に動作します。

-Ksimd_reg_size={128|256|512}オプションで指定したベクトルレジスタのサイズと、実行するCPUアーキテクチャのベクトルレジスタのサイズが異なる場合、実行時異常終了する可能性があります。また、実行結果は保証されません。

なお、-Ksimd_reg_size={128|256|512}オプションで指定したベクトルレジスタのサイズが、実行するCPUアーキテクチャのベクトルレジスタのサイズより小さいことが明らかな場合、prctl(2)システムコールなどを利用して有効なベクトルレジスタのサイズを設定する必要があります。

-Ksimd_reg_size=agnosticオプションが有効な場合、実行可能プログラムは、CPUのSVEのベクトルレジスタのサイズに関わらず実行可能です。

SVEのベクトルレジスタのサイズを変更する場合の注意事項は、“C.2.8 SVEのベクトルレジスタのサイズの変更について”も参照してください。

3.6.3 ハードウェアストライドプリフェッチャーを利用する最適化

-Kprefetch_stride=hard_autoオプション、-Kprefetch_stride=hard_alwaysオプション、prefetch_stride hard_auto指示子、およびprefetch_stride hard_always指示子による最適化は、実行時にハードウェアストライドプリフェッチャーを利用します。

ハードウェアストライドプリフェッチャーを利用するためには、システムのプリフェッチ機構の設定変更が許可されている必要があります。許可されていない場合は、これらの最適化の効果は得られません。

設定変更が許可されているかどうかは、システム管理者にお問い合わせください。

3.6.4 誤りがあるプログラムに対する注意事項

以下に示すような誤ったプログラムに対して最適化を行った場合、最適化のレベルにより実行結果が異なることがあります。実行結果が異なる現象が発生した場合は、以下の点を確認してプログラムを修正してください。

- 値が定義されていない変数を引用している
- 許されない形式の値が格納されている変数を引用している
- 配列宣言を超える添字で配列要素を引用している
- C言語の文法に違反した使い方をしている

第4章 並列化機能

本章では、LLVM OpenMPライブラリを使用して、C言語プログラムを並列処理する方法について説明します。

富士通OpenMPライブラリを使用する場合は、“付録K 富士通OpenMPライブラリ”をお読みください。

注意

以下の条件を満たしている場合、本処理系はLLVM OpenMPライブラリを使用します。

- プログラムのリンク時に、-Nlibompオプションが有効である。

参考

並列処理用ライブラリ(LLVM OpenMPライブラリおよび富士通OpenMPライブラリ)と結合可能なオブジェクトファイルの組合せを下表に示します。

	Fortran オブジェクト ファイル	C/C++オブジェクトファイル	
		tradモード (-Nnoclangオプション)	clangモード (-Nclangオプション)
LLVM OpenMPライブラリ (-Nlibompオプション)	結合可	結合可	結合可
富士通OpenMPライブラリ (-Nfjomplibオプション)	結合可	結合可	**結合不可**

4.1 並列処理の概要

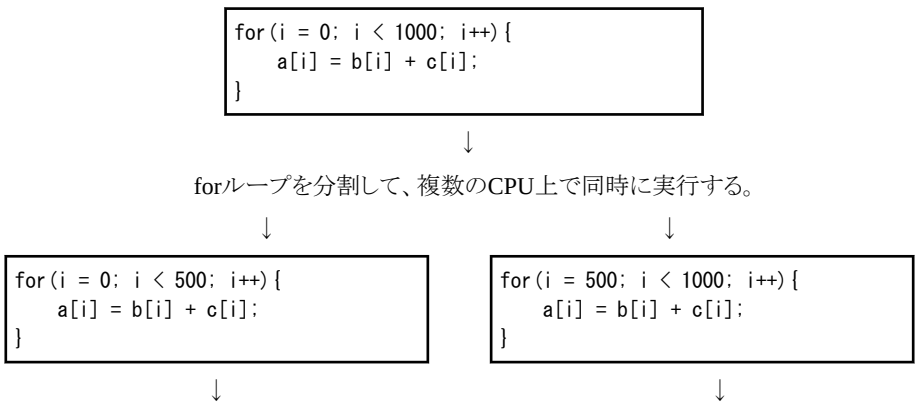
ここでは、C言語プログラムの並列処理の概要と、本処理系の並列機能の特徴について説明しています。

4.1.1 並列処理とは

並列処理という言葉は広い意味に使われていますが、ここで述べる並列処理とは、独立に動作可能な複数のCPUを同時に使って、1つのプログラムを実行することを意味します。ここでは、複数のプログラムを同時に実行するマルチジョブのことを、並列処理とは言わないことにします。

“図4.1 並列処理のイメージ”に、同時に複数のCPUを利用して並列処理する様子を示します。

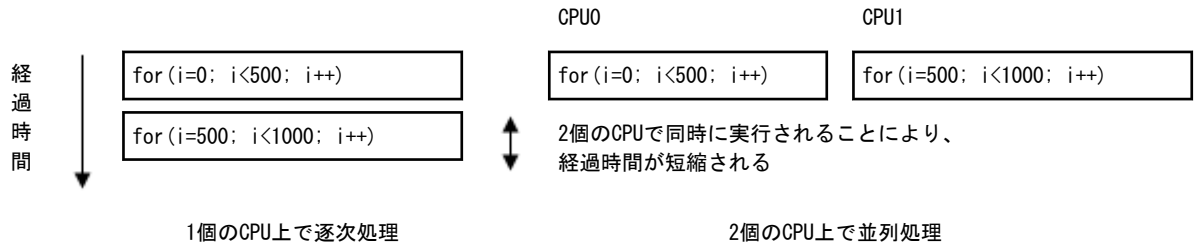
図4.1 並列処理のイメージ



4.1.2 並列処理の効果

並列処理の効果は、複数のCPUを同時に使うことによって、プログラムの経過時間が短縮されることです。例えば、“図4.1 並列処理のイメージ”に示すように、forループを分割して2つのforループを並列実行できたとすると、このforループを実行するのにかかる時間は、理想的には半分になります(“図4.2 並列処理による経過時間の短縮”)。

図4.2 並列処理による経過時間の短縮



しかし、並列処理によってプログラム実行の経過時間の短縮は可能ですが、プログラム実行に要するCPU時間を減らすことはできません。なぜなら、並列処理では複数のCPUを利用しますが、それぞれのCPUの消費するCPU時間の合計は、プログラムを逐次的に実行した場合のCPU時間と同等か、または、並列処理に必要なオーバヘッドの分だけ増加するためです。

並列処理プログラムの性能は、一般に、実行に要したCPU時間の短縮ではなく、経過時間の短縮で評価することに注意してください。逐次的に動作していたプログラムを並列処理によって実行した場合、並列処理のためのオーバヘッドによってプロセッサ全体で消費するCPU時間の合計は、逐次処理時のCPU時間よりも大きくなります。

4.1.3 並列処理で効果を得るための条件

並列処理で経過時間を短縮するためには、複数のCPUを同時に使える計算機環境が必要です。本処理系を使用して作成した並列処理プログラムは、単一のCPUしかもたないハードウェアでも実行可能ですが、この場合には経過時間の短縮は望めません。また、複数のCPUをもつハードウェア上でも、他のジョブが動いていて計算機全体としてCPU時間が不足しているような状況下では、経過時間の短縮は難しくなります。これは、並列処理プログラムの実行のために、同時に複数のCPUが割り当てられる機会が少なくなるためです。

つまり、並列処理で効果を得るためには、複数のCPUをもつハードウェア上で、かつ、システム全体としてCPU処理能力に余裕のある環境下で、並列処理プログラムを実行する必要があります。

また、並列処理のためのオーバヘッドを相対的に小さくするため、ループの繰返し数またはループ中の実行文数が多いことが必要です。

4.1.4 本処理系の並列機能の特徴

本処理系は、自動並列化機能およびOpenMP仕様に基づいた並列化機能を提供します。

自動並列化は、本処理系が自動的にプログラムを並列処理することです。自動並列化の対象となるのは、本処理系が並列化可能であると認識できるforループ、whileループ、do-whileループ、およびif-gotoループ(以降、単にループと呼びます)に限られますが、プログラムの書き換えなどのユーザー負担を軽減できます。ただし、ループ外からの飛込み、またはループ外への飛出しのあるループは、自動並列化の対象となりません。

自動並列化の機能を利用する場合、逐次実行可能なソースプログラムに、何ら手を加える必要はありません。したがって、プログラムがC言語規格に合致している限り、他の処理系ソースプログラムを移植するのは容易です。

また、本処理系には、自動並列化を促進する最適化制御行が用意されています。最適化制御行は、利用者がコンパイラに自動並列化の参考になる情報を通知し、効率の良いオブジェクト生成を促進するために用いられます。

自動並列化については、“4.2 自動並列化”をお読みください。

OpenMP仕様による並列化については、“4.3 OpenMP仕様による並列化”をお読みください。

4.2 自動並列化

ここでは、本処理系で提供している自動並列化について説明します。

4.2.1 翻訳の方法

自動並列化の機能を使用するには、翻訳コマンドに以下のオプションを指定します。

```
-Kparallel [-Nlibomp]
```

4.2.1.1 自動並列化のための翻訳時オプション

自動並列化機能に関連するオプションを以下に示します。各オプションの詳細については、“2.2 翻訳時オプション”をお読みください。

```
-K{parallel|parallel_strong|visimpact}
[,array_private,dynamic_iteration,instance=N,loop_part_parallel,ocl,optmsg=2,parallel_fp_precision,parallel_iteration=N,
reduction,region_extension] [-Nlibomp]
```

4.2.2 実行の方法

自動並列化されたプログラムを実行させるときは、環境変数OMP_NUM_THREADSで、並列に動作させるスレッドの数を指定することができます。また、環境変数OMP_STACKSIZEで、スレッドごとのスタック領域の大きさを指定することができます。さらに、環境変数OMP_WAIT_POLICYで、スレッドの同期待ちの方法を変更することができます。

その他の実行に関する手続きは、逐次実行のときと同じです。

4.2.2.1 スレッド数

並列に動作するスレッドの数は、以下の優先順位で決定されます。

1. 環境変数OMP_NUM_THREADSの指定値
2. システムで利用できるCPU数

自動並列では以下のOpenMP仕様の環境変数を使用することが可能です。OpenMP仕様の環境変数の詳細については、OpenMP仕様書をお読みください。

環境変数OMP_NUM_THREADS

実行中に使用するスレッドの数を指定します。

4.2.2.2 実行時の領域

環境変数OMP_STACKSIZEで、スレッドごとのスタック領域の大きさを指定することができます。

環境変数OMP_STACKSIZE

環境変数OMP_STACKSIZEで、スレッドごとのスタック領域の大きさをバイト、Kバイト、Mバイト、Gバイト、Tバイト単位で指定することができます。デフォルト値は8Mバイトです。

4.2.2.3 同期待ち処理

環境変数OMP_WAIT_POLICYで、同期待ち処理の動作を指定することができます。

環境変数OMP_WAIT_POLICY

ACTIVE	同期が獲得できるまでスピン待ちを行います。
PASSIVE	スピン待ちを行わず、サスペンド待ちを行います。

デフォルトはPASSIVEです。

経過時間を重視する場合には、ACTIVEを選択してください。CPU時間を重視する場合は、PASSIVEを選択してください。

4.2.2.4 スレッドのCPUバインド

環境変数GOMP_CPU_AFFINITYまたはOMP_PROC_BINDを使用することにより、スレッドの固定のCPUへのバインドを制御できます。GOMP_CPU_AFFINITYはOMP_PROC_BINDよりも優先されます

環境変数GOMP_CPU_AFFINITY

スレッドを指定されたcpuid順にCPUバインドします。

スレッド数が、指定したcpuidの数を超える場合は、先頭から繰り返して使用します。

各cpuidは、コンマ(',')または空白文字(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

```
cpuid1[-cpuid2[:inc]]
```

cpuid1:範囲指定の最初のcpuid (0≦*cpuid1*＜CPU_SETSIZE)
cpuid2:範囲指定の最後のcpuid (0≦*cpuid2*＜CPU_SETSIZE)
inc:増分値 (1≦*inc*＜CPU_SETSIZE)

なお、以下の条件を満たす必要があります。

```
cpuid1≦cpuid2
```

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべて指定した場合と等価になります。

cpuidは、上記のように指定できますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)をお読みください。

注意

cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーを出力しプログラムが終了しますので、設定値を見直してください。実行開始時のプロセスのCPU affinityについては、FLIB_HPCFUNC_INFO=TRUE、またはtasksetやnumactlなどのシステムのコマンドで、確認することができます。環境変数FLIB_HPCFUNC_INFOについては、“付録I 高速化機能の利用”を参照してください。システムのコマンドについては、各manマニュアルを参照してください。

例

環境変数GOMP_CPU_AFFINITYの指定例

- 例1:

```
$ export GOMP_CPU_AFFINITY="12,14,13,15"
```

12,14,13,15の順でスレッドにバインドします。
スレッド数5以上の場合は、先頭から繰り返して使用します。

- 例2:

```
$ export GOMP_CPU_AFFINITY="12-19"
```

12,13,14,15,16,17,18,19の順でスレッドにバインドします。
スレッド数9以上の場合は、先頭から繰り返して使用します。

- 例3:

```
$ export GOMP_CPU_AFFINITY="12-19:2"
```

12,14,16,18の順でスレッドにバインドします。
スレッドが5以上の場合は、先頭から繰り返して使用します。

― 例4:

```
$ export GOMP_CPU_AFFINITY="12-16:2,13,19"
```

12,14,16,13,19の順でスレッドにバインドします。
スレッドが6以上の場合は、先頭から繰り返して使用します。

環境変数OMP_PROC_BIND

スレッドアフィニティを制御します。true、false、またはカンマで区切られたmaster、close、spreadのリストのいずれかを設定します。リストの値は、ネストレベルに対応するparallelリージョンで使用するスレッドアフィニティポリシーを設定します。

デフォルトはcloseです。

falseを設定した場合、スレッドアフィニティは無効になり、parallel構文のproc_bind指示節も無視されます。

false以外を設定した場合、スレッドアフィニティは有効になり、初期スレッドはプレースリストの最初のプレースにバインドされます。

masterを設定した場合、すべてのスレッドはマスタースレッドと同じプレースにバインドされます。

closeを設定した場合、スレッドはマスタースレッドがバインドされているプレースの次から連続したプレースにバインドされます。

spreadを設定した場合、マスタースレッドのプレースパーティションが分割され、スレッドは各サブパーティションの1つのプレースにバインドされます。

trueを指定した場合は、spreadを指定した場合と同様の効果になります。

プレースについては環境変数OMP_PLACESを参照してください。

環境変数OMP_PLACES

スレッドアフィニティを制御するプレースリストを定義します。

プレースとは、スレッドが実行されるプロセッサの順序付けされていない集合です。プレースリストとは、利用可能なプレースの順序付けされたリストです。

環境変数OMP_PLACESには、抽象名または明示的なプレースリストのどちらかを、以下の形式で設定します。

{ *abstract-name*[(*num-places*)] | *place-list* }

abstract-name

抽象名です。抽象名には以下を指定します。デフォルトはcoresです。

抽象名	意味
threads	プレースは、1ハードウェアスレッド単位になります。 本処理系ではシステムのCPU資源の最小単位になりcoresと等価になります。
cores	プレースは、1コア単位になります。 本処理系ではシステムのCPU資源の最小単位になりthreadsと等価になります。
sockets	プレースは、1ソケット単位になります。 本処理系ではNUMAノード単位になります。

num-places

抽象名形式のプレース数です。1以上の正の整数です。

place-list

明示的なプレースリストです。以下の形式で指定します。

{ *place*:*length*[:*stride*] | [!]*place*[,*place-list*] } （注1）

place

“{”*place1*“}” (注2)

place1

{ *cpuid*:*length*[:*stride*] | [!]*cpuid*[,*place1*] }

length

インターバル指定の要素数です。1以上の正の整数です。

stride

インターバル指定の増分値です。1以上の正の整数で、デフォルト値は1です。

cpuid

システムのCPU資源の最小単位の識別番号です。

注1) 排他演算子!は直後の指定を除外することを指示します。

注2) “{”および“}”は選択記号ではありません。半角中括弧を使用して指定することを意味します。

例

環境変数OMP_PLACESの指定例

- 例1:

```
$ export OMP_PLACES="cores"
```

抽象名“cores”のプレースリストを定義しています。

- 例2:

```
$ export OMP_PLACES="cores(4)"
```

抽象名“cores”の4プレースのプレースリストを定義しています。

- 例3:

```
$ export OMP_PLACES="{12,13,14},{15,16,17},{18,19,20},{21,22,23}"
$ export OMP_PLACES="{12:3},{15:3},{18:3},{21:3}"
$ export OMP_PLACES="{12:3}:4:3"
```

上記3つの定義は、すべて等価です。

cpuid 12～14、15～17、18～20、および21～23の同じ4つのプレースを定義しています。

4.2.3 翻訳・実行の例

例1:

```
$ fccpx -Kparallel,reduction,ocl -Nlibomp test1.c
$ ./a.out
```

リダクションの最適化と最適化制御行を有効にして、ソースプログラムを翻訳します。このプログラムが実行時に使用するスレッド数は、実行時環境によって決まります。スレッド数の詳細については、“4.2.2.1 スレッド数”をお読みください。

例2:(shの場合)

```
$ fccpx -Kparallel -Nlibomp test2.c
$ OMP_NUM_THREADS=2
$ export OMP_NUM_THREADS
$ ./a.out
$ OMP_NUM_THREADS=4
```

```
$ export OMP_NUM_THREADS
$ ./a.out
```

環境変数OMP_NUM_THREADSに値2を設定して、2個のスレッドを使用して動作させます。

次に、環境変数OMP_NUM_THREADSに値4を設定して、4個のスレッドを使用して動作させます。

4.2.4 並列化プログラムのチューニング

並列化したプログラムをチューニングする手段として、プロファイラで提供される実行時間のサンプリング機能があります。実行時間の情報により、プログラムのどの文の実行コストが高いかを知ることができます。プログラムを高速に実行させるには、実行コストの高い文が並列に動作するようにプログラミングします。プロファイラについては、“プロファイラ使用手引書”をお読みください。

4.2.5 自動並列化機能の詳細

ここでは、本処理系のもつ自動並列化機能について説明します。

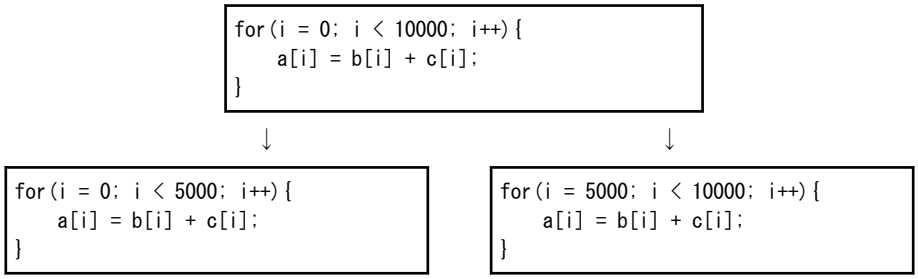
4.2.5.1 自動並列化の対象

自動並列化機能は、C言語ソースプログラム中のループ(多重ループも含みます)とポインタ演算を並列化の対象にします。

4.2.5.2 ループスライスとは

自動並列化機能は、ループを複数に分割します。そして分割されたループを並列に実行することで、経過時間の短縮を図ります。この並列化をループスライスと呼びます。“図4.3 ループスライスのイメージ”に、ループスライスのイメージを示します。

図4.3 ループスライスのイメージ

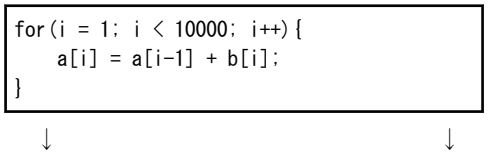


4.2.5.3 コンパイラによる自動ループスライス

本処理系は、ループスライスを行ってもデータの定義引用順序が変わらないようなループを自動並列化の対象として選択して、並列実行の結果が逐次実行の結果と同じになるようにします。

“図4.4 ループスライスができないループの例”に、自動ループスライスの対象にならない例を示します。このforループでは、本来、for文の制御変数iが4999の時に定義された配列要素a[4999]の値を、for文の制御変数iが5000の時に引用しなければなりません。しかし、このforループを単純にループスライスして並列に実行すると、a[4999]を定義する前に引用する可能性があり、実行結果を誤ることになります。

図4.4 ループスライスができないループの例



```
for(i = 1; i < 5000; i++){
    a[i] = a[i-1] + b[i];
}
```

```
for(i = 5000; i < 10000; i++){
    a[i] = a[i-1] + b[i];
}
```

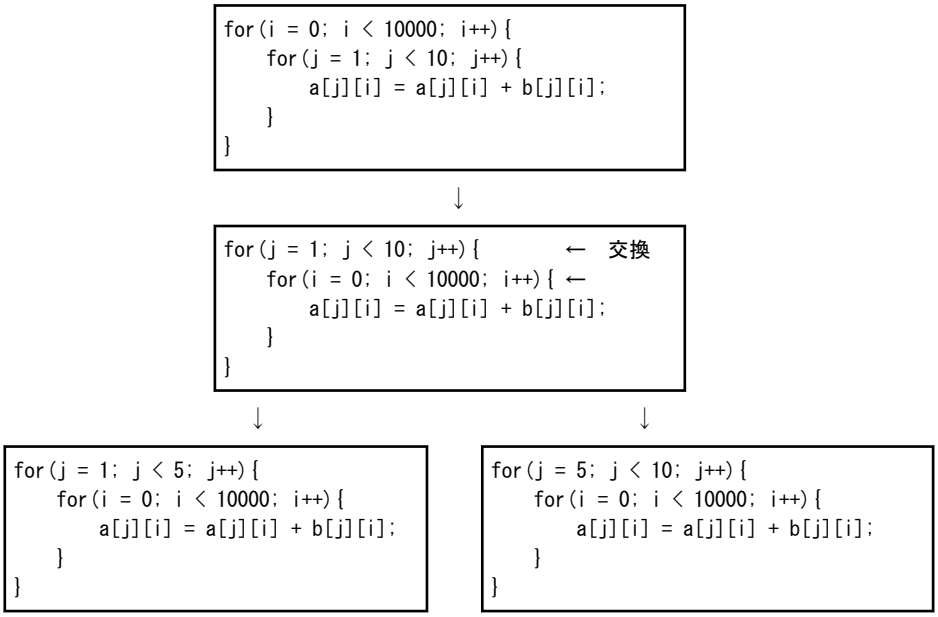
4.2.5.4 ループ交換と自動ループスライス

多重ループに対してループスライスを実施する場合、本処理系は、ループスライスが可能なループを選択し、可能ならばループの交換を行い、できるだけ外側のループをループスライスします。これは、できるだけ並列処理制御の回数を少なくしてオーバヘッドを削減し、実行性能を向上させるためです。

ループ交換は、-Kparallelオプションに加え、-Kevalオプションが指定された時にだけ行われます。

“図4.5 多重ループでのループ交換”に、多重ループにおいてforループが交換されてループスライスされた例を示します。制御変数jについて並列化されますが、jを外側のループにすることで、並列処理制御の回数を少なくすることができます。

図4.5 多重ループでのループ交換



4.2.5.5 ループ分割と自動ループスライス

“図4.6 ループの分割”のループでは、配列aは、ループスライスを行うとデータの定義引用順序が変わるため、ループスライスすることができません。配列bは、データの定義引用順序が変わらないので、ループスライスできます。このような場合、配列aを定義している文と配列bを定義している文を2つのループに分割して、配列bを定義するループの方をループスライスします。

このようなループ分割による部分的な自動並列化は、-Kloop_part_parallelオプションが有効、かつ、ループスライスの効果が十分見込まれるとコンパイラが判断した場合に適用されます。

“図4.6 ループの分割”に、forループが分割されてループスライスされた例を示します。

図4.6 ループの分割

```
for(i = 1; i < 10000; i++){
    a[i] = a[i-1] + c[i];
```

```
    b[i] = b[i] + c[i];
}
```

↓

```
for(i = 1; i < 10000; i++){
    a[i] = a[i-1] + c[i];
}

for(i = 1; i < 10000; i++){
    b[i] = b[i] + c[i];
}
```

↓ 2つ目のループを並列化 ↓

```
for(i = 1; i < 5000; i++){
    b[i] = b[i] + c[i];
}
```

```
for(i = 5000; i < 10000; i++){
    b[i] = b[i] + c[i];
}
```

4.2.5.6 ループ融合と自動ループスライス

“図4.7 ループの融合”では、繰返し数が同じループが2つ連続しています。このような場合、ループを1つに融合することで、ループ制御のオーバヘッドを軽減するとともに、並列処理制御の回数を減らすことができます。

“図4.7 ループの融合”に、ループを融合してループスライスされた例を示します。

図4.7 ループの融合

```
for(i = 0; i < 10000; i++){
    a[i] = b[i] + c[i];
}
for(i = 0; i < 10000; i++){
    d[i] = e[i] + f[i];
}
```

↓

```
for(i = 0; i < 10000; i++){
    a[i] = b[i] + c[i];     ← 融合
    d[i] = e[i] + f[i];     ←
}
```

↓ ↓

```
for(i = 0; i < 5000; i++){
    a[i] = b[i] + c[i];
    d[i] = e[i] + f[i];
}
```

```
for(i = 5000; i < 10000; i++){
    a[i] = b[i] + c[i];
    d[i] = e[i] + f[i];
}
```

4.2.5.7 リダクションによるループスライス

-Kparallelオプションと-Kreductionオプションを同時に指定したときに、自動並列化される場合があります。この並列化によって実行結果に計算誤差を生じることがありますが、加算および乗算などの交換可能な演算の順序を変更してループスライスを行います。

リダクションの対象になるのは、ループ内に以下の演算が含まれる場合です。

- 総和を求める演算がある場合

  例:s = s + a[i];

- 総積を求める演算がある場合

  例：p = p * a[i];

- 内積を求める演算がある場合

  例：p = p + a[i] * b[i];

- 最小値を求める演算がある場合

  例：x = min(x, a[i]);

- 最大値を求める演算がある場合

  例：y = max(y, a[i]);

- ビットOR演算がある場合

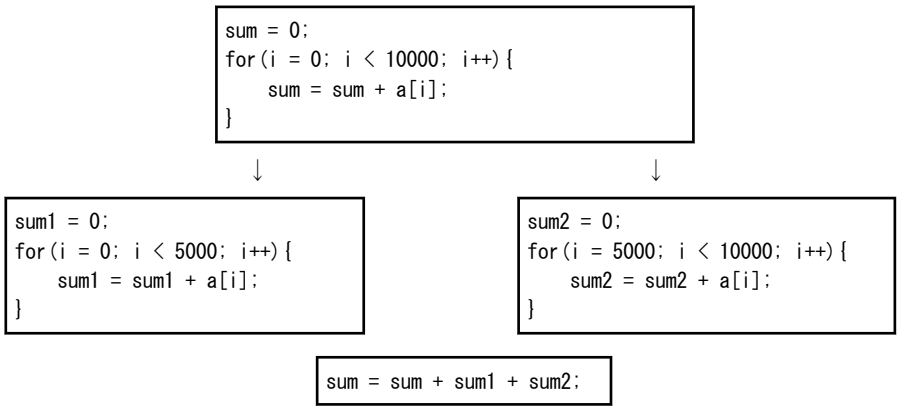
  例：n = n | a[i];

- ビットAND演算がある場合

  例：m = m & a[i];

“図4.8 リダクションによる自動ループスライス”に、リダクションによるループスライスの例を示します。

図4.8 リダクションによる自動ループスライス



4.2.5.8 ループスライスされないループ

以下に示すループは、ループスライスの対象となりません。

- 並列実行しても実行時間が短縮されないと予想される場合
- 関数の引用を含む場合
- ループの形が複雑な場合
- 標準ライブラリ関数を含む場合
- データの定義引用順序が逐次実行のときと変わるおそれがある場合

並列実行しても実行時間が短縮されないと予想される場合

ループの繰返し数が小さい場合や、ループ内の演算数が少ない場合は、並列実行に伴うオーバヘッドのため、ループスライスを行うと、逐次処理した場合よりも性能が低下することがあります。そのため、性能が向上しないと予想されるループについては、本処理系はループスライスを行いません。

“図4.9 繰返し数が小さく、演算数が少ないループ”に、繰返し数が小さく、演算数が少ないループの例を示します。

図4.9 繰返し数が小さく、演算数が少ないループ

```
for(i = 0; i < 10; i++){     ←  繰返し数が小さく、かつ演算数も少ないため、
    a[i] = a[i] + b[i];          ループスライスの対象となりません。
}
```

関数の引用を含む場合

関数の引用を含むループはループスライスの対象となりません。内部ループに、関数の引用が含まれているループも同様です。

ただし、最適化制御行によって、並列化を促進することができます。詳細については、“4.2.6 最適化制御行”をお読みください。また、関数がインライン展開された場合は、並列化の対象になります。

“図4.10 関数の引用を含むループ”に、関数の引用を含むループの例を示します。

図4.10 関数の引用を含むループ

```
for(j = 0; j < 10; j++){
    for(i = 0; i < 10000; i++){          ←  関数呼出しの引用が含まれるため、
        a[j][i] = a[j][i] + b[j][i];         ループスライスの対象となりません。
        func(a);
    }
}
```

ループの形が複雑な場合

以下に示すループは、形が複雑なため、ループスライスの対象となりません。

－ ループの内側から外側へ飛出しがあるループ

“図4.11 飛出しのあるループ”に、飛出しのあるループの例を示します。

図4.11 飛出しのあるループ

```
for(j = 0; j < 10; j++){
    for(i = 0; i < 10000; i++){          ←  forループ外への飛出しがあるため、
        a[j][i] = a[j][i] + b[j][i];         ループスライスの対象となりません。
        if(a[j][i]<0) goto out;
    }
}
out:;
```

標準ライブラリ関数を含む場合

標準ライブラリ関数は、ループスライスの対象になるものとならないものがありますが、診断メッセージによって知ることができます。

データの定義引用順序が逐次実行のときと変わるおそれがある場合

“図4.4 ループスライスができないループの例”で説明したように、データの定義引用順序が逐次実行のときと変わるループはループスライスの対象となりません。

4.2.5.9 自動並列化状況の出力

自動並列化の状況を知りたいときは、-Kparallelオプションおよび-Koptmsg=2オプションを同時に指定してください。翻訳時の診断メッセージに、自動並列化の状況が出力されます。

自動並列化が行われなかった理由も、同様に知ることができます。

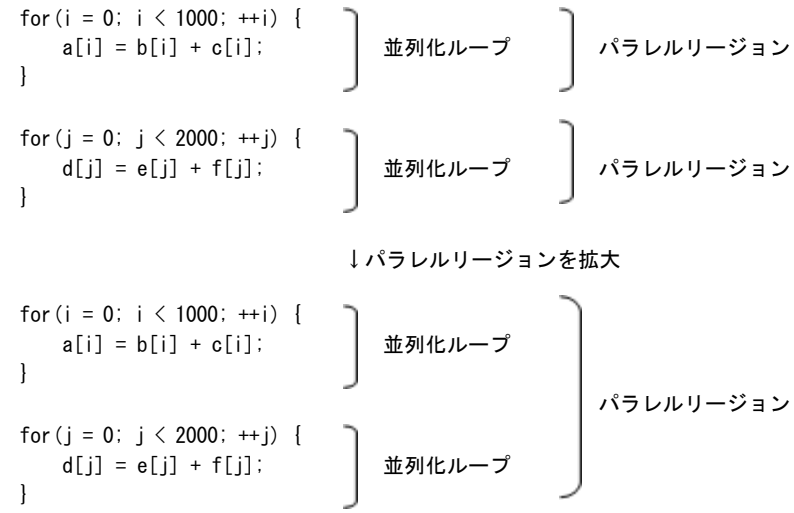
4.2.5.10 パラレルリージョンの拡大

-Kparallelオプションと-Kregion_extensionオプションを同時に指定したときに、パラレルリージョンが拡大されることで並列化オーバヘッドを削減する場合があります。

パラレルリージョンとは、自動並列化のために複数のスレッドを生成してから解放するまでの範囲です。通常、このパラレルリージョンは、“図4.12 パラレルリージョンの拡大の例”のように、1つのパラレルリージョンに対して1つの並列化ループが生成されます。このとき、実行時にパラレルリージョンを生成するためのコストが、並列化オーバヘッドとなります。

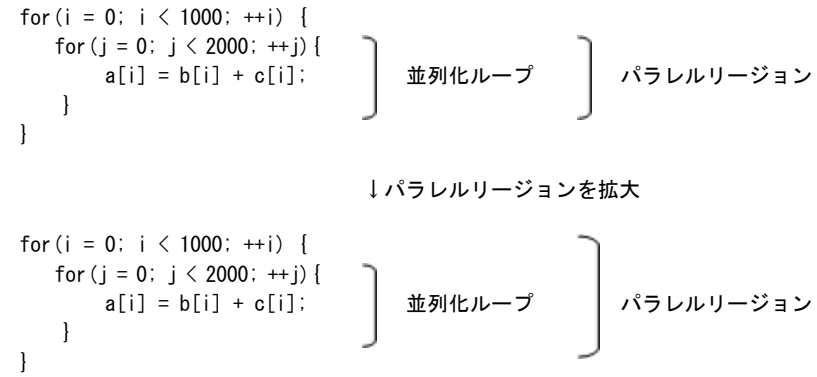
この2つの並列化ループに対して、パラレルリージョンの拡張を行うことにより、1つのパラレルリージョンに対して2つの並列化ループが生成されます。その結果、パラレルリージョンを生成する回数が1回だけになり、並列化オーバヘッドコストを削減することができます。

図4.12 パラレルリージョンの拡大の例



上記以外にも、“図4.13 パラレルリージョンの拡大の例(多重ループ)”のようなループに対してもパラレルリージョンの拡大を行うことにより、並列化オーバヘッドを削減することができます。この場合、パラレルリージョンを生成する回数が1000回だったのに対して、最適化後は1回だけになります。

図4.13 パラレルリージョンの拡大の例(多重ループ)



4.2.5.11 ブロック分割とサイクリック分割

自動並列化ループの分割方法には、ブロック分割とサイクリック分割があります。

ブロック分割とは、ループの繰返し数をスレッド数で分割したブロックを各スレッドに割り当てる方式です。

サイクリック分割とは、ループの繰返し数を任意のサイズで分割したブロックを各スレッドに順番に割り当てる方式です。自動並列化の分割方式のデフォルトは、ブロック分割です。最適化指示子parallel_cyclicが指定された場合、サイクリック分割します。

図4.14 ブロック分割とサイクリック分割

```
for(i = 0; i < 20; i++) {
    for(j = i; j < 20; j++){
        a[i][j] = b[i][j];
    }
}
```

並列化ループ

"図4.14 ブロック分割とサイクリック分割"に対してブロック分割またはサイクリック分割を適用した場合、ループの繰返し数は以下のように各スレッドに割り当てられます。

- ブロック分割の場合

  ループの繰返し数をスレッド数2で割ったブロックを以下のように各スレッドに割り当てます。

```
スレッド0:  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
スレッド1:  {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

- サイクリック分割のブロックサイズが2の場合

  ループの繰返し数を2回転(ブロックサイズ2)で分割したブロックを以下のように各スレッドに順番に割り当てます。

```
スレッド0:  {1,  2},  {5, 6},  {9,  10},  {13, 14},  {17, 18}
スレッド1:  {3,  4},  {7, 8},  {11, 12},  {15, 16},  {19, 20}
```

4.2.6 最適化制御行

本処理系には、自動並列化を促進するために、最適化制御行が用意されています。

自動並列化を促進する最適化制御行を有効にするには、-Kparallelオプションと-Koclオプションを同時に指定してください。

4.2.6.1 最適化指示子の種類

最適化制御行は、指定される最適化指示子の種類によって、機能が異なります。

"表4.1 自動並列化用の最適化指示子一覧"に、最適化指示子の種類と機能を示します。自動並列化用の最適化指示子は、利用者がコンパイラに自動並列化の参考になる情報を通知し、効率の良いオブジェクトプログラムを生成するために用いられます。

表4.1 自動並列化用の最適化指示子一覧

最適化指示子	意味概略	指定できる最適化制御行(注1)		
		global行	procedure行	loop行
array_private	プライベート配列化機能を有効にすることを指示します。	○	○	○
noarray_private	プライベート配列化機能を無効にすることを指示します。	○	○	○
independent [*ext*[,*ext*]...](注2)	関数(*ext*)引用のあるループスライスを指示します。	×	○	○
loop_part_parallel	ループを分割して部分的に自動並列化する機能を有効にすることを指示します。	○	○	○

最適化指示子	意味概略	指定できる最適化制御行(注1)		
		global行	procedure行	loop行
loop_nopart_parallel	ループを分割して部分的に自動並列化する機能を無効にすることを指示します。	○	○	○
serial	自動並列化機能を無効にすることを指示します。	○	○	○
parallel	自動並列化機能を有効にすることを指示します。	○	○	○
parallel_cyclic [*n*]	ブロックサイズ(*n*)でサイクリック分割することを指示します。	×	×	○
parallel_strong	ループの繰返し数が小さい、または演算数が少ないために並列化されていないループを並列化させることを指示します。	○	○	○
reduction	リダクション演算を並列化することを指示します。	○	○	○
noreduction	リダクション演算を並列化しないことを指示します。	○	○	○
temp [*var*[,*var*]...](注3)	ループ内で一時的に使用している変数*var*を指示します。	○	○	○
temp_private *var* [,*var*]...(注3)	変数(*var*)をスレッド内でローカルな領域に割り付けることを指示します。これにより、スレッド間のデータ依存が解消され、自動並列化が促進されます。	×	×	○
first_private *var* [,*var*]...(注3)	変数(*var*)をスレッド内でローカルな領域に割り付けることを指示します。また、変数の初期値の引用があることを指示します。これにより、スレッド間のデータ依存が解消され、自動並列化が促進されます。	×	×	○
last_private *var* [,*var*]...(注3)	変数(*var*)をスレッド内でローカルな領域に割り付けることを指示します。また、ループ終了後に変数の引用があることを指示します。これにより、スレッド間のデータ依存が解消され、自動並列化が促進されます。	×	×	○
*var1 op var2*または *var1 op const*(注4)	変数*var1*と変数*var2*の大小関係または変数*var1*と定数*const*の大小関係を指示します。*op*は比較演算子です。	○	○	○

注1)
○:最適化指示子を最適化制御行に指定できます。
×:最適化指示子を最適化制御行に指定できません。

注2) *ext*は使用する前に宣言してください。

注3) *var*は使用する前に宣言してください。

注4) *var1*、*var2*は使用する前に宣言してください。

4.2.6.2 自動並列化と最適化指示子

自動並列化用の最適化指示子を指定した場合であっても、ループスライスの対象とならないforループについては、最適化指示子の効果は無効になります。ループスライスの対象とならないforループについては、“4.2.5.8 ループスライスされないループ”をお読みください。

4.2.6.3 自動並列化用の最適化指示子

自動並列化用の最適化指示子には、次の種類があります。

array_private指示子

array_private指示子は、ループ内のプライベート化可能な配列を認識し、プライベート化することにより並列化を促進させることを指示します。

“図4.15 array_private指示子の使用例”の場合、-Karray_privateオプションを指定しなくても、array_private指示子を指定することにより、配列aがプライベート化可能であると認識すればプライベート化するため、並列化を促進することができます。

図4.15 array_private指示子の使用例

```
#pragma loop array_private
for(i = 0; i < 100; i++) {
    for(j = 0; j < 1000; j++) {
        a[j] = i + d[j];
        b[i][j] = a[1] + c[j];
    }
}
```

noarray_private指示子

noarray_private指示子は、ループ内のプライベート化可能な配列に対してプライベート化しないことを指示します。

“図4.16 noarray_private指示子の使用例”の場合、noarray_private指示子を指定することにより、配列aがプライベート化可能であってもプライベート化による並列化を抑止することができます。

図4.16 noarray_private指示子の使用例

```
#pragma loop noarray_private
for(i = 0; i < 100; i++) {
    for(j = 0; j < 1000; j++) {
        a[j] = i + d[j];
        b[i][j] = a[1] + c[j];
    }
}
```

independent指示子

independent指示子は、ループ内の関数を引用しても逐次実行のときと動作が変わらないことを指示します。これにより、関数の引用のあるループをループスライスの対象にします。

independent指示子にはループスライスに影響しない関数名を指定できます。また、関数名を省略すると、対象範囲内のすべての関数に有効となります。independent指示子に指定した関数は、-Kparallelオプションを指定して翻訳しなければなりません。

“図4.17 independent指示子のないループの例”の場合、関数funcを引用しているため、本処理系はループがループスライス可能であるか否かを判断できません。

図4.17 independent指示子のないループの例

```
#include <math.h>
main()
{
    int i;
    double a[10000],j;
    double func(double);
    for(i = 0; i < 10000; i++){
        j = 1;
        a[i] = func(j);
    }
    ...
}
double func(double j)
{
```

```
    return sqrt(pow(j, 2) + 3 * j + 6);
}
```

もし関数funcの引用のあるループをループスライスしても、実行結果に影響を与えないと分かっているのであれば、“図4.18 independent指示子の使用例”のようにindependentを使用することにより、このループはループスライスされます。

図4.18 independent指示子の使用例

```
#include <math.h>
main()
{
    int i;
    double a[10000],j;
    double func(double);

#pragma loop independent func
    for(i = 0; i < 10000; i++){
        j = 1;                          ↕ 並列動作
        a[i] = func(j);
    }
    ...

}
double func(double j)
{
    return sqrt(pow(j, 2) + 3 * j + 6);
}
```

注意

independent指示子に、ループスライス不可能な手続きを誤って指定した場合、本処理系は誤ったループスライスを行うことがあります。なお、ループスライス不可能な手続きの例として、以下のようなループがあります。

- 手続き間で依存関係がある手続き

loop_part_parallel指示子

loop_part_parallel指示子は、ループを分割して部分的に自動並列化することを指示します。

“図4.19 loop_part_parallel指示子の使用例”の場合、-Kloop_part_parallelオプションを指定しなくても、loop_part_parallel指示子を指定することにより、ループを分割して部分的に並列化できます。

図4.19 loop_part_parallel指示子の使用例

```
#pragma loop loop_part_parallel
for(i = 1; i < n; i++){
    a[i] = a[i] - b[i] + log(c[i]);     ↕ 並列動作
    d[i] = d[i-1] + a[i];               ↕ 逐次動作
}
```

loop_nopart_parallel指示子

loop_nopart_parallel指示子は、部分的に自動並列化しないことを指示します。

“図4.20 loop_nopart_parallel指示子の使用例”の場合、loop_nopart_parallel指示子を指定することにより、対象とするループは部分的に並列化されません。

図4.20 loop_nopart_parallel指示子の使用例

```
#pragma loop loop_nopart_parallel
for(i = 1; i < n; i++){
    a[i] = a[i] - b[i] + log(c[i]);
    d[i] = d[i-1] + a[i];
}
```

逐次動作

serial指示子

serial指示子は、ループのループスライスを抑止する場合に使用します。

例えば、逐次実行させた方が速いとわかっているループに使用します。

“図4.21 serial指示子のないプログラム例”のプログラムで、ループ2をループスライスしたくない場合、“図4.22 serial指示子の使用例”のようにserial指示子を指定することにより、ループ2のループスライスを止めることができます。

図4.21 serial指示子のないプログラム例

```
for(j = 0; j < 10; j++){
    for(i = 0; i < l; i++){      ←  ループ1
        a1[j][i] = a1[j][i] + b1[j][i];
    }
}
...
for(j = 0; j < 10; j++){
    for(i = 0; i < m; i++){      ←  ループ2
        a2[j][i] = a2[j][i] + b2[j][i];
    }
}
...
for(j = 0; j < 10; j++){
    for(i = 0; i < n; i++){      ←  ループ3
        a3[j][i] = a3[j][i] + b3[j][i];
    }
}
```

並列動作

並列動作

並列動作

図4.22 serial指示子の使用例

```
for(j = 0; j < 10; j++){
    for(i = 0; i < l; i++){    ←ループ1
        a1[j][i] = a1[j][i] + b1[j][i];
    }
}
...
```

並列動作

```
#pragma loop serial
for(j = 0; j < 10; j++){
    for(i = 0; i < m; i++){   ←ループ2          逐次動作
        a2[j][i] = a2[j][i] + b2[j][i];
    }
}
...

for(j = 0; j < 10; j++){
    for(i = 0; i < n; i++){   ←ループ3          並列動作
        a3[j][i] = a3[j][i] + b3[j][i];
    }
}
```

parallel指示子

parallel指示子は、serial指示子の効果を打ち消して、一部のループだけをループスライスの対象とする場合に使用します。

“図4.23 parallel指示子のないプログラム例”で、ループ2のループだけをループスライスの対象にしたい場合、“図4.24 parallel指示子の使用例”のようにserial指示子とparallel指示子を併用することにより、ループ2のループだけをループスライスの対象にすることができます。

図4.23 parallel指示子のないプログラム例

```
void foo(){
  ...
  for(j = 0; j < m1; j++){      ←ループ1
      for(i = 0; i < 100; i++){              並列動作
          a1[j][i] = a1[j][i] + b1[j][i];
      }
  }
  ...

  for(j = 0; j < m2; j++){      ←ループ2
      for(i = 0; i < 100; i++){
          a2[j][i] = a2[j][i] + b2[j][i];    並列動作
      }
  }
  ...

  for(j = 0; j < m3; j++){      ←ループ3
      for(i = 0; i < 100; i++){
          a3[j][i] = a3[j][i] + b3[j][i];    並列動作
      }
  }
  ...
}
```

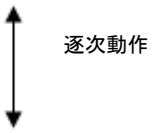
図4.24 parallel指示子の使用例

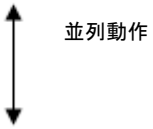
```
#pragma global serial
void foo(){
  ...
```

```
  for(j = 0; j < m1; j++){      ←ループ1
      for(i = 0; i < 100; i++){
          a1[j][i] = a1[j][i] + b1[j][i];
      }
  }
  ...

#pragma loop parallel
  for(j = 0; j < m2; j++){      ←ループ2
      for(i = 0; i < 100; i++){
          a2[j][i] = a2[j][i] + b2[j][i];
      }
  }
  ...

  for(j = 0; j < m3; j++){      ←ループ3
      for(i = 0; i < 100; i++){
          a3[j][i] = a3[j][i] + b3[j][i];
      }
  }
  ...
}
```





---

parallel_cyclic指示子

parallel_cyclic指示子は、コンパイラが自動並列化可能と解析したループに対してサイクリック分割を行うことを指示します。parallel_cyclic指示子に続いて指定する1から10000の整数値*n*によって、ブロックサイズを指定します。*n*の指定を省略した場合、ブロックサイズは1になります。

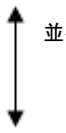
本最適化指示子が適用された場合、並列効果の見積もりは行わずに並列実行します。

サイクリック分割については、“4.2.5.11 ブロック分割とサイクリック分割”をお読みください。

“図4.25 parallel_cyclic指示子の使用例”の場合、ループはブロックサイズを2とするサイクリック分割が行われます。

図4.25 parallel_cyclic指示子の使用例

```
#pragma loop parallel_cyclic 2
for (i = 0; i < 1000; i++) {
    for (j = i; j < 1000; j++) {
        a[i][j] = b[i][j];
    }
}
```



---

parallel_strong指示子

parallel_strong指示子は、ループの繰返し数が小さい、または演算数が少ないために並列化されていないループを並列化させることを指示するために使用します。

“図4.26 parallel_strong指示子の使用例”の場合、ループの繰返し数が小さいことによる性能向上が得られないため並列化が抑止されますが、parallel_strong指示子を指定することにより、並列化することができます。

図4.26 parallel_strong指示子の使用例

```
#pragma loop parallel_strong
for (i = 0; i < 10; i++) {
    a[i] = a[i] + b[i] + c[i];
}
```

並列動作

---

reduction指示子

reduction指示子は、リダクション演算が含まれるループを並列化することを指示します。

“図4.27 reduction指示子の使用例”の場合、-Kreductionオプションを指定しなくても、reduction指示子を指定することにより、リダクション演算を含むループを並列化することができます。

図4.27 reduction指示子の使用例

```
#pragma loop reduction
for(i = 0; i < 5000; i++){
    s = s + a[i];
}
```

並列動作

---

noreduction指示子

noreduction指示子は、リダクション演算が含まれるループを並列化しないことを指示します。

“図4.28 noreduction指示子の使用例”の場合、noreduction指示子を指定することにより、リダクション演算を含むループの並列化を抑止することができます。

図4.28 noreduction指示子の使用例

```
#pragma loop noreduction
for(i = 0; i < 5000; i++){
    s = s + a[i];
}
```

逐次動作

---

temp指示子

temp指示子は、ループ内で引用されている変数がそのループの中で一時的に使用されていることを指示します。

これにより、並列化したループの実行性能を向上させることができます。

“図4.29 temp指示子のないプログラム例”の場合は、変数tがループの中でしか使用されていなくても、tが外部変数であるため、本処理系はtが関数funcの中で参照されていると判断し、tの値が正しくなるようなループスライスを行います。

図4.29 temp指示子のないプログラム例

```
int t;
main()
{
    ...
```

```
    for(j = 0; j < 50; j++){                    ↑
        for(i = 0; i < 1000; i++){              |  並列動作
            t = a[j][i] + b[j][i];              |
            c[j][i] = t + d[j][i];              |
        }                                       ↓
    }
    ...

    func();
}
```

この場合、ループの終了時のtの値が、関数funcの中で参照されないことが分かるのであれば、“図4.30 temp指示子の使用例”のようにtempでtを指定します。これにより、ループ終了時のtの値を正しくする命令が不要になるので、実行性能が向上します。

図4.30 temp指示子の使用例

```
int t;
main()
{
    ...

#pragma loop temp t
    for(j = 0; j < 50; j++){                    ↑
        for(i = 0; i < 1000; i++){              |  並列動作
            t = a[j][i] + b[j][i];              |
            c[j][i] = t + d[j][i];              |
        }                                       ↓
    }
    ...

    func();
}
```

注意

temp指示子に、一時的に使用されている変数以外の変数を誤って指定した場合、本処理系は誤ったループスライスを行うことがあります。

temp_private指示子

temp_private指示子は、自動並列化の対象となっているループ内において、指定された変数を各スレッドでローカルな変数として扱うことを指示します。

なお、以下のような最適化メッセージが出力された場合、本指示子で対象データを各スレッドでローカルな領域に割り付けることにより、自動並列化の阻害要因が解消され、該当ループの自動並列化が促進されます。

jwd5208p-i “*file*”, line *n*: 変数’*var*’ を定義・参照する順序がわからないため、定義・参照する順序が逐次実行と変わる可能性があり、このループは並列化できません。

本指示子は、指定した直後のループのみ対象となります。

本指示子に指定できる変数は以下のとおりです。

- 整数型または浮動小数点型の変数
- 整数型または浮動小数点型を要素にもつ配列型で、かつ、翻訳時に大きさが確定している変数

本指示子が適用された場合、以下の最適化メッセージが出力されます。

```
jwd5013p-i "file", line n: 最適化指示子temp_privateを変数'var'に適用しました。
```

“図4.31 temp_private指示子のないプログラム例”の場合、外側ループでは配列aの定義引用順序が不明のため、並列化ができません。

図4.31 temp_private指示子のないプログラム例

```
for (i = 0; i < 100; i++) {       ←  自動並列化の対象となるループ
    for (j = 0; j < m; j++) {     ←  変数mの値が不明のため、配列aの定義される範囲が不明
        a[j] = b[j];
    }
    for (j = 0; j < n; j++) {     ←  変数nの値が不明のため、配列aの参照される範囲が不明
        c[i][j] = a[j];
    }
}
```

“図4.32 temp_private指示子の使用例”の場合、temp_private指示子を指定し、配列aの依存関係を明確にすることにより、外側ループで並列化することができます。

図4.32 temp_private指示子の使用例

```
#pragma loop temp_private a
for(i = 0; i < 100; i++){
    for(j = 0; j < m; j++){
        a[j] = b[j];
    }
    for(j = 0; j < n; j++){
        c[i][j] = a[j];
    }
}
```

並列動作

注意

temp_private指示子を誤って指定した場合、実行結果に誤りが生じることがあります。

first_private指示子

first_private指示子は、自動並列化の対象となっているループ内において、指定された変数を各スレッドでローカルな変数として扱い、また、スレッドごとに変数の初期値の引用があることを指示します。

なお、以下のような最適化メッセージが出力された場合、本指示子で対象データを各スレッドでローカルな領域に割り付けることにより、自動並列化の阻害要因が解消され、該当ループの自動並列化が促進されます。

```
jwd5208p-i "file", line n: 変数'var'を定義・参照する順序がわからないため、定義・参照する順序が逐次実行と変わる可能性があり、このループは並列化できません。
```

本指示子は、指定した直後のループのみ対象となります。

本指示子に指定できる変数は以下のとおりです。

- 整数型または浮動小数点型の変数
- 整数型または浮動小数点型を要素にもつ配列型で、かつ、翻訳時に大きさが確定している変数

本指示子が適用された場合、以下の最適化メッセージが出力されます。

```
jwd5014p-i "file", line n: 最適化指示子first_privateを変数'var'に適用しました。
```

“図4.33 first_private指示子のないプログラム例”の場合、外側ループでは配列aの定義引用順序が不明のため、並列化ができません。

図4.33 first_private指示子のないプログラム例

```
for (i = 0; i < 100; i++) {       ←  自動並列化の対象となるループ
    for (j = 0; j < m; j++) {     ←  変数mの値が不明のため、配列aの定義される範囲が不明
        b[i][j] = a[j];
    }
    for (j = n; j < 100; j++) {   ←  変数nの値が不明のため、配列aの参照される範囲が不明
        a[j] = b[i][j] + d[j];
        c[i][j] = a[j];
    }
}
```

“図4.34 first_private指示子の使用例”の場合、first_private指示子を指定し、配列aの依存関係を明確にすることにより、外側ループで並列化することができます。

図4.34 first_private指示子の使用例

```
#pragma loop first_private a
for(i = 0; i < 100; i++){
    for(j = 0; j < m; j++){
        b[i][j] = a[j];
    }
    for(j = n; j < 100; j++){
        a[j] = b[i][j] + d[j];
        c[i][j] = a[j];
    }
}
```

並列動作

注意

first_private指示子を誤って指定した場合、実行結果に誤りが生じることがあります。

last_private指示子

last_private指示子は、自動並列化の対象となっているループ内において、指定された変数を各スレッドでローカルな変数として扱い、また、ループ終了後に変数の引用があることを指示します。

なお、以下のような最適化メッセージが出力された場合、本指示子で対象データを各スレッドでローカルな領域に割り付けることにより、自動並列化の阻害要因が解消され、該当ループの自動並列化が促進されます。

```
jwd5208p-i "file", line n: 変数'var'を定義・参照する順序がわからないため、定義・参照する順序が逐次実行と変わる可能性があり、このループは並列化できません。
```

本指示子は、指定した直後のループのみ対象となります。

本指示子に指定できる変数は以下のとおりです。

- 整数型または浮動小数点型の変数
- 整数型または浮動小数点型を要素にもつ配列型で、かつ、翻訳時に大きさが確定している変数

本指示子が適用された場合、以下の最適化メッセージが出力されます。

```
jwd5015p-i "file", line n: 最適化指示子last_privateを変数'var'に適用しました。
```

“図4.35 last_private指示子のないプログラム例”の場合、外側ループでは配列aの定義引用順序が不明のため、並列化ができません。

図4.35 last_private指示子のないプログラム例

```
for (i = 0; i < 100; i++) {       ←  自動並列化の対象となるループ
    for (j = 0; j < m; j++) {
        b[i][j] = i;
    }
    for (j = n; j < 100; j++) {   ←  変数nの値が不明のため、配列aの参照される範囲が不明
        a[j] = b[i][j] + d[j];
        c[i][j] = a[j];
    }
}
foo(a, c);
```

“図4.36 last_private指示子の使用例”の場合、last_private指示子を指定し、配列aの依存関係を明確にすることにより、外側ループで並列化することができます。

図4.36 last_private指示子の使用例

```
#pragma loop last_private a
for(i = 0; i < 100; i++){
    for(j = 0; j < m; j++){
        b[i][j] = i;
    }
    for(j = n; j < 100; j++){
        a[j] = b[i][j] + d[j];
        c[i][j] = a[j];
    }
}
foo(a, c);
```

並列動作

注意

last_private指示子を誤って指定した場合、実行結果に誤りが生じることがあります。

***var1 op var2*指示子**
***var1 op const*指示子**

これらの指示子は、変数と変数の大小関係または変数と定数の大小関係を指示します。

ここで、*var1*および*var2*は、変数名です。*const*は、整数定数です。*op*は、関係演算子(<, <=, >, >=)または等価演算子(==, !=)です。

“図4.37 var1 op const指示子の使用例”の場合、最適化制御行が記述されていないと配列要素の添字の値の大小関係が分からないため、配列aの定義引用順序が逐次実行のときと変わる可能性があり並列化されません。

この場合、*var1 op const*指示子を指定することにより、配列aの定義引用順序が逐次実行のときと変わらないと分かるので並列化することができます。

図4.37 var1 op const指示子の使用例

```
int i, m;
float a[10000], b[10000];
...

#pragma loop m > 2000
for (i = 0; i < 2000; i++){
    a[i] = a[i + m] + b[i];
}
```

並列動作

4.2.7 自動並列化機能を使うときの留意事項

本処理系の自動並列化機能を使用する場合の留意事項について説明します。

4.2.7.1 並列処理の入れ子での注意

ループスライスされたループ内で関数を引用していて、引用先の関数中にループスライスされるループを含む場合、結果的に並列実行部分が入れ子になります。このような場合、内側のループは逐次実行されます。このようなループを含むソースプログラムは-Kparallel,instance=*N*を指定して翻訳してはなりません。

“図4.38 並列処理が入れ子になった場合”に、スライスされたループが逐次実行される例を示します。このようなループを含むソースプログラムを-Kparallel,instance=*N*オプションを指定して翻訳すると、実行結果は保証されません。

図4.38 並列処理が入れ子になった場合

```
#include <stdio.h>
#define M 4000
double a[M][M], b[M][M];
void f1(int);
void f2(int);

main()
{
    f1(M);
    printf("%e %e¥n", a[0][0], b[0][0]);
}

void f1(int n)
{
    int i;
#pragma loop independent
    for(i = 0; i < n; i++){     ←  このループは並列実行される
        f2(i);
    }
}

void f2(int i)
{
    int j;
    for(j = 0; j < i + 1; j++){  ←  このループは逐次実行される
        a[i][j] = 3.8;
    }
    for(j = 0; j < i + 1; j++){  ←  このループは逐次実行される
        b[i][j] = 4.8;
    }
}
```

“図4.38 並列処理が入れ子になった場合”のソースプログラムa.cを以下のように翻訳した場合、実行結果は保証できません。

```
$ fccpx -Kparallel,instance=4 -Nlibomp a.c （誤った使い方）
```

このような誤りを防ぐために、ループスライスされるループ内で引用している関数に、serial指示子を以下のように指定します。

```
void f2(int i)
{
    int j;
#pragma procedure serial
    for(j = 0; j < i + 1; j++){  ←  逐次化
        a[i][j] = 3.8;
    }
    for(j = 0; j < i + 1; j++){  ←  逐次化
        b[i][j] = 4.8;
    }
}
```

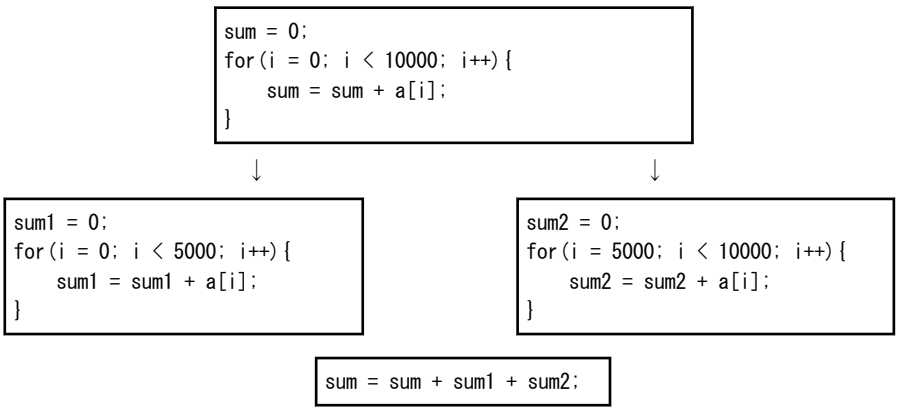
4.2.7.2 -Kparallel,reduction指定時の注意

-Kparallelオプションおよび-Kreductionオプションを指定して翻訳したとき、並列実行の結果が、逐次実行の結果と異なることがあります。これは、並列実行のときの演算順序が、逐次実行の演算順序と異なる場合があるためです。

“図4.39 リダクションによる計算誤差”は、“4.2.5.7 リダクションによるループスライス”の説明で使用した例です。変数sumは、逐次実行では、配列要素a[0]からa[9999]まで順次に加算されます。並列実行では、配列要素a[0]からa[4999]まで順次に加算した値を変数sum1として、配列要素a[5000]からa[9999]まで加算した値をsum2とします。そして、sum1とsum2を加算した値をsumとします。

つまり、逐次実行と並列実行では、配列aの要素の値を累計する順番が違うので、演算結果に計算誤差が生じることがあります。

図4.39 リダクションによる計算誤差



4.2.7.3 最適化制御行の使い方の注意

本処理系は、最適化指示子の使い方が誤っている場合、誤ったループスライスを行います。

norecurrence指示子、temp指示子、independent指示子、temp_private指示子、およびlast_private指示子の誤った使い方を、以下に示します。

“図4.40 norecurrence指示子の誤った使用例”では、配列aに対して、誤ったnorecurrence指示子を指定しています。ループスライスを行うと配列aの定義引用順序が変わるため、誤った動作をすることがあります。

図4.40 norecurrence指示子の誤った使用例

```
#pragma loop norecurrence a
for(i = 1; i < 10000; i++){
    a[i] = a[i-1] + b[i];
}
```

"図4.41 temp指示子の誤った使用例"では、変数tに対して、誤ってtemp指示子を指定しています。ループの実行後、変数tの値は保証されないので、変数lastに期待した値が代入されないことがあります。

図4.41 temp指示子の誤った使用例

```
#pragma loop temp t
for(i = 0; i < 1000; i++){
    t = a[i] + b[i];
    c[i] = t + d[i];
}
last = t;
```

"図4.42 independent指示子の誤った使用例"では、関数funcに対して、誤ってindependent指示子を指定しています。関数funcの引用によって配列aの定義引用の順序が変わるため、結果が正しくない場合があります。

図4.42 independent指示子の誤った使用例

```
int a[1000], b[1000];

main()
{
    int i;
    void func(int);
#pragma procedure independent func
    for(i = 1; i < 1000; i++){
        a[i] = b[i] + 1;
        func(i-1);
    }
}

void func(int j)
{
    a[j] = a[j] + 1;
}
```

"図4.43 temp_private 指示子の誤った使用例"では、変数aに対して、誤ってtemp_private指示子を指定しています。各スレッドで不定な値を参照することになるため、実行結果は保証されません。

図4.43 temp_private 指示子の誤った使用例

```
#pragma loop temp_private a
for(i = 0; i < 1000; i++){
    b[i] = a;
}
```

“図4.44 last_private 指示子の誤った使用例”では、配列aに対して、誤ってlast_private指示子を指定しています。配列aが最終スレッドで定義されない場合があるため、実行結果は保証されません。

図4.44 last_private 指示子の誤った使用例

```
#pragma loop last_private a
for(j = 0; j < 1000; j++){
    for(i = j; i < 500; i++){
        a[i] = b[i];
    }
}
foo(a);
```

4.2.7.4 並列処理中の標準ライブラリ関数

ループスライスされたループ内で関数を引用していて、引用先の関数中にループスライスの対象とならない標準ライブラリ関数を含む場合、動作は保証されません。並列実行に伴うオーバヘッドのため、逐次処理した場合より実行性能が低下する場合があります。

4.3 OpenMP仕様による並列化

ここでは、本処理系で提供しているOpenMP仕様による並列化について説明します。OpenMP仕様については、“OpenMP Architecture Review Board”のウェブサイトをお読みください。

本処理系は、以下の仕様をサポートしています。ただし、デバイスに関連する構文、環境変数、およびランタイムライブラリルーチンはサポートしていません。

コンパイラのモード	サポートする仕様
tradモード	OpenMP 3.1
	OpenMP 4.0の一部 (注1)
	OpenMP 4.5の一部 (注1)
clangモード	OpenMP 4.5 (注2)
	OpenMP 5.0の一部 (注3)

注1) 以下の機能を使用できます。

- simd構文
- declare simd構文
- parallel構文のproc_bind指示節
- task構文のdepend指示節
- taskgroup構文

注2) 以下の機能は使用できません。

- declare simd構文
- taskloop simd構文のlinear指示節

注3) 以下の機能を使用できます。

- task構文のin_reduction指示節
- taskgroup構文のtask_reduction指示節
- taskloop構文のreduction指示節およびin_reduction指示節

4.3.1 翻訳の方法

ソースプログラムの翻訳およびリンクを行うには、翻訳コマンドに以下のオプションを指定します。

指定するオプションは、OpenMP仕様による並列化を行うか否かで異なります。

OpenMP仕様による並列化	指定するオプション	
	翻訳時	リンク時
OpenMP仕様による並列化およびSIMD化を行う場合	-Kopenmp	-Kopenmp -Nlibomp
OpenMP仕様によるSIMD化のみを行い、並列化を行わない場合	-Kopenmp_simd	-

-: 指定するオプションはありません。

4.3.1.1 OpenMPプログラムのための翻訳時オプション

ここでは、OpenMPプログラムを翻訳するための翻訳時オプションについて説明します。

-Kopenmp

-Kopenmpオプションは、OpenMP指示文を有効にし、ソースプログラムを翻訳します。

-Kopenmpオプションは、リンク時にも指定する必要があります。-Kopenmpオプションを指定して翻訳されたオブジェクトプログラムが含まれる場合は、リンク時に-Kopenmpオプションを指定する必要があります。

例

```
$ fccpx a.c -Kopenmp -c
$ fccpx a.o -Kopenmp -Nlibomp
```

-Kopenmp_simd

-Kopenmp_simdオプションは、OpenMP仕様のsimd構文およびdeclare simd構文のみを有効にし、ソースプログラムを翻訳します。

-Nlibomp

OpenMPライブラリとして、LLVM OpenMPライブラリを使用することを指示します。

-Nlibompオプションは、リンク時に指定する必要があります。

4.3.1.2 OpenMPプログラムの最適化情報を出力するための翻訳時オプション

ここでは、OpenMPプログラムの最適化情報を出力するための、tradモード用翻訳時オプションについて説明します。

-Nsrc, -Nlst

-Kopenmpオプションが-Nsrcオプションまたは-Nlstオプションと同時に指定された場合、OpenMP仕様の構文によって指示された実行文に対する並列化情報の出力は、以下のようになります。

- 並列に実行される可能性がある文に対しては、pが出力されます。冗長に実行される文に対しては、出力されません。for指示構文またはsections指示構文に直接囲まれた実行文がこれに該当します。
- 同時に1スレッドだけで実行される文に対しては、sが出力されます。master指示構文、single指示構文、critical指示構文またはordered指示構文に直接囲まれた実行文がこれに該当します。atomic指示構文の直後の実行文に対しては、その全体が排他的に1スレッドで実行される場合、sが出力されます。
- 1つの実行文の中で、並列に実行される可能性がある部分と、1スレッドだけで実行される部分が混在している場合、mが出力されます。

ネストした並列実行では、その実行文を囲む最も内側のparallelリージョンに対する並列化情報を出力します。そのparallelリージョンがどのように呼ばれるか(並列実行中に呼ばれるか、それとも逐次実行中に呼ばれるか)は、出力に影響を与えません。

-Koptmsg=2

OpenMP仕様のSIMD化やcollapseなど実行性能に関連する診断メッセージを出力します。

4.3.1.3 OpenMPプログラムの制限事項

tradモードでは、OpenMP指示文を含んでいる関数はインライン展開されません。

4.3.2 実行の方法

OpenMPプログラムを実行させる手続きは、逐次実行のときと同じです。

4.3.2.1 実行時の環境変数

環境変数OMP_WAIT_POLICY

ACTIVE	同期が獲得できるまでスピン待ちを行います。
PASSIVE	スピン待ちを行わず、サスペンド待ちを行います。

デフォルトはPASSIVEです。

経過時間を重視する場合には、ACTIVEを選択してください。CPU時間を重視する場合は、PASSIVEを選択してください。

環境変数OMP_PROC_BIND

スレッドアフィニティを制御します。true、false、またはカンマで区切られたmaster、close、spreadのリストのいずれかを設定します。リストの値は、ネストレベルに対応するparallelリージョンで使用するスレッドアフィニティポリシーを設定します。

デフォルトはcloseです。

falseを設定した場合、スレッドアフィニティは無効になり、parallel構文のproc_bind指示節も無視されます。

false以外を設定した場合、スレッドアフィニティは有効になり、初期スレッドはプレースリストの最初のプレースにバインドされます。

masterを設定した場合、すべてのスレッドはマスタースレッドと同じプレースにバインドされます。

closeを設定した場合、スレッドはマスタースレッドがバインドされているプレースの次から連続したプレースにバインドされます。

spreadを設定した場合、マスタースレッドのプレースパーティションが分割され、スレッドは各サブパーティションの1つのプレースにバインドされます。

trueを指定した場合は、spreadを指定した場合と同様の効果になります。

プレースについては環境変数OMP_PLACESを参照してください。

環境変数OMP_PLACES

スレッドアフィニティを制御するプレースリストを定義します。

プレースとは、スレッドが実行されるプロセッサの順序付けされていない集合です。プレースリストとは、利用可能なプレースの順序付けされたリストです。

環境変数OMP_PLACESには、抽象名または明示的なプレースリストのどちらかを、以下の形式で設定します。

```
{ abstract-name[(num-places)] | place-list }
```

abstract-name

抽象名です。抽象名には以下を指定します。デフォルトはcoresです。

抽象名	意味
threads	プレースは、1ハードウェアスレッド単位になります。 本処理系ではシステムのCPU資源の最小単位になりcoresと等価になります。
cores	プレースは、1コア単位になります。 本処理系ではシステムのCPU資源の最小単位になりthreadsと等価になります。
sockets	プレースは、1ソケット単位になります。 本処理系ではNUMAノード単位になります。

num-places

抽象名形式のプレース数です。1以上の正の整数です。

place-list

明示的なプレースリストです。以下の形式で指定します。

```
{ place:length[:stride] | [!]place[,place-list] }  (注1)
```

place

“{”*place1*“}” (注2)

place1

{ *cpuid*:*length*[:*stride*] | [!]*cpuid*[,*place1*] }

length

インターバル指定の要素数です。1以上の正の整数です。

stride

インターバル指定の増分値です。1以上の正の整数で、デフォルト値は1です。

cpuid

システムのCPU資源の最小単位の識別番号です。

注1) 排他演算子!は直後の指定を除外することを指示します。

注2) “{”および“}”は選択記号ではありません。半角中括弧を使用して指定することを意味します。

例

環境変数OMP_PLACESの指定例

- 例1:

```
$ export OMP_PLACES="cores"
```

抽象名“cores”のプレースリストを定義しています。

- 例2:

```
$ export OMP_PLACES="cores(4)"
```

抽象名“cores”の4プレースのプレースリストを定義しています。

- 例3:

```
$ export OMP_PLACES="{12,13,14},{15,16,17},{18,19,20},{21,22,23}"
$ export OMP_PLACES="{12:3},{15:3},{18:3},{21:3}"
$ export OMP_PLACES="{12:3}:4:3"
```

上記3つの定義は、すべて等価です。

cpuid 12～14、15～17、18～20、および21～23の同じ4つのプレースを定義しています。

環境変数OMP_STACKSIZE

利用者は、環境変数OMP_STACKSIZEの値として、スレッドごとのスタック領域の大きさをバイト、Kバイト、Mバイト、Gバイト単位で指定することができます。

デフォルト値は8Mバイトです。

環境変数GOMP_CPU_AFFINITY

スレッドを指定されたcpuid順にCPUバインドします。

スレッド数が、指定したcpuidの数を超える場合は、先頭から繰り返して使用します。

各cpuidは、コンマ(',')または空白(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

```
cpuid1[-cpuid2[:inc]]
```

cpuid1: 範囲指定の最初のcpuid (0≦*cpuid1*＜CPU_SETSIZE)
cpuid2: 範囲指定の最後のcpuid (0≦*cpuid2*＜CPU_SETSIZE)
inc: 増分値 (1≦*inc*＜CPU_SETSIZE)

なお以下の条件を満たす必要があります。

```
cpuid1≦cpuid2
```

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべてを指定した場合と等価になります。

cpuidは、上記のように指定はできますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)を参照してください。

注意

cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーを出力しプログラムが終了しますので、設定値を見直してください。実行開始時のプロセスのCPU affinityについては、FLIB_HPCFUNC_INFO=TRUE、またはtasksetやnumactlなどのシステムのコマンドで、確認することができます。環境変数FLIB_HPCFUNC_INFOについては、“付録I 高速化機能の利用”を参照してください。システムのコマンドについては、各manマニュアルを参照してください。

例

環境変数GOMP_CPU_AFFINITYの指定例

- 例1:

```
$ export GOMP_CPU_AFFINITY="12,14,13,15"
```

12,14,13,15の順でスレッドにバインドします。
スレッド数5以上の場合は、先頭から繰り返して使用します。

- 例2:

```
$ export GOMP_CPU_AFFINITY="12-19"
```

12,13,14,15,16,17,18,19の順でスレッドにバインドします。
スレッド数9以上の場合は、先頭から繰り返して使用します。

- 例3:

```
$ export GOMP_CPU_AFFINITY="12-19:2"
```

12,14,16,18の順でスレッドにバインドします。
スレッドが5以上の場合は、先頭から繰り返して使用します。

- 例4:

```
$ export GOMP_CPU_AFFINITY="12-16:2,13,19"
```

12,14,16,13,19の順でスレッドにバインドします。
スレッドが6以上の場合は、先頭から繰り返して使用します。

4.3.2.2 OpenMP仕様の環境変数

利用者は、以下のOpenMP仕様の環境変数を使用することが可能です。

なお、OpenMP 4.0以降で使用可能になった環境変数について、環境変数名の後の括弧内にサポートしているOpenMPの仕様を示します。

OpenMP仕様の環境変数の詳細については、OpenMP仕様書をお読みください。

環境変数OMP_SCHEDULE

schedule指示節にruntimeを持つ、for指示文またはparallel for指示文に対して、実行すべきスケジュールタイプとチャンクサイズを指定します。

環境変数OMP_NUM_THREADS

実行中に使用するスレッドの数を指定します。

環境変数OMP_DYNAMIC

動的スレッド調整機能の有効または無効を指定します。

環境変数OMP_PROC_BIND

スレッドをCPUにバインドする方法を指定します。

環境変数OMP_NESTED

parallelリージョンのネスト機能の有効または無効を指定します。

環境変数OMP_STACKSIZE

スレッドごとのスタック領域の大きさを指定します。

環境変数OMP_WAIT_POLICY

同期待ち処理の動作を指定します。

環境変数OMP_MAX_ACTIVE_LEVELS

ネストしている活動状態のparallelリージョンの最大数を指定します。

環境変数OMP_THREAD_LIMIT

OpenMPプログラムで実行するスレッド数の最大数を指定します。

環境変数OMP_PLACES (OpenMP 4.0以降)

スレッドが利用できるプレースリストを指定します。

環境変数OMP_CANCELLATION (OpenMP 4.0以降)

キャンセレーション機能の有効または無効を指定します。

環境変数OMP_MAX_TASK_PRIORITY (OpenMP 4.5以降)

タスク優先度の最大値を指定します。

環境変数OMP_DISPLAY_ENV (OpenMP 4.0以降)

OpenMPバージョンとOpenMP内部制御変数の初期値を表示するかどうかを指定します。

OpenMPバージョンの値は201611です。

4.3.2.3 実行時の注意事項

本処理系のOpenMP仕様による並列化機能を使用する場合の注意事項を、以下に示します。

実行時の変数割付け

スレッドごとのスタック領域の大きさを、特定の大きさで確保したい場合には、環境変数OMP_STACKSIZEを使用してください。

環境変数OMP_STACKSIZEについては、“4.3.2.1 実行時の環境変数”をお読みください。

CPU数の上限

CPU数の上限値は、システムで利用できるCPU数です。

スレッド数

OpenMPのスレッド数は、以下の優先順位で決定されます。自動並列化のスレッド数は、“4.2 自動並列化”の記述に従います。

1. parallel指示文のnum_threads指示節の指定値
2. omp_set_num_threads関数の指定値
3. 環境変数OMP_NUM_THREADSの指定値
4. CPU数の上限値

動的スレッド調整機能が有効な場合、上記の優先順位で決まったスレッド数はCPU数の上限内に収まるように調整されます。

動的スレッド調整機能が無効な場合、上記の優先順位で決まったスレッド数となります。1CPU当たりのスレッド数が1を超える場合には、並列に実行されることを意図したスレッドが時分割で実行されることになります。このような場合、スレッド間の同期処理のオーバヘッドが大きくなり、実行性能が低下することがあります。システムによる負荷も考慮して、1CPU当たりのスレッド数が1以下になるようにスレッド数を設定することをお勧めします。

スレッドのCPUバインド

スレッドは固定のCPUにバインドされませんが、環境変数OMP_PROC_BINDまたはGOMP_CPU_AFFINITYを使用することによりスレッドの固定のCPUへのバインドを制御することができます。環境変数GOMP_CPU_AFFINITYは環境変数OMP_PROC_BINDよりも優先されます。

4.3.3 処理系依存の仕様

OpenMP仕様において、処理系の実現に任されている仕様については、本処理系では以下のように実現しています。各項目の詳細については、OpenMP仕様書をお読みください。

メモリモデル

4バイトを超えるか4バイト境界を跨ぐ変数に対して、

- 2つのスレッドからの書き込みが同時に起こるとき

  明示的な排他制御が行われなければ、変数の値はどちらの値にもならず不定となることがあります。

- 2つのスレッドから書き込みと読み出しが同時に起こるとき

  明示的な排他制御が行われなければ、読み出される変数の値は書き込み前の値でも書き込み後の値でもなく不定となることがあります。

内部制御変数

各内部制御変数の初期値は以下のとおりです。

- nthreads-varの初期値はCPU数の上限値です。
- dyn-varの初期値はFALSEです。
- run-sched-varの初期値は、チャンクサイズなしのstaticです。
- def-sched-varの初期値は、チャンクサイズなしのstaticです。
- bind-varの初期値はFALSEです。
- stacksize-varの初期値は、8Mバイトです。
- wait-policy-varの初期値は、PASSIVEです。
- thread-limit-varの初期値は、2147483647です。
- max-active-levels-varの初期値は、2147483647です。
- place-partition-varの初期値は、coresです。本処理系では、threadsと等価です。

動的スレッド調整機能

本処理系のOpenMP仕様による並列化機能では、動的スレッド調整機能を実現しています。この機能の効果については、“4.3.2.3 実行時の注意事項”をお読みください。

デフォルトの状態では、動的スレッド調整機能は無効です。

ループ指示文

一重化したループの繰返し数の計算に使用する変数の型は、long int型です。

内部制御変数run-sched-varにautoが設定されたときのschedule(runtime)指示節の効果は、schedule(static)となります。

sections構文

sections構文内の構造化ブロックのスレッドへの割当ては、staticスケジュールと同じ方法で行われます。

single構文

singleリージョンは、最初にそのリージョンに到達したスレッドによって実行されます。

taskloop構文

grainsize指示節およびnum_tasks指示節の指定がない場合、これらの値はループ繰返し数を考慮して適切に設定されます。

一重化したループの繰返し数の計算に使用する変数の型は、8バイト整数型です。

critical構文

本処理系ではロックのヒントをサポートしていません。したがってhint指示節を使用した場合、hint指示節は指定されてないものとみなされます。

atomic構文

2つのatomicリージョンは、更新する変数の型が異なっていれば、独立に(排他的でなく)実行されます。型が一致しているとき、以下の場合にはアドレスが異なっていても排他的に実行される場合があります。

- 論理型、複素数型、1バイト整数型、2バイト整数型、または4倍精度(16バイト)実数型の変数の更新
- 配列要素の更新であり、添字式が字面上一致していないとき
- 対象の式が明示的または暗黙の型変換を伴うとき

omp_set_num_threads関数

omp_set_num_threads関数の呼び出しは、引数が0以下の値のとき、1を設定したとみなされます。この値として、システムがサポートするスレッド数を超える数値を設定してはなりません。

omp_set_schedule関数

実装依存のスケジュールタイプはありません。

omp_set_max_active_levels関数

omp_set_max_active_levels関数が明示的なparallelリージョンから呼ばれた場合、呼び出しは無効になります。
omp_set_max_active_levels関数の引数が0以上の整数ではない場合、呼び出しは無効になります。

omp_get_max_active_levels関数

omp_get_max_active_levels関数はプログラム内のどの場所からでも呼び出されることができ、内部制御変数max-active-levels-varの値を返します。

omp_get_place_proc_ids関数

指定されたプレースの利用できるプロセッサを識別する数値を返します。プロセッサの定義は上記されているオペレーティングシステムが管理するCPUになります。

omp_init_lock_with_hintおよびomp_init_nest_lock_with_hint関数

本処理系ではロックのヒントをサポートしていません。したがって本関数を使用した場合、hintがないものとみなされます。

環境変数OMP_SCHEDULE

コンパイラのモードによって、動作が一部異なります。

環境変数OMP_SCHEDULEに指定されたスケジュールタイプが有効なスケジュールタイプではない場合、指定は無効となり、以下のデフォルト値が適用されます。

コンパイラのモード	デフォルト値
tradモード	チャンクサイズなしのstatic

コンパイラのモード	デフォルト値
clangモード	チャンクサイズ1のstatic

OMP_SCHEDULEに指定されたスケジュールタイプがstatic、dynamic、またはguidedであり、指定されたチャンクサイズが正ではない場合、チャンクサイズは次のようになります。

コンパイラのモード	スケジュールタイプ	チャンクサイズ
tradモード	static	チャンクサイズなし
	dynamic	1
	guided	
clangモード	static	1
	dynamic	
	guided	

環境変数OMP_NUM_THREADS

環境変数OMP_NUM_THREADSのリストに0を設定すると、1を設定した場合と同じ効果をもちます。0未満を設定すると、“4.3.2.3 実行時の注意事項”のスレッド数に従ってスレッド数が決定されます。この値として、システムがサポートするスレッド数を超える数値を設定してはなりません。

環境変数OMP_PROC_BIND

環境変数OMP_PROC_BINDに使用できない値を設定すると、指定は無効となり、デフォルト値(close)が適用されます。以下の条件をすべて満たす場合、余ったスレッドは可能な限り均等にプレースに割り当てられます。

- OMP_PROC_BINDにspreadまたはcloseが指定されている
- T(スレッド数)がP(プレース数)より大きい
- T/Pが割り切れない

環境変数OMP_PLACES

環境変数OMP_PLACESに設定された値が定義された形式を満たしていない場合、指定は無効となり、デフォルト値(cores)が適用されます。

環境変数OMP_DYNAMIC

環境変数OMP_DYNAMICにTRUEでもFALSEでもない値を設定すると、指定は無効となり、デフォルト値(FALSE)が適用されます。

環境変数OMP_NESTED

環境変数OMP_NESTEDにTRUEでもFALSEでもない値を設定すると、指定は無効となり、デフォルト値(FALSE)が適用されます。

環境変数OMP_STACKSIZE

環境変数OMP_STACKSIZEに設定された値が定義された形式を満たしていない場合、指定は無効となり、デフォルト値(8Mバイト)が適用されます。

環境変数OMP_WAIT_POLICY

ACTIVEの振舞いはスピン待ちです。PASSIVEの振舞いはサスペンド待ちです。

環境変数OMP_MAX_ACTIVE_LEVELS

環境変数OMP_MAX_ACTIVE_LEVELSに設定された値が0以上の整数ではない場合、指定は無効となり、デフォルト値(2147483647)が適用されます。

環境変数OMP_THREAD_LIMIT

環境変数OMP_THREAD_LIMITに設定された値が正の整数ではない場合、指定は無効となり、デフォルト値(2147483647)が適用されます。

4.3.4 プログラミングの注意事項

ここでは、OpenMP仕様のプログラミングにおける注意事項について説明します。

4.3.4.1 parallelリージョンおよび明示的なtaskリージョンの実現

parallel構文またはtask構文の構造化ブロックは、翻訳処理により、内部関数化されます。

parallel構文またはtask構文から生成される内部関数は、コンパイラのモードによって名前が異なります。内部関数の名前を以下に示します。

コンパイラのモード	内部関数の種別	内部関数名
tradモード	parallel構文から生成された内部関数	"._OMP_*識別番号A*"という名前が付加されます
	task構文から生成された内部関数	"._TSK_*識別番号B*"という名前が付加されます
clangモード	parallel構文から生成された内部関数	.omp_outlined.[_debug__][.*識別番号C*] (注1) (注2)
	task構文から生成された内部関数	.omp_task_entry.[.*識別番号C*] (注3)

識別番号A: ソースファイル内でparallel構文を識別する一意の数字

識別番号B: ソースファイル内でtask構文を識別する一意の数字

識別番号C:ソースファイル内で一意の数字。ソースファイル内でparallel構文およびtask構文がそれぞれ複数存在した場合、*識別番号C*にはparallel構文およびtask構文を跨った一意の番号が割り振られます。

注1) "_debug__"は、-gオプションが指定されている場合に付加されます。

注2) "*.識別番号C*"は、ソースファイル内にparallel構文が2つ以上指定されている場合に付加されます。

注3) "*.識別番号C*"は、ソースファイル内にtask構文が2つ以上指定されている場合に付加されます。

4.3.4.2 threadprivate変数の実現

threadprivate変数はスレッド・ローカル・ストレージ(Thread-Local Storage)を使用して実現されています。スレッド・ローカル・ストレージのサイズがオフセットの範囲を超えた場合、リンク時にエラーが発生します。

エラーメッセージ例:

```
relocation truncated to fit: R_AARCH64_TLSLE_ADD_TPREL_HI12 against symbol 'varname'
```

このエラーが発生した場合、tradモードの-Ktls_size={12|24|32|48}オプションまたはclangモードの-mfj-tls-size={12|24|32|48}オプションにより、適切なオフセットのサイズを指定してください。-Ktls_sizeオプションの詳細は"2.2.2.5 -Kオプション"を、-mfj-tls-sizeオプションの詳細は"9.1.2.2.6 コード生成関連オプション"をお読みください。

4.3.4.3 OpenMPプログラムの自動並列化

-Kopenmpオプションと-Kparallelオプションは、同時に指定可能です。同時に指定した場合、自動並列化されるループは以下のように制限されます。

- OpenMPの構文内にあるループは、自動並列化の対象とはなりません。
- OpenMPの指示文を静的に内部に含むループは、自動並列化の対象とはなりません。
- OpenMPの指示文で並列化されるループがある場合、以下のループは自動並列化の対象とはなりません。
  - OpenMPの指示文で並列化されるループ自身
  - OpenMPの指示文で並列化されるループの内側にあるループ

4.3.4.4 clangモードにおける共有ライブラリの作成

clangモードで以下のような共有ライブラリを作成する場合は、翻訳時オプション-fopenmpを指定してください。翻訳時オプション-fopenmpを指定せずに作成した場合、その共有ライブラリをリンクしたプログラムは異常終了することがあります。

- 翻訳時オプション-fopenmpを指定して作成された共有ライブラリをリンクする共有ライブラリ

- 翻訳時オプション-fopenmpを指定して作成したプログラムから、dlopen関数などで動的にオープンして使用される共有ライブラリ

例

- 例1

```
$ fccpx shraed1.c -Nclang -fopenmp -shared -o libsample_linked.so
$ fccpx shared2.c -Nclang -fopenmp -shared -o libsample.so -L. -lsample_linked
```

リンクするlibsample_linked.soが翻訳時オプション-fopenmpを指定して作成されているため、libsample.so作成時も翻訳時オプション-fopenmpを指定します。

- 例2

```
$ fccpx main.c -Nclang -fopenmp -o main.exe
$ fccpx shared.c -Nclang -fopenmp -shared -o libsample.so
```

libsample.soを動的にオープンするプログラムmain.exeが翻訳時オプション-fopenmpを指定して作成されているため、libsample.so作成時も翻訳時オプション-fopenmpを指定します。

4.4 実行時メッセージ

LLVM OpenMPライブラリは、実行時メッセージを以下の形式で標準エラーに出力します。

```
OMP: type message
```

以下にLLVM OpenMP ライブラリの実行時メッセージ形式の意味を説明します。

OMP:

LLVM OpenMPライブラリのメッセージであることを示します

type

メッセージ種別を表します。以下のいずれかの値が出力されます。

値	説明
Hint	OpenMPに関する助言(ヒント)です。
Info	OpenMPに関する情報です。
Warning	OpenMPに関する警告メッセージです。致命的なエラーではありません。本メッセージが出力された場合でも、実行を継続します。
Error	OpenMPに関する致命的なエラーメッセージです。本メッセージが出力された場合は、実行を中断します。
System error	システムに関する致命的なエラーメッセージです。本メッセージが出力された場合は、実行を中断します。

message

メッセージ本文です。

例:

```
$ export OMP_STACKSIZE=1Z
$ ./a.out
OMP: Warning #80: OMP_STACKSIZE="1Z": value too large.
OMP: Info #107: OMP_STACKSIZE value "9223372036854775807" will be used.
OMP: Error #34: System unable to allocate necessary resources for OMP thread:
OMP: System error #11: Resource temporarily unavailable
OMP: Hint Try decreasing the value of OMP_NUM_THREADS.
```

上記の例は環境変数OMP_STACKSIZEにスタックサイズとしてFXシステムの許容値を超える値を設定したとき、表示される一連の実行時メッセージです。

注意

以下の実行時環境変数は、LLVM OpenMPライブラリの実行時メッセージの制御には使用できません。実行時環境変数の詳細は、“2.5.1 実行時環境変数”をお読みください。

- FLIB_C_MESSAGE

第5章 出力情報

本章では、C言語で記述されたプログラムの翻訳および実行において、本処理系が出力する情報について説明します。

5.1 翻訳時情報

ここでは、C言語で記述されたプログラムの翻訳において、本処理系が出力する情報について説明します。

5.1.1 ヘッダ

-Nlstオプション、-Nlst_out=*file*オプション、-Nsrcオプションおよび-Nstaオプションのうち少なくとも1つが指定された場合、翻訳時に出力する各情報に対して、以下の形式でヘッダを出力します。

5.1.1.1 出力形式

ヘッダの出力形式を以下に示します。

ヘッダの出力形式

```
Fujitsu C/C++ Version バージョン　日付
```

バージョン	コンパイラのバージョンを出力します。
日付	翻訳した日時をasctime関数の形式で出力します。

5.1.2 ソースリスト

以下の翻訳時オプションによって、ソースリストを出力することができます。

- -Nlst[={p|t}]

  ソースリストと統計情報をファイルに出力します。

- -Nlst_out=*file*

  ソースリストと統計情報を出力するファイル名を指定します。-Nlst=pオプションも有効になります。

- -Nsrc

  ソースリストを標準出力に出力します。

  ただし、-Nlstオプションまたは-Nlst_out=*file*オプションが同時に指定された場合、ソースリストはファイルに出力されます。

ソースリストには、適用された最適化および並列化を示す記号が付加されます。

5.1.2.1 出力形式

ソースリストの出力形式を以下に示します。

ソースリストの出力形式

```
Compilation information
  Current directory : ディレクトリ名
  Source file       : ソースファイル名
(line-no.)(optimize)
 nnnnnnnn   pi mmmmv  ソース
   ...
                      <<< Loop-information Start >>>
                      <<<  詳細な最適化情報
                      <<< Loop-information  End >>>
 nnnnnnnn   pi mmmmv  ソース
   ...
```

ディレクトリ名	ソースファイルを格納しているディレクトリ名
ソースファイル名	ソースファイル名
nnnnnnnn	ソースの行番号 (可変長)
p	並列化の表示記号
i	インライン展開の表示記号
mmmm	ループアンローリングによる展開数 (可変長)
v	SIMD化の表示記号
ソース	ソース行
<<< Loop-information Start >>>(注)	詳細な最適化情報のヘッダ
詳細な最適化情報(注)	直後の文に適用された最適化情報の詳細
<<< Loop-information End >>>(注)	詳細な最適化情報のフッタ

注) -Nlst=tオプションが有効な場合だけ出力されます。

5.1.2.2 ソースリストに含まれる情報

5.1.2.2.1 ソースの行番号

ソースの行番号を出力します。

5.1.2.2.2 並列化の表示記号

並列化の表示記号は、次の条件で出力します。

- ループ制御文のある行では、ループ制御文に対する表示記号を出力します。
  ループ制御文は、for文、while文、do-while文、またはif-goto文です。
- 複数のループ制御文がある行では、一番左にあるループ制御文に対する表示記号を出力します。

並列化の表示記号には、ループ制御文に対する表示記号とループ制御文以外の文に対する表示記号があります。

ループ制御文に対する表示記号

p	並列化されたことを示します。-Nlst=tの場合、並列化された範囲の先頭行では"pp"となります。
m	並列化された部分とされなかった部分があることを示します。
s	並列化されなかったことを示します。
空白	並列化対象ループでないことを示します。

ループ制御文以外の文に対する表示記号

p	並列化されたことを示します。(注1)
m	並列化された部分とされなかった部分があることを示します。(注2)
s	並列化されなかったことを示します。
空白	並列化対象ループでないことを示します。

注1) ループ制御文に対する表示記号が"s"の場合、コンパイラは並列化効果が得られないと判断し、並列化を行いませんが、並列化可能であったことを意味します。

注2) ループ制御文に対する表示記号が"s"の場合、コンパイラは並列化効果が得られないと判断し、並列化を行いませんが、並列化可能であった部分と不可能であった部分があったことを意味します。

5.1.2.2.3 インライン展開の表示記号

関数がインライン展開された場合、"i"を出力します。

空白の場合、インライン展開されなかったことを示します。

5.1.2.2.4 ループアンローリングによる展開数

ループアンローリングが適用された場合、展開数を出力します。

ループがフルアンローリングされた場合は、展開数ではなく"f"を出力します。

空白の場合は、ループアンローリングが適用されなかったことを示します。

5.1.2.2.5 SIMD化の表示記号

SIMD化の表示記号は、次の条件で出力します。

- ループ制御文のある行では、ループ制御文に対する表示記号を出力します。
  ループ制御文は、for文、while文、do-while文またはif-goto文です。
- 複数のループ制御文がある行では、一番左にあるループ制御文に対する表示記号を出力します。

SIMD化の表示記号には、ループ制御文に対する表示記号とループ制御文以外の文に対する表示記号があります。

ループ制御文に対する表示記号

v	SIMD化されたことを示します。
m	SIMD化された部分とされなかった部分があることを示します。
s	SIMD化されなかったことを示します。
空白	SIMD化対象ループでないことを示します。

ループ制御文以外の文に対する表示記号

v	SIMD化されたことを示します。(注1)(注3)
m	SIMD化された部分とされなかった部分があることを示します。(注2)(注3)
s	SIMD化されなかったことを示します。
空白	SIMD化対象ループでないことを示します。

注1) ループ制御文に対する表示記号が"s"の場合、コンパイラはSIMD化効果が得られないと判断し、SIMD化を行いませんが、SIMD化可能であったことを意味します。

注2) ループ制御文に対する表示記号が"s"の場合、コンパイラはSIMD化効果が得られないと判断し、SIMD化を行いませんが、SIMD化可能であった部分と不可能であった部分があったことを意味します。

注3) ループ制御文に対する表示記号が"v"または"m"の場合、SIMD命令が使用されない実行文に対しても"v"が表示されることがあります。ループをSIMD化できない要因が含まれている場合には、"s"が表示されますので、チューニングの指針としてください。

5.1.2.2.6 詳細な最適化情報

-Nlst=tオプションが指定された場合、適用された最適化の情報がソースリストに追加されて、ファイルへ出力されます。

以下の最適化情報が、ループ単位で出力されます。

自動並列化に関する情報

- Standard iteration count: *N*

  ループの繰返し数が、*N*以上のときは並列実行し、*N*未満のときは逐次実行することを意味します。

ループに対して行った最適化情報

- INTERCHANGED(nest: *nest-no*)

  ループ交換されたことを意味します。

  - *nest-no*は、ループ交換の最適化によって移動されたループの入れ子の深さを表します。
  - (nest:)は、一重化などほかの最適化が行われた場合は表示されないことがあります。

- FUSED(lines: *num1*,*num2*[,*num3*]...)

  ループ融合が行われたことを意味します。

- *num1*の行番号のループに、*num2*以降のループが融合されたことを表します。先行ループに吸収された*num2*以降の行番号のループについては、(lines:)は表示されません。

― FISSION(num: *N*)

ループ分割が行われたことを意味します。

- *N*は、ループ分割後のループ数を表します。

― COLLAPSED

ループが一重化されたことを意味します。

― SOFTWARE PIPELINING(IPC: *ipc*, ITR: *itr*, MVE: *mve*, POL: *pol*)

ループに対しソフトウェアパイプラインされたことを意味します。

- *ipc*は、ソフトウェアパイプライニングが適用されたループの、サイクル当たりの命令数(Instructions Per Cycle)の予測値です。
- *itr*は、ソフトウェアパイプライニングが適用されたループが実行時に選択されるために、必要なループの繰返し数です。翻訳時メッセージjwd8205o-iで出力される値と同じです。
- *mve*は、ソフトウェアパイプライニングによるループの展開数です。
- *pol*は、使用された命令スケジューリングアルゴリズムを示します。
  Sは小さなループ、Lは大きなループに適したアルゴリズムが使用されたことを意味します。
  それぞれ、-Kswp_policy=smallオプション、-Kswp_policy=largeオプションを指定したときに使用されるアルゴリズムです。

― SIMD

SIMD化されたことを意味します。以下のいずれかで表示されます。

- SIMD(VL: *length*[,*length*]...)

  SIMD化されたことを意味します。*length*は、1つのSIMD命令で処理する配列の要素数を表します。ループ分割された各ループで*length*が異なる場合、*length*が複数表示されます。

- SIMD(VL: AGNOSTIC; VL: *length*[,*length*]... in 128-bit)

  SVEのベクトルレジスタを特定のサイズとみなさずSIMD化されたことを意味します。*length*は、SVEのベクトルレジスタサイズを128ビットとみなして、1つのSIMD命令で処理する配列の要素数を表します。ループ分割された各ループで*length*が異なる場合、*length*が複数表示されます。

― STRIPING

ループストライピングされたことを意味します。

― MULTI-OPERATION FUNCTION

マルチ関数化された関数を含むことを意味します。

― PATTERN MATCHING(matmul)

ループをライブラリ呼出し(matmul)に変換したことを意味します。

― UNSWITCHING

ループアンスイッチングされたことを意味します。

― FULL UNROLLING

ループがフルアンローリングされたことを意味します。

― CLONE

ループがclone最適化されたことを意味します。

― LOOP VERSIONING

ループバージョニングされたことを意味します。

プリフェッチに関する情報

ハードウェアプリフェッチ

ー PREFETCH(HARD) Expected by compiler:

ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用することを期待しprefetch命令を生成しないことを示します。

ー 配列名,...

ハードウェアプリフェッチを利用することを期待した配列名を示します。コンパイラが内部的に生成した配列は"(unknown)"と表示します。

prefetch命令

ー PREFETCH(SOFT) : *N*

ループ中に存在するすべてのprefetch命令の個数を示します。
また、本表示に続けて、プリフェッチのアクセス種別ごとに、以下の形式でprefetch命令の個数と配列名を示します。

アクセス種別 : *N*
配列名 : *N*,...

- アクセス種別

- SEQUENTIAL : *N*
ループ内で使用される連続的にアクセスされる配列データに対するprefetch命令の個数を示します。

- STRIDE : *N*
ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対するprefetch命令の個数を示します。

- INDIRECT : *N*
ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対するprefetch命令の個数を示します。

- SPECIFIED : *N*
ビルトイン関数__builtin_prefetchの指定によって生成したprefetch命令の個数を示します。

- 配列名

- 配列名 : *N*,...
配列ごとのprefetch命令の個数を示します。コンパイラが内部的に生成した配列は"(unknown)"と表示します。

zfillの最適化に関する情報

ー ZFILL

zfillの最適化が適用されたことを示します。

ー 配列名,...

zfillの最適化が適用された配列名を示します。コンパイラが内部的に生成した配列は"(unknown)"と表示します。

レジスタに関する情報

ー SPILLS :

最内ループ中に存在するレジスタの退避・復元命令の個数をレジスタ種別ごとに示します。

- GENERAL : SPILL *N* FILL *N*

汎用レジスタのメモリへの退避・復元命令の個数を示します。

- SIMD&FP : SPILL *N* FILL *N*

SIMDと浮動小数点レジスタのメモリへの退避・復元命令の個数を示します。

- SCALABLE : SPILL *N* FILL *N*

拡張レジスタのメモリへの退避・復元命令の個数を示します。

- PREDICATE : SPILL *N* FILL *N*

プレディケートレジスタのメモリへの退避・復元命令の個数を示します。

5.1.2.3 ソースリストの出力例

以下にソースリストの出力例を示します。

ソースリストの出力例1

```
Compilation information
  Current directory : ディレクトリ名
  Source file       : ソースファイル名
(line-no.)(optimize)
       1              #include <stdio.h>
       2
       3              float sub(int i);
       4
       5              int main() {
       6                int i;
       7                float a[1000];
       8
       9   p     2v     for(i = 0; i < 1000; i++) {
      10   p     2v       a[i] = i;
      11   p     2v     }
      12
      13                printf("%lf¥n", a[0]);
      14
      15                i = 0;
      16           s    _loop:
      17           s    if(i < 1000) {
      18                  goto _next;
      19                }
      20           s    a[i] = i;
      21           s    i++;
      22           s    goto _loop;
      23           s    _next:
      24
      25                printf("%lf¥n", a[0]);
      26
      27   s       s    for(i = 0; i < 1000; i++) {
      28   s       s      a[i] = sub(i);
      29   s       s    }
      30
      31           f    for(i = 0; i < 2; i++) {
      32           f      a[i] = sub(i);
      33           f    }
      34
      35                printf("%lf¥n", a[0]);
      36                return 0;
      37              }
      38
      39              float sub(int j) {
      40                return (float)j;
      41              }
```

この例では、以下のことがわかります。

- 9行目のfor文によるループが並列化されました。
- 9行目のfor文によるループがSIMD化されました。
- 9行目のfor文に含まれる文がアンローリングされました。
- 16行目から23行目までのif-goto文によるループはSIMD化されませんでした。
- 27行目のfor文によるループは並列化されませんでした。
- 27行目のfor文によるループはSIMD化されませんでした。

ー 31行目のfor文に含まれる文がフルアンローリングされました。

ソースリストの出力例2

```
Compilation information
  Current directory : ディレクトリ名
  Source file       : ソースファイル名
(line-no.)(optimize)
       1              #include <stdio.h>
       2
       3              float sub(int i);
       4
       5              int main() {
       6                int i;
       7                float a[10000];
       8
                       <<< Loop-information Start >>>
                       <<<  [PARALLELIZATION]
                       <<<    Standard iteration count: 1000
                       <<<  [OPTIMIZATION]
                       <<<    SIMD(VL: 16)
                       <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 80, MVE: 5, POL: S)
                       <<< Loop-information  End >>>
       9  pp     2v     for(i = 0; i < 10000; i++) {
      10   p     2v       a[i] = i;
      11   p     2v     }
      12
      13                printf("%lf¥n", a[0]);
      14
      15                i = 0;
      16           s    _loop:
      17           s    if(i < 10000) {
      18                  goto _next;
      19                }
      20           s    a[i] = i;
      21           s    i++;
      22           s    goto _loop;
      23           s    _next:
      24
      25                printf("%lf¥n", a[0]);
      26
                       <<< Loop-information Start >>>
                       <<<  [PARALLELIZATION]
                       <<<    Standard iteration count: 1000
                       <<<  [OPTIMIZATION]
                       <<<    SIMD(VL: 16)
                       <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 80, MVE: 5, POL: S)
                       <<< Loop-information  End >>>
      27  pp       v    for(i = 0; i < 10000; i++) {
      28   pi      v      a[i] = sub(i);
      29                }
      30
                       <<< Loop-information Start >>>
                       <<<  [PARALLELIZATION]
                       <<<    Standard iteration count: 1000
                       <<<  [OPTIMIZATION]
                       <<<    SIMD(VL: 8)
                       <<< Loop-information  End >>>
      31   s       v    for(i = 0; i < 8; i++) {
      32   p       v      a[i] = i;
      33   p       v    }
      34
      35                printf("%lf¥n", a[0]);
      36                return 0;
```

```
      37            }
      38
      39            float sub(int j) {
      40              return (float)j;
      41            }
Total prefetch num: 0
```

この例では、以下のことがわかります。

- 9行目のfor文によるループは、繰返し数が1000回以上の場合、並列化されます。
- 9行目のfor文によるループは、ソフトウェアパイプライニングされました。
- 9行目のfor文によるループが並列化されました。
- 9行目のfor文によるループがSIMD化されました。
- 9行目のfor文に含まれる文がアンローリングされました。
- 16行目、17行目、20行目、21行目、22行目、23行目の文は、SIMD化されませんでした。
- 27行目のfor文によるループは、繰返し数が1000回以上の場合、並列化されます。
- 27行目のfor文によるループは、ソフトウェアパイプライニングされました。
- 27行目のfor文によるループが並列化されました。
- 27行目のfor文によるループがSIMD化されました。
- 28行目の関数呼出しがインライン展開されました。
- 31行目のfor文によるループは、繰返し数が1000回以上の場合、並列化されます。
- 31行目のfor文によるループがSIMD化されました。
- 32行目の文は並列化可能でしたが並列化されませんでした。
- 33行目の文は並列化可能でしたが並列化されませんでした。

5.1.2.4 翻訳時情報(ソースリスト)の注意事項

翻訳時オプション -Nlst=p、-Nlst=t または -Nsrc で出力される翻訳時情報(最適化情報)は、以下のようなケースで正しく出力されない場合があります。

- インライン展開が行われた場合、最適化の動作はインライン展開された場所によって異なる可能性があります。このようなケースでは、以下のように出力される可能性があります。
  - 1つのループに対して、複数の最適化メッセージが出力される。
  - 翻訳時情報(最適化情報)と反対の意味の最適化メッセージが出力される。
  - 翻訳時情報(最適化情報)が出力されない。

  また、インライン展開された関数では、prefetch数は累計されて出力されます。
- SIMD化、自動並列化、ループ融合、ループ分割など詳細な最適化情報として出力される最適化が、1つのループに対して複数適用されるケースでも、以下のように出力される可能性があります。
  - 行番号に対する最適化情報がずれる。
  - 翻訳時情報(最適化情報)と反対の意味の最適化メッセージが出力される。
- ループアンスイッチング最適化は、if文の条件が成立した場合のループと成立しない場合のループを生成しますが、その複数のループのうちいずれかの翻訳情報と最適化メッセージが出力されます。また、行番号に対する最適化情報が出力されない場合があります。
- 1行に複数のループを記述した場合、複数ループのうちいずれかの最適化情報が出力されることになります。最適化情報を出力したいループは可能な限り、同一行に記述することは避けてください。
- 条件文とgoto文で構成されるループでは詳細な最適化情報は出力されません。行番号に対する最適化情報もずれる可能性があります。

・ 翻訳時オプション-Karray_privateまたは最適化指示子array_private、first_private、last_privateにより、自動並列化されたループがあるとき、コンパイラはループを生成することがあります。翻訳情報(最適化情報)および最適化メッセージが、その生成ループに対して出力されることがあります。

5.1.3 並列化メッセージ

並列化メッセージは、並列化できなかった原因、またはどのように並列化および最適化されたかを示します。

-Koptmsg=2オプションを-Kparallelオプションと同時に指定することで、並列化メッセージを標準エラーに出力することができます。以下に、出力例を示します。

並列化メッセージの出力例

```
Parallelization messages
  jwd5001p-i "ソースファイル名", line n1: ループ制御変数'i'のループを並列化しました。
  jwd6001s-i "ソースファイル名", line n1: ループ制御変数'i'のループをSIMD化しました。
  jwd8202o-i "ソースファイル名", line n1: このループを展開数8回でループアンローリングしました。
  jwd5122p-i "ソースファイル名", line n2: ループ内に自動並列化の制約となる関数呼出しが存在します。
```

5.1.4 統計情報

以下のオプションによって、ソースファイルの統計情報を出力することができます。

・ -Nlst[={p|t}]

  ソースリストと統計情報をファイルに出力します。

・ -Nlst_out=*file*

  ソースリストと統計情報を出力するファイル名を指定します。-Nlst=pオプションも有効になります。

・ -Nsta

  統計情報を標準出力に出力します。

  ただし、-Nlstオプションまたは-Nlst_out=*file*オプションが同時に指定された場合、統計情報はファイルに出力されます。

統計情報では、有効になった最適化関連オプションだけを出力します。

出力されていない最適化関連オプションは無効にされています。

5.1.4.1 出力形式

統計情報の出力形式を以下に示します。

統計情報の出力形式

```
Statistics information
  Option information
    Profile file         : 翻訳時プロフィルファイルで指定されたオプションの一覧（注）
    Environment variable : 翻訳時オプション設定用の環境変数で指定されたオプションの一覧（注）
    Command line options : 翻訳コマンドで指定されたオプションの一覧
    Effective options    : 有効になった最適化関連オプションの一覧
```

注) オプションは「モード共通 モード固有」の順序で、1行にまとめて出力されます。

以下の場合、Profile file行およびEnvironment variable行は出力されません。

・ 翻訳時プロフィルファイルが存在しない場合および翻訳時プロフィルファイルのファイルサイズが0の場合

・ 翻訳時オプション設定用の環境変数が存在しない場合

5.1.4.2 出力対象外のオプション

統計情報では、最適化関連オプションを出力対象としています。

ld関連オプションなどは出力対象外です。

出力対象外のオプション

```
-# / -### / -A- / -Aname[(tokens)] / -B{dynamic|static} / -C / -Dname[=tokens] / -E / -H / -Idir / -Ldir / -
Nlst_out=file / -P / -S / -Uname / -Wtool,arg1[,arg2]... / -Yitem,dir / -V / -c / {-help|--help} / -lname / -o
pathname / -shared
```

5.1.4.3 コンパイラが解釈して出力するオプション

以下のオプションは、コンパイラが解釈した内容で出力されます。

コンパイラが解釈する内容については、“2.2.2.5 -Kオプション”をお読みください。

- -Kfast
- -Knoprefetch
- -Kvisimpact

5.2 実行時情報

ここでは、C言語で記述されたプログラムの実行において、本処理系が出力する情報について説明します。

5.2.1 実行時メッセージ

本処理系が実行時に出力するメッセージの詳細については、以下のマニュアルをお読みください。

- “jwe”から始まるメッセージについては、“Fortran/C/C++実行時メッセージ”をお読みください。
- 上記以外の実行時メッセージについては、システムのマニュアルをお読みください。

5.2.2 トレースバック情報

環境変数FLIB_C_MESSAGEにNO_MESSAGEが定義されていない、または、-NRtrapオプションが指定されている場合には、エラーまたは割込み事象の発生時にトレースバックマップを標準エラーへ出力します。この機能により、main関数からエラーが発生したプログラム単位までの呼出し関係を知ることができます。

トレースバックマップ情報は、スタートアップで獲得されたヒープ領域に保持されているデバッグ情報から取得します。情報が出力されないとき、環境変数FLIB_TRACEBACK_MEM_SIZEでヒープ領域の大きさを省略値より大きくすることにより出力される場合があります。なお、ヒープ領域の不足が原因で情報が取得できない場合は、診断メッセージjwe1655i-iが出力されます。この診断メッセージはエラー処理中に出力されるため、診断メッセージが出力される順序が正しくない場合があります。

環境変数FLIB_C_MESSAGEおよびFLIB_TRACEBACK_MEM_SIZEの詳細については、“2.5.1 実行時環境変数”をお読みください。

-NRtrapオプションの詳細については、“2.2.2.6 -Nオプション”をお読みください。

トレースバック情報の詳細については、“Fortran使用手引書”をお読みください。

第6章 言語仕様

本章では、本処理系で拡張した言語仕様、および言語仕様のサポート状況について説明します。

なお、本章では下記の構文表記を使用します。

- 構文要素(非終端記号)は、英数字を含む日本語で示し、リテラル語および文字集合の要素(終端記号)は英数字および特殊記号で示します。
- 非終端記号に続くコロン(:)は、その後に非終端記号の定義があることを示します。
- 選択可能な定義は、“次のいずれか”と前置きしている場合を除き、別々の行に表示します。
- 省略可能な記号は、添字“*opt*”で示します。したがって、{式*opt*}は、中括弧が省略可能な式を囲んでいることを示します。

6.1 本処理系で拡張した言語仕様

6.1.1 long long型

signed char、short int、intおよびlong intの4つの符号付き整数型に加えて、long long int型が追加されています。

接尾語

整数接尾語:

- 符号なし接尾語 長語接尾語*opt*
- 長語接尾語 符号なし接尾語*opt*
- 符号なし接尾語 長長語接尾語*opt*
- 長長語接尾語 符号なし接尾語*opt*

符号なし接尾語:

以下のいずれか

- u
- U

長長語接尾語:

- 長語接尾語 長語接尾語

長語接尾語:

以下のいずれか

- l
- L

意味規則

整数定数の型は、次の並びのうちでその値を表現できる最初の型とします。

接尾語なしの10進数:

- int
- long int
- unsigned long int
- long long int
- unsigned long long int

接尾語なしの8進数または16進数:

- int
- unsigned int
- long int
- unsigned long int
- long long int
- unsigned long long int

文字uまたはUが接尾語として付く場合:

- unsigned int
- unsigned long int
- unsigned long long int

文字lまたはLが接尾語として付く場合:

- long int
- unsigned long int
- long long int
- unsigned long long int

文字uまたはU、および文字lまたはLの両方が接尾語として付く場合:

- unsigned long int
- unsigned long long int

文字lまたはL、および文字lまたはLの両方が接尾語として付く場合:

- long long int
- unsigned long long int

型指定子としての制約

型指定子の各並びは、以下の組のいずれか1つに属していなければなりません。1つの項目に2つ以上の組がある場合は、コンマで区切っています。型指定子はどんな順序で記述してもよく、また、ほかの宣言指定子と混在して記述することもできます。

- long long, signed long long, long long int, signed long long int
- unsigned long long, unsigned long long int

記述例

```
long long int lli=10LL;
unsigned long long int ulli=10ULL;
```

6.1.2 pragma指令

pragma指令は、処理系への動作を指示します。

意味

#pragma ident 文字列リテラル 改行

文字列リテラル内の文字列を、オブジェクトプログラムの中に注釈として付加することを、処理系に指示します。これは、#ident指令と同じ意味を持つ#pragma指令です。一般に、オブジェクトプログラム中にバージョン管理情報を付加するために利用します。

#pragma weak 識別子 改行

識別子をウィーク・シンボルとして定義することを、処理系に指示します。

#pragma weak 識別子1=識別子2 改行

識別子1を識別子2と同じ値および型を持つウィーク・シンボルとして定義することを、処理系に指示します。

#pragma unknown_control_flow (識別子の並び) 改行

上記の形式の前処理指令は、識別子で指定された関数が処理系で認識できない制御の流れを持つことを、処理系に指示します。

#pragma redefine_extname <旧関数名> <新関数名> 改行

オブジェクトコードの中の<旧関数名>である外部参照名をすべて<新関数名>に置き換えることを、処理系に指示します。

6.1.3 ident指令

ident指令は、Cプログラムに注釈を付加することを指示します。

意味

#ident 文字列リテラル 改行

指定した文字列リテラルがオブジェクトプログラムの".commentセクション"に置かれます。".commentセクション"は、プログラムの実行時に記憶域にローディングされないセクションです。

#ident 前処理字句 改行

指定した前処理字句が通常のテキストとして処理され、現在マクロ名として定義している各識別子は、その置換えの並びで置き換えられます。置換え後のすべての#ident指令は、前の形式と一致しなければなりません。

6.1.4 assert指令

assert指令は、プレディケート名を定義し、アサーション字句をプレディケート名に関連付けることを指示します。

意味

#assert 識別子 アサーション指定子*opt* 改行

#assert指令に続く識別子がプレディケート名であると定義します。また、プレディケート名に続くアサーション指定子の識別子の並びがアサーション字句であると定義します。

#assert指令は、プレディケート名を定義し、アサーション字句をプレディケート名に関連付けます。

識別子の並びのない#assert指令は、対応する#unassert指令を記述しているところまで有効です。

#演算式の各識別子

#演算式の各識別子は、#assert指令で指定したプレディケート名とアサーション字句にそれぞれ相当します。アサーション字句がプレディケート名に関連付けられている場合だけ1となり、その他の場合、0となります。例えば、#system(unix)は1です。

以下のアサーションは、処理系によってあらかじめ定義されています(既定義アサーション)。

- #assert system(unix)

記述例

```
#assert langlevel(ansi)
#assert langlevel(sysv)
```

上記の指令は、プレディケート名langlevelを定義し、アサーション字句ansiとsysvをプレディケート名に関連付けます。

```
#assert float
```

上記の指令は、プレディケート名floatを定義します。アサーション字句は持ちません。

6.1.5 unassert指令

unassert指令は、プレディケート名とアサーション字句の関連付けを取り消すことを指示します。

意味

#unassert 識別子 アサーション指定子*opt* 改行

アサーション字句の関連付けを取り消します。識別子の並び(アサーション指定子)がない#unassert指令は、プレディケート名のアサーション字句の関連付けをすべて取り消します。

指定した識別子がプレディケート名でない場合、#unassert指令は無視されます。また、関連付けていないアサーション字句を指定した場合も、#unassert指令は無視されます。

6.1.6 あらかじめ定義されたマクロ名

あらかじめ定義されたマクロ(以降、既定義マクロと呼びます)のいくつかは、翻訳時オプション-std=*level*の指定に応じて値が変わります。

以下の表に、既定義マクロの値を一部示します。すべての既定義マクロを表示するためには、翻訳時オプション-Eおよび-dMを指定してください。

表6.1 翻訳時オプション-std=levelの指定に応じて値が変化する既定義マクロ

識別子	翻訳時オプション-std=*level*に指定された言語仕様のレベル					
	c89	gnu89	c99	gnu99	c11	gnu11
__STDC_NO_ATOMICS__	-		-		1	
__STDC_NO_THREADS__	-		-		1	
__STDC_VERSION__	-		199901L		201112L	
__STRICT_ANSI__	1	-	1	-	1	-
linux	-	1	-	1	-	1
unix	-	1	-	1	-	1

-: 定義なし

表6.2 値が変化しない既定義マクロ

識別子	値
_LP64	1
_OPENMP (注1)	201107
_REENTRANT (注2)	1
__ARM_ARCH	8
__ASSEMBLER__ (注3)	1
__DATE__	翻訳日付(asctime関数の形式)
__ELF__	1
__EXCEPTIONS (注4)	1
__FCC_major__	コンパイラのメジャーバージョン番号
__FCC_minor__	コンパイラのマイナーバージョン番号
__FCC_patchlevel__	コンパイラのパッチレベル番号
__FCC_version__	コンパイラのバージョンを表す文字列
__FILE__	ソースファイル名
__FUJITSU	1
__GNUC__	6
__GNUC_MINOR__	1
__GNUC_PATCHLEVEL__	0
__LINE__	ソースファイルの行番号

識別子	値
__LP64__	1
__MT__ (注5)	1
__OPTIMIZE__ (注6)	1
__PIC__	1 (注7)
	2 (注8)
__PRAGMA_REDEFINE_EXTNAME	1
__PTRDIFF_TYPE__	long int
__SIGNED_CHARS__	1 (注9)
	- (注10)
__SIZE_TYPE__	long unsigned int
__STDC__	1
__STDC_HOSTED__	1
__TIME__	翻訳時刻(asctime関数の形式)
__USER_LABEL_PREFIX__	空文字列
__WCHAR_TYPE__	int (注11)
__aarch64__	1
__linux	1
__linux__	1
__pic__	1 (注7)
	2 (注8)
__unix	1
__unix__	1

-: 定義なし

注1) 翻訳時オプション-Kopenmpまたは-fopenmp有効時

注2) 翻訳時オプション-mtまたは-pthread有効時

注3) 翻訳時オプション-x assembler-with-cpp有効時

注4) 翻訳時オプション-Nexceptionsまたは-fexceptions有効時

注5) 翻訳時オプション-mt有効時

注6) 翻訳時オプション-O1以上有効時

注7) 翻訳時オプション-fpicまたは-fpie有効時

注8) 翻訳時オプション-fPICまたは-fPIE有効時

注9) 翻訳時オプション-fsigned-char有効時

注10) 翻訳時オプション-funsigned-char有効時

注11) clangモードと値が異なるため、clangモードのオブジェクトと結合している場合に意図しない動作をする可能性があります。

6.2 言語仕様のサポート状況

本処理系では、下記の規格をサポートしています。

- C89規格

・ C99規格

・ C11規格

6.2.1 C99規格の言語仕様

ここでは、本処理系におけるC99仕様のサポート状況について説明します。

本処理系では、以下の項目がサポートされていません。

配列宣言子内の型修飾子リスト・staticの記述

配列宣言子内の型修飾子リスト(restrictを除く)およびstaticの記述は、文法エラーになります。

#pragma STDCプラグマ

#pragma STDCプラグマはサポートしていません。

単純文字列リテラルとワイド文字列リテラルの文字列結合

隣り合った単純文字列リテラルとワイド文字列リテラルの文字列結合はサポートしていません。

C99規格では、単純文字列リテラルとワイド文字列リテラルの文字列結合が翻訳段階6で行われ、その後、翻訳段階7でmbstowcsによってマルチバイト並びからワイド文字列へ変換が行われるとされていますが、これは既存のソースプログラムのポータビリティを損なうものです。C89規格ではこの使い方はエラーでした。C11規格では、結合する文字列のいずれかが接頭辞を持つ場合、すべての文字列が同じ接頭辞を持つものとして扱われます。

浮動小数点環境

浮動小数点環境を扱う機能(fenv.h)はサポートしていません。

_Imaginary

型指定子_Imaginary はサポートしていません。

6.2.2 C11規格の言語仕様

本処理系におけるC11仕様のサポート状況は以下のとおりです。

表6.3 C11仕様のサポート状況

Language Features	サポート
_Alignas, _Alignof, max_align_t, stdalign.h	○
_Atomic, stdatomic.h	○ (注)
_Generic	○
_Noreturn, stdnoreturn.h	○
_Static_assert	○
_Thread_local	○
Anonymous struct and union	○
Typedef redefinition	○
New macros in float.h	○
Macros for Complex values	○
Unicode strings	○

○:サポート

×:未サポート

注) -Nclangオプションが有効な場合にサポートします。-Nnoclangオプションが有効な場合は、既定義マクロ__STDC_NO_ATOMICS__を定義します。

第7章 言語およびtrad/clang間結合における注意事項

本章では、プログラム言語やモード(trad/clang)が異なるオブジェクトプログラムを結合する場合の注意事項を説明します。

7.1 結合時の翻訳コマンドと必須オプション

Fortran、C言語、およびC++言語の翻訳コマンドには、クロスコンパイラとネイティブコンパイラの2種類があります。さらに、プログラムがMPIライブラリを利用しているかどうかにより、下表のように使い分けます。

以降の説明では、クロスコンパイラの翻訳コマンド(MPIライブラリの利用なし)を用います。MPIライブラリやネイティブコンパイラを利用する場合は、下表に従い翻訳コマンドを読み替えてください。

プログラム言語	MPIライブラリの利用	クロスコンパイラ	ネイティブコンパイラ
Fortran	なし	frtpx	frt
	あり	mpifrtpx	mpifrt
C言語	なし	fccpx	fcc
	あり	mpifccpx	mpifcc
C++言語	なし	FCCpx	FCC
	あり	mpiFCCpx	mpiFCC

Fortran、C言語 tradモード、C言語 clangモード、C++言語 tradモード、およびC++言語 clangモードのオブジェクトプログラムを組み合わせて結合することができます。

オブジェクトプログラムの組合せによっては、結合時に使用できない翻訳コマンドがあります。オブジェクトプログラムの組合せと、結合時に使用する翻訳コマンドおよび必須オプションは、“表7.1 オブジェクトプログラムの組合せと結合時に使用する翻訳コマンド/必須オプション”を参照してください。翻訳コマンドとオプションの詳細は、各言語の使用手引書を参照してください。

表7.1 オブジェクトプログラムの組合せと結合時に使用する翻訳コマンド/必須オプション

結合するオブジェクトプログラム						COARRAY機能	翻訳コマンドと必須オプション
				C++ clang			
Fortran	C trad	C clang	C++ trad	C++標準ライブラリにlibc++使用	C++標準ライブラリにlibstdc++使用		
○	○	–	–	–	–	なし	frtpx または fccpx --linkfortran
						あり	frtpx -Ncoarray または fccpx --linkcoarray
○	–	○	–	–	–	なし	frtpx または fccpx -Nclang --linkfortran
						あり	frtpx -Ncoarray または fccpx -Nclang --linkcoarray
○	–	–	○	–	–	なし	frtpx --linkstl=libfjc++ または FCCpx --linkfortran

結合するオブジェクトプログラム						COARRAY機能	翻訳コマンドと必須オプション
Fortran	C trad	C clang	C++ trad	C++ clang			
				C++標準ライブラリにlibc++使用	C++標準ライブラリにlibstdc++使用		
						あり	frtpx --linkstl=libfjc++ -Ncoarray または FCCpx --linkcoarray
○	―	―	―	○	―	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray
				―	○	なし	frtpx --linkstl=libstdc++ または FCCpx -Nclang --linkfortran
						あり	frtpx --linkstl=libstdc++ -Ncoarray または FCCpx -Nclang --linkcoarray
―	○	○	―	―	―	―	fccpx -Nclang
―	○	―	○	―	―	―	FCCpx
―	○	―	―	○	―	―	FCCpx -Nclang
				―	○		FCCpx -Nclang -stdlib=libstdc++
―	―	○	○	―	―	―	FCCpx
―	―	○	―	○	―	―	FCCpx -Nclang
				―	○		FCCpx -Nclang -stdlib=libstdc++
―	―	―	○	○	―	―	FCCpx -Nclang -stdlib=libc++ (“注1”参照)
				―	○		結合できません。 (“注2”参照)
○	○	○	―	―	―	なし	frtpx または fccpx -Nclang --linkfortran
						あり	frtpx -Ncoarray または fccpx -Nclang --linkcoarray
○	○	―	○	―	―	なし	frtpx --linkstl=libfjc++ または FCCpx --linkfortran
						あり	frtpx --linkstl=libfjc++ -Ncoarray または FCCpx --linkcoarray
○	○	―	―	○	―	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++

結合するオブジェクトプログラム						COARRAY機能	翻訳コマンドと必須オプション
Fortran	C trad	C clang	C++ trad	C++ clang			
				C++標準ライブラリにlibc++使用	C++標準ライブラリにlibstdc++使用		
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray
				−	○	なし	frtpx --linkstl=libstdc++ または FCCpx -Nclang --linkfortran
						あり	frtpx --linkstl=libstdc++ -Ncoarray または FCCpx -Nclang --linkcoarray
○	−	○	○	−	−	なし	frtpx --linkstl=libfjc++ または FCCpx --linkfortran
						あり	frtpx --linkstl=libfjc++ -Ncoarray または FCCpx --linkcoarray
○	−	○	−	○	−	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray
				−	○	なし	frtpx --linkstl=libstdc++ または FCCpx -Nclang --linkfortran
						あり	frtpx --linkstl=libstdc++ -Ncoarray または FCCpx -Nclang --linkcoarray
○	−	−	○	○	−	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++ (“注1”参照)
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray (“注1”参照)
				−	○	なし	結合できません。 (“注2”参照)
						あり	
−	○	○	○	−	−	−	FCCpx
−	○	○	−	○	−	−	FCCpx -Nclang -stdlib=libc++
				−	○		FCCpx -Nclang -stdlib=libstdc++
−	○	−	○	○	−	−	FCCpx -Nclang -stdlib=libc++ (“注1”参照)

結合するオブジェクトプログラム						COARRAY機能	翻訳コマンドと必須オプション
Fortran	C trad	C clang	C++ trad	C++ clang			
				C++標準ライブラリにlibc++使用	C++標準ライブラリにlibstdc++使用		
				－	○		結合できません。 (“注2”参照)
－	－	○	○	○	－	－	FCCpx -Nclang -stdlib=libc++ (“注1”参照)
				－	○		結合できません。 (“注2”参照)
○	○	○	○	－	－	なし	frtpx --linkstl=libfjc++ または FCCpx --linkfortran
						あり	frtpx --linkstl=libfjc++ -Ncoarray または FCCpx --linkcoarray
○	○	○	－	○	－	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray
				－	○	なし	frtpx --linkstl=libstdc++ または FCCpx -Nclang --linkfortran
						あり	frtpx --linkstl=libstdc++ -Ncoarray または FCCpx -Nclang --linkcoarray
○	○	－	○	○	－	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++ (“注1”参照)
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray (“注1”参照)
				－	○	なし あり	結合できません。 (“注2”参照)
○	－	○	○	○	－	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++ (“注1”参照)
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray (“注1”参照)

結合するオブジェクトプログラム						COARRAY機能	翻訳コマンドと必須オプション
Fortran	C trad	C clang	C++ trad	C++ clang			
				C++標準ライブラリにlibc++使用	C++標準ライブラリにlibstdc++使用		
				－	○	なし	結合できません。 (“注2”参照)
						あり	
－	○	○	○	○	－	－	FCCpx -Nclang -stdlib=libc++ (“注1”参照)
				－	○		結合できません。 (“注2” 参照)
○	○	○	○	○	－	なし	frtpx --linkstl=libc++ または FCCpx -Nclang --linkfortran -stdlib=libc++ (“注1”参照)
						あり	frtpx --linkstl=libc++ -Ncoarray または FCCpx -Nclang -stdlib=libc++ --linkcoarray (“注1”参照)
				－	○	なし	結合できません。 (“注2”参照)
						あり	

7.1.1 C++ tradモードとC++ clangモードのオブジェクト結合

C++ tradモードで作成したオブジェクトプログラムとC++ clangモードで作成したオブジェクトプログラムの結合可否は、翻訳時に使用したC++標準ライブラリに依存します。

C++ tradモード	C++ clangモード	翻訳コマンドと必須オプション
(libc++使用のみ)	C++標準ライブラリにlibc++使用	FCCpx -Nclang -stdlib=libc++ (“注1”参照)
	C++標準ライブラリにlibstdc++使用	結合できません。 (“注2”参照)

注1

以下の条件をすべて満たす場合、動作が不定になる可能性があり、動作保証できません。

1. C++ tradモードで作成したオブジェクトプログラムと、C++ clangモードで作成したオブジェクトとで、関数間のインタフェースを持つ。
2. 条件1の関数の引数または復帰値がクラス型である。

注2

C++ clangモードが使用するC++標準ライブラリのデフォルトはlibstdc++です。

結合させるには、オブジェクト作成時に-stdlib=libc++オプションを指定し、C++標準ライブラリにlibc++を使用してください。

7.1.2 C++ clangモード同士のオブジェクト結合

C++ clangモードで作成したオブジェクトプログラム同士の結合可否は、翻訳時に使用したC++標準ライブラリに依存します。

翻訳時に使用したC++標準ライブラリ		翻訳コマンドと必須オプション
libc++	libc++	FCCpx -Nclang -stdlib=libc++

翻訳時に使用したC++標準ライブラリ		翻訳コマンドと必須オプション
libc++	libstdc++	結合できません。
libstdc++	libstdc++	FCCpx -Nclang -stdlib=libstdc++ (なお、-stdlib=libstdc++オプションはデフォルトなので省略可能)

7.1.3 MPIオブジェクトプログラム結合後のプロファイラ利用

結合時に以下のオプションも指定してください。

- frtpxコマンドで結合する場合： -lfjprofmpiオプション
- fccpxまたはFCCpxコマンドで結合する場合： -lfjprofmpifオプション

7.1.4 -fltoオプションを指定して作成したclangモードのオブジェクト

“-Nclang -flto”オプションを指定して作成したオブジェクトは、LLVM内部で使用される形式(通常のオブジェクトと異なる形式)になるため、clangモードでしか結合できません。

言語間結合で“-Nclang -flto”オプションを指定して作成したオブジェクトを含む場合は、結合時に“-Nclang -flto”オプションを指定してください。

7.2 C++言語との結合

以下の点に注意してください。

- C++で記述された関数からC言語で記述された関数を呼び出す場合、C++側で、呼び出される関数をC言語へのリンケージ指定(extern"C")付きで宣言する必要があります。
- C言語で記述された関数からC++で記述された関数を呼び出す場合、C++側で、呼び出される関数をC言語へのリンケージ指定(extern"C")付きで宣言する必要があります。
- C言語へのリンケージ指定(extern"C")付きで宣言された関数の引数および復帰値は、C言語で許されている型が指定できます。

以下に呼出しの例を示します。

例1: C++のプログラムに最初に制御を渡す場合

C++プログラム: cplusmain.cc

```
#include <iostream>

extern "C" {
    void cfunc(short int);
    int cpfunc(int);
}

int cpfunc(int i) {
    std::cout << "C++: cpfunc called, i=" << i << std::endl;
    return i;
}

int main() {
    std::cout << "C++ main()" << std::endl;
    cfunc(10);
    return 0;
}
```

Cプログラム: csub.c

```
#include <stdio.h>

int cpfunc(int);
```

```
void cfunc(short int si) {
    printf("C:   cfunc called, si=%d¥n", si);
    cpfunc(si);
}
```

翻訳

```
$ fccpx -c csub.c
$ FCCpx csub.o cplusmain.cc
```

実行

```
$ ./a.out
```

結果

```
C++ main()
C:   cfunc called, si=10
C++: cpfunc called, i=10
```

例2: C言語のプログラムに最初に制御を渡す場合

C++プログラム: cplussub.cc

```
#include <iostream>

extern "C" {
    void cfunc(short int);
    int cpfunc(int);
}

int cpfunc(int i) {
    std::cout << "C++: cpfunc called, i=" << i << std::endl;
    cfunc(i);
    return i;
}
```

Cプログラム: cmain.c

```
#include <stdio.h>

int cpfunc(int);

int main() {
    printf("C main()¥n");
    cpfunc(10);
    return 0;
}

void cfunc(short int si) {
    printf("C:   cfunc called, si=%d¥n", si);
}
```

翻訳

```
$ fccpx -c cmain.c
$ FCCpx cmain.o cplussub.cc
```

実行

```
$ ./a.out
```

結果

```
C main()
C++: cpfunc called, i=10
C:   cfunc called, si=10
```

7.3 Fortranとの結合

以下の点に注意してください。

- 最初に制御を渡すプログラムがC言語で記述される場合、その関数名はMAIN__またはmainでなければなりません。詳細については、“Fortran使用手引書”をお読みください。
- 次の場合には、実行可能プログラムの復帰値にFortranの復帰コードが設定されます。Fortranの復帰コードの詳細については、“Fortran使用手引書”をお読みください。

  ー 最初に制御を渡すプログラムがFortranで記述されている場合
- 手続き呼出しおよびデータの受渡しについては、“Fortran使用手引書”をお読みください。
- Cから呼び出すFortranの関数名およびFortranから呼び出されるCの関数名の最後には、_を付けてください。

以下に呼出しの例を示します。

COARRAY仕様を利用する場合は、--linkfortranオプションの代わりに--linkcoarrayオプションを指定してください。COARRAY仕様については、“Fortran使用手引書 別冊 COARRAY”をお読みください。

例1: Fortranのプログラムに最初に制御を渡す場合

Fortranプログラム: fortranmain.f95

```
print *,"Fortran: main program"
call cfunc(10)
end
```

Cプログラム: csub.c

```
#include <stdio.h>

int cfunc_(int *p) {
    printf(" C: cfunc_ called, *p=%d¥n", *p);
    return *p;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -c csub.c
$ frtpx csub.o fortranmain.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortranmain.f95
$ fccpx --linkfortran csub.c fortranmain.o
```

実行

```
$ ./a.out
```

結果

```
 Fortran: main program
 C: cfunc_ called, *p=10
```

例2: C言語のプログラムに最初に制御を渡す場合(MAIN__)

Fortranプログラム: fortransub.f95

```
integer function func(x)
integer*4 x
print *,"Fortran: func() called"
call cfunc(x)
func=x
end
```

Cプログラム: cmain.c

```
#include <stdio.h>

int func_(int *);

int cfunc_(int *p) {
    printf(" C: cfunc_ called. *p=%d¥n", *p);
    return *p;
}

int MAIN__() {
    int i=10;
    printf(" C MAIN__()¥n");
    func_(&i);
    return 0;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -c cmain.c
$ frtpx cmain.o fortransub.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortransub.f95
$ fccpx --linkfortran cmain.c fortransub.o
```

実行

```
$ ./a.out
```

結果

```
 C MAIN__()
 Fortran: func() called
 C: cfunc_ called. *p=10
```

例3: C言語のプログラムに最初に制御を渡す場合(main)

Fortranプログラム: fortransub.f95

```
integer function func(x)
integer*4 x
print *,"Fortran: func() called"
call cfunc(x)
func=x
end
```

Cプログラム: cmain.c

```
#include <stdio.h>
```

```
int func_(int *);

int cfunc_(int *p) {
    printf(" C: cfunc_ called. *p=%d¥n", *p);
    return *p;
}

int main() {
    int i=10;
    printf(" C main()¥n");
    func_(&i);
    return 0;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -c cmain.c
$ frtpx -mlcmain=main cmain.o fortransub.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortransub.f95
$ fccpx --linkfortran cmain.c fortransub.o
```

実行

```
$ ./a.out
```

結果

```
 C main()
 Fortran: func() called
 C: cfunc_ called. *p=10
```

第8章 プログラムのデバッグ

実行可能プログラムでエラーが発生したとき、異常終了したとき、または利用者が目的とした結果を得られないときは、その原因を追求しプログラムの修正を行う必要があります。

本章では、原因を見出す手段として、本処理系が用意している各種のデバッグ機能について説明します。

8.1 デバッグのための検査機能

ここでは、Cプログラムのデバッグのための検査機能について説明します。

-Nquickdbgオプションは、プログラムの実行時にデバッグを行うための検査機能です。

-Nquickdbgオプションの検査機能には、以下の検査があります。

- 配列範囲の検査("8.1.1 配列範囲の検査(subchk機能)"を参照)
- ヒープメモリの検査("8.1.2 ヒープメモリの検査(heapchk機能)"を参照)

エラー検出時の診断メッセージを出力するオプションには、-Nquickdbg=inf_detailまたは-Nquickdbg=inf_simpleオプションがあります。これらのオプションは、配列範囲の検査時における実行性能への影響と診断メッセージ内容を制御します。

-Nquickdbg=inf_detailオプションが有効な場合、エラー検出時に、エラーの原因を識別するためのメッセージとエラーが発生した行番号に加えて、エラー発生の原因となった変数名などの情報を含む診断メッセージを出力して、実行を継続します。

-Nquickdbg=inf_simpleオプションが有効な場合、エラー検出時に、エラーが発生したことを通知するメッセージとエラーが発生した行番号を含む診断メッセージを出力して、実行を終了します。ただし、-Nnolineオプションを指定した場合、行番号の値は保証されません。

-Nquickdbg=inf_simpleオプションが有効な場合は、-Nquickdbg=inf_detailオプションが有効な場合と比較して、診断メッセージに出力される情報を限定することにより、検査機能による実行性能への影響が軽減されます。

ただし、ヒープメモリの検査においては、-Nquickdbg=inf_detailオプション、-Nquickdbg=inf_simpleオプションのどちらを指定した場合にも、エラーの原因を識別するためのメッセージとエラーが発生した行番号などを含む診断メッセージを出力して、実行を継続します。実行性能は双方とも同等となります。

-Nquickdbgオプションについては、"2.2.2.6 -Nオプション"をお読みください。

8.1.1 配列範囲の検査(subchk機能)

-Nquickdbg=subchkオプションを指定すると、配列の引用が、配列の宣言範囲内に対して行われているか検査されます。配列の宣言範囲外を引用した場合、診断メッセージ(-Nquickdbg=inf_detailの場合jwe1601i-w、-Nquickdbg=inf_simpleの場合jwe1606i-u)が出力されます。

ただし、以下の場合については、検査されません。

- 関数の仮引数として宣言された配列
- 翻訳時に大きさが確定しない配列(可変長配列、フレキシブル配列、要素数0の配列要素)
- 配列の引用時に、添字として副作用(side effect)を引き起こす可能性がある式が指定されている
- 配列の引用時に、添字として式中の複合文(注)が指定されている

  注) 式中の複合文(Statements and Declarations in Expressions)は、GNU C拡張仕様です。

8.1.2 ヒープメモリの検査(heapchk機能)

-Nquickdbg=heapchkオプションを指定すると、malloc関数、calloc関数、realloc関数、valloc関数、posix_memalign関数、memalign関数で割り当てたメモリに対して、以下の検査が行われます。

- ヒープメモリの解放時に、メモリの解放検査および領域外書き込みの検査が行われます。
- プログラムの終了時にメモリリーク検査が行われます。

検査項目について、以下に説明します。

注意

malloc関数、calloc関数、realloc関数、valloc関数、posix_memalign関数、memalign関数以外を使用して割り当てたメモリを解放した場合、診断メッセージjwe1602i-wまたはjwe1603i-wが出力されることがあります。

8.1.2.1 メモリの解放検査

ヒープメモリの解放時に、解放するメモリについて次の検査が行われます。

- 解放済みメモリに対する2重解放
- 不当メモリの解放

解放済みメモリに対する2重解放が検出された場合、診断メッセージ(jwe1602i-w)が出力されます。不当メモリの解放が検出された場合、診断メッセージ(jwe1603i-w)が出力されます。

8.1.2.2 ヒープメモリの領域外書き込み検査

ヒープメモリの解放時に、ヒープオーバランによる領域外書き込みが発生していないか検査されます。

ヒープメモリの領域外書き込みの検査では、獲得したヒープメモリの後方に続く8byteの領域に対して、領域外書き込みの検査が行われます。領域外書き込みが検出された場合、診断メッセージ(jwe1604i-w)が出力されます。

ただし、領域外書き込みが生じた場合でも、ヒープメモリの後方に続く8byteの領域が次の値を持つ場合には、領域外書き込みは検出されません。

- 0x8b8b8b8b8b8b8b8b

上記の値は、環境変数FLIB_HEAPCHK_VALUEを指定することで変更できます。

```
FLIB_HEAPCHK_VALUE hex
```

ヒープメモリの領域外書き込み検査における検査値を変更します。*hex*には16進表記の値が指定可能です。有効となる値は00≦*hex*≦FFの範囲になります。環境変数が指定されていない場合、または、*hex*の値が無効である場合、本処理系の省略値が有効となります。省略値は8Bです。

以下に環境変数FLIB_HEAPCHK_VALUEの指定例を示します。

```
$ FLIB_HEAPCHK_VALUE=8C
$ export FLIB_HEAPCHK_VALUE
```

この指定により、領域外書き込み検査における検査値として、0x8c8c8c8c8c8c8c8cが適用されます。

8.1.2.3 メモリリーク検査

プログラムの実行終了時に、ヒープメモリが未解放のまま残されていないか検査されます。未解放のヒープメモリが存在している場合、診断メッセージ(jwe1605i-w)が出力されます。

8.2 異常終了プログラムのデバッグ

主プログラムの翻訳時およびリンク時に-NRtrapオプションを指定して作成した実行ファイルにおいて、プログラムの実行中に異常終了事象が発生した場合、シグナルが捕捉され異常終了の原因追及の手助けとなる情報が出力されます。

-NRnotrapオプションが有効な場合、異常終了事象の原因追及の手助けとなる情報は出力されません。

-NRtrapオプションおよび-NRnotrapオプションについては、“2.2.2.6 -Nオプション”をお読みください。

8.2.1 異常終了の原因

捕捉するシグナル(注)とそれに対応するシグナルコードを以下に示します。

注) Armアーキテクチャである富士通製CPU A64FXでサポートしているシグナルだけが対象です。そのため、サポートしていないシグナルは、-NRtrapオプションを有効にしても検出できません。-NRtrapオプションについては、“2.2.2.6 -Nオプション”をお読みください。

表8.1 捕捉シグナルと対応するシグナルコード

シグナル番号	シグナル番号の意味	シグナルコード		シグナルコードの意味
SIGILL(04)	不当な命令の実行	1	ILL_ILLOPC	不正な命令コード
		2	ILL_ILLOPN	不正なオペランド
		3	ILL_ILLADR	不正なアドレッシングモード
		4	ILL_ILLTRP	不正なトラップ
		5	ILL_PRVOPC	不正な特権命令コード
		6	ILL_PRVREG	不正な特権レジスタ
		7	ILL_COPROC	コプロセッサエラー
		8	ILL_BADSTK	内部スタックエラー
SIGFPE(08)	浮動小数点例外 (注1)	3	FPE_FLTDIV	浮動小数点ゼロ除算
		4	FPE_FLTOVF	浮動小数点オーバフロー
		5	FPE_FLTUND	浮動小数点アンダフロー (注2)
		7	FPE_FLTINV	無効な浮動小数点演算
		―	―	浮動小数点例外
SIGBUS(10)	記憶保護例外	1	BUS_ADRALN	無効なアドレス境界付け
		2	BUS_ADRERR	実在しない物理アドレス
		3	BUS_OBJERR	オブジェクト固有のハードウェアエラー
SIGSEGV(11)	セグメンテーション例外	1	SEGV_MAPERR	オブジェクトにマップされていないアドレス
		2	SEGV_ACCERR	マップされたオブジェクトへの許可が無効
SIGXCPU(30)	CPU占有時間による打切り	―	―	―

注1) Armアーキテクチャである富士通製CPU A64FXでは、FPE_INTDIV(除数が0の場合の整数除算例外)は検出されません。詳細は“C.2.9 除数が0の場合の整数除算例外について”をお読みください。

注2) -NRtrapオプションおよび環境変数FLIB_EXCEPT=u(浮動小数点アンダフロー割込み検出指示)を指定した場合に捕捉されるシグナルです。

8.2.2 異常終了時の出力情報

捕捉されたシグナルに応じて、以下の情報が標準エラーに出力されます。

8.2.2.1 一般的な異常終了時の出力情報

SIGILL、SIGBUS、またはSIGSEGVのどれかを捕捉した場合に出力されます。これらの情報に続いてトレースバックマップが出力されます。

```
jwe0019i-u The program was terminated abnormally with signal number SSSSSSS.
           signal identifier = NNNNNNNNNN, (Detailed information.)
```

SSSSSSS	SIGILL、SIGBUS、またはSIGSEGVを示します。
NNNNNNNNNN	異常終了の要因となったシグナルコードを示します。
(Detailed information.)	上記シグナルコード*NNNNNNNNNN*に対応する詳細情報を示します。

SIGBUSまたはSIGSEGVの原因がスタックオーバフローである可能性が高い場合、以下の情報を付加します。

```
The cause of this exception may be stack-overflow.
```

8.2.2.2 SIGXCPUの出力情報

SIGXCPUが捕捉された場合、以下の情報が出力されます。

```
jwe0017i-u The program was terminated with signal number SIGXCPU.
```

8.2.2.3 異常終了処理中に再び異常終了が発生した場合の出力情報

異常終了処理中に再び異常終了が発生した場合、以下の情報が出力されます。

```
jwe0020i-u An error was detected during an abnormal termination process.
```

8.3 フック機能

フック機能は、Cプログラムの特定箇所から、または、一定時間間隔でユーザー定義関数を呼び出す機能です。ユーザー定義関数において、トレース情報または特定の変数の値を出力することで、プログラムの動作を確認することができます。

ここでは、フック機能について説明します。

8.3.1 ユーザー定義関数の形式

ユーザー定義関数名は固定(user_defined_proc)となります。

ユーザー定義関数は、以下の形式で定義してください。

- 形式

```
#include "fjhook.h"
void user_defined_proc(int *FLAG, char *NAME, int *LINE, int *THREAD)
```

- 引数

  ー FLAG:ユーザー定義関数の呼出し元を示します。

  = 0:プログラムの入口

  = 1:プログラムの出口

  = 2:関数の入口

  = 3:関数の出口

  = 4:パラレルリージョン(OpenMP/自動並列化)の入口

  = 5:パラレルリージョン(OpenMP/自動並列化)の出口

  = 6:一定時間間隔

  = 7～99:システムリザーブ

  = 100～:利用者側で利用可能

  ー NAME:呼出し元の関数名を示します。

  NAMEは、FLAGが2、3、4、5、または100以上の場合のみ参照可能です。

  ー LINE:呼出し元の行番号を示します。

  LINEは、FLAGが2、3、4、5、または100以上の場合のみ参照可能です。

  ー THREAD:OpenMP、自動並列化でのスレッド並列において、ユーザー定義関数を呼び出したスレッドの識別番号を示します。

  THREADは、FLAGが2、3、4、5、または100以上の場合のみ参照可能です。

8.3.2 フック機能の注意事項

フック機能の使用時には、以下の注意事項があります。

- 関数名“user_defined_proc”をほかの用途に使用してはなりません。
- ユーザー定義関数が定義されていない場合、動作は保証されません。
- ユーザー定義関数の引数に値を設定してはなりません。
- -Nnolineオプションを指定した場合、引数LINEの値は保証されません。
- スレッド並列化されたプログラムでは、複数のスレッドからユーザー定義関数が呼び出されることがあります。
- setjmp関数またはlongjmp関数を呼び出す場合、もしくは、例外が捕捉された場合には、ユーザー定義関数が呼び出されないことがあります。
- ユーザー定義関数の関数入口/出口は、プログラム内の任意箇所からの呼出しを除き、ユーザー定義関数の呼出元になりません。
- ユーザー定義関数内で、直接的および間接的にexit(3)関数を実行した場合、動作は保証されません。
- ユーザー定義関数内で使用可能なOpenMP仕様は、以下のOpenMP関数だけです。
  - omp_get_active_level
  - omp_get_ancestor_thread_num
  - omp_get_dynamic
  - omp_get_level
  - omp_get_max_active_levels
  - omp_get_max_threads
  - omp_get_nested
  - omp_get_num_procs
  - omp_get_num_threads
  - omp_get_proc_bind
  - omp_get_schedule
  - omp_get_thread_limit
  - omp_get_thread_num
  - omp_get_wtick
  - omp_get_wtime
  - omp_in_final
  - omp_in_parallel
  - omp_set_dynamic
  - omp_set_max_active_levels
  - omp_set_nested
  - omp_set_num_threads
  - omp_set_schedule

8.3.3 特定箇所からのユーザー定義関数呼出し

-Nhook_funcオプションをオブジェクトプログラムおよび実行可能プログラムの作成時に指定した場合、以下の箇所からユーザー定義関数が呼び出されます。

- プログラムの入口/出口
- 関数の入口/出口

また、-Kopenmpまたは-Kparallelオプションが有効な場合、上記の箇所に加えて以下の箇所からもユーザー定義関数が呼び出されます。

・ パラレルリージョン(OpenMP/自動並列化)の入口/出口

OpenMP、自動並列化の入口/出口では、引数NAMEに次の内部関数名が渡されます。

・ OpenMP “_OMP_識別番号_”

・ 自動並列化 “_PRL_識別番号_”

FortranプログラムとCプログラムの結合プログラムにおいて、-Nhook_funcオプションが指定された場合の動作を以下に示します。

・ FortranプログラムとCプログラムにおいて、それぞれのプログラム中にユーザー定義サブルーチンまたはユーザー定義関数が定義されている場合、Fortranプログラムからは Fortranプログラム中で定義されたユーザー定義サブルーチンが、Cプログラムからは Cプログラム中で定義されたユーザー定義関数が呼び出されます。

・ FortranプログラムとCプログラムにおいて、どちらか一方のプログラム中にのみユーザー定義サブルーチンまたはユーザー定義関数が定義されている場合、定義されたユーザー定義サブルーチンまたはユーザー定義関数はFortranプログラムとCプログラムの両方から呼び出されます。

特定箇所でのフック機能の使用例を以下に示します。

ユーザー定義関数のプログラム例

```
#include <stdio.h>
#include "fjhook.h"

void sub() {
   printf("SUB¥n");
}

int main() {
   printf("HELLO¥n");
   sub();
}

void user_defined_proc(int *FLAG, char *NAME, int *LINE, int *THREAD) {
   switch(*FLAG){
   case 0:
      printf("PROGRAM START¥n");
   break ;
   case 1:
      printf("PROGRAM END¥n");
   break ;
   case 2:
      printf("PROC START: %s LINE: %d¥n",NAME,*LINE);
   break ;
   case 3:
        printf("PROC END: %s LINE: %d¥n",NAME,*LINE);
   break ;
  }
}
```

-Nhook_funcを指定してプログラムを翻訳する。

```
$ fccpx -Nhook_func test.c
```

実行結果

```
PROGRAM START
PROC START: main LINE: 8
HELLO
PROC START: sub LINE: 4
SUB
PROC END: sub LINE: 6
PROC END: main LINE: 11
PROGRAM END
```

8.3.3.1 特定箇所からの呼出しにおける注意事項

-Nhook_funcオプションの指定時には、以下の注意事項があります。

- 計算量の少ない関数が繰り返し呼び出されるような場合、関数の入口/出口からのユーザー定義関数呼出しは、実行性能に影響を与える可能性があります。
- 計算量の少ないパラレルリージョンが繰り返し実行されるような場合、パラレルリージョンの入口/出口からのユーザー定義関数呼出しは、実行性能に影響を与える可能性があります。

8.3.4 一定時間間隔でのユーザー定義関数呼出し

実行可能プログラムのリンク時に-Nhook_timeオプションを指定した場合、一定のユーザー時間が経過するたびにユーザー定義関数が呼び出されます。

ユーザー定義関数の呼出し間隔は、環境変数FLIB_HOOK_TIMEにより指定可能です。環境変数FLIB_HOOK_TIMEを指定しない場合は、1分ごとにユーザー定義関数が呼び出されます。環境変数FLIB_HOOK_TIMEについては、“2.5.1 実行時環境変数”をお読みください。

8.3.4.1 一定時間間隔での呼出しにおける注意事項および制限事項

-Nhook_timeオプションの指定時には、以下の注意事項があります。

- 環境変数FLIB_HOOK_TIMEの指定は、プログラムの実行途中に変更してはなりません。
- 環境変数FLIB_HOOK_TIMEに0～2147483647以外の値を指定した場合、動作は保証されません。
- ユーザー定義関数の呼出しが、実行性能に影響する可能性があります。

また、一定時間間隔での呼出しでは、非同期シグナルからユーザー定義関数が呼び出されるため、以下の制限事項があります。

- ユーザー定義関数では、非同期シグナルセーフ関数を除く関数の使用はできません。
- -Nhook_timeオプションを指定する場合、SIGVTALRMに対応するシグナル処理ルーチンを定義してはなりません。
- -Nhook_timeオプションを指定する場合、setitimer(2)を使用してはなりません。
- ユーザー定義関数による大域変数の参照は、volatile sig_atomic_t型の変数だけが保証されます。
- -Nhook_timeオプションを指定して作成された実行ファイルに対して、プロファイラを使用してはなりません。

8.3.5 プログラム内の任意箇所からの呼出し

引数(FLAG:100～、NAME、LINE、THREAD)を設定して、プログラム中の任意箇所からユーザー定義関数を呼び出すことが可能です。

第9章 clangモード

本章では、clangモードについて説明します。また、tradモードとの差異についても説明します。

9.1 翻訳から実行まで

9.1.1 翻訳コマンド

翻訳コマンドについて説明します。

9.1.1.1 翻訳コマンドの形式

clangモードで翻訳する場合は、-Nclangオプションを有効にしてください。

“表9.1 翻訳コマンドの形式”に、翻訳コマンドの形式を示します。

表9.1 翻訳コマンドの形式

コマンド名		オペランド
クロスコンパイラ	fccpx	[□-Nclangオプションを含むオプションの並び]□ファイル名の並び
ネイティブコンパイラ	fcc	[□-Nclangオプションを含むオプションの並び]□ファイル名の並び

□:1個以上の空白が必要であることを意味します。

参考

“-Nclangオプションを含むオプションの並び”と“ファイル名の並び”の指定順序に制約はありません。

“-Nclangオプションを含むオプションの並び”と“ファイル名の並び”を混在して指定してもかまいません。

9.1.1.2 翻訳コマンドの入力ファイル

“表9.2 入力ファイルの形式”に、翻訳コマンドに入力ファイルとして指定できるファイルを示します。

表9.2 入力ファイルの形式

ファイルの種別	ファイルのサフィックス	渡し先
ヘッダ	.h	プリプロセッサ (注)
	.H	
Cソースファイル	.c	プリプロセッサおよび コンパイラ
	.i	
アセンブラソースファイル	.s	アセンブラ
前処理を要するアセンブラソースファイル	.S	プリプロセッサおよび アセンブラ
オブジェクトファイル	.o	リンカ

注) -Eオプションが有効な場合

9.1.1.3 翻訳コマンドの復帰値

“表9.3 翻訳コマンドが設定する復帰値”に、翻訳コマンドが設定する復帰値を示します。

表9.3 翻訳コマンドが設定する復帰値

復帰値	意味
0	正常に終了しました。
1	翻訳時またはリンク時にエラーが発生しました。

9.1.1.4 翻訳時の注意事項

翻訳時の注意事項を説明します。

9.1.1.4.1 翻訳時のスタックサイズ

ループ内の演算処理が非常に多いソースプログラムを翻訳する場合、コンパイラのスタック領域が不足し、セグメンテーション例外が発生することがあります。

この場合、コンパイラ実行環境のスタック領域を十分な大きさに拡張する必要があります。

プロセスのスタック領域は、ulimitコマンド(bash組み込みコマンド)などで設定できます。

9.1.2 翻訳時オプション

翻訳コマンドに指定できる翻訳時オプションの形式と意味について説明します。

翻訳時オプションは、以下の方法で指定することができます。

- 翻訳コマンドのオペランド
- 環境変数 (“9.1.3 翻訳コマンドの環境変数”参照)
- 翻訳時プロフィルファイル (“9.1.4 翻訳時プロフィルファイル”参照)

翻訳時オプションの指定の優先順位は、“2.4 翻訳時プロフィルファイル”を参照してください。

注意

- Clang/LLVMのオプションについては、ほぼすべてのオプションを利用することができます。ただし、本章内に掲載していないオプションを指定した場合、動作保証しません。
- “表9.6 clangモードでは形式や名前が置換されるオプション”に記載がないtradモードの翻訳時オプションは、clangモードで指定できません。指定された場合、エラーメッセージを出力し、翻訳を中断します。

9.1.2.1 翻訳時オプションの形式

翻訳コマンドのオペランドで翻訳時オプションを指定することができます。

翻訳時オプションの形式を以下に示します。

コンパイラ全般に関連するオプション

```
[ -### ] [ -C ] [ -Dname[=tokens] ] [ -E ] [ -Idir ] [ -Ldir ] [ -M ] [ -MD ] [ -MF filename ] [ -MM ] [ -MMD ] [ -MP ]
[ -MT target ] [ -Nopt ] [ -P ] [ -S ] [ -SSL2 ] [ -SSL2BLAMP ] [ -Uname ] [ -Wtool,arg1[,arg2]... ] [ -c ] [ -fopt ]
[ {-g|-g0} ] [ -lname ] [ -o pathname ] [ -shared ] [ -v ] [ --coverage ] [ --verbose ]
```

メッセージ関連オプション

```
[ -Rpass=.* ] [ -Rpass-analysis=.* ] [ -Rpass-missed=.* ] [ -w ] [ -Koptmsg=2 ]
```

最適化関連オプション

```
[ -O[n] ] [ -m{omit-leaf-frame-pointer|no-omit-leaf-frame-pointer} ] [ -msve-vector-bits={512|scalable} ]
[ -fopt ] [ -Kopt ]
```

言語仕様関連オプション

```
[ -std=name ] [ --linkcoarray ] [ --linkfortran ]
```

CPU/アーキテクチャ関連オプション

```
[ -march=arch[+features]... ] [ -Karch ] [ -mcpu=cpuname[+features]... ] [ -Kcpuname ]
```

コード生成関連オプション

```
[ -fopt ] [ -mcmodel=name ] [ -mfj-tls-size={12|24|32|48} ] [ -Kopt ]
```

9.1.2.2 翻訳時オプションの意味

翻訳時オプションの意味を説明します。

9.1.2.2.1 コンパイラ全般に関連するオプション

翻訳で使用される一般的なオプションについて説明します。

-###

翻訳コマンドによって実行されるコマンドを出力します。

ただし、プログラムの翻訳処理は行いません。

-C

前処理の段階で通常削除される注釈を、前処理指令の行にあるもの以外、この段階では削除されないようにします。

-D*name*[=*tokens*]

#define前処理指令と同様に、マクロを定義します。*name*にはマクロ名を指定します。*tokens*には*name*に関連付ける値を指定します。

-E

指定されたCソースファイルについて、前処理だけを実行し、結果を標準出力に出力します。

-N*opt*

“表9.6 clangモードでは形式や名前が置換されるオプション”に掲載されている-N*opt*オプションは、対応するclangモードのオプションに置き換わります。

置き換わるオプションの対応は、“表9.6 clangモードでは形式や名前が置換されるオプション”を参照してください。

-I*dir*

名前が/以外で始まるヘッダの検索を、*dir*で指定されたディレクトリを先に検索し、その後、通常のディレクトリを検索するように変更します。

複数の-Iオプションでディレクトリが複数指定された場合、指定された順に検索します。

-L*dir*

リンカがライブラリを検索するディレクトリのリストに、*dir*を加えます。

本オプションは、リンカに渡されます。

-M

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。

本オプションを指定すると、前処理だけが実施されます。

-MD

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。

結果をサフィックス.dの付いたファイルに格納します。

-Eおよび-oオプションと同時に指定された場合、結果を-oオプションで指定されたファイルに格納します。

-MFオプションと同時に指定された場合、結果を-MFオプションで指定されたファイルに格納します。

-MF *filename*

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係を*filename*に出力することを指示します。

-MM

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。ただし、標準ヘッダは出力対象に含まれません。

本オプションを指定すると、前処理だけが実施されます。

-MMD

ソースの依存関係をmakeコマンドが認識する書式で出力することを指示します。ただし、標準ヘッダは出力対象に含まれません。

結果をサフィックス. dの付いたファイルに格納します。

-Eおよび-oオプションと同時に指定された場合、結果を-oオプションで指定されたファイルに格納します。

-MFオプションと同時に指定された場合、結果を-MFオプションで指定されたファイルに格納します。

-MP

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係に疑似ターゲットを追加することを指示します。

-MT *target*

-M、-MM、-MD、または-MMDオプションが有効な場合、依存関係のターゲットを*target*に変更することを指示します。

-P

-Eオプションによる出力結果に、linemarker(行情報)を含めないことを指示します。

-S

アセンブリ言語出力を、サフィックス. sの付いた対応するファイルに残します。

本オプションと同時に-fltoオプションを指定した場合、生成されるファイルにはコンパイラの内部情報が出力されます。そのため、tradモードまたはアセンブラ(asコマンド)の入力ファイルに指定することはできません。

本オプションと同時に-gオプションまたは-ffj-lineオプションが有効な場合、生成されるファイルにはclangモードで拡張された情報が出力されます。そのため、tradモードまたはアセンブラ(asコマンド)の入力ファイルに指定することはできません。

-SSL2

C-SSLII、C-SSLIIスレッド並列機能、およびBLAS/LAPACKを結合することを指示します。

-SSL2BLAMP

C-SSLII、C-SSLIIスレッド並列機能、およびBLAS/LAPACKスレッド並列版を結合することを指示します。

-SSL2オプションとの違いは、BLAS/LAPACKにおいてスレッド並列版が結合されることです。

-U*name*

#undef前処理指令と同様に、マクロ定義を無効にします。*name*には無効にするマクロ名を指定します。

-W*tool*,*arg1*[,*arg2*]...

指定された*arg1*[,*arg2*]...をそれぞれ別の引数として*tool*に渡します。各引数は、直前の引数とコンマだけで区切られていなければなりません。

*tool*には、以下のいずれかの文字を指定します。

*too*lに指定する文字	*arg1*[,*arg2*]...を渡す先
p	プリプロセッサ
a	アセンブラ
l	リンカ

なお、-Waオプションと同時に-fltoオプションを指定した場合、-fltoオプションは無効になります。

例

リンカに対して、共有ライブラリのアドレスのシンボル解決に遅延リンクを使わないことを指示する場合は、以下のように指定します。

```
-Wl,-z,now
```

-c

翻訳処理の最後の段階であるリンクを行わないようにします。

本オプションと同時に-fltoオプションを指定した場合に生成されるファイルは、-Nclangオプションが有効な場合のみリンクできます。

-f*opt*

*opt*には、以下のいずれかを指定します。

```
{ {debug-info-for-profiling|no-debug-info-for-profiling} | {fj-fjcex|fj-no-fjcex} | {fj-fjprof|fj-no-fjprof} | {fj-hook-time|fj-no-hook-time} | {fj-line|fj-no-line} | fj-lst[={p|t}] | fj-lst-out=file | fj-src | profile-dir=dir_name | sanitize={address|undefined} | {unwind-tables|no-unwind-tables} }
```

-f{debug-info-for-profiling|no-debug-info-for-profiling}

プロファイラで利用可能なシグネチャを追加情報として生成するか否かを指示します。シグネチャは、関数の多重定義解決に用いる情報です。この追加情報は、基本プロファイラで-Minlinedオプションを指定した時に必要になります。

翻訳時オプション-fdebug-info-for-profilingが指定された場合、シグネチャを追加情報として生成します。この場合、基本プロファイラで-Minlineオプションを指定すると、インライン展開された関数についてシグネチャ単位で関数コストを表示します。

翻訳時オプション-fno-debug-info-for-profilingが指定された場合、シグネチャを追加情報として生成しません。この場合、基本プロファイラで-Minlinedオプションを指定すると、インライン展開された関数について関数名単位で関数コストを表示します。また、シグネチャを生成しない分、オブジェクトプログラムのサイズは翻訳時オプション-fdebug-info-for-profiling指定時より小さくなります。

省略時は、翻訳時オプション-fdebug-info-for-profilingが適用されます。本オプションは、翻訳時オプション-ffj-lineが有効な場合に意味があります。

プロファイラについては、“プロファイラ使用手引書”を参照してください。

-f{fj-fjcex|fj-no-fjcex}

clangモードが提供する富士通拡張関数を利用するか否かを指示します。

-ffj-fjcexオプションが有効な場合、利用します。省略時は、-ffj-no-fjcexオプションが適用されます。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

富士通拡張関数については、“付録G 富士通拡張関数”をお読みください。

-f{fj-fjprof|fj-no-fjprof}

プロファイラ機能を有効にするか否かを指示します。-ffj-fjprofオプションが指定された場合、有効にします。省略時は、-ffj-fjprofオプションが適用されます。本オプションはリンク時に指定する必要があります。

プロファイラ機能については、“プロファイラ使用手引書”をお読みください。

-f{fj-hook-time|fj-no-hook-time}

一定時間間隔での呼出しによるフック機能を利用するか否かを指示します。-ffj-hook-timeオプションが指定された場合、フック機能を利用します。省略時は、-ffj-no-hook-timeオプションが適用されます。

-ffj-hook-timeオプションが指定された場合、一定時間間隔でユーザー定義関数を呼び出します。

ユーザー定義関数を呼び出す時間間隔は、環境変数FLIB_HOOK_TIMEで指定可能です。

環境変数FLIB_HOOK_TIMEを指定しない場合は、1分ごとにユーザー定義関数が呼び出されます。環境変数FLIB_HOOK_TIMEについては、“9.1.6.1 実行時環境変数”を参照してください。

本オプションは、リンク時に指定する必要があります。フック機能については、“8.3.4 一定時間間隔でのユーザー定義関数呼出し”を参照してください。

-f{fj-line|fj-no-line}

プロファイラで提供される実行時間のサンプリング機能に必要な追加情報を生成するか否かを指示します。-ffj-lineオプションが指定された場合、追加情報を生成します。省略時は、-ffj-lineオプションが適用されます。

-ffj-lineオプションと同時に-fltoオプションを指定した場合、プロファイラで提供される実行時間のサンプリング機能に必要なループに関する情報が生成されません。プロファイラについては、“プロファイラ使用手引書”を参照してください。

-ffj-lst[={p|t}]

-ffj-lstオプションは、翻訳情報をファイルに出力することを指示します。翻訳情報は、サフィックス.lstの付いたファイルへ出力します。Cソースファイルを複数指定した場合には、*各ファイル名*.lstファイルに出力します。

-ffj-lstオプションが引数を省略して指定された場合、-ffj-lst=pが適用されます。

-ffj-lst=p

翻訳情報として、ソースリストを出力することを指示します。

ソースリストに含まれる最適化情報により、インライン展開、ループアンローリングなどの状況がわかります。

-ffj-lst=t

-ffj-lst=pでの出力に加えて、より詳細な最適化情報を出力することを指示します。

注意

翻訳を行ったディレクトリに“*各ファイル名*.opt.yaml”ファイルが既に存在する場合、翻訳情報は出力されません。

-ffj-lst-out=*file*

指定されたファイル名*file*に翻訳情報を出力することを指示します。

本オプションを指定した場合、-ffj-lst=pオプションも有効になります。

-ffj-src

ソースリストを標準出力に出力することを指示します。

ソースリストに含まれる最適化情報により、自動並列化、インライン展開、ループアンローリングなどの状況がわかります。

-ffj-lstオプションまたは-ffj-lst-out=*file*オプションが同時に指定された場合、ソースリストはファイルに出力されます。

-fprofile-dir=*dir_name*

コードカバレッジ機能使用時に必要な.gcdaファイルの格納先ディレクトリを指示します。*dir_name*には、格納先ディレクトリ名を相対パスまたは絶対パスで指定します。

.gcdaファイルは、コードカバレッジ用の情報を含むオブジェクトファイルを結合した実行可能プログラムを実行すると、生成されます。実行時に、*dir_name*で指定されたディレクトリが存在しない場合は、新規に作成されます。

本オプションは、--coverageおよび、-Sまたは-cオプションが有効な場合に意味があります。

コードカバレッジ機能および省略時の格納先ディレクトリについては、“9.9 コードカバレッジ機能”をお読みください。

-fsanitize={address|undefined}

-fsanitizeオプションはCプログラムをデバッグするための機能です。Cプログラムのオブジェクトプログラム内にデバッグするために必要な情報を組み込み、実行時に自動検査します。検査は、-fsanitize=undefinedまたは-fsanitize=addressオプションが有効な場合に行われます。

複数の-fsanitizeオプションを指定することで、複数のデバッグ機能を組み合わせて利用することができます。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

-fsanitizeオプションが有効な場合、-ffj-largepageオプションは無効となります。

-fsanitize=address

ヒープメモリに対して、メモリの不当な解放、領域外書込み、およびメモリリークを検査することを指示します。

-fsanitize=undefined

配列の使用時において、宣言した配列のサイズと使用時の添字範囲の正当性を検査することを指示します。

-f{unwind-tables|no-unwind-tables}

スタックトレースなどのデバッグに必要な情報を格納するためのunwindテーブルを生成するか否かを指示します。-funwind-tablesが指定された場合、unwindテーブルを生成します。省略時は、-funwind-tablesオプションが適用されます。

{-g|-g0}

-gオプションは、実行可能ファイルにデバッガで利用可能な追加情報を含めることを指示します。

-g0オプションは、実行可能ファイルにデバッガで利用可能な追加情報を含めないことを指示します。

省略時は、-g0オプションが適用されます。

-l*name*

lib*name*.soまたはlib*name*.aというファイル名のライブラリを結合します。

シンボルは指定された順序にライブラリで解決されるため、コマンド行でのライブラリの指定順序が重要になります。本オプションは、ソースファイル名のあとに指定してください。

本オプションは、リンカに渡されます。

-o *pathname*

*pathname*で指定された名前のファイルを作成します。

-shared

リンカに対して、共有オブジェクトを作成することを指示します。

本オプションは、リンカに渡されます。

-v

翻訳コマンドが呼び出すコンパイラ、アセンブラおよびリンカの実行コマンドを表示します。

--coverage

コードカバレッジ機能を利用するための情報を生成することを指示します。--coverageオプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

--coverageオプションが有効な場合、-fltoオプションは無効となります。

--coverageオプションを指定した場合、実行回数を計測する命令がオブジェクトファイル内に追加されるため、実行性能が低下する場合があります。

コードカバレッジ機能については、“9.9 コードカバレッジ機能”をお読みください。

--verbose

翻訳コマンドが呼び出すコンパイラ、アセンブラおよびリンカの実行コマンドを表示します。

9.1.2.2.2 メッセージ関連オプション

翻訳時メッセージに関するオプションについて説明します。

-Rpass=.*

最適化情報を出力します。

例

-Rpass=.*の使用例

```
$ fccpx -Nclang test01.c -Ofast -Rpass=.* -S
test01.c:9:20: remark: sinking zext [-Rpass=licm]
      a_array[i] = b_array[i] * c_array[i];
                   ^
test01.c:8:3: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
```

```
  for(i = 0; i < N; i++) {
  ^
```

-Rpass-analysis=.*

コンパイラが最適化を適用しなかった理由を出力します。

例

-Rpass-analysis=.*の使用例

```
$ fccpx -Nclang test02.c -Ofast -Rpass-analysis=.* -S
test02.c:9:18: remark: loop not vectorized: call instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]
  for(i = 0; i < foo(); i++) {
                 ^
test02.c:9:3: remark: loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-
vectorize]
  for(i = 0; i < foo(); i++) {
  ^
```

-Rpass-missed=.*

最適化が適用されなかったことを示すメッセージを出力します。

ただし、出力されるメッセージには、コンパイラが最適化を適用しなかった理由は含まれません。

例

-Rpass-missed=.*の使用例

```
$ fccpx -Nclang test02.c -Ofast -Rpass-missed=.* -S
test02.c:9:3: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
  for(i = 0; i < foo(); i++) {
  ^
```

-w

警告メッセージの出力を抑止します。

-Koptmsg=2

本オプションを指定した場合、-Rpass=.*オプションに置き換わります。

9.1.2.2.3 最適化関連オプション

最適化に関するオプションについて説明します。

-O[*n*]

*n*には、最適化レベル0、1、2、3、またはfastを指定します。最適化レベル*n*が指定されていない場合、-O2オプションが適用されます。最適化レベルが高いほど、実行時間が短縮されますが、翻訳時間は増加します。高位の最適化レベルは、低位の最適化レベルを機能的に包含します。

-O[*n*]オプションを指定しない場合、-O0オプションが適用されます。

- 最適化レベル0

  最適化されません。

- 最適化レベル1

  基本的な最適化が行われます。

  最適化レベル0に比べてオブジェクトプログラムの大きさが縮小され、実行時間も大幅に短縮されます。

ー 最適化レベル2

最適化レベル1に加えて、以下の最適化が行われます(括弧内のオプションが有効となります)。

- SIMD拡張命令を利用したオブジェクトを生成(-fvectorize)
- ループアンローリング(-funroll-loops)
- ループ分割(-ffj-loop-fission)
- prefetch命令の生成(-ffj-prefetch-sequential)
- インライン展開(-finline-functions)
- スーパーワードレベルの並列化(-fslp-vectorize)
- 最適化機能の繰返し実施
  最適化機能の繰返し実施は、最適化レベル1で行われる最適化機能を、最適化の余地がなくなるまで繰り返して実施します。

ー 最適化レベル3

最適化レベル2よりも高度な最適化を実施します。最適化レベル2よりも翻訳時間が増加する場合や、オブジェクトプログラムの大きさが増加する場合があります。

ー 最適化レベル`fast`

ターゲットマシン上で高速に実行するオブジェクトプログラムを作成することを指示します。

本オプションは、-O3 -ffj-fast-matmul -ffast-math -ffp-contract=fast -ffj-fp-relaxed -ffj-ilfunc -fomit-frame-pointer -finline-functionsを指定した場合と等価です。

本オプションは、-ffj-fast-matmul、-ffast-math、-ffp-contract=fast、-ffj-fp-relaxed、および-ffj-ilfuncオプションの影響により、計算結果に副作用を生じることがあります。最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

-m{omit-leaf-frame-pointer|no-omit-leaf-frame-pointer}

リーフ関数(他の関数を呼び出さない関数)のフレームポインタレジスタを保証しない最適化を行うか否かを指示します。

-momit-leaf-frame-pointerオプションが指定された場合、最適化を行います。トレースバック情報は保証されません。

-mno-omit-leaf-frame-pointerオプションが指定された場合、最適化は行いません。リーフ関数にフレームポインタレジスタを保持する命令が出るため、性能が低下する場合があります。

省略時は、-momit-leaf-frame-pointerオプションが適用されます。

以下に、-f{omit-frame-pointer|no-omit-frame-pointer}オプションとの関係を示します。

ー -momit-leaf-frame-pointerオプションは-fomit-frame-pointerオプションから誘導されます。

ー -mno-omit-leaf-frame-pointerオプションは、-fno-omit-frame-pointerオプションを個別に指定した場合のみ誘導され、有効になります。-mno-omit-leaf-frame-pointerオプションを個別に指定しても有効になりません。

-msve-vector-bits={512|scalable}

SVEのベクトルレジスタサイズを指定します。単位はビットです。

-msve-vector-bits=512オプションが指定された場合、翻訳時にベクトルレジスタのサイズを指定された固定値とみなして最適化を実施します。そのため、最適化が促進され実行性能の向上が期待されます。

ただし、生成した実行可能プログラムは、翻訳時に指定したサイズのベクトルレジスタのSVEを実装しているCPUアーキテクチャにおいてのみ正常に動作します。詳細は、“9.1.2.3.5 SVEのベクトルレジスタサイズを指定する場合の注意事項”をお読みください。

-msve-vector-bits=scalableオプションが指定された場合、SVEのベクトルレジスタを特定のサイズとみなさず翻訳を行い、実行時にベクトルレジスタサイズを決定する実行可能プログラムを作成します。この実行可能プログラムは、CPUアーキテクチャに実装されたSVEのベクトルレジスタサイズによらず実行可能です。

省略時は、-msve-vector-bits=scalableオプションが適用されます。

本オプションは、-marchオプションの*features*でsveが有効な場合に意味があります。

-msve-vector-bits=512オプションが指定された場合、-fslp-vectorizeオプションは無効となります。また、SIMD組込み関数を使用することはできません。

-msve-vector-bits=512オプションと-fltoオプションを同時に指定した場合、-msve-vector-bits=512オプションは無効になり、-msve-vector-bits=scalableオプションが有効になります。

-f*opt*

*opt*には、以下のいずれかを指定します。

```
{ {fj-eval-concurrent|fj-no-eval-concurrent} | {fj-fast-matmul|fj-no-fast-matmul} | {fj-fp-precision|fj-no-fp-
precision} | {fj-fp-relaxed|fj-no-fp-relaxed} | {fj-hpctag|fj-no-hpctag} | {fj-ilfunc[={loop|procedure}]|fj-no-ilfunc}
| {fj-interleave-loop-insns[=N]|fj-no-interleave-loop-insns} | {fj-loop-fission|fj-no-loop-fission} | fj-loop-fission-
threshold=N | fj-no-prefetch | {fj-ocl|fj-no-ocl} | {fj-optlib-string|fj-no-optlib-string} | {fj-preex|fj-no-preex} |
fj-prefetch-cache-level={1|2|all} | {fj-prefetch-conditional|fj-no-prefetch-conditional} | fj-prefetch-iteration=N |
fj-prefetch-iteration-L2=N | fj-prefetch-line=N | fj-prefetch-line-L2=N | {fj-prefetch-sequential[={auto|soft}]|fj-no-
prefetch-sequential} | {fj-prefetch-stride|fj-no-prefetch-stride} | {fj-prefetch-strong|fj-no-prefetch-strong} | {fj-
prefetch-strong-L2|fj-no-prefetch-strong-L2} | {fj-promote-licm-addressing|fj-no-promote-licm-addressing} | {fj-
regalloc-using-latency|fj-no-regalloc-using-latency} | {fj-sched-insn-contiguous|fj-no-sched-insn-contiguous} | {fj-
swp|fj-no-swp} | {fj-zfill[=N]|fj-no-zfill} | {builtin|no-builtin} | {fast-math|no-fast-math} | {finite-math-only|no-
finite-math-only} | fp-contract={fast|on|off} | {inline-functions|no-inline-functions} | {lto|no-lto} | {omit-frame-
pointer|no-omit-frame-pointer} | {openmp|no-openmp} | {openmp-simd|no-openmp-simd} | {reroll-loops|no-reroll-loops} |
{signed-char|unsigned-char} | {slp-vectorize|no-slp-vectorize} | {strict-aliasing|no-strict-aliasing} | {unroll-loops|
no-unroll-loops} | {vectorize|no-vectorize} }
```

-f{fj-eval-concurrent|fj-no-eval-concurrent}

tree-height-reduction最適化において、浮動小数点演算命令の並列性を優先するか否かを指示します。

-ffj-eval-concurrentオプションが指定された場合、tree-height-reduction最適化において、命令の並列性を優先することを指示します。-ffj-no-eval-concurrentオプションが指定された場合、tree-height-reduction最適化において、命令の並列性を抑え、FMA命令の利用を優先することを指示します。省略時は、-ffj-no-eval-concurrentオプションが適用されます。

本オプションは、-O1オプション以上が有効、かつ、-ffast-mathオプションが有効な場合に意味があります。

-f{fj-fast-matmul|fj-no-fast-matmul}

行列積のループを高速なライブラリ呼び出しに変換するか否かを指示します。

-ffj-fast-matmulオプションが指定された場合、行列積のループを高速なライブラリ呼び出しに変換します。省略時は、-ffj-no-fast-matmulオプションが適用されます。

-ffj-fast-matmulオプションを指定した場合、実行結果に副作用(計算誤差)を生じることがあります。最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

本オプションは、-O2オプション以上が有効な場合に意味があります。ただし、OpenMP指示文が有効なループは変換しません。

-ffj-fast-matmulオプションは、-sharedオプションと同時には指定できません。

ファイル名の並びに-ffj-fast-matmulオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-ffj-fast-matmulオプションを指定する必要があります。

-f{fj-fp-precision|fj-no-fp-precision}

浮動小数点演算の計算誤差が生じないようなオプションの組合せを誘導するか否かを指示します。

-ffj-fp-precisionオプションが指定された場合、浮動小数点演算の計算誤差が生じないようなオプションの組合せを誘導します。省略時は、-ffj-no-fp-precisionオプションが適用されます。

-ffj-fp-precisionオプションは、-ffj-fp-precisionオプションを

-fno-fast-math -ffp-contract=off -ffj-no-fast-matmul -ffj-no-fp-relaxed -ffj-no-ilfunc

に置き換えた場合と等価です。そのため、-ffj-fp-precisionオプションが有効な場合、一部の最適化機能が制限されるので、実行性能が低下する可能性があります。

-ffj-no-fp-precisionオプションは、-ffj-fp-precisionオプションの指定を無効にしますが、-ffj-fp-precisionオプションが誘導する個々のオプションを同時に指定していても、その指定には影響を与えません。

-f{fj-fp-relaxed|fj-no-fp-relaxed}

単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数について、逆数近似演算命令とFloating-Point Multiply-Add/Subtract命令を利用した逆数近似演算を行うか否かを指示します。

-ffj-fp-relaxedオプションが指定された場合、逆数近似演算を行います。省略時は、-ffj-no-fp-relaxedオプションが適用されます。

-ffj-fp-relaxedオプションを指定した場合、実行結果に副作用を生じることがあります。最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-hpctag|fj-no-hpctag}

A64FXプロセッサのHPCタグアドレスオーバライド機能を利用するか否かを指示します。-ffj-hpctagオプションが指定された場合、HPCタグアドレスオーバライド機能を利用します。省略時は-ffj-hpctagオプションが適用されます。

HPCタグアドレスオーバライド機能を利用することで、ハードウェアプリフェッチアシスト機能を有効にできます。

本オプションは、-mcpu=a64fxオプションが有効な場合に意味があります。

本オプションはプログラムの翻訳時およびリンク時に指定する必要があります。

-f{fj-ilfunc[={loop|procedure}]|fj-no-ilfunc}

数学関数をインライン展開するか否かを指示します。

-fbuiltinオプションが有効、かつ-ffj-ilfuncオプションが指定された場合、インライン展開します。省略時は、-ffj-no-ilfuncオプションが適用されます。

-ffj-ilfuncオプションの={loop|procedure}が省略された場合、-ffj-ilfunc=procedureオプションが適用されます。

-ffj-ilfuncオプションを指定した場合、実行結果に副作用を生じることがあります。最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

以下に、対象となる関数を示します。

- 単精度実数型および倍精度実数型の数学関数

```
atan, atan2, cos, exp, log, log10, pow, sin, tan
```

- 単精度複素数型および倍精度複素数型の数学関数

```
abs, exp
```

-ffj-ilfunc=loop

ループ内の数学関数をインライン展開します。コンパイラは、データ型や関数呼出しの有無などから最適化が促進される可能性がある場合に、インライン展開します。

-ffj-ilfunc=procedure

関数内のすべての数学関数をインライン展開します。

-f{fj-interleave-loop-insns[=*N*]|fj-no-interleave-loop-insns}

SVEを利用してSIMD化されたループに対して、ループインターリーブの最適化を行うか否かを指示します。

-ffj-interleave-loop-insnsオプションが指定された場合、最適化を行います。*N*はインターリーブ展開数です。*N*は、2から10までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に値を決定します。省略時は、-ffj-no-interleave-loop-insnsオプションが適用されます。

本オプションは、-fvectorizeオプションが有効、かつ、-marchオプションの*features*でsveが有効な場合に意味があります。

ループインターリーブは、tradモードでのループストライピングと同様の最適化を行います。

-f{fj-loop-fission|fj-no-loop-fission}

ソフトウェアパイプライニングの促進、SIMD化の促進、およびレジスタ不足の解消のために、ループを複数のループに分割する最適化を行うか否かを指示します。

-ffj-loop-fissionオプションおよび-ffj-oclオプションが有効な場合に、loop loop_fission_target指示子が指定されたループを自動で分割します。-O1オプションが有効な場合、常に-ffj-no-loop-fissionオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-ffj-loop-fissionオプションが適用されます。

本オプションは、loop loop_fission_target指示子と組み合わせて使用した場合にのみ、動作します。

本オプションで動作するループ分割のアルゴリズムは、tradモードのクラスタリングアルゴリズムに相当します。

本オプションで動作するループ分割では、メモリのスタック領域を多く使用する傾向があります。

-ffj-loop-fission-threshold=*N*

自動ループ分割における分割後のループの粒度(ループ内の命令数やレジスタ数など)を決める閾値*N*を指示します。

*N*は、1から100までの整数値です。*N*の値を小さくした場合、分割後のループが小さくなり、ループの分割数が増える傾向があります。省略時は-ffj-loop-fission-threshold=50が適用されます。

最適化制御行にloop loop_fission_target指示子が指定されており、-ffj-loop-fissionオプション、-ffj-oclオプション、および-O2オプション以上が有効な場合に、本オプションは有効になります。

-ffj-no-prefetch

prefetch命令を使用したオブジェクトを生成しないことを指示します。

-f{fj-ocl|fj-no-ocl}

clangモードがサポートする富士通コンパイラ独自の最適化制御行(プラグマディレクティブ)を有効にするか否かを指示します。

-ffj-oclオプションが指定された場合、有効にします。省略時は、-ffj-no-oclオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-optlib-string|fj-no-optlib-string}

文字列操作関数において、最適化版のライブラリをリンクするか否かを指示します。-ffj-optlib-stringオプションが指定された場合、最適化版のライブラリを静的にリンクします。省略時は、-ffj-no-optlib-stringオプションが適用されます。

-ffj-optlib-stringオプションは、-mcpu=a64fx[+sve]オプションと同時に指定した場合に有効になります。

-ffj-optlib-stringオプションは、-sharedオプション指定時に-fltoオプションと同時には指定できません。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

以下に、対象となる文字列操作関数を示します。

```
bcopy, bzero, memchr, memcmp, memccpy, memcpy, memmove, memset, strcat, strcmp, strcpy, strlen, strncmp, strncpy, strncat
```

-f{fj-preex|fj-no-preex}

ソースプログラムに対して不変式の先行評価の最適化を行うか否かを指示します。

-ffj-preexオプションが指定された場合、最適化を行います。省略時は、-ffj-no-preexオプションが適用されます。

-ffj-preexオプションを指定した場合、プログラムの論理上、実行されないはずの命令が実行され、実行結果に副作用(実行時の例外発生など)を生じることがあります。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-ffj-prefetch-cache-level={1|2|all}

どのキャッシュレベルにデータをプリフェッチするかを指示します。

省略時は、-ffj-prefetch-cache-level=allオプションが適用されます。

本オプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうちの1つ以上が有効な場合に意味があります。

-ffj-prefetch-cache-level=1

データを1次キャッシュにプリフェッチすることを指示します。1次キャッシュにだけプリフェッチするprefetch命令を使用します。

-ffj-prefetch-cache-level=2

データを2次キャッシュにプリフェッチすることを指示します。2次キャッシュにだけプリフェッチするprefetch命令を使用します。

-ffj-prefetch-cache-level=all

-ffj-prefetch-cache-level=1および-ffj-prefetch-cache-level=2オプションの機能を同時に有効にします。2種類のprefetch命令を組み合わせることにより、より効果的なプリフェッチを実現することができます。

-f{fj-prefetch-conditional|fj-no-prefetch-conditional}

if文やswitch文に含まれるブロックの中で使用される配列データに対してprefetch命令を生成するか否かを指示します。

-ffj-prefetch-conditionalオプションが指定された場合、prefetch命令を生成します。省略時は、-ffj-no-prefetch-conditionalオプションが適用されます。

本オプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうち1つ以上が有効な場合に意味があります。

-ffj-prefetch-iteration=*N*

prefetch命令を生成する際、ループのN回転後に引用されるデータを対象とすることを指示します。Nは、1から10000までの整数値です。

本オプションは、1次キャッシュにのみプリフェッチするprefetch命令を対象とします。

本オプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうちの1つ以上が指定されている、かつ、-ffj-prefetch-cache-level=1または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

本オプションは、-ffj-prefetch-lineオプションと同時には指定できません。

-ffj-prefetch-iteration-L2=*N*

prefetch命令を生成する際、ループの*N*回転後に引用されるデータを対象とすることを指示します。*N*は、1から10000までの整数値です。

本オプションは、2次キャッシュにだけプリフェッチするprefetch命令を対象とします。

本オプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうちの1つ以上が指定されている、かつ、-ffj-prefetch-cache-level=2または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

本オプションは、-ffj-prefetch-line-L2オプションと同時には指定できません。

-ffj-prefetch-line=*N*

prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示します。*N*は、1から100までの整数値です。

本オプションは、1次キャッシュにプリフェッチするprefetch命令を対象とします。

本オプションは、-ffj-prefetch-sequentialオプションが指定されている、かつ、-ffj-prefetch-cache-level=1または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

本オプションは、-ffj-prefetch-iterationオプションと同時には指定できません。

-ffj-prefetch-line-L2=*N*

prefetch命令を生成する際、*N*キャッシュライン先に該当するデータをプリフェッチの対象とすることを指示します。*N*は、1から100までの整数値です。

本オプションは、2次キャッシュにだけプリフェッチするprefetch命令を対象とします。

本オプションは、-ffj-prefetch-sequentialオプションが指定されている、かつ、-ffj-prefetch-cache-level=2または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

本オプションは、-ffj-prefetch-iteration-L2オプションと同時には指定できません。

-f{fj-prefetch-sequential[={auto|soft}]|fj-no-prefetch-sequential}

ループ内で使用される連続的にアクセスされる配列データに対して、prefetch命令を使用したオブジェクトを生成するか否かを指示します。

-ffj-prefetch-sequentialオプションが指定された場合、prefetch命令を使用したオブジェクトを生成します。-O1オプションが有効な場合、省略時は、-ffj-no-prefetch-sequentialオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-ffj-prefetch-sequentialオプションが適用されます。

-ffj-prefetch-sequentialオプションの={auto|soft}が省略された場合、-ffj-prefetch-sequential=autoオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-ffj-prefetch-sequential=auto

ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用するか、prefetch命令を出力するかをコンパイラが自動的に選択します。

-ffj-prefetch-sequential=soft

ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用せずに、prefetch命令を出力します。

-f{fj-prefetch-stride|fj-no-prefetch-stride}

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を使用したオブジェクトを生成するか否かを指示します。プリフェッチするアドレスが翻訳時に確定しないループを含みます。

-ffj-prefetch-strideオプションが指定された場合、prefetch命令を使用したオブジェクトを生成します。省略時は、-ffj-no-prefetch-strideオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-prefetch-strong|fj-no-prefetch-strong}

1次キャッシュに対するprefetch命令を生成する際、strong prefetch命令を生成するか否かを指示します。

-ffj-prefetch-strongが指定された場合、strong prefetch命令を生成します。省略時は、-ffj-prefetch-strongが適用されます。

本オプションは、1次キャッシュにプリフェッチするprefetch命令だけを対象とします。

-ffj-prefetch-strongオプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうちの1つ以上が指定されている、かつ、-ffj-prefetch-cache-level=1または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

-ffj-no-prefetch-strongオプションは、-ffj-hpctagオプションが有効な場合に意味があります。

-f{fj-prefetch-strong-L2|fj-no-prefetch-strong-L2}

2次キャッシュに対するprefetch命令を生成する際、strong prefetch命令を生成するか否かを指示します。

-ffj-prefetch-strong-L2が指定された場合、strong prefetch命令を生成します。省略時は、-ffj-prefetch-strong-L2が適用されます。

本オプションは、2次キャッシュにプリフェッチするprefetch命令だけを対象とします。

-ffj-prefetch-strong-L2オプションは、-ffj-prefetch-sequentialまたは-ffj-prefetch-strideオプションのうちの1つ以上が指定されている、かつ、-ffj-prefetch-cache-level=2または-ffj-prefetch-cache-level=allオプションが指定されている場合に有効になります。

-ffj-no-prefetch-strong-L2オプションは、-ffj-hpctagオプションが有効な場合に意味があります。

-f{fj-promote-licm-addressing|fj-no-promote-licm-addressing}

ループ不変式移動の最適化を促進するために、SIMD化前に、アドレス計算におけるループ不変項をまとめる最適化を行うか否かを指示します。

-ffj-promote-licm-addressingが指定された場合、この最適化を行います。

-O1オプション以上が有効な場合、省略時は、-ffj-promote-licm-addressingオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-regalloc-using-latency|fj-no-regalloc-using-latency}

ループ内のレジスタ割付け処理において、メモリへ退避するレジスタを命令のレイテンシを考慮して決定するか否かを指示します。

-ffj-regalloc-using-latencyが指定された場合、命令のレイテンシを考慮してメモリへ退避するレジスタを決定します。

-O1オプション以上が有効な場合、省略時は、-ffj-regalloc-using-latencyオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-sched-insn-contiguous|fj-no-sched-insn-contiguous}

レジスタ割付け前とレジスタ割付け後の命令スケジューリングにおいて、浮動小数点演算命令、ロード命令、ストア命令を、それぞれ連続するようにスケジューリングするか否かを指示します。

-ffj-sched-insn-contiguousが指定された場合、可能な範囲で連続するようにスケジューリングします。

-O1オプション以上が有効な場合、省略時は、-ffj-sched-insn-contiguousオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{fj-swp|fj-no-swp}

ソフトウェアパイプライニングを行うか否かを指示します。

-ffj-swpオプションが指定された場合、ソフトウェアパイプライニングを行います。省略時は、-ffj-no-swpオプションが適用されます。本オプションは、-O1オプション以上および-mcpu=a64fxオプションが有効な場合に意味があります。

以下のループに対しては最適化が行われません。

- パイプライニングの効果がないループ
- -msve-vector-bits=scalableオプションによりベクトル化されたループ

-ffj-swpオプションが有効な場合、-msve-vector-bits=512オプションを誘導します。ただし、-fltoオプションが有効な場合は誘導しません。また、本オプションの後ろに-msve-vector-bits=scalableオプションが指定されている場合も誘導しません。-msve-vector-bits=512オプションを誘導するか否かは、loop swp指示子の有無に依存しません。

-g有効時は最適化が行われない場合があります。

-Rpass=.* 、-Rpass-analysis=.* 、および-Rpass-missed=.*オプションを指定する場合、ソフトウェアパイプライニングの最適化メッセージは、ソースプログラムに明示的に書かれた単一のループを元にコンパイラが生成した1つまたは複数のループに対するメッセージになります。このため、コンパイラによって複数のループが生成された場合、ソースプログラムに書かれた1つのループに対して異なった複数のメッセージが出力されることがあります。

-Rpass-analysis=.* および-Rpass-missed=.*オプションを指定せず、-Rpass=.*オプションを指定することにより、コンパイラによって生成されるループによるメッセージの混乱を避けることができます。この場合、メッセージ“remark: loop pipelined”は、ソフトウェアパイプライニングがこれらのループの1つに適用されることを意味します。

-f{fj-zfill[=*N*]|fj-no-zfill}

zfillの最適化を行うか否かを指示します。zfillの最適化は、ループ内で書き込みのみを行う配列データについて、データをメモリからロードすることなく、キャッシュ上に書き込み用の領域を確保する命令(DC ZVA)を使って書き込み動作を高速化します。zfillの最適化は対象となるストア命令の指すアドレスを起点として、256バイトを1ブロックとする単位で*N*ブロック分先のデータを最適化の対象とします。*N*は、1から100までの整数値です。*N*の指定を省略した場合、コンパイラが自動的に値を決定します。-O1オプションが有効な場合、常に-ffj-no-zfillオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-ffj-no-zfillオプションが適用されます。

-ffj-zfillオプションは、-mcpu=a64fxオプション、-msve-vector-bits=512オプション、および-O2オプション以上がすべて有効な場合に意味があります。-ffj-zfillオプションが有効な場合、-msve-vector-bits=512オプションを誘導します。ただし、-fltoオプションが有効な場合は誘導しません。また、本オプションの後ろに-msve-vector-bits=scalableオプションが指定されている場合も誘導しません。-msve-vector-bits=512オプションを誘導するか否かは、loop zfill指示子の有無に依存しません。

なお、-ffj-zfillオプション指定して翻訳したオブジェクトプログラムをDC ZVA命令の1回のキャッシュ書き込み動作が256バイトであるCPU以外で実行した場合、実行時異常終了または実行結果に誤りが生じることがあります。A64FXでのDC ZVA命令の1回のキャッシュ書き込み動作は256バイトです。

また、以下の場合に本最適化を適用すると実行性能が低下することがあります。

- メモリバンド幅のボトルネック影響を受けていないプログラム
- 繰返し数が小さいループ
- -ffj-zfill=*N*オプションでブロック数を指定している、かつ、ループによって書き込まれるメモリ領域のサイズが*N*ブロックよりも小さい場合

本最適化による実行性能への影響はループ単位で異なります。このため、本最適化は、-ffj-zfill[=*N*]オプションでプログラム全体へ適用せず、loop zfill指示子でループ単位に適用することを推奨します。

zfillについては、“3.3.5 zfill”をお読みください。

-f{builtin|no-builtin}

標準ライブラリ関数の動作を認識して、最適化を促進させるか否かを指示します。

-fbuiltinオプションが指定された場合、最適化を促進させます。省略時は、-fbuiltinオプションが適用されます。

利用者が標準ライブラリ関数と同名の関数を定義した場合、利用者の意図した結果にならない場合があります。

対象となる標準ライブラリ関数は、以下のとおりです。

```
abort, abs, acos, acosf, acoshf, asin, asinf, asinhf, atan, atan2, atan2f, atanf, atanhf, calloc, cbrt, cbrtf, ceil,
ceilf, clearerr, copysign, copysignf, cos, cosf, cosh, coshf, csqrt, csqrtf, erf, erfc, erfcf, erff, exit, exp,
exp2, exp2f, expf, expm1f, fabs, fabsf, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, floor, floorf, fma,
fmaf, fmax, fmaxf, fmin, fminf, fmod, fmodf, fputc, fputs, free, frexp, fseek, fwrite, getenv, hypotf, ilogb,
```

```
ilogbf, isalnum, isalpha, iscntrl, isdigit, isgraph, islower, ispunct, isspace, isupper, isxdigit, ldexp, lgammaf,
log, log10, log10f, log1pf, log2, log2f, logbf, logf, malloc, memchr, memcmp, memcpy, memmove, memset, modf,
nextafterf, perror, pow, powf, printf, putchar, rand, realloc, remainderf, remove, rename, rint, rintf, scalbnf,
scanf, setvbuf, sin, sinf, sinh, sinhf, sprintf, sqrt, sqrtf, srand, sscanf, strcat, strchr, strcmp, strcpy,
strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtod, strtok, strtol,
strtoul, system, tan, tanf, tanh, tanhf, tolower, toupper, vprintf, vsprintf
```

-f{fast-math|no-fast-math}

演算の評価方法を変更する最適化を行うか否かを指示します。-ffast-mathオプションが指定された場合、最適化を行います。省略時は、-fno-fast-mathオプションが適用されます。

-ffast-mathオプションを指定した場合、最適化やflush-to-zeroモード(演算のオペランドや結果が非正規化数のときにそれらを0に置き換える)使用による影響で、実行結果に副作用(計算誤差や実行時の例外発生など)を生じることがあります。

最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

本オプションは、-O1オプション以上が有効な場合に意味があります。

本オプションは、プログラムの翻訳時およびリンク時に指定する必要があります。

-f{finite-math-only|no-finite-math-only}

引数または演算結果において有限の数値のみであるということを仮定し、浮動小数点演算の最適化を促進させるか否かを指示します。

-ffinite-math-onlyオプションが指定された場合、最適化を促進させます。

本オプションは-ffast-mathから誘導されます。

-ffp-contract={fast|on|off}

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うか否かを指示します。

-ffp-contract=fastオプションが指定された場合、最適化を行います。-ffp-contract=onオプションが指定された場合、言語規格で規定された#pragma STDC FP_CONTRACT ONを用いた場合と同様に最適化を行います。-ffp-contract=offオプションが指定された場合、最適化を行いません。省略時は、-ffp-contract=offオプションが適用されます。

-ffp-contract={fast|on}オプションが有効な場合、実行結果に副作用(丸め誤差程度の違い)を生じることがあります。最適化の副作用については、“9.1.2.3.2 浮動小数点演算に対する最適化とその副作用”を参照してください。

-ffp-contract={fast|off}オプションは、-O1オプション以上が有効な場合に意味があります。

-ffp-contract=onオプションは、-O0オプション以上が有効な場合に意味があります。

-f{inline-functions|no-inline-functions}

ソースプログラム上で定義された関数をインライン展開の対象にするか否かを指示します。-O2オプション以上が有効な場合、省略時は、-finline-functionsオプションが適用されます。

本オプションは、-O2オプション以上と同時に指定した場合に有効となります。

インライン関数においても、コンパイラ内でインライン展開すべきでないと判断した場合は、インライン展開の対象にはなりません。

-f{lto|no-lto}

リンク時最適化を行うか否かを指示します。-fltoオプションを指定した場合、リンク時最適化を行います。-fltoオプションは、プログラムの翻訳時およびリンク時に指定する必要があります。省略時は、-fno-ltoオプションが適用されます。

-fltoオプションと-ffj-lineオプションを同時に指定した場合、プロファイラで提供される実行時間のサンプリング機能に必要なループに関する情報が生成されません。プロファイラについては、“プロファイラ使用手引書”を参照してください。

-fltoオプションを指定した場合、同時に指定できる最適化関連オプションは下記のみです。

- -O[*n*]オプション
- “9.1.2.2.3 最適化関連オプション”のうち、プログラムのリンク時に指定するオプション

-fltoオプションは、-sharedオプション指定時に-ffj-optlib-stringオプションと同時には指定できません。

-fltoオプションと-msve-vector-bits=512オプションを同時に指定した場合、-msve-vector-bits=512オプションは無効になり、-msve-vector-bits=scalableオプションが有効になります。

-f{omit-frame-pointer|no-omit-frame-pointer}

関数呼出しにおける、フレームポインタレジスタを保証しない最適化を行うか否かを指示します。

-fomit-frame-pointerオプションが指定された場合、最適化を行います。トレースバック情報は保証されません。

-fomit-frame-pointerオプションを指定した場合、-momit-leaf-frame-pointerを誘導します。

-fno-omit-frame-pointerオプションを指定した場合、-mno-omit-leaf-frame-pointerを誘導します。

省略時は、-fno-omit-frame-pointerが適用されますが、この場合は-mno-omit-leaf-frame-pointerを誘導しません。

-f{openmp|no-openmp}

OpenMP仕様の指示文を受け入れるか否かを指示します。-fopenmpオプションが指定された場合、受け入れます。省略時は、-fno-openmpオプションが適用されます。

ファイル名の並びに-fopenmpオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-fopenmpオプションを指定する必要があります。

-f{openmp-simd|no-openmp-simd}

OpenMP仕様のsimd構文、declare reduction構文、およびsimd指示節をもつordered構文のみを有効にするか否かを指示します。省略時は、-fno-openmp-simdオプションが適用されます。

-fopenmp-simdオプションが指定された場合、OpenMP仕様によるSIMD化だけを行い、並列化は行いません。

-fno-openmp-simdオプションが指定された場合、-fopenmp-simdオプションを無効にします。

-O1オプション以下が有効な場合、simd構文が有効であったとしても、SIMD化は行われません。

-f{reroll-loops|no-reroll-loops}

ループリローリングの最適化を行うか否かを指示します。-freroll-loopsオプションが指示された場合、最適化を行います。省略時は、-fno-reroll-loopsオプションが適用されます。

-fsigned-char

char 型で宣言した変数を singed char 型として扱うことを指示します。

-funsigned-char

char 型で宣言した変数を unsinged char 型として扱うことを指示します。

-f{slp-vectorize|no-slp-vectorize}

スーパーワードレベルの並列化(以降、SLPと呼びます)を行うことを指示します。

-msve-vector-bits=512オプションが指定された場合、本オプションは無効となります。

-O1オプションが有効な場合、省略時は、-fno-slp-vectorizeオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-fslp-vectorizeオプションが適用されます。

-f{strict-aliasing|no-strict-aliasing}

言語規格で規定された厳密なaliasing ruleに従って、メモリ領域の重なりを考慮した最適化(Strict Aliasing)を行うか否かを指示します。-O0オプション有効な場合、常に-fno-strict-aliasingオプションが適用されます。-O1オプション以上が有効な場合、省略時は、-fstrict-aliasingオプションが適用されます。

本オプションは、-O1オプション以上が有効な場合に意味があります。

-f{unroll-loops|no-unroll-loops}

ループアンローリングの最適化を行うか否かを指示します。-funroll-loopsオプションが指定された場合、最適化を行います。展開数はコンパイラが自動的に最適な値を決定します。-O1オプションが有効な場合、省略時は、-fno-unroll-loopsオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-funroll-loopsオプションが適用されます。

-f{vectorize|no-vectorize}

ループ内の演算に対し、SIMD拡張命令を利用したオブジェクトを生成するか否かを指示します。-fvectorizeオプションが指定された場合、ループ内の命令に対してSIMD拡張命令を利用したオブジェクトを生成します。-O1オプションが有効な場合、省略時は、-fno-vectorizeオプションが適用されます。-O2オプション以上が有効な場合、省略時は、-fvectorizeオプションが適用されます。

-K*opt*

“表9.6 clangモードでは形式や名前が置換されるオプション”に掲載されている-K*opt*オプションは、対応するclangモードのオプションに置き換わります。

9.1.2.2.4 言語仕様関連オプション

言語仕様に関係するオプションについて説明します。

-std=*name*

コンパイラ(プリプロセッサを含む)が解釈する言語仕様のレベルを指定します。

*name*には、以下のいずれかを指定します。

```
{ c89 | c99 | c11 | gnu89 | gnu99 | gnu11 }
```

省略時は、-std=gnu11オプションが適用されます。

-std=c89

C89仕様に基づいて解釈することを指定します。

-std=c99

C99仕様に基づいて解釈することを指定します。

-std=c11

C11仕様に基づいて解釈することを指定します。

-std=gnu89

C89仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

-std=gnu99

C99仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

-std=gnu11

C11仕様に加えて、GNU C拡張仕様に基づいて解釈することを指定します。

--linkcoarray

COARRAY仕様を利用しているFortranとの言語間結合で必要なライブラリの検索を行うよう指示します。

本オプションを指定しない場合、検索処理を行わないため、翻訳時間を短縮することができます。

--linkfortran

COARRAY仕様を利用していないFortranとの言語間結合で必要なライブラリの検索を行うよう指示します。

本オプションを指定しない場合、検索処理を行わないため、翻訳時間を短縮することができます。

9.1.2.2.5 CPU/アーキテクチャ関連オプション

プロセッサ向けの最適化関連オプションおよびアーキテクチャ関連に関するオプションについて説明します。

-march=*arch*[+*features*]...

指定したアーキテクチャの命令セットを利用したオブジェクトファイルを生成することを指示します。

*arch*には、以下のいずれかを指定します。

```
{ armv8-a | armv8.1-a | armv8.2-a | armv8.3-a }
```

-march=armv8-a

Armv8-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-march=armv8.1-a

Armv8-AおよびArmv8.1-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-march=armv8.2-a

Armv8-A、Armv8.1-A、およびArmv8.2-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

-march=armv8.3-a

Armv8-A、Armv8.1-A、Armv8.2-A、およびArmv8.3-Aで定義されている命令を利用したオブジェクトファイルを生成することを指示します。

*features*には、以下のいずれかを指定できます。+で区切って、複数の*features*を指定することもできます。

```
{ sve | nosve | fp16 }
```

sve

Armv8-Aアーキテクチャの拡張であるSVEを利用したオブジェクトファイルを出力することを指示します。

nosve

Armv8-Aアーキテクチャの拡張であるSVEを利用しないオブジェクトファイルを出力することを指示します。

SVEをサポートしないプロセッサ向けのオブジェクトファイルを作成したい場合は、本オプションを指定してください。

fp16

半精度浮動小数点型について言語仕様を拡張したオブジェクトファイルを出力することを指示します。言語仕様については"9.5.2 半精度(16ビット)浮動小数点型について"を参照してください。

-marchオプションは、-mcpuオプションに対応して省略値が変わります。省略時は、下記のように-marchオプションが適用されます。

表9.4 -marchオプションの省略値

有効な-mcpuオプション	-marchオプションの省略値
-mcpu=a64fx[+sve]	-march=armv8.2-a+sve+fp16
-mcpu=a64fx+nosve	-march=armv8.2-a+nosve+fp16
-mcpu=generic[+nosve]	-march=armv8-a+nosve
-mcpu=thunderx[+nosve]	
-mcpu=generic+sve	-march=armv8-a+sve
-mcpu=thunderx+sve	
-mcpu=thunderx2t99[+nosve]	-march=armv8.1-a+nosve
-mcpu=thunderx2t99+sve	-march=armv8.1-a+sve

注意

-mcpuオプションと-marchオプションが同時に指定された場合、-marchオプションに指定されたアーキテクチャや*features*が優先されます。

-K*arch*

"表9.6 clangモードでは形式や名前が置換されるオプション"に掲載されている-K*arch*オプションは、clangモードのオプションに置き換わります。

-mcpu=*cpuname*[+*features*]...

指定したプロセッサ向けのオブジェクトファイルを出力することを指示します。

*cpuname*には、以下のいずれかを指定します。

```
{ a64fx | generic | thunderx | thunderx2t99 }
```

-mcpu=a64fx

A64FXプロセッサ向けのオブジェクトファイルを出力することを指示します。

-mcpu=generic

Armプロセッサ向けのオブジェクトファイルを出力することを指示します。

-mcpu=thunderx

ThunderXプロセッサ向けのオブジェクトファイルを出力することを指示します。

-mcpu=thunderx2t99

ThunderX2プロセッサ向けのオブジェクトファイルを出力することを指示します。

*features*には、以下のいずれかを指定できます。+で区切って、複数の*features*を指定することもできます。

```
{ sve | nosve | fp16 }
```

sve

Armv8-Aアーキテクチャの拡張であるSVEを利用したオブジェクトファイルを出力することを指示します。

nosve

Armv8-Aアーキテクチャの拡張であるSVEを利用しないオブジェクトファイルを出力することを指示します。

SVEをサポートしないプロセッサ向けのオブジェクトファイルを作成したい場合は、本オプションを指定してください。

fp16

半精度浮動小数点型について言語仕様を拡張したオブジェクトファイルを出力することを指示します。言語仕様については"9.5.2 半精度(16ビット)浮動小数点型について"を参照してください。

省略時は、-mcpu=a64fxオプションが適用されます。

注意

-mcpuオプションと-marchオプションが同時に指定された場合、-marchオプションに指定されたアーキテクチャや*features*が優先されます。

-K*cpuname*

"表9.6 clangモードでは形式や名前が置換されるオプション" に掲載されている-K*cpuname*オプションは、clangモードのオプションに置き換わります。

9.1.2.2.6 コード生成関連オプション

コード生成に関するオプションについて説明します。

-f*opt*

*opt*には、以下のいずれかを指定します。

```
{ {PIC|pic} | {fj-largepage|fj-no-largepage} }
```

-f{PIC|pic}

位置独立コード(PIC)を生成することを指示します。

-f{fj-largepage|fj-no-largepage}

作成する実行可能プログラムがラージページ機能を使用するか否かを指示します。-ffj-largepageオプションが指定された場合、ラージページ機能を使用します。省略時は、-ffj-largepageオプションが適用されます。

-ffj-largepageオプションは、-fsanitizeオプションと同時に指定した場合、無効となります。

本オプションは、リンク時に指定する必要があります。

-mcmodel=*name*

実行可能プログラムおよび共有オブジェクトの、コード領域と静的データ領域の最大値を指示します。

*name*には、以下のいずれかを指定します。

```
{small||large}
```

省略時は、-mcmodel=smallオプションが適用されます。

-mcmodel=small

リンク後のコード領域と静的データ領域の合計が4GB以内になると仮定し、効率の良いオブジェクトプログラムを生成します。

-mcmodel=large

リンク後のコード領域が4GB以内になると仮定します。静的データ領域の大きさに制約はありません。静的データ領域が大きくリンク時にエラーが発生する場合は、本オプションを指定します。

-mfj-tls-size={12|24|32|48}

スレッド・ローカル・ストレージ(Thread-Local Storage)へのアクセスに必要なオフセットのサイズを指定します。単位はビットです。

スレッド・ローカル・ストレージのサイズに合わせ、-mfj-tls-size=12(4Kバイト)、-mfj-tls-size=24(16Mバイト)、-mfj-tls-size=32(4Gバイト)、-mfj-tls-size=48(256Tバイト)を指定することができます。

スレッド・ローカル・ストレージのサイズがオフセットの範囲を超えた場合、リンク時にエラーが生じます。

本オプションと-fltoオプションを同時に指定することはできません。

-K*opt*

“表9.6 clangモードでは形式や名前が置換されるオプション”に掲載されている-K*opt*オプションは、対応するclangモードのオプションに置き換わります。

9.1.2.3 翻訳時オプションの注意事項

翻訳時オプションの注意事項を説明します。

9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応

clangモードとtradモードの翻訳時オプション体系は異なります。そのため、プログラムの翻訳に使っていたコンパイラのモードを変更する場合には注意が必要です。

clangモードとtradモードにおいて、オプション名も機能概要も同じである翻訳時オプションを“表9.5 clang/tradモード共通の翻訳時オプション”に示します。機能概要は同じでオプション名が異なる翻訳時オプションを“表9.6 clangモードでは形式や名前が置換されるオプション”に示します。

- “表9.6 clangモードでは形式や名前が置換されるオプション”に記載がないtradモードの翻訳時オプションは、clangモードで指定できません。指定された場合、エラーメッセージを出力し、翻訳を中断します。
  - tradモードの-K{SVE|NOSVE}オプションは、clangモードでは指定できません。指定された場合、エラーメッセージを出力し、翻訳を中断します。SVEを利用するか否かを指示する場合は、-mcpuオプションまたは-marchオプションに+{sve|nosve}を付加してください。
- “表9.6 clangモードでは形式や名前が置換されるオプション”に記載があるtradモードの翻訳時オプションが指定された場合は、対応するclangモードの翻訳時オプションが指定されたものとみなします。
- 機能概要が同じ翻訳時オプションを指定しても、clangモードとtradモードでは最適化の適用結果が異なる場合があります。
  - -Kfastオプションが指定された場合、誘導されるオプションやデフォルトで有効になるアーキテクチャがtradモードとは異なります。
    - 誘導されるオプションについては、clangモードは“9.1.2.2.3 最適化関連オプション”の-Ofastオプションの説明を、tradモードは“2.2.2.5 -Kオプション”の-Kfastオプションの説明を参照してください。
    - デフォルトで有効になるアーキテクチャについては、clangモードは“9.1.2.2.5 CPU/アーキテクチャ関連オプション”の-m*arch*オプションを、tradモードは“2.2.2.5 -Kオプション”の-K*arch*オプションを参照してください。
- 以下のオプションは、tradモードとclangモードで省略時の動作が異なります。
  - -K{strict_aliasing|nostrict_aliasing}オプション

    tradモード:-Knostrict_aliasing

    clangモード:-O0が有効な時は-fnostrict_aliasing、-O1以上が有効な時は-fstrict-aliasing

ー -K{omitfp|noomitfp}オプション

tradモード:-Knoomitfp

clangモード:-fno-omit-frame-pointer、加えて-m{omit-leaf-frame-pointer|no-omit-leaf-frame-pointer}の指定がない時は-momit-leaf-frame-pointer

表9.5 clang/tradモード共通の翻訳時オプション

オプション名
-C
-D*name*[=*tokens*]
-E
-I*dir*
-L*dir*
-M
-MD
-MF*filename*
-MM
-MMD
-MP
-MT*target*
-P
-S
-SSL2
-SSL2BLAMP
-U*name*
-W*tool*,*arg1*[,*arg2*]...
-c
-g
-gO
-l*name*
-o *pathname*
-shared
-v
-w

表9.6 clangモードでは形式や名前が置換されるオプション

tradモードでのオプション名	clangモードでのオプション名
-KA64FX	-mcpu=a64fx
-KARMV8_1_A	-march=armv8.1-a
-KARMV8_2_A	-march=armv8.2-a
-KARMV8_3_A	-march=armv8.3-a
-KARMV8_A	-march=armv8-a
-KGENERIC_CPU	-mcpu=generic
-K{PIC|pic}	-f{PIC|pic}
-Kcmodel={small|large}	-mcmodel={small|large}
-K{eval|noeval}	-f{fast-math|no-fast-math}
-K{eval_concurrent|eval_noconcurrent}	-ffj-{eval-concurrent|no-eval-concurrent}
-Kfast	-Ofast
-K{fast_matmul|nofast_matmul}	-ffj-{fast-matmul|no-fast-matmul}
-K{fp_contract|nofp_contract}	-ffp-contract={fast|off}
-K{fp_relaxed|nofp_relaxed}	-ffj-{fp-relaxed|no-fp-relaxed}
-K{ilfunc[={loop|procedure}]|noilfunc}	-ffj-{ilfunc[={loop|procedure}]|no-ilfunc}
-K{largepage|nolargepage}	-ffj-{largepage|no-largepage}
-K{lib|nolib}	-f{builtin|no-builtin}
-K{loop_fission|loop_nofission}	-ffj-{loop-fission|no-loop-fission}
-Kloop_fission_threshold=*N*	-ffj-loop-fission-threshold=*N*
-Knoprefetch	-ffj-no-prefetch
-K{ocl|noocl}	-ffj-{ocl|no-ocl}
-K{omitfp|noomitfp}	-f{omit-frame-pointer|no-omit-frame-pointer}
-K{openmp|noopenmp}	-f{openmp|no-openmp}
-K{openmp_simd|noopenmp_simd}	-f{openmp-simd|no-openmp-simd}
-Koptmsg=2	-Rpass=.*
-K{preex|nopreex}	-ffj-{preex|no-preex}
-Kprefetch_cache_level={1|2|all}	-ffj-prefetch-cache-level={1|2|all}
-K{prefetch_conditional|prefetch_noconditional}	-ffj-{prefetch-conditional|no-prefetch-conditional}
-Kprefetch_iteration=*N*	-ffj-prefetch-iteration=*N*
-Kprefetch_iteration_L2=*N*	-ffj-prefetch-iteration-L2=*N*

tradモードでのオプション名	clangモードでのオプション名
-Kprefetch_line=*N*	-ffj-prefetch-line=*N*
-Kprefetch_line_L2=*N*	-ffj-prefetch-line-L2=*N*
-K{prefetch_sequential[={auto|soft}]|prefetch_nosequential}	-ffj-{prefetch-sequential[={auto|soft}]|no-prefetch-sequential}
-K{prefetch_stride|prefetch_nostride}	-ffj-{prefetch-stride|no-prefetch-stride}
-K{prefetch_strong|prefetch_nostrong}	-ffj-{prefetch-strong|no-prefetch-strong}
-K{prefetch_strong_L2|prefetch_nostrong_L2}	-ffj-{prefetch-strong-L2|no-prefetch-strong-L2}
-K{simd|nosimd}	-f{vectorize|no-vectorize}
-K{strict_aliasing|nostrict_aliasing}	-f{strict-aliasing|no-strict-aliasing}
-K{swp|noswp}	-ffj-{swp|no-swp}
-Kswp_strong	-ffj-swp
-Kswp_weak	
-K{unroll|nounroll}	-f{unroll-loops|no-unroll-loops}
-K{zfill[=*N*]|nozfill}	-ffj-{zfill[=*N*]|no-zfill}
-N{exceptions|noexceptions}	-f{exceptions|no-exceptions}
-N{fjcex|nofjcex}	-ffj-{fjcex|no-fjcex}
-N{fjprof|nofjprof}	-ffj-{fjprof|no-fjprof}
-N{hook_time|nohook_time}	-ffj-{hook-time|no-hook-time}
-N{line|noline}	-ffj-{line|no-line}
-Nlst	-ffj-lst
-Nlst=p	-ffj-lst=p
-Nlst=t	-ffj-lst=t
-Nlst_out=*file*	-ffj-lst-out=*file*
-Nsrc	-ffj-src
-V	--version
{-x-|-x0}	-f{inline-functions|no-inline-functions}

9.1.2.3.2 浮動小数点演算に対する最適化とその副作用

最適化オプションや最適化指示子で浮動小数点演算を最適化した場合、副作用を生じる可能性があります。

ここでは、その副作用(主に計算誤差)について説明します。

本処理系は、基本的にはIEEE 754に準拠したオブジェクトを作成しますが、“表9.7 浮動小数点演算に対する最適化の副作用”の計算誤差を伴う最適化により、数値演算の取り扱い方法がIEEE 754に準拠しなくなる場合があります。

各オプションの詳細は“9.1.2.2.3 最適化関連オプション”を、各最適化指示子の詳細は“9.2.2.1.2 最適化指示子の種類”を参照ください。

表9.7 浮動小数点演算に対する最適化の副作用

最適化オプション	最適化指示子	副作用の内容
-ffj-fast-matmul	—	行列積のループを高速なライブラリ呼び出しに変換する最適化により、計算誤差が生じることがあります。
-ffj-ilfunc[={loop|procedure}]	—	数学関数のインライン展開で、逆数近似演算命令や三角関数補助命令などを利用したアルゴリズムを用いるため、-ffj-fp-relaxedと同様の副作用が生じる可能性があります。さらに、-ffp-contract=offオプションや-ffj-no-fp-relaxedオプションの指定のある・なし、指定順序に関係なく、逆数近似演算命令およびFloating-Point Multiply-Add/Subtract命令を使用するようになります。
-ffj-fp-relaxed	—	単精度浮動小数点除算、倍精度浮動小数点除算、およびsqrt関数に対し、逆数近似演算命令を利用する最適化を行うので、以下のような副作用が生じる可能性があります。 ・ 実行結果に丸め誤差程度の違いが生じる。 ・ 引数および戻り値に現れる非正規化数を0で置き換える。 ・ 引数および戻り値に現れる負の0を正の0で置き換える。 ・ 引数または戻り値がNaN、±Inf、正規化数の最大値に近い値、正規化数の最小値に近い値のいずれかの場合、IEEE 754に準拠しない動作となる。
-Ofast	—	-ffast-mathオプションおよび-ffj-ilfuncオプションなどを誘導するため、計算誤差が生じることがあります。
-ffast-math	—	演算の評価順序を変更する最適化を行うので、計算誤差が生じることがあります。また、IEEE 754に準拠しない動作をすることがあります。 例 x*y + x*z → x*(y+z) また、flush-to-zeroモードを使用するため、アンダーフローによって非正規化数となる演算結果やソースオペランドが同符号の0.0に変わります。
-ffp-contract={fast|on}	fp contract({fast|on})	Floating-Point Multiply-Add/Substract演算命令を利用する最適化を行うので、丸め誤差程度の計算誤差が生じることがあります。

また、本処理系は#pragma STDC FENV_ACCESS ONをサポートしません。そのため、#pragma STDC FENV_ACCESS ONをプログラムに記述した場合でも、プログラムの論理上は発生しない浮動小数点例外が実行時に発生することがあります。例えば、以下の現象が発生することがあります。

- 標準ライブラリ関数に含まれるfetestexcept関数を用いて浮動小数点状態フラグにアクセスしたときに、プログラムの論理上はセットされないフラグがセットされている。
- GNU Cライブラリに含まれるfeenableexcept関数を用いて浮動小数点例外のトラップを有効にしたときに、プログラムの論理上は発生しないSIGFPEシグナルが生成される。

ただし、翻訳時オプション-O0を指定することで最適化が抑止され、現象を回避できる場合があります。

9.1.2.3.3 SVE利用時の注意事項

-ffj-prefetch-sequentialオプションまたは-ffj-prefetch-strideオプションが有効な場合でも、以下の条件をすべて満たすループではprefetch命令は生成されません。

- 誘導変数の増分値が1よりも大きい

- 配列の型のサイズが8バイトより小さい

9.1.2.3.4 SIMD組込み関数利用時の注意事項

SIMD組込み関数利用時に注意すべき点について説明します。

- 共有ライブラリ内に、SIMD組込み関数のvector typeを引数に持つ関数がある場合、以下のいずれかの対処を行ってください。以下の対処を行わない場合、実行結果に誤りが生じることがあります。

  ― リンク時に-Wl,-z,nowオプションを指定する。

  ― 実行時に、環境変数LD_BIND_NOWに1を設定する。

  環境変数LD_BIND_NOWについては、“9.1.6.1 実行時環境変数”を参照してください。

例

SIMD組込み関数のvector typeを引数に持つ関数

```
#include <arm_sve.h>
extern void sub1(svint64_t p_val);
int main() {
  svint64_t p_val;
  p_val = svdup_n_s64(123);
  sub1(p_val);    /* vector typeを引数に持つ関数の呼出し*/
  return 0;
}
```

- -msve-vector-bits=512オプションが指定された場合、SIMD組込み関数を使用することはできません。

9.1.2.3.5 SVEのベクトルレジスタサイズを指定する場合の注意事項

-msve-vector-bits=512オプションが指定された場合、SVEのベクトルレジスタのサイズを512ビットであるとみなして最適化を実施します。そのため、生成した実行可能プログラムは、SVEのベクトルレジスタが512ビットのCPUアーキテクチャにおいてのみ正常に動作します。SVEのベクトルレジスタが512ビット以外のCPUアーキテクチャでは実行時異常終了する可能性があります。また、実行結果は保証されません。

-msve-vector-bits=scalableオプションが指定された場合、実行可能プログラムは、CPUのSVEのベクトルレジスタのサイズに関わらず実行可能です。

SVEのベクトルレジスタのサイズを変更する場合の注意事項は、“C.2.8 SVEのベクトルレジスタのサイズの変更について”も参照してください。

9.1.3 翻訳コマンドの環境変数

ここでは、clangモードでのみ有効な環境変数について説明します。

clang/tradモード共通の翻訳コマンドの環境変数については、“2.3 翻訳コマンドの環境変数”を参照してください。

fccpx_clang_ENV
fcc_clang_ENV

翻訳時オプション設定用の環境変数です。

環境変数fccpx_clang_ENVはクロスコンパイラ用、環境変数fcc_clang_ENVはネイティブコンパイラ用です。

利用者は、これらの環境変数の値として、clangモード固有の翻訳時オプションおよびclang/tradモード共通の翻訳時オプションを設定することができます。clang/tradモード共通の翻訳時オプションについては、“9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応”を参照してください。

これらの環境変数に設定された翻訳時オプションは、clangモードでのみ有効となります。

翻訳時オプションの優先順位については、“2.4 翻訳時プロフィルファイル”を参照してください。

9.1.4 翻訳時プロフィルファイル

翻訳時プロフィルファイルを設定することにより、翻訳時オプションのデフォルトを変更することができます。

使用可能な翻訳時プロフィルファイルを以下に示します。

表9.8 翻訳時プロフィルファイル(clangモード)

翻訳コマンドの種別	モード	ファイル名
クロスコンパイラ	clang/tradモード共通	/etc/opt/FJSVstclanga/fccpx_PROF
	clangモード固有	/etc/opt/FJSVstclanga/fccpx_clang_PROF
ネイティブコンパイラ	clang/tradモード共通	/etc/opt/FJSVstclanga/fcc_PROF
	clangモード固有	/etc/opt/FJSVstclanga/fcc_clang_PROF

clang/tradモード共通の翻訳時プロフィルファイルには、clang/tradモード共通の翻訳時オプションを設定することができます。clang/tradモード共通の翻訳時オプションについては、“9.1.2.3.1 clangモードとtradモードにおける翻訳時オプションの対応”を参照してください。

clangモード固有の翻訳時プロフィルファイルには、clangモード固有の翻訳時オプションおよびclang/tradモード共通の翻訳時オプションを設定することができます。

翻訳時プロフィルファイルの記述形式および翻訳時オプションの優先順位については、“2.4 翻訳時プロフィルファイル”を参照してください。

9.1.5 あらかじめ定義されたマクロ名

あらかじめ定義されたマクロ(以降、既定義マクロと呼びます)のいくつかは、翻訳時オプション-std=*name*の指定に応じて値が変わります。

以下の表に、既定義マクロの値を一部示します。すべての既定義マクロを表示するためには、翻訳時オプション-Eおよび-dMを指定してください。

表9.9 翻訳時オプション-std=nameの指定に応じて値が変化する既定義マクロ

識別子	翻訳時オプション-std=*name*に指定された言語仕様のレベル					
	c89	gnu89	c99	gnu99	c11	gnu11
__STDC_VERSION__	-		199901L		201112L	
__STRICT_ANSI__	1	-	1	-	1	-
linux	-	1	-	1	-	1
unix	-	1	-	1	-	1

-: 定義なし

表9.10 値が変化しない既定義マクロ

識別子	値
_LP64	1
_OPENMP (注1)	201511
_REENTRANT (注2)	1
__ARM_ARCH	8
__ASSEMBLER__ (注3)	1
__CLANG_FUJITSU	1
__DATE__	翻訳日付(asctime関数の形式)
__ELF__	1
__EXCEPTIONS (注4)	1
__FCC_major__	コンパイラのメジャーバージョン番号
__FCC_minor__	コンパイラのマイナーバージョン番号
__FCC_patchlevel__	コンパイラのパッチレベル番号
__FCC_version__	コンパイラのバージョンを表す文字列

識別子	値
__FILE__	ソースファイル名
__GNUC__	4
__GNUC_MINOR__	2
__GNUC_PATCHLEVEL__	1
__LINE__	ソースファイルの行番号
__LP64__	1
__OPTIMIZE__ (注5)	1
__PIC__	1 (注6)
	2 (注7)
__PRAGMA_REDEFINE_EXTNAME	1
__PTRDIFF_TYPE__	long int
__SIZE_TYPE__	long unsigned int
__STDC__	1
__STDC_HOSTED__	1
__TIME__	翻訳時刻(asctime関数の形式)
__USER_LABEL_PREFIX__	空文字列
__WCHAR_TYPE__	unsigned int (注8)
__aarch64__	1
__clang_major__	Clang/LLVMのメジャーバージョン番号
__clang_minor__	Clang/LLVMのマイナーバージョン番号
__clang_patchlevel__	Clang/LLVMのパッチレベル番号
__clang_version__	Clang/LLVMのバージョンを表す文字列
__linux	1
__linux__	1
__pic__	1 (注6)
	2 (注7)
__unix	1
__unix__	1

-: 定義なし

注1) 翻訳時オプション-Kopenmpまたは-fopenmp有効時

注2) 翻訳時オプション-pthread有効時

注3) 翻訳時オプション-x assembler-with-cpp有効時

注4) 翻訳時オプション-Nexceptionsまたは-fexceptions有効時

注5) 翻訳時オプション-O1以上有効時

注6) 翻訳時オプション-fpicまたは-fpie有効時

注7) 翻訳時オプション-fPICまたは-fPIE有効時

注8) tradモードと値が異なるため、tradモードのオブジェクトと結合している場合に意図しない動作をする可能性があります。

9.1.6 実行の手続き

ここでは、プログラムを実行するための手続きについて説明します。

9.1.6.1 実行時環境変数

実行時の制御は環境変数を指定することで変更できます。“表9.11 実行時環境変数(clangモード)”に、実行時に指定できる環境変数を示します。

なお、OpenMP仕様の環境変数については、“4.3.2.2 OpenMP仕様の環境変数”をお読みください。

表9.11 実行時環境変数(clangモード)

環境変数名	オペランド	意味
FLIB_HOOK_TIME	*time*	一定時間間隔呼出しによるフック機能において、*time*ミリ秒ごとにユーザー定義関数が呼び出されます。 *time*は0～2147483647の値です。*time*に0が指定された場合、一定時間間隔での呼出しは無効となります。 詳細については、“8.3 フック機能”をお読みください。
LD_BIND_NOW	1	共有ライブラリのアドレスのシンボル解決に遅延リンクを使わないことを指示します。

9.1.6.2 実行時の注意事項

clangモードで作成されたプログラムを実行する場合の注意事項を、以下に示します。

- 実行時の変数割付け

  本処理系で作成されたプログラムは、関数内でローカルな変数およびプライベート変数をスタック領域に割り付けます。関数内でローカルな変数およびプライベート変数に必要な領域が大きい場合には、スタック領域を十分な大きさに拡張する必要があります。プロセスのスタック領域は、ulimitコマンド(bash組み込みコマンド)などで設定できます。

9.2 最適化機能

9.2.1 最適化の概要

最適化の概要については、“3.1 最適化の概要”を参照してください。

注意

最適化の適用結果

clangモードとtradモードでは最適化の適用結果が異なる場合があります。

翻訳時メッセージ

clangモードでは、英文の翻訳時メッセージだけ出力します。

また、clangモードとtradモードでは、メッセージの出力形式が異なります。

9.2.2 最適化機能の活用方法

最適化機能を有効に活用するための機能について説明します。

9.2.2.1 最適化制御行(プラグマディレクティブ)の利用

最適化にとって有効な情報をソースプログラム上に記述することにより、最適化の効果をより高めることができます。

clangモードでは、以下の2種類の最適化制御行(プラグマディレクティブ)を使用することができます。

- clangモードがサポートする富士通コンパイラ独自の最適化制御行

・ Clang/LLVMでサポートしている最適化制御行

clangモードがサポートする富士通コンパイラ独自の最適化制御行は、-ffj-oclまたは-ffj-no-oclオプションで制御することができます。

Clang/LLVMでサポートしている最適化制御行は常に有効になります。

注意

本章内に掲載していない最適化制御行を指定した場合、動作保証しません。

9.2.2.1.1 最適化制御行(プラグマディレクティブ)の種類

clangモードがサポートする富士通コンパイラ独自の最適化制御行(プラグマディレクティブ)は、以下の形式で指定します。指定された最適化指示子を直後のループで有効にします。

書式

```
#pragma fj 最適化指示子
```

挿入位置

対象にしたいループの直前に記述します。また、Clang/LLVMの最適化制御行のloop行またはほかの富士通コンパイラ独自の最適化制御行のloop行を、ループとの間に挟むことができます。

clangモードがサポートするClang/LLVMの最適化制御行(プラグマディレクティブ)は、以下の形式で指定します。指定された最適化指示子を、直後のループまたは複合文で有効にします。

書式

```
#pragma clang 最適化指示子
```

挿入位置

- loopから始まる最適化指示子の場合

  対象にしたいループの直前に記述します。また、ほかのClang/LLVMの最適化制御行のloop行または富士通コンパイラ独自の最適化制御行のloop行を、ループとの間に挟むことができます。

- fpから始まる最適化指示子の場合

  対象にしたい複合文の中のすべての明示的な宣言および文の前に記述します。

9.2.2.1.2 最適化指示子の種類

clangモードで指定可能な最適化指示子を“表9.12 指定可能な最適化指示子”に示します。

注意

tradモードの最適化指示子を指定した場合は、clangモードの最適化指示子に置き換わります。

置き換わる最適化指示子の対応は、“表9.13 clangモードでは形式や名前が置換される最適化指示子”を参照してください。

“表9.12 指定可能な最適化指示子”に記載がない最適化指示子を指定した場合は、無視されます。

表9.12 指定可能な最適化指示子

最適化指示子	最適化制御行書式	機能概要
loop clone *var*==*n*(注)	#pragma fj loop clone *var*==*n*	指定された変数*var*と値*n*の等式を条件式とする分岐を生成し、ループを複製することを指示します。
loop eval_concurrent	#pragma fj loop eval_concurrent	tree-height-reduction最適化において、浮動小数点演算命令の並列性を優先することを指示します。

最適化指示子	最適化制御行書式	機能概要
		-ffj-eval-concurrentオプションで適用される最適化をループ単位で指定することができます。
loop eval_noconcurrent	#pragma fj loop eval_noconcurrent	tree-height-reduction最適化において、浮動小数点演算命令の並列性を抑え、FMA命令の利用を優先することを指示します。
loop loop_fission_target	#pragma fj loop loop_fission_target [cl]	コンパイラによる自動ループ分割を行うことを指示します。最内ループに対してのみ指定できます。clを指定した場合のコンパイラの動作は、省略時と同じです。
loop loop_fission_threshold *n*	#pragma fj loop loop_fission_threshold *n*	自動ループ分割における分割後のループの粒度を決める閾値*n*を指示します。*n*は、1から100までの整数値です。最内ループに対してのみ指定できます。
loop preex	#pragma fj loop preex	不変式の先行評価を行うことを指示します。 -ffj-preexオプションで適用される最適化をループ単位で指定することができます。
loop nopreex	#pragma fj loop nopreex	不変式の先行評価を行わないことを指示します。
loop prefetch	#pragma fj loop prefetch	コンパイラの自動prefetch機能を有効にすることを指示します。 -ffj-prefetch-sequentialオプションまたは-ffj-prefetch-strideオプションで適用される最適化をループ単位で指定することができます。
loop noprefetch	#pragma fj loop noprefetch	コンパイラの自動prefetch機能を無効にすることを指示します。
loop prefetch_sequential [auto|soft]	#pragma fj loop prefetch_sequential [auto|soft]	連続的にアクセスされる配列データに対してprefetch命令を生成することを指示します。 -ffj-prefetch-sequential[={auto|soft}]オプションで適用される最適化をループ単位で指定することができます。
loop prefetch_nosequential	#pragma fj loop prefetch_nosequential	連続的にアクセスされる配列データに対してprefetch命令を生成しないことを指示します。
loop prefetch_stride	#pragma fj loop prefetch_stride	ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成することを指示します。
loop prefetch_nostride	#pragma fj loop prefetch_nostride	ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成しないことを指示します。
loop prefetch_strong	#pragma fj loop prefetch_strong	1次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。 -ffj-prefetch-strongオプションで適用される最適化をループ単位で指定することができます。
loop prefetch_nostrong	#pragma fj loop prefetch_nostrong	1次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。
loop prefetch_strong_L2	#pragma fj loop prefetch_strong_L2	2次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。 -ffj-prefetch-strong-L2オプションで適用される最適化をループ単位で指定することができます。
loop prefetch_nostrong_L2	#pragma fj loop prefetch_nostrong_L2	2次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。

最適化指示子	最適化制御行書式	機能概要
loop swp	#pragma fj loop swp	ソフトウェアパイプライニングを行うことを指示します。
loop noswp	#pragma fj loop noswp	ソフトウェアパイプライニングを行わないことを指示します。
loop zfill [*M*]	#pragma fj loop zfill [*M*]	zfillの最適化を有効にします。 *M*はDC ZVA命令の書き込みブロック数を表す1～100の整数値です。
loop nozfill	#pragma fj loop nozfill	zfillの最適化を無効にします。
fp contract(fast)	#pragma clang fp contract(fast)	FMA命令を出力することを指示します。 -ffp-contract=fastオプションで適用される最適化を、セクション単位で指定することができます。
fp contract(on)	#pragma clang fp contract(on)	FMA命令を出力することを指示します。 #pragma STDC FP_CONTRACT(ON)を用いた場合と等価です。
fp contract(off)	#pragma clang fp contract(off)	FMA命令を出力しないことを指示します。
loop unroll(enable)	#pragma clang loop unroll(enable)	ループをアンローリングすることを指示します。 -funroll-loopsオプションで適用される最適化をループ単位で指定することができます。unroll(full)を指定した場合はループを全展開します。
loop unroll(full)	#pragma clang loop unroll(full)	
loop unroll_count(*n*)	#pragma clang loop unroll_count(*n*)	
loop unroll(disable)	#pragma clang loop unroll(disable)	ループをアンローリングしないことを指示します。
loop vectorize(assume_safety)	#pragma clang loop vectorize(assume_safety)	ループ中で配列の要素またはポインタ変数に対して依存がないことを指示します。
loop vectorize(enable)	#pragma clang loop vectorize(enable)	SIMD化を有効にすることを指示します。 -fvectorizeオプションで適用される最適化をループ単位で指定することができます。vectorize_width(*n*, scalable)を指定した場合は、SVEを利用したSIMD化においてSIMD長が*n*の定数倍になります。
loop vectorize_width(*n*, scalable)	#pragma clang loop vectorize_width(*n*, scalable)	
loop vectorize(disable)	#pragma clang loop vectorize(disable)	SIMD化を無効にすることを指示します。

注) *var*は使用する前に宣言してください。

loop clone指示子

指定された変数*var*と値*n*の等式を条件式とする分岐を生成し、ループを複製することを指示します。同一ループに対して複数のloop clone指示子を指定した場合、分岐は指定した値の順に生成されます。多重ループの場合、異なるネストレベルのループにloop clone指示子を指定することはできません。本機能により、フルアンローリングなどのほかの最適化を促進します。clone最適化はループを複製するため、オブジェクトプログラムの大きさおよび翻訳時間が増加する場合があります。本最適化指示子は、-O3オプションが有効な場合に意味があります。

*var*はint型の変数です。ただし、int型であっても、以下の場合は指定できません。

- 構造体メンバ変数
- 共用体メンバ変数
- 配列要素
- threadprivate対象変数

*n*は、-2147483648から2147483647までの整数値です。

例

最適化指示子の指定

- 例1: loop clone指示子を1つのみ指定

```
#pragma fj loop clone N==10
for(i = 0; i < N; ++i) {
   a[i] = i;
}
```

[最適化適用後のソースイメージ]

```
if(N == 10) {
    for(i = 0; i < N; ++i) {
        a[i] = i;
    }
} else {
    for(i = 0; i < N; ++i) {
        a[i] = i;
    }
}
```

分岐を生成し、ループを複製します。

- 例2: 同一ループに複数のloop clone指示子を指定

```
#pragma fj loop clone N==10
#pragma fj loop clone M==20
for(i = 0; i < N; i++) {
    a[i] = M;
}
```

[最適化適用後のソースイメージ]

```
if(N == 10) {
    for(i = 0; i < N; ++i) {
        a[i] = M;
    }
} else if (M == 20) {
    for(i = 0; i < N; ++i) {
        a[i] = M;
    }
} else {
    for(i = 0; i < N; ++i) {
        a[i] = M;
    }
}
```

同一ループに対して複数のloop clone指示子を指定した場合、指定した順番にループが複製されます。

- 例3: フラグを利用したloop clone指示子を指定

```
int flag = (start == 0 && end == 10) ? 1 : 0;
#pragma fj loop clone flag==1
for(i = start; i < end; ++i) {
    a[i] = i;
}
```

[最適化適用後のソースイメージ]

```
int flag = (start == 0 && end == 10) ? 1 : 0;
if(frag == 1) {
    for(i = start; i < end; ++i) {
        a[i] = i;
```

```
    }
} else {
    for(i = start; i < end; ++i) {
        a[i] = i;
    }
}
```

分岐を生成し、ループを複製します。

- 例4: loop clone指示子の誤った指定

```
#pragma fj loop clone M==10
for(i = 0; i < M; ++i) {
#pragma fj loop clone N==20
    for(j = 0; j < N; ++j) {
        a[i][j] = 0;
    }
}
```

多重ループの場合、異なるネストレベルのループにloop clone指示子を指定することはできません。

loop eval_concurrent指示子

tree-height-reduction最適化において、浮動小数点演算命令の並列性を優先することを指示します。

 例

最適化指示子の指定

```
#pragma fj loop eval_concurrent
for(i = 0; i < n; i++) {
    x[i] = a[i] * b[i] + c[i] * d[i] + e[i] * f[i] + g[i] * h[i];
}
```

loop eval_noconcurrent指示子

tree-height-reduction最適化において、浮動小数点演算命令の並列性を抑え、FMA命令の利用を優先することを指示します。

 例

最適化指示子の指定

```
#pragma fj loop eval_noconcurrent
for(i = 0; i < n; i++) {
    x[i] = a[i] * b[i] + c[i] * d[i] + e[i] * f[i] + g[i] * h[i];
}
```

loop loop_fission_target指示子

指定されたループに対して、自動ループ分割を行うことを指示します。

最内ループに対してのみ指定できます。

指示子に続いてclを指定した場合のコンパイラの動作は、省略時と同じです。

例

最適化指示子の指定

```
#pragma fj loop loop_fission_target
for (i = 0; i < n; i++) {
    ...
}
```

loop loop_fission_threshold指示子

指示子に続いて指定する1から100の整数値*n*によって、分割後のループの粒度の閾値を指示します。

最内ループに対してのみ指定できます。

例

最適化指示子の指定

```
#pragma fj loop loop_fission_target
#pragma fj loop loop_fission_threshold 10
for (i = 0; i < n; i++) {
    ...
}
```

loop preex指示子

不変式の先行評価の最適化を行うことを指示します。

例

最適化指示子の指定

```
#pragma fj loop preex
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] / b[k];
    }
}
```

[最適化適用後のソースイメージ]

```
t = 1 / b[k];
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] * t;
    }
}
```

対象となるループ中で、不変式の先行評価の最適化を行います。

loop nopreex指示子

不変式の先行評価の最適化を抑止することを指示します。

例

最適化指示子の指定

```
#pragma fj loop nopreex
for(i = 0; i < n; i++) {
    if(m[i] != 0) {
        a[i] = a[i] / b[k];
    }
}
```

対象となるループ中で、不変式の先行評価の最適化を行いません。

loop prefetch指示子

コンパイラの自動プリフェッチ機能を有効にすることを指示します。自動プリフェッチ機能とは、コンパイラが自動的にprefetch命令を挿入するのに最適な位置を判断し、prefetch命令を生成する機能です。

例

最適化指示子の指定

```
#pragma fj loop prefetch
for(i = 0; i < 10; i=i+m) {
    a[i] = b[i];
}
```

loop noprefetch指示子

コンパイラの自動プリフェッチ機能を無効にすることを指示します。

例

最適化指示子の指定

```
#pragma fj loop noprefetch
for(i = 0; i < 10; i=i+m) {
    a[i] = b[i];
}
```

forループに対して自動プリフェッチ機能が無効になります。

loop prefetch_sequential指示子

連続的にアクセスされる配列データに対してprefetch命令を生成することを指示します。

prefetch_sequential指示子に続けてautoを指定した場合、ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用するか、prefetch命令を生成するか否かを、コンパイラが自動的に選択することを指示します。

prefetch_sequential指示子に続けてsoftを指定した場合、ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用せずに、prefetch命令を生成することを指示します。

prefetch_sequential指示子に続くautoやsoftを省略した場合は、autoを指定したものとみなします。

例

最適化指示子の指定

- 例1:

```
#pragma fj loop prefetch_sequential auto
for(i = 0; i < n; i++) {
    a1[i] = a2[i] + a3[i] + a4[i] + a5[i]
      + a6[i] + a7[i] + a8[i] + a9[i] + a10[i]
      + a11[i] + a12[i] + a13[i] + a14[i] + a15[i]
      + a16[i] + a17[i];
}
```

対象となるループにおいて、ハードウェアプリフェッチを利用するか、prefetch命令を生成するか否かを、コンパイラが自動的に選択します。

- 例2:

```
#pragma fj loop prefetch_sequential soft
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループにおいて、ハードウェアプリフェッチを利用せずにprefetch命令を生成します。

loop prefetch_nosequential指示子

連続的にアクセスされる配列データに対してprefetch命令を生成しないことを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_nosequential
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループ内の連続的にアクセスされる配列データに対してprefetch命令を生成しません。

loop prefetch_stride指示子

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成することを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_stride
for(i = 0; i < n; i=i+m) {
    a[i] = b[i];
}
```

対象となるループにおいて、prefetch命令を生成します。

loop prefetch_nostride指示子

ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch命令を生成しないことを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_nostride
for(i = 0; i < n; i=i+m) {
    a[i] = b[i];
}
```

対象となるループにおいて、prefetch命令を生成しません。

loop prefetch_strong指示子

1次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_strong
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループで生成される1次キャッシュのprefetch命令はstrong prefetchになります。

loop prefetch_nostrong指示子

1次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_nostrong
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループで生成される1次キャッシュのprefetch命令はstrong prefetchとなりません。

loop prefetch_strong_L2指示子

2次キャッシュに対して生成されるprefetch命令をstrong prefetchとすることを指示します。

例

最適化指示子の指定

```
#pragma fj loop prefetch_strong_L2
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループで生成される2次キャッシュのprefetch命令はstrong prefetchになります。

loop prefetch_nostrong_L2指示子

2次キャッシュに対して生成されるprefetch命令をstrong prefetchとしないことを指示します。

最適化指示子の指定

```
#pragma fj loop prefetch_nostrong_L2
for(i = 0; i < n; i++) {
    a[i] = a[i] + b[i];
}
```

対象となるループで生成される2次キャッシュのprefetch命令はstrong prefetchとなりません。

loop swp指示子

ソフトウェアパイプライニングを行うことを指示します。

最適化指示子の指定

```
#pragma fj loop swp
for (i = 0; i < n; i++) {
    a[i] = b[i]/c[i];
}
```

loop noswp指示子

ソフトウェアパイプライニングを行わないことを指示します。

最適化指示子の指定

```
#pragma fj loop noswp
for (i = 0; i < n; i++) {
    a[i] = b[i]/c[i];
}
```

loop zfill指示子

ループ内で書き込みのみを行う配列データについて、データをメモリからロードすることなく、キャッシュ上に書き込み用の領域を確保する命令を使って、書き込み動作を高速にする最適化を行うことを指示します。zfill指示子に続けて指定する1から100までの整数値*N*によって、256バイトを1ブロックとする単位で*N*ブロック分先のデータをzfill最適化の対象とします。*N*の値が省略された場合、コンパイラが自動的に値を決定します。

zfillについては、“3.3.5 zfill”をお読みください。

以下に、指定の例を示します。

最適化指示子の指定

- 例1

```
#pragma fj loop zfill
for(i = 0; i < n; i++) {
    ...
}
```

何ブロック先のデータをzfillの最適化の対象にするかはコンパイラが自動で判断します。

- 例2

```
#pragma fj loop zfill 1
for(i = 0; i < n; i++) {
    ...
}
```

1ブロック先のデータをzfillの最適化の対象にすることを指示します。

loop nozfill指示子

zfillの最適化を実施しないことを指示します。

以下に、指定の例を示します。

 例

最適化指示子の指定

```
#pragma fj loop nozfill
for(i = 0; i < n; i++) {
    ...
}
```

データはzfillの最適化の対象となりません。

fp contract(fast)指示子

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うことを指示します。なお、Floating-Point Multiply-Add/Subtract演算命令を使用した場合、演算結果に丸め誤差程度の違いが生じることがあります。

 例

最適化指示子の指定

```
for(i = 0; i < n; i++) {
    #pragma clang fp contract(fast)
    a[i] = a[i] + b[i] * c[i];
}
```

対象となる複合文に対して、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行います。

fp contract(on)指示子

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を、言語規格で規定された#pragma STDC FP_CONTRACT ONを用いた場合と同様に行うことを指示します。なお、Floating-Point Multiply-Add/Subtract演算命令を使用した場合、演算結果に丸め誤差程度の違いが生じることがあります。

例

最適化指示子の指定

```
for(i = 0; i < n; i++) {
    #pragma clang fp contract(on)
    a[i] = a[i] + b[i] * c[i];
}
```

対象となる複合文に対して、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行います。

fp contract(off)指示子

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行わないことを指示します。

例

最適化指示子の指定

```
for(i = 0; i < n; i++) {
    #pragma clang fp contract(off)
    a[i] = a[i] + b[i] * c[i];
}
```

対象となる複合文に対して、Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行いません。

loop unroll(enable)指示子

ループアンローリングの最適化を行うことを指示します。

ループ展開数は、コンパイラが自動的に最適な値を決定します。

なお、この指示子は指定した直後のループだけが対象になります。

例

最適化指示子の指定

```
#pragma clang loop unroll(enable)
for(i = 0; i < n; i++) {
    ...
}
```

loop unroll(full)指示子

ループアンローリングの最適化を行うことを指示します。

指定した直後のループを対象として、ソースプログラムの繰返し数だけ実行文を展開します。繰返し数が不明な場合は、最適化を行いません。

なお、この指示子は指定した直後のループだけが対象になります。

例

最適化指示子の指定

```
#pragma clang loop unroll(full)
for(i = 0; i < 8; i++) {
```

```
    ...
}
```

対象となるループ中の文を全展開します。

loop unroll_count指示子

指示子に続いて2から100までの整数値で、ループ展開数の上限を指定します。上限の指定を省略することはできません。

なお、この指示子は指定した直後のループだけが対象になります。

例

最適化指示子の指定

```
#pragma clang loop unroll_count(8)
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループ中の文を展開数8で展開します。

loop unroll(disable)指示子

ループアンローリングの最適化を抑止することを指示します。

なお、この指示子は指定した直後のループだけが対象になります。

例

最適化指示子の指定

```
#pragma clang loop unroll(disable)
for(i = 0; i < n; i++) {
    ...
}
```

対象となるループ中の文に対してループアンローリング最適化を行いません。

loop vectorize(assume_safety)指示子

ループ中の配列の要素またはポインタ変数に対して依存がないことを指示します。この指定により配列、またはポインタ変数の定義引用順序が不明でSIMD化できなかったループに対してSIMD化の対象にします。ただし、演算の種類やループ構造によっては指定したとしてもSIMD化の対象とならない場合があります。

注意

loop vectorize(assume_safety)指示子がループ中で依存する配列の要素またはポインタ変数に対して誤って指定された場合、実行結果は保証されません。

例

最適化指示子の指定

```
#pragma clang loop vectorize(assume_safety)
for (int i = 0; i < count; i++) {
```

```
    a[index[i]] = a[index[i]]+ 1;
}
```

loop vectorize(enable)指示子

SIMD化することを指示します。ただし、演算の種類やループ構造によりSIMD化しない場合もあります。

SIMD長はコンパイラが自動的に決定する値になります。

例

最適化指示子の指定

```
double a[10], b[10];
...
#pragma clang loop vectorize(enable)
for(i = 0; i < 10; i++) {
    a[i] = a[i] + b[i];
}
```

loop vectorize_width(*n*, scalable)指示子

SVEのベクトルレジスタを特定のサイズとみなさずSVEを利用してSIMD化することを指示します。ただし、演算の種類やループ構造によりSIMD化しない場合もあります。

本最適化指示子は、-msve-vector-bits=scalableオプションが有効な場合に意味があります。

SVEのベクトルレジスタサイズを最小の128ビットとみなした場合の、1つのSIMD命令で処理するデータの要素数を、*n*で指定します。1つのSIMD命令で実際に処理されるデータの要素数(SIMD長)は、実行時のCPUアーキテクチャに実装されたSVEのベクトルレジスタサイズに依存します。例として、SVEのベクトルレジスタサイズが512ビットの場合、(512÷128)×*n*となり、4×*n*がSIMD長になります。1つのSIMD命令でそのSIMD長を処理できない場合は、複数のSIMD命令で処理します。*n*に指定できる値は、2または4です

例

最適化指示子の指定

```
float a[100];
double b[100];
...
#pragma clang loop vectorize_width(4, scalable)
for(i = 0; i < 100; i++) {
    a[i] += 1.0;
    b[i] += 1.0;
}
```

例えばSVEのベクトルレジスタサイズが512ビットの場合は、a[0]からa[15]まで、b[0]からb[7]まで、b[8]からb[15]までをそれぞれ1つのSIMD命令で処理します。

loop vectorize(disable)指示子

SIMD化しないことを指示します。

例

最適化指示子の指定

```
double a[10], b[10];
...
#pragma clang loop vectorize(disable)
```

```
for(i = 0; i < 10; i++) {
    a[i] = a[i] + b[i];
}
```

9.2.2.1.3 最適化制御行/最適化指示子の注意事項

最適化制御行および最適化指示子の注意事項を説明します。

最適化制御行の種類によるコンパイラの動作の違い

"9.2.2.1.1 最適化制御行(プラグマディレクティブ)の種類"で説明した最適化制御行の種類によって、以下のようにコンパイラの動作が異なる場合があります。

- 同種の最適化制御行を同一箇所に複数指定した場合

指定した最適化指示子		コンパイラの動作	
		Clang/LLVMでサポートしている最適化制御行	富士通コンパイラ独自の最適化制御行
同一の最適化指示子	loop clone指示子	（対応する指示子はありません）	翻訳可能です。 指定されたすべてのloop clone指示子が有効になります。
	その他	翻訳時エラーになります。	翻訳可能です。 最後に指定した最適化制御行が有効になります。
ある最適化の適用と抑止のように、互いに相反する最適化指示子	以下の最適化指示子 ・ fp contract(fast) ・ fp contract(on) ・ fp contract(off)	翻訳可能です。 最後に指定した最適化制御行が有効になります。	
	その他 例 loop unroll(enable)指示子とloop unroll(disable)指示子	翻訳時エラーになります。	
ある最適化について適用方法が異なる最適化指示子 例 loop unroll(full)指示子とloop unroll_count(*n*)指示子		翻訳時エラーになります。	
パラメタ値が異なる最適化指示子 例 数値*n*が異なるloop unroll_count(*n*)指示子		翻訳時エラーになります。	

- 最適化制御行に誤りがある場合

指定誤りの内容	コンパイラの動作	
	Clang/LLVMでサポートしている最適化制御行	富士通コンパイラ独自の最適化制御行
最適化制御行の指示子または指示子に続けて指定した要素の誤り	翻訳時エラーになります。	翻訳可能です。

指定誤りの内容	コンパイラの動作	
	Clang/LLVMでサポートしている最適化制御行	富士通コンパイラ独自の最適化制御行

例
・ 例1
`#pragma loop unrecognized_specifier`
・ 例2
`#pragma loop prefetch_sequential unrecognized_parameter`
		警告メッセージが出力されて最適化制御行が無視されます。

clangモードとtradモードにおける最適化指示子の対応

clangモードとtradモードにおいて、最適化内容が同じで名前が異なる最適化指示子を、“表9.13 clangモードでは形式や名前が置換される最適化指示子”に示します。

― 表に記載があるtradモードの最適化制御行は、対応するclangモードの最適化制御行が使用されたものとみなします。

― 最適化内容が同じでも、最適化の適用結果が異なる場合があります。

― 表に記載がないtradモードの最適化制御行を使用した場合は、警告メッセージが出力されて無視されます。

表9.13 clangモードでは形式や名前が置換される最適化指示子

tradモード	clangモード
`#pragma loop clone` *var*`==`*n*	`#pragma fj loop clone` *var*`==`*n* (注1)
`#pragma loop eval_concurrent`	`#pragma fj loop eval_concurrent`
`#pragma loop eval_noconcurrent`	`#pragma fj loop eval_noconcurrent`
`#pragma loop loop_fission_target [cl]`	`#pragma fj loop loop_fission_target [cl]`
`#pragma loop loop_fission_threshold` *n*	`#pragma fj loop loop_fission_threshold` *n*
`#pragma loop preex`	`#pragma fj loop preex`
`#pragma loop nopreex`	`#pragma fj loop nopreex`
`#pragma loop prefetch`	`#pragma fj loop prefetch`
`#pragma loop noprefetch`	`#pragma fj loop noprefetch`
`#pragma loop prefetch_sequential [auto|soft]`	`#pragma fj loop prefetch_sequential [auto|soft]`
`#pragma loop prefetch_nosequential`	`#pragma fj loop prefetch_nosequential`
`#pragma loop prefetch_stride`	`#pragma fj loop prefetch_stride`
`#pragma loop prefetch_nostride`	`#pragma fj loop prefetch_nostride`
`#pragma loop prefetch_strong`	`#pragma fj loop prefetch_strong`
`#pragma loop prefetch_nostrong`	`#pragma fj loop prefetch_nostrong`
`#pragma loop prefetch_strong_L2`	`#pragma fj loop prefetch_strong_L2`
`#pramga loop prefetch_nostrong_L2`	`#pragma fj loop prefetch_nostrong_L2`
`#pramga loop swp`	`#pragma fj loop swp`
`#pramga loop noswp`	`#pragma fj loop noswp`
`#pragma loop zfill [`*N*`]`	`#pragma fj loop zfill [`*N*`]`
`#pragma loop nozfill`	`#pragma fj loop nozfill`
`#pragma[ fj] loop simd` (注2)	`#pragma clang loop vectorize(enable)`

tradモード	clangモード
#pragma[fj] loop nosimd	#pragma clang loop vectorize(disable)
#pragma[fj] loop unroll	#pragma clang loop unroll(enable)
#pragma[fj] loop unroll *n*	#pragma clang loop unroll_count(*n*)
#pragma[fj] loop unroll "full"	#pragma clang loop unroll(full)
#pragma[fj] loop nounroll	#pragma clang loop unroll(disable)

注1) tradモードとclangモードとでは、引数に指定可能な表記法、型、および値が異なります。

注2) 引数なし。

9.2.3 SIMD化

一般的なSIMD化については、“3.2.7.1 一般的なSIMD化”を参照してください。

9.2.3.1 SIMD化可能な数学関数

以下は、-fvectorizeオプションが有効な場合にSIMD化可能な数学関数です。

```
ceil, ceilf, copysign, copysignf, fabs, fabsf, floor, floorf, fmax, fmaxf, fmin, fminf, round, roundf, trunc, truncf
```

以下は、-fvectorizeオプションおよび-fbuiltin オプションが有効な場合にSIMD化可能な数学関数です。

```
abs, cimag, cimagf, creal, crealf, fma, fmaf, nearbyint, nearbyintf, rint, rintf
```

以下は、-fvectorizeオプション、-fbuiltin オプション、および-ffast-mathオプションが有効な場合にSIMD化可能な数学関数です。

```
cabs, cabsf, sqrt, sqrtf
```

-fvectorizeオプション、-fbuiltinオプション、および-ffj-ilfuncオプションが有効な場合、インライン展開の対象となる数学関数がSIMD化可能です。
インライン展開の対象となる数学関数については、“9.1.2.2.3 最適化関連オプション”の-ffj-ilfuncオプションの説明も参照してください。

9.3 並列化機能

9.3.1 並列化処理の概要

並列処理とは

並列処理という言葉は広い意味に使われていますが、ここで述べる並列処理とは、独立に動作可能な複数のCPUを同時に使って、1つのプログラムを実行することを意味します。ここでは、複数のプログラムを同時に実行するマルチジョブのことを、並列処理とは言わないことにします。

並列処理の効果

並列処理の効果は、複数のCPUを同時に使うことによって、プログラムの経過時間が短縮されることです。例えば、ループを分割して2つのループを並列実行できたとすると、このループを実行するのにかかる時間は、理想的には半分になります。

しかし、並列処理によってプログラム実行の経過時間の短縮は可能ですが、プログラム実行に要するCPU時間を減らすことはできません。なぜなら、並列処理では複数のCPUを利用しますが、それぞれのCPUの消費するCPU時間の合計は、プログラムを逐次的に実行した場合のCPU時間と同等か、または、並列処理に必要なオーバヘッドの分だけ増加するためです。

並列処理プログラムの性能は、一般に、実行に要したCPU時間の短縮ではなく、経過時間の短縮で評価することに注意してください。逐次的に動作していたプログラムを並列処理によって実行した場合、並列処理のためのオーバヘッドによってプロセッサ全体で消費するCPU時間の合計は、逐次処理時のCPU時間よりも大きくなります。

並列処理で効果を得るための条件

並列処理で経過時間を短縮するためには、複数のCPUを同時に使える計算機環境が必要です。本処理系を使用して作成した並列処理プログラムは、単一のCPUしかもたないハードウェアでも実行可能ですが、この場合には経過時間の短縮は望めません。また、複数のCPUをもつハードウェア上でも、他のジョブが動いていて計算機全体としてCPU時間が不足しているような状況下では、経過時間の短縮は難しくなります。これは、並列処理プログラムの実行のために、同時に複数のCPUが割り当てられる機会が少なくなるためです。

つまり、並列処理で効果を得るためには、複数のCPUをもつハードウェア上で、かつ、システム全体としてCPU処理能力に余裕のある環境下で、並列処理プログラムを実行する必要があります。

また、並列処理のためのオーバヘッドを相対的に小さくするため、ループの繰返し数またはループ中の実行文数が多いことが必要です。

9.3.2 OpenMP仕様による並列化

OpenMP仕様による並列化については、“4.3 OpenMP仕様による並列化”を参照してください。

9.4 出力情報

9.4.1 翻訳時情報

翻訳時に出力する情報について説明します。

9.4.1.1 ヘッダ

-ffj-lstオプション、-ffj-lst-out=*file*オプションおよび-ffj-srcオプションのうち少なくとも1つが指定された場合、翻訳時に出力する各情報に対して、以下の形式でヘッダを出力します。

ヘッダの出力形式

```
Fujitsu C/C++ Version バージョン 日付
```

バージョン	コンパイラのバージョンを出力します。
日付	翻訳した日時をasctime関数の形式で出力します。

9.4.1.2 ソースリスト

以下の翻訳時オプションによって、ソースリストを出力することができます。

- -ffj-lst[={p|t}]

  ソースリストをファイルに出力します。

- -ffj-lst-out=*file*

  ソースリストを出力するファイル名を指定します。-ffj-lst=pオプションも有効になります。

- -ffj-src

  ソースリストを標準出力に出力します。

  ただし、-ffj-lstオプションまたは-ffj-lst-out=*file*オプションが同時に指定された場合、ソースリストはファイルに出力されます

ソースリストには、適用された最適化を示す記号が付加されます。

9.4.1.2.1 出力形式

ソースリストの出力形式を以下に示します。

ソースリストの出力形式

```
Compilation information
  Current directory : ディレクトリ名
  Source file : ソースファイル名
(line-no.)(optimize)
```

```
nnnnnnnn i mmmmv ソース
...
          <<< Loop-information Start >>>
          <<< 詳細な最適化情報
          <<< Loop-information End >>>
nnnnnnnn i mmmmv ソース
...
```

ディレクトリ名	ソースファイルを格納しているディレクトリ名
ソースファイル名	ソースファイル名
nnnnnnnn	ソースの行番号 (可変長)
i	インライン展開の表示記号
mmmm	ループアンローリングによる展開数 (可変長)
v	SIMD化の表示記号
ソース	ソース行
<<< Loop-information Start >>> (注)	詳細な最適化情報のヘッダ
詳細な最適化情報 (注)	直後の文に適用された最適化情報の詳細
<<< Loop-information End >>> (注)	詳細な最適化情報のフッタ

注) -ffj-lst=tオプションが有効な場合だけ出力されます。

9.4.1.2.2 ソースリストに含まれる情報

ソースの行番号

ソースの行番号を出力します。

インライン展開の表示記号

関数呼び出しがある行に対して、以下の表示記号を出力します。

表示記号	説明
i	関数がインライン展開されたことを示します。
空白	関数がインライン展開されなかったことを示します。

ループアンローリングによる展開数

ループアンローリングが適用された場合、展開数を出力します。

ループがフルアンローリングされた場合は、展開数ではなく"f"を出力します。

空白の場合は、ループアンローリングが適用されなかったことを示します。

SIMD化の表示記号

ループ制御文(for文、while文、do-while文、およびif-goto文)がある行は、ループ制御文に対して、以下の表示記号を出力します。

表示記号	説明
v	SIMD化されたことを示します。
m	SIMD化された部分とされなかった部分があることを示します。
s	SIMD化されなかったことを示します。
空白	SIMD化対象ループでないことを示します。

詳細な最適化情報

翻訳時オプション-ffj-lst=tが指定された場合、適用された最適化の情報がソースリストに追加されて、ファイルへ出力されます。

以下の最適化情報が、ループ単位で出力されます。

ループに対して行った最適化情報

- FISSION(num: *N*)

  ループ分割が行われたことを意味します。

  - *N*は、ループ分割後のループ数を表します。

- SOFTWARE PIPELINING

  ループに対しソフトウェアパイプラインされたことを意味します。

- SIMD

  SIMD化されたことを意味します。以下のいずれかで表示されます。

  - SIMD(VL: *length*[,*length*]... Interleave: *num*[,*num*])

    SIMD化されたことを意味します。*length*は、1つのSIMD命令で処理する配列の要素数を表します。ループ分割された各ループで*length*が異なる場合、*length*が複数表示されます。

    *num*は、SIMD化後のメモリアクセスを効率化するために、SIMD化後のループ内の命令展開数が表示されます。

  - SIMD(VL: AGNOSTIC; VL: *length*[,*length*]... in 128-bit Interleave: *num*[,*num*])

    SVEのベクトルレジスタを特定のサイズとみなさずSIMD化されたことを意味します。*length*は、SVEのベクトルレジスタサイズを128ビットとみなして、1つのSIMD命令で処理する配列の要素数を表します。ループ分割された各ループで*length*が異なる場合、*length*が複数表示されます。

    *num*は、SIMD化後のメモリアクセスを効率化するために、SIMD化後のループ内の命令展開数が表示されます。

- PATTERN MATCHING(matmul)

  ループをライブラリ呼出し(matmul)に変換したことを意味します。

- FULL UNROLLING

  ループがフルアンローリングされたことを意味します。

- CLONE

  ループがclone最適化されたことを意味します。

プリフェッチに関する情報

prefetch命令

- PREFETCH(SOFT) : *N*

  ループ中に存在するすべてのprefetch命令の個数を示します。

  また、本表示に続けて、プリフェッチのアクセス種別ごとに、以下の形式でprefetch命令の個数と配列名を示します。

  ```
  アクセス種別 : N
  配列名 : N,...
  ```

  アクセス種別

  - SEQUENTIAL : *N*

    ループ内で使用される連続的にアクセスされる配列データに対するprefetch命令の個数を示します。

  - STRIDE : *N*

    ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対するprefetch命令の個数を示します。

  配列名

- 配列名：*N*,...

  配列ごとのprefetch命令の個数を示します。コンパイラが内部的に生成した配列は"(unknown)"と表示します。

レジスタに関する情報

ー SPILLS :

最内ループ中に存在するレジスタの退避・復元命令の個数をレジスタ種別ごとに示します。

- GENERAL : SPILL *N* FILL *N*

  汎用レジスタのメモリへの退避・復元命令の個数を示します。

- SIMD&FP : SPILL *N* FILL *N*

  SIMDと浮動小数点レジスタのメモリへの退避・復元命令の個数を示します。

- SCALABLE : SPILL *N* FILL *N*

  拡張レジスタのメモリへの退避・復元命令の個数を示します。

- PREDICATE : SPILL *N* FILL *N*

  プレディケートレジスタのメモリへの退避・復元命令の個数を示します。

9.4.1.2.3 ソースリストの出力例

以下にソースリストの出力例を示します。

例

ソースリストの出力例1

```
Compilation information
  Current directory : ディレクトリ名
  Source file : ソースファイル名
(line-no.)(optimize)
        1             #include <stdio.h>
        2
        3             float sub(int i);
        4
        5             int main() {
        6               int i;
        7               float a[1000];
        8
        9          v    for(i = 0; i < 1000; i++) {
       10                 a[i] = i;
       11               }
       12
       13               printf("%lf¥n", a[0]);
       14               i = 0;
       15              _loop:
       16               if(i < 1000) {
       17                 goto _next;
       18               }
       19               a[i] = i;
       20               i++;
       21               goto _loop;
       22              _next:
       23
       24               printf("%lf¥n", a[0]);
       25
       26          v    for(i = 0; i < 1000; i++) {
       27    i             a[i] = sub(i);
       28               }
       29         f     for(i = 0; i < 2; i++) {
```

```
       30                a[i] = i;
       31              }
       32              printf("%lf¥n", a[0]);
       33              return 0;
       34            }
       35
       36            float sub(int j) {
       37              return (float)j;
       38            }
```

この例では、以下のことがわかります。

- 9行目のfor文によるループがSIMD化されました。
- 26行目のfor文によるループがSIMD化されました。
- 27行目の関数がインライン展開されました。
- 29行目のfor文に含まれる文がフルアンローリングされました。

例

ソースリストの出力例2

```
Compilation information
  Current directory : ディレクトリ名
  Source file : ソースファイル名
(line-no.)(optimize)
       1            #include <stdio.h>
       2
       3            float sub(int i);
       4
       5            int main() {
       6              int i;
       7              float a[10000];
       8
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 4 Interleave: 1)
                      <<< Loop-information End >>>
       9          v    for(i = 0; i < 10000; i++) {
      10                a[i] = i;
      11              }
      12
      13              printf("%lf¥n", a[0]);
      14
      15              i = 0;
      16            _loop:
      17              if(i < 10000) {
      18                goto _next;
      19              }
      20              a[i] = i;
      21              i++;
      22              goto _loop;
      23            _next:
      24              printf("%lf¥n", a[0]);
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 4 Interleave: 1)
                      <<< Loop-information End >>>
      25          v    for(i = 0; i < 10000; i++) {
      26    i             a[i] = sub(i);
      27              }
```

```
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information End >>>
28         f     for(i = 0; i < 8; i++) {
29                 a[i] = i;
30               }
31               printf("%lf¥n", a[0]);
32               return 0;
33             }
34
35             float sub(int j) {
36               return(float)j;
37             }
```

この例では、以下のことがわかります。

- 9行目のfor文によるループは、SIMD化されました。
- 25行目のfor文によるループは、SIMD化されました。
- 26行目の関数呼出しがインライン展開されました。
- 28行目のfor文に含まれる文がフルアンローリングされました。

9.4.1.2.4 翻訳時情報(ソースリスト)の注意事項

-ffj-lst=p、-ffj-lst=t、または-ffj-srcオプションを指定して出力される翻訳時情報(最適化情報)が正確でない場合があります。

- インライン展開は、展開した場所によって最適化の動作が異なることがあります。この場合、以下の翻訳時情報(最適化情報)が出力されることがあります。
  - 1つのループに対して、複数の最適化メッセージが出力される。
  - 翻訳時情報(最適化情報)と反対の意味の最適化メッセージが出力される。
  - 翻訳時情報(最適化情報)が出力されない。

  なお、ループ内の関数がインライン展開された場合、その関数のprefetch数はループの翻訳時情報(最適化情報)に計上されます。
- 1つのループに対して、SIMD化、ループ融合、ループ分割などの最適化が複数適用されると、以下の翻訳時情報が出力されることがあります。
  - 行番号に対する最適化情報がずれる。
  - 翻訳時情報(最適化情報)と反対の意味の最適化メッセージが出力される。
- ループアンスイッチングは、if文の条件が成立する場合のループと成立しない場合のループを生成しますが、どちらか一方の翻訳情報と最適化メッセージしか出力されません。また、行番号に対する最適化情報が出力されないことがあります。
- 1行に複数のループを記述した場合、その中のどれか1つのループに対する最適化情報だけが出力されます。最適化情報を出力したいループは、他のループと同じ行に記述しないでください。
- 条件文とgoto文で構成されるループの詳細な最適化情報は出力されません。また、行番号に対する最適化情報がずれることがあります。
- リンク時最適化が適用された場合、以下の現象が発生することがあります。
  - 翻訳情報で出力される最適化とは異なった最適化が実行時に適用される。
  - 一部の翻訳時情報(最適化情報)が出力されない。
- インターリーブ機能はSIMD化の機能の一部を使っているため、SIMD化が行われずインターリーブのみ動作した場合でも、SIMD化の表示記号"v"が出力されます。この場合、詳細な最適化情報として"VL: *length*"が出力されないため、SIMD化されていないことが判断できます。

例

― SIMD化とインターリーブが動作した場合

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 4 Interleave: 1)
                    <<< Loop-information End >>>
 9          v     for(i = 0; i < 10000; i++) {
10                  a[i] = i;
11                }
```

― SIMD化が行われずインターリーブのみ動作した場合

```
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(Interleave: 1)
                    <<< Loop-information End >>>
 9          v     for(i = 0; i < 10000; i++) {
10                  a[i] = i;
11                }
```

9.5 言語仕様

9.5.1 言語規格のサポート範囲

言語規格のサポート範囲を説明します。

clangモードでは、下記の規格をサポートしています。

- C89規格
- C99規格
- C11規格

9.5.2 半精度(16ビット)浮動小数点型について

下記2種類の半精度(16ビット)浮動小数点型をサポートします。

__fp16データ型: ARM C言語規格の拡張で定義(IEEE 754-2008)

__fp16データ型は算術のデータ型ではありません。__fp16データ型は、値を格納するため、または型変換するために使用します。よって、__fp16データ型で宣言した値を算術演算で使用した場合、単精度浮動小数点データ型に自動的に型変換されます。また算術演算後に、単精度浮動小数点データ型は、__fp16データ型に型変換されます。

_Float16データ型: C11規格の拡張で定義(ISO/IEC TS 18661-3:2015)

_Float16データ型は算術データ型です。_Float16データ型で宣言された値を用いて演算した場合は、半精度の算術演算が使用されます。_Float16データ型を使用することで、単精度浮動小数点データ型への型変換が不要になることから、実行性能が向上する場合があります。

ただし、_Float16データ型と単精度浮動小数点データ型の間に、暗黙の型変換はありません。よって、_Float16データ型で宣言した値を単精度浮動小数点データ型の引数として渡す場合、明示的に型変換する必要があります。

例

_Float16/単精度浮動小数点データ型の明示的型変換

```
void half_function(void)
{
```

```
    _Float16 value = 1.0f16;
    printf("%f",(float)value);
}
```

9.6 言語およびtrad/clang間結合における注意事項

プログラム言語やモード(trad/clang)が異なるオブジェクトプログラムを結合する場合の注意事項を説明します。

9.6.1 結合時の翻訳コマンドと必須オプション

“7.1 結合時の翻訳コマンドと必須オプション”を参照してください。

9.6.2 C++言語との結合

以下の点に注意してください。

- C++で記述された関数からC言語で記述された関数を呼び出す場合、C++側で、呼び出される関数をC言語へのリンケージ指定(extern"C")付きで宣言する必要があります。
- C言語で記述された関数からC++で記述された関数を呼び出す場合、C++側で、呼び出される関数をC言語へのリンケージ指定(extern"C")付きで宣言する必要があります。
- C言語へのリンケージ指定(extern"C")付きで宣言された関数の引数および復帰値は、C言語で許されている型が指定できます。

以下に呼出しの例を示します。

例1: C++のプログラムに最初に制御を渡す場合

C++プログラム: cplusmain.cc

```
#include <iostream>

extern "C" {
    void cfunc(short int);
    int cpfunc(int);
}

int cpfunc(int i) {
    std::cout << "C++: cpfunc called, i=" << i << std::endl;
    return i;
}

int main() {
    std::cout << "C++ main()" << std::endl;
    cfunc(10);
    return 0;
}
```

Cプログラム: csub.c

```
#include <stdio.h>

int cpfunc(int);

void cfunc(short int si) {
    printf("C:   cfunc called, si=%d¥n", si);
    cpfunc(si);
}
```

翻訳

```
$ fccpx -Nclang -c csub.c
$ FCCpx -Nclang csub.o cplusmain.cc
```

実行

```
$ ./a.out
```

結果

```
C++ main()
C:   cfunc called, si=10
C++: cpfunc called, i=10
```

例2: C言語のプログラムに最初に制御を渡す場合

C++プログラム: cplussub.cc

```
#include <iostream>

extern "C" {
    void cfunc(short int);
    int cpfunc(int);
}

int cpfunc(int i) {
    std::cout << "C++: cpfunc called, i=" << i << std::endl;
    cfunc(i);
    return i;
}
```

Cプログラム: cmain.c

```
#include <stdio.h>

int cpfunc(int);

int main() {
    printf("C main()¥n");
    cpfunc(10);
    return 0;
}

void cfunc(short int si) {
    printf("C:   cfunc called, si=%d¥n", si);
}
```

翻訳

```
$ fccpx -Nclang -c cmain.c
$ FCCpx -Nclang cmain.o cplussub.cc
```

実行

```
$ ./a.out
```

結果

```
C main()
C++: cpfunc called, i=10
C:   cfunc called, si=10
```

9.6.3 Fortranとの結合

以下の点に注意してください。

- 最初に制御を渡すプログラムがC言語で記述される場合、その関数名は、MAIN__またはmainでなければなりません。詳細については、“Fortran使用手引書”を参照してください。

- 最初に制御を渡すプログラムがFortranで記述されている場合、実行可能プログラムの復帰値にFortranの復帰コードが設定されます。Fortranの復帰コードの詳細については、“Fortran使用手引書”を参照してください。
- 手続き呼出しおよびデータの受渡しについては、“Fortran使用手引書”を参照してください。
- Cから呼び出すFortranの関数名およびFortranから呼び出されるCの関数名の最後には、_を付けてください。

以下に呼出しの例を示します。

COARRAY仕様を利用する場合は、--linkfortranオプションの代わりに--linkcoarrayオプションを指定してください。COARRAY仕様については、“Fortran使用手引書 別冊 COARRAY”をお読みください。

例

例1: Fortranのプログラムに最初に制御を渡す場合

Fortranプログラム: fortranmain.f95

```
print *,"Fortran: main program"
call cfunc(10)
end
```

Cプログラム: csub.c

```
#include <stdio.h>
int cfunc_(int *p) {
    printf(" C: cfunc_ called, *p=%d¥n", *p);
    return *p;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -Nclang -c csub.c
$ frtpx csub.o fortranmain.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortranmain.f95
$ fccpx -Nclang --linkfortran csub.c fortranmain.o
```

実行

```
$ ./a.out
```

結果

```
Fortran: main program
C: cfunc_ called, *p=10
```

例2: C言語のプログラムに最初に制御を渡す場合(MAIN__)

Fortranプログラム: fortransub.f95

```
integer function func(x)
integer*4 x
print *,"Fortran: func() called"
call cfunc(x)
func=x
end
```

Cプログラム: cmain.c

```
#include <stdio.h>
int func_(int *);
int cfunc_(int *p) {
    printf(" C: cfunc_ called. *p=%d¥n", *p);
    return *p;
}
int MAIN__() {
    int i=10;
    printf(" C MAIN__()¥n");
    func_(&i);
    return 0;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -Nclang -c cmain.c
$ frtpx cmain.o fortransub.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortransub.f95
$ fccpx -Nclang --linkfortran cmain.c fortransub.o
```

実行

```
$ ./a.out
```

結果

```
C MAIN__()
Fortran: func() called
C: cfunc_ called. *p=10
```

例3: C言語のプログラムに最初に制御を渡す場合(main)

Fortranプログラム: fortransub.f95

```
integer function func(x)
integer*4 x
print *,"Fortran: func() called"
call cfunc(x)
func=x
end
```

Cプログラム: cmain.c

```
#include <stdio.h>
int func_(int *);
int cfunc_(int *p) {
    printf(" C: cfunc_ called. *p=%d¥n", *p);
    return *p;
}
int main() {
    int i=10;
    printf(" C main()¥n");
    func_(&i);
    return 0;
}
```

翻訳

- リンクにFortran用の翻訳コマンドを用いる場合

```
$ fccpx -Nclang -c cmain.c
$ frtpx -mlcmain=main cmain.o fortransub.f95
```

- リンクにC言語用の翻訳コマンドを用いる場合

```
$ frtpx -c fortransub.f95
$ fccpx -Nclang --linkfortran cmain.c fortransub.o
```

実行

```
$ ./a.out
```

結果

```
C main()
Fortran: func() called
C: cfunc_ called. *p=10
```

9.7 SIMD組込み関数

clangモードでは、SIMD(Single Instruction Multi Data)組込み関数を使用できます。

SIMD組込み関数の詳細については、Arm社が開発者向けに公開しているドキュメント“ARM C Language Extensions for SVE”を参照してください。

clangモードでは、ドキュメントバージョン00bet1(First public release)に記載されているSIMD組込み関数をサポートしています。ドキュメントバージョン00bet2以降で追加されたSIMD組込み関数はサポートしていません。ご注意ください。詳細は、“ARM C Language Extensions for SVE - 1.1.2. Change history”を参照してください。

9.8 GNU C互換機能

clangモードがサポートするGNU Cコンパイラの言語仕様(GNU C拡張仕様)および翻訳時オプション(GNU C互換オプション)について説明します。

9.8.1 GNU C拡張仕様

clangモードがサポートするGNU C拡張仕様については、clangおよびGCCのウェブサイトをお読みください。

9.8.2 GNU C互換オプション

clangモードでは、以下のGNU C互換オプションをサポートします。GNU C互換オプションの詳細については、GCCのウェブサイトを参照してください。

{--print-file-name|-print-file-name}=include

clangモードが提供するヘッダを格納するディレクトリを表示することを指示します。

{--print-prog-name|-print-prog-name}={as|ld|objdump|ranlib|ar}

翻訳コマンドが呼び出すプログラムを表示することを指示します。

--shared

リンカに対して、共有オブジェクトを作成するよう指示します。

本オプションは、リンカに渡されます。

--version

コンパイラのバージョンと著作権の情報を標準出力に出力します。

-Wp,-MD,*filename*

本オプションは、-MD -MF *filename*を指定した場合と等価です。

-Xlinker *option*

*option*をリンカへのオプションとして渡すことを指示します。

-dM

-Eオプションが有効な場合、すべてのマクロ定義を出力します。

-f{exceptions|no-exceptions}

既定義マクロ__EXCEPTIONSを定義するか否かを指示します。省略時は、-fno-exceptionsオプションが適用されます。

-fno-common

初期値指定のない大域変数を、オブジェクトファイルのデータセクションに割り当てることを指示します。

複数のソースファイルで、同一の名前の変数が`extern`指定子なしに宣言されている場合に、リンク時にエラーとして検出できるようになります。

-f{optimize-sibling-calls|no-optimize-sibling-calls}

末尾呼出しの最適化を行うか否かを指示します。翻訳時オプション-O1以上が有効な場合、省略時は、翻訳時オプション-foptimize-sibling-callsが適用されます。

本オプションは、翻訳時オプション-O1以上が有効な場合に意味があります。

-f{pie|PIE}

位置独立実行可能プログラム(PIE)を生成することを指示します。

-fpieオプションを指定した場合、-fPIEオプションに比べて命令シーケンスが短くなるため高速になりますが、リンク時に参照できるユニークな外部シンボル数は少なくなります。-fPIEオプションを指定した場合、-fpieオプションに比べて命令シーケンスが長くなるため低速になりますが、リンク時に参照できるユニークな外部シンボル数は多くなります。これらの場合の参照できるユニークな外部シンボル数は、同時に結合されるすべてのライブラリの合計です。

-f{pic|PIC}オプションとの違いは、リンク時に-pieオプションを指定することで、実行可能プログラムを生成できることです。

-fvisibility={default|internal|hidden|protected}

実行可能プログラムまたは共有オブジェクト(以下コンポーネント)内での大域シンボルの可視属性を指定します。省略時は、-fvisibility=defaultオプションが適用されます。

-fvisibility=default

シンボルは、別のコンポーネントから参照することができます。

-fvisibility=hidden

シンボルは、別のコンポーネントから参照できません。ただし、シンボルのアドレスを渡された別のコンポーネントからは参照できます。

-fvisibility=internal

本オプションは、-fvisibility=hiddenオプションと等価です。

-fvisibility=protected

シンボルは、別のコンポーネントから参照できます。ただし、別のコンポーネントにある同じ名前の定義で上書きすることはできません。

-g{dwarf|dwarf-4}

DWARF Version 4形式のデバッグ情報を出力します。

本オプションは、-gオプションと等価です。

-idirafter *dir*

ヘッダの検索時に、*dir*を標準のディレクトリのあとに追加します。

複数の-idirafterオプションでディレクトリが複数指定された場合、指定された順に検索します。

ヘッダの検索順の詳細については、“9.1.2.2.1 コンパイラ全般に関連するオプション”の-Iオプションの説明を参照してください。

ヘッダが絶対パス名で指定された場合、指定された絶対パス名だけ検索します。ディレクトリが存在しない場合、本オプションは無効になります。

-include *file*

ソースファイルの先頭で*file*をインクルードするよう指示します。

複数の-includeオプションが指定された場合、指定された順にインクルードします。

-isystem *dir*

ヘッダの検索時に、*dir*を標準のディレクトリとして追加します。

複数の-isystemオプションでディレクトリが複数指定された場合、指定された順に検索します。

ヘッダの検索順の詳細については、“9.1.2.2.1 コンパイラ全般に関連するオプション”の-Iオプションの説明を参照してください。

ヘッダが絶対パス名で指定された場合、指定された絶対パス名だけ検索します。ディレクトリが存在しない場合、本オプションは無効になります。

-nostartfiles

リンク時に、標準のシステムスタートアップファイルを使用しないことを指示します。

-nostdinc

ヘッダの検索時に、標準のディレクトリを検索しないことを指示します。

-nostdlib

リンク時に、標準のシステムスタートアップファイルおよび標準ライブラリを使用しないことを指示します。

-pthread

POSIXスレッドライブラリを利用するマルチスレッドオブジェクトを生成することを指示します。

-rdynamic

リンカに対して、すべてのシンボルを動的シンボルテーブルに追加するよう指示します。本オプションは、リンカに-export-dynamicオプションを渡します。

-undef

システム固有およびGNU C固有の既定義マクロを抑止することを指示します。

-w

すべての警告を抑止することを指示します。

-x *language*

本オプションよりあとに指定された入力ファイルの種別を指示します。*language*には、cまたはassembler-with-cppを指定します。

9.9 コードカバレッジ機能

ここでは、コードカバレッジ機能について説明します。

コードカバレッジ機能は、実行時に実行文の実行回数を計測し、プログラムのコード網羅率を調べる機能です。

コードカバレッジ機能では、オープンソースソフトウェアであるコンパイラ基盤LLVMに含まれるコードカバレッジツール(以降、llvm-covコマンドと呼びます)を使用します。llvm-covコマンドの使用方法については、manコマンドを使用して、llvm-covコマンドのオンラインマニュアルを参照してください。

9.9.1 コードカバレッジ機能の使用方法

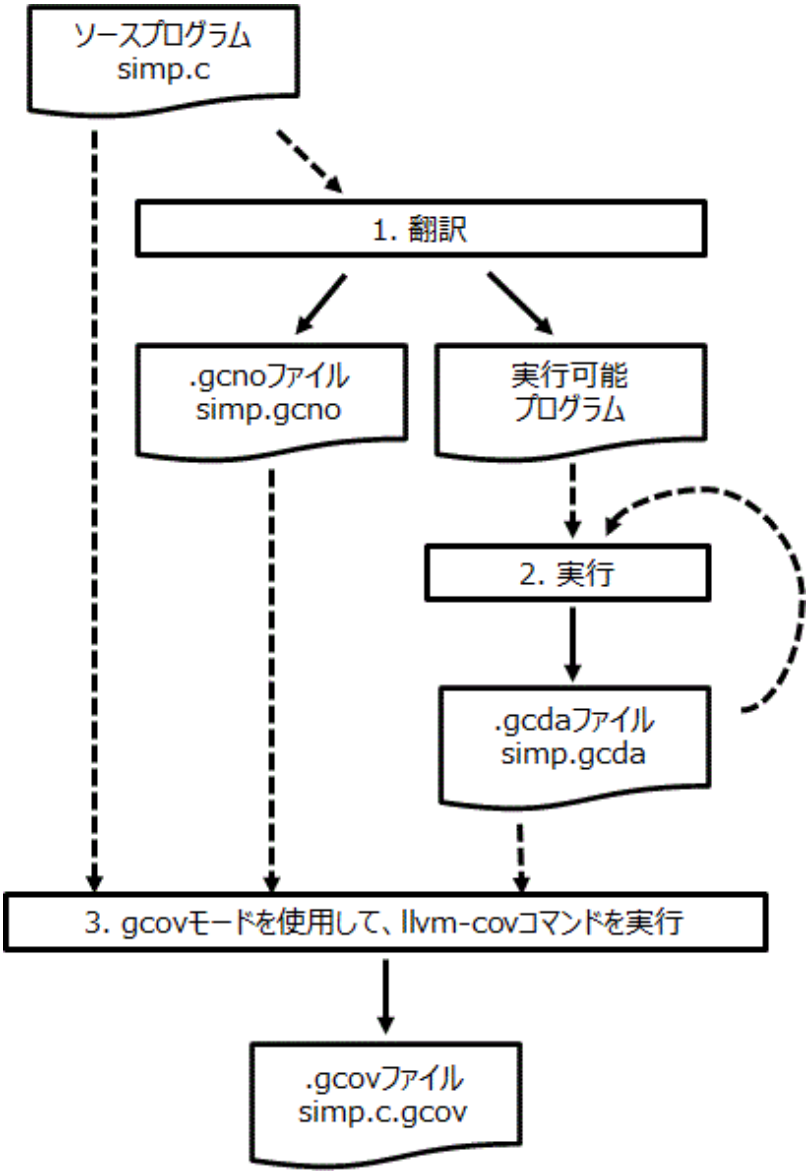
コードカバレッジ機能は、以下の手順で使用します。

1. 翻訳
2. 実行

3. gcovモードを使用して、llvm-covコマンドを実行

以下の図に、コードカバレッジ機能の手続きを示します。

図9.1 コードカバレッジ機能の手続き



9.9.2 コードカバレッジ機能使用時に必要なファイル

コードカバレッジ機能を使用する時に必要なファイルについて説明します。

9.9.2.1 .gcnoファイル

.gcnoファイルは、翻訳時に得られる情報を含んだバイナリファイルです。

ファイル生成タイミング	翻訳時に生成されます。
ファイル名	ソースプログラムの拡張子をgcnoに変更したファイル名になります。
ファイル生成場所	翻訳を行ったディレクトリに生成されます。

以下に、.gcnoファイルの生成例を示します。

例

```
$ fcc -Nclang --coverage ../simp.c
$ ls
a.out simp.gcno
```

-oオプションを指定してアセンブラソースファイルおよびオブジェクトファイルを生成した場合は、-oオプションが、.gcnoファイルにも適用されます。

以下に、-oオプションを指定した場合の生成例を示します。

例

```
$ fcc -Nclang --coverage ../simp.c -c -o www/yyy.o
$ ls www/
yyy.gcno yyy.o
```

9.9.2.2 .gcdaファイル

.gcdaファイルは、実行時に得られる情報を含んだバイナリファイルです。

ファイル生成タイミング	実行時に生成されます。 同一名のファイルがすでに存在する場合は、そのファイルに情報が累積されます。
ファイル名	ソースプログラムの拡張子をgcdaに変更したファイル名になります。
ファイル生成場所	翻訳を行ったディレクトリに生成されます。

以下に、.gcdaファイルの生成例を示します。

例

```
$ fcc -Nclang --coverage ../simp.c
$ ls
a.out simp.gcno
$ ./a.out
$ ls
a.out simp.gcda simp.gcno
```

-oオプションを指定してアセンブラソースファイルおよびオブジェクトファイルを生成した場合は、-oオプションが.gcdaファイルにも適用されます。

以下に、-oオプションを指定した場合の生成例を示します。

例

```
$ fcc -Nclang --coverage ../simp.c -c -o www/yyy.o
$ ls www/
yyy.gcno yyy.o
$ fcc -Nclang --coverage www/yyy.o
$ ./a.out
$ ls www/
yyy.gcda yyy.gcno yyy.o
```

.gcdaファイルの格納先ディレクトリは-fprofile-dir=*dir_name*オプションで変更できます。

以下に、-fprofile-dir=*dir_name*オプションを指定した場合の生成例を示します。

 例

```
$ fcc -Nclang --coverage ../simp.c -c -o www/yyy.o -fprofile-dir=/tmp
$ ls www/
yyy.gcno yyy.o
$ fcc -Nclang --coverage www/yyy.o
$ ./a.out
$ ls www/
yyy.gcno yyy.o
$ ls /tmp/www/
yyy.gcda
```

9.9.3 コードカバレッジ機能の注意事項

コードカバレッジ機能使用時の注意事項について説明します。

- 実行回数を計測する命令がオブジェクトファイル内に追加されるため、実行性能が低下する場合があります。
- 実行時に生成される.gcdaファイルの格納先ディレクトリは、翻訳時に決定されます。
  - 翻訳後に実行可能プログラムを移動しても、.gcdaファイルの格納先ディレクトリは変わりません。
  - .gcdaファイルの格納先ディレクトリを変更したい場合は、翻訳時に-fprofile-dir=*dir_name*オプションを使用して変更してください。
- -mcmodel=largeオプションを必要とするプログラムでは、使用できません。
- 以下の場合、実行回数を正しく計測できないことがあります。
  - 1行に複数の実行文が含まれる場合
  - exit(3)関数が呼ばれた場合
  - setjmp関数またはlongjmp関数を呼び出す場合
  - 例外が捕捉された場合
  - ソースプログラムに#line指令が含まれる場合
  - コンパイラの最適化が適用された場合

9.10 制限事項および注意事項

ここでは、clangモードにおける制限事項および注意事項について説明します。

9.10.1 complex.hのマクロCMPLX、CMPLXF、およびCMPLXLにおける制限

complex.hのマクロCMPLX、CMPLXF、およびCMPLXLは定義されません。

complex.hのマクロCMPLX、CMPLXF、またはCMPLXLを用いて複素数型の変数を初期化した場合、以下が発生します。

- Cプログラム:翻訳時エラーまたはリンクエラー
- C++プログラム:翻訳時エラー

複素数型の変数は、complex.hのマクロCMPLX、CMPLXF、およびCMPLXLを使用せずに初期化してください。

 例

マクロCMPLXを使用しない初期化の例

Cプログラム

```
double complex a = 1.0 + 2.0 * I;
```

C++プログラム

```
std::complex<double> a(1.0, 2.0);
```

付録A　処理系依存の仕様

ここでは、正しいプログラム構成要素および正しいデータに関する動作のうち、本処理系の特徴に依存した項目について説明します。

以下に、処理系依存の項目と、その項目に引き続いて本処理系の動作を説明しています。

A.1　翻訳

■どのような方法で診断メッセージを識別するか。

診断メッセージは、標準エラーに出力されます。

診断メッセージの形式は、以下のとおりです。

```
"filename", line nn: メッセージ本体
```

filename:　　エラーが起こったファイル名

nn:　　エラーが起こった行番号

A.2　環境

■main関数への実引数の意味。

以下に、ホスト環境が与える値を示します。

```
int main(int argc, char *argv[]) {/*...*/}
```

argc:　　実引数の個数

argv:　　実引数となる文字列へのポインタ配列
(argv[0]はプログラム名または空文字列)

■対話型装置がどのようなもので構成されるか。

コンソールおよび端末などの表示装置で構成されます。

A.3　識別子

■外部結合でない識別子において(31以上の)意味がある先頭の文字数。

4096文字までです。

■外部結合である識別子において(6以上の)意味がある先頭の文字数。

4096文字までです。

■外部結合である識別子において英小文字と英大文字の区別に意味があるか否か。

英小文字と英大文字を区別します。

■識別子にマルチバイトキャラクタが記述されたとき、それが国際文字名にどう対応するか。

マルチバイトキャラクタを識別子に記述できません。

A.4　文字

■ソースおよび実行文字集合の要素で、この規格で明示的に規定しているもの以外の要素。

ソース文字集合および実行文字集合の要素の値は、環境変数LANGに設定された、日本語UTF-8、日本語EUCのいずれかの値となります。

■多バイト文字のコード化のために使用されるシフト状態。

シフトコードに依存した表現形式ではありません。

■実行文字集合の文字におけるビット数。

1バイト中のビット数は、8です。

■(文字定数内および文字列リテラル内の)ソース文字集合の要素と実行文字集合の要素との対応付け。

ソース文字集合の要素と実行文字集合の要素との対応付けは同じです。

■基本実行文字集合で表現できない文字もしくは拡張表記を含む単純文字定数の値、またはワイド文字定数に対しては拡張文字集合で表現できない文字もしくは拡張表記を含むワイド文字定数の値。

規定されていない拡張表記が文字定数または文字列リテラルの中に現れている場合、¥だけが無視されて単なる文字として扱われます。

■2文字以上の文字を含む単純文字定数または2文字以上の多バイト文字を含むワイド文字定数の値。

単純文字定数として、2つ以上の文字を記述した場合、その値は、左から4文字取り出されて、その文字が'C1C2...C*n*'(2≦*n*≦5)の場合、以下の値を取ります。

$$\sum_{k=1}^{n-1} 256^{k-1} \times (\text{'}Ck\text{'}\&255) + 256^{n-1} \times \text{'}Cn\text{'},(2 \leqq n \leqq 4)$$

5文字以上の場合は、5文字以降が無視されます。

ワイド文字定数として2つ以上の多バイト文字を記述した場合、左から1文字取り出されて、その値は対応するコードの値になります。2文字以上の場合には、2文字以降が無視されます。

記述例

表現'¥x123'の値: '¥x23'

表現'¥0223'の値: '¥022'＋'3'×256

表現L'¥1234'の値: L'¥123'

■ワイド文字定数に対して、多バイト文字を対応するワイド文字(コード)に変換するために使用されるロケール。

環境変数LANGで定義されます。

■"単なる" charがsigned charと同じ値の範囲を持つか、unsigned charと同じ値の範囲を持つか。

unsigned charと同じ値の範囲を持ちます。

■実行時キャラクタセットのメンバの値

実行時キャラクタセットは、翻訳時キャラクタセットと一致しなければなりません。

■実行時キャラクタセットでは表現できないマルチバイトキャラクタあるいは拡張表記を含む文字列の値

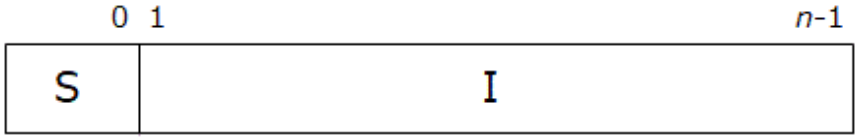
翻訳時に解釈された値がそのまま文字列内に含まれます。

A.5 整数

■整数のさまざまな型の表現方法および値の集合。

“図A.1 汎整数型の表現および値”に、汎整数型の表現および値を示します。

図A.1 汎整数型の表現および値



S：符号(0のとき正、1のとき負)
I：整数部

$$x = -S \times 2^{n-1} + \sum_{k=1}^{n-1} ik \times 2^{n-k-1}$$

文字集合で列挙した要求されるソース文字集合の任意の要素以外の量を、char型オブジェクトに格納した場合、その動作は、量数の下位バイトが格納され、その値は符号付き整数として扱われます。

■整数をより短い符号付き整数に変換した結果、または符号なし整数を長さの等しい符号付き整数に変換した結果で、値が表現できない場合の変換結果。

その値は、変換される型のビット数のビットパターンを 2の補数表現として解釈した値となります。

■符号付き整数に対してビット単位の演算を行った結果。

詰め物ビット(padding bits)は存在しないため、符号を考慮した結果が演算の結果になります。

■整数除算における剰余の符号。

整数同士の演算で結果が割り切れない場合、両オペランドが正の値であれば、/ 演算子の結果は、商の真の値より小さい最大の整数となり、% 演算子の結果は正となります。

いずれか一方のオペランドが負の値の場合、/ 演算子の結果は、商の真の値より大きい最小の整数となります。また、% 演算子の結果の符号は、被除数の符号と同じになります。

両オペランドが負の値の場合、/ 演算子の結果は、商の真の値より小さい最大の整数となり、% 演算子の結果の符号は、被除数の符号と同じになります。

■負の値をもつ符号付き汎整数型の右シフトの結果。

E1>>E2の結果は、E1をE2ビット右にシフトしたものです。

E1が符号付きで負の値を持つ場合、結果は以下に示す式を計算した値となります。

$$E1/2^{E2} - 1 + (E1\%2^{E2} == 0)$$

■処理系によって拡張された整数型

拡張された整数型はありません。

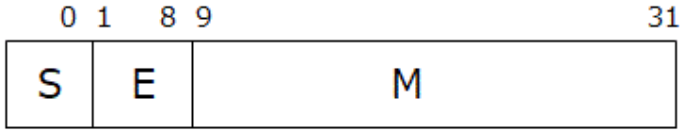
A.6 浮動小数点数

■浮動小数点数のさまざまな型の表現方法および値の集合。

“図A.2 IEEE754準拠形式 (指数部表現の基底が 2の場合)浮動小数点型の表現および値”に、浮動小数点型の表現および値を示します。

図A.2 IEEE754準拠形式 (指数部表現の基底が 2の場合)浮動小数点型の表現および値

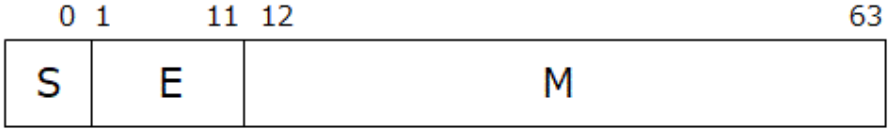
float型



S ：符号(0のとき正、1のとき負)
E ：指数部
M ：仮数部

$$x = (-1)^{s} \times (1 + m/2^{23}) \times 2^{e-127}$$

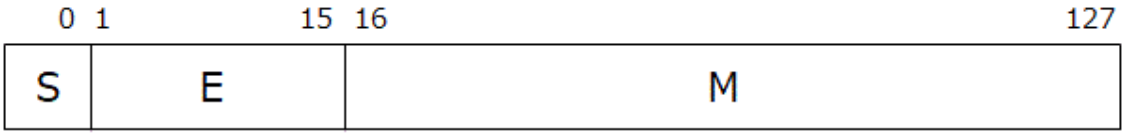
double型



S ：符号(0のとき正、1のとき負)
E ：指数部
M ：仮数部

$$x = (-1)^{s} \times (1 + m/2^{52}) \times 2^{e-1023}$$

long double型



S ：符号(0のとき正、1のとき負)
E ：指数部
M ：仮数部

$$x = (-1)^{s} \times (1 + m/2^{112}) \times 2^{e-16383}$$

■汎整数の値を元の値に正確に表現することができない浮動小数点数に変換する場合の切り捨ての方向。

そのときのFLT_ROUNDSに従って、丸めが行われます。FLT_ROUNDSの初期値は1(最も近傍へ)です。

■浮動小数点数をより狭い浮動小数点数に変換する場合の切り捨てまたは丸めの方向。

そのときのFLT_ROUNDSに従って、丸めが行われます。FLT_ROUNDSの初期値は1(最も近傍へ)です。

A.7 配列とポインタ

■配列の大きさの最大値を保持するために必要な整数の型。すなわち、sizeof演算子の型size_t。

size_tの型は、unsigned long int型として扱います。

■ポインタを整数へキャストした結果およびその逆の場合の結果。

ポインタは、整数型に変換できます。要求されるサイズは、unsigned long int型です。結果の値は、unsigned long long型の内部表現がポインタの内部表現と一致する場合の値です(変換において、ビットパターンは変換しません)。

任意の整数は、ポインタに変換できます。結果の値は、unsigned long int型に変換したあとの内部表現と一致する内部表現を持つ変換後の型の値となります(変換において、ビットパターンは変換しません)。

■同じ配列内の、2つの要素へのポインタ間の差を保持するために必要な整数の型、すなわちptrdiff_tの型。

ptrdiff_tの型は、long型です。

A.8 レジスタ

■register記憶域クラス指定子を使用することによって、実際にオブジェクトをレジスタに置くことができる範囲。

この指定が効果的である範囲は、レジスタに格納されるオブジェクトの型が算術型(long double型は含まれない)とポインタで、レジスタには、registerの指定に関係なく使用状況に応じて、効果の高いオブジェクトから順に割り当てられる場合です。

A.9 構造体、共用体、列挙型およびビットフィールド

■共用体オブジェクトのメンバを異なる型のメンバを用いてアクセスする場合。

その値は、参照したメンバの型で評価された値となります。

■構造体のメンバの詰め物および境界調整。

ある処理系によって書かれたバイナリデータを他の処理系で読まない限り、問題とはなりません。

“表A.1 算術型の大きさと境界整列値”で示すように、型に適した境界に整列している最も小さなオフセットに割り付けられます。単位はバイトです。

表A.1 算術型の大きさと境界整列値

型	大きさ	境界整列値
char	1	1
signed char	1	1
unsigned char	1	1
signed short int	2	2
unsigned short int	2	2
signed int	4	4
unsigned int	4	4
signed long int	8	8
unsigned long int	8	8
signed long long int	8	8
unsigned long long int	8	8
float	4	4
double	8	8
long double	16	16

■"単なる" int型のビットフィールドが、signed intのビットフィールドとして扱われるか、unsigned intのビットフィールドとして扱われるか。

unsigned intとして扱われます。

■単位内のビットフィールドの割付け順序。

単位内のビットは、指定した型の下位ビットから順に割り当てられます。

■ビットフィールドを記憶域単位の境界にまたがって割り付けるか否か。

ビットフィールドは、それを保持するのに十分な大きさの最も小さいアドレスの記憶単位に割り当てられます。十分な領域が残っている場合、構造体内のビットフィールドの直後にある別のビットフィールドは、同じ単位の隣接したビットに詰め込まれます。十分な領域がない場合、合わないビットフィールドは、隣接した単位の境界をまたがって割り当てられないで、次の単位に詰め込まれます。

■列挙型の値を表現するために選択される整数型。

列挙型の値を表現する型は、int型です。

■_Bool, signed int, unsigned int以外にビットフィールドに使用できる整数型

すべての整数型が使用できます。

A.10 修飾子

■volatile修飾型のオブジェクトにアクセスする構成要素。

volatile宣言は、メモリに割り付けられた入出力ポートに対応するオブジェクト、または非同期割込み機能により、アクセスされるオブジェクトを記述するために使用できます。このように宣言したオブジェクトの動作は、処理系による"最適化"、または式の評価規則により許されている範囲外の評価順序の変更を受けることはありません。

A.11 宣言子

■算術型、構造体型または共用体型を修飾する宣言子の最大数。

特に制限ありません(記憶域の大きさによる)。

A.12 文

■switch文におけるcase値の最大数。

特に制限ありません(記憶域の大きさによる)。

A.13 前処理指令

■条件付き取り込みを制御する定数式中の単一文字からなる文字定数の値が実行文字集合中の同じ文字定数の値に一致するか否か。このような文字定数が負の値をもつことがあるか否か。

一致します。また、単一の文字からなる文字定数は、基本文字集合は負の値を持ちません。その他の値は負の値となることがあります。

■取込み可能なソースファイルを探すための方法。

“2.2.2 翻訳時オプションの意味”の-Iオプションの説明をお読みください。

■取込み可能なソースファイルに対する"で囲まれた名前の探索。

“2.2.2 翻訳時オプションの意味”の-Iオプションの説明をお読みください。

■ソースファイル名と文字列との対応付け。

アルファベットの大文字と小文字は区別されます。

■認識される#pragma指令の動作。

“6.1.2 pragma指令”の説明をお読みください。

■翻訳日付および翻訳時刻がそれぞれ有効でない場合における__DATE__および__TIME__の定義。

asctime関数で生成されるものと同じです。

■定数式の単純文字の文字定数の値が負の値を持つことができるか。

符号付きと解釈されます。

■#演算子が文字列内の国際文字名の最初の¥に対して¥を付加するか。

付加します。

A.14 標準ライブラリ関数

標準ライブラリ関数の処理系依存の仕様については、システム付属のリファレンスマニュアル、またはmanコマンドの説明をお読みください。

A.15 OpenMP仕様

OpenMP仕様の処理系依存の仕様については、“4.3.3 処理系依存の仕様”をお読みください。

付録B　翻訳限界

“表B.1 翻訳限界”に、本処理系における翻訳限界を示します。

表B.1 翻訳限界

項目	C言語規格 (各値以上)	本処理系の値
複合文、繰返し文および選択文の入れ子のレベル数	127	制限なし(記憶域の大きさによる)
条件組込みにおける入れ子のレベル数	63	制限なし(記憶域の大きさによる)
宣言内の1つの算術型、構造体型、共用体型または不完全型を修飾する(任意の組合せの)ポインタ、配列および関数宣言子の数	12	制限なし(記憶域の大きさによる)
1つの完全宣言子の中の括弧で囲んだ入れ子の宣言子数	63	制限なし(記憶域の大きさによる)
1つの完全式の中の括弧で囲んだ入れ子の式数	63	制限なし(記憶域の大きさによる)
内部結合として宣言した識別子、またはマクロ名の有効先頭文字数	63	4096
外部結合として宣言した識別子の有効先頭文字数	31	4096
1つの翻訳単位の中の外部結合として宣言した識別子の数	4095	制限なし(記憶域の大きさによる)
1つのブロックの中で宣言されるブロック範囲を持つ識別子の数	511	制限なし(記憶域の大きさによる)
1つの翻訳単位の中で同時に定義されるマクロ識別子の数	4095	制限なし(記憶域の大きさによる)
1つの関数定義の中の仮引数の数	127	制限なし(記憶域の大きさによる)
1つの関数呼出しの中の実引数の数	127	制限なし(記憶域の大きさによる)
1つのマクロ定義の中の仮引数の数	127	制限なし(記憶域の大きさによる)
1つのマクロ呼出しの中の仮引数の数	127	制限なし(記憶域の大きさによる)
1つの論理ソース行の中の文字数	4095	制限なし(記憶域の大きさによる)
(連結後の)1バイトの文字列リテラルまたはワイド文字列リテラルの中の文字数	4095	制限なし(記憶域の大きさによる)
ホスト環境における1つのオブジェクトのバイト数	65535	制限なし(記憶域の大きさによる)
#includeファイルに対する入れ子のレベル数	15	511
(任意の入れ子になったswitch文を除いた)1つのswitch文中でのcase名札の数	1023	制限なし(記憶域の大きさによる)
1つの構造体または共用体のメンバ数	1023	制限なし(記憶域の大きさによる)
1つの列挙の中の列挙定数の数	1023	制限なし(記憶域の大きさによる)
1つの構造体宣言の並びの中の入れ子の構造体または共用体定義のレベル数	63	制限なし(記憶域の大きさによる)

付録C　制限事項および注意事項

C.1　制限事項

ありません。

C.2　注意事項

C.2.1　可変個の引数を持つ関数の呼出し

本処理系は、可変個の引数を持つ関数を呼び出す場合、引数として複素数(complex)、構造体(struct)、および共用体(union)を直接受け渡すことはできません。

引数として受け渡したい場合、直接ではなくポインタで受け渡すようにしてください。

C.2.2　符号付き整数型の式評価結果について

符号付き整数型の演算において、式の評価結果が元の符号付き整数型の範囲を超える場合、演算結果は保証されません。

符号付き整数型の演算では、オーバフローが発生しないようにしてください。

C.2.3　未定義の動作について

文法が規定した未定義の動作は保証されません。

C.2.4　可変長配列について

本処理系は、可変長配列をスタック領域に割り付けます。

このため、可変長配列に必要な領域が大きい場合や、可変長配列を持つ関数がインライン展開される場合には、スタック領域を十分な大きさに拡張する必要があります。

プロセスのスタック領域は、ulimitコマンド(bash組み込みコマンド)などで設定できます。

C.2.5　浮動小数点型のasmレジスタ宣言について

本処理系では、long double型および複素数型を持つ変数をasmレジスタ宣言することはできません。

C.2.6　asm文レジスタ制約への浮動小数点型の式の記述について

本処理系では、long double型および複素数型を持つ式を、asm文レジスタ制約のオペランドに記述することはできません。

C.2.7　デバッガを用いたデバッグについて

ここでは、tradモードを使用して生成した実行可能ファイルを、デバッガを利用してデバッグする際の注意事項について説明します。

デバッガ利用時の注意事項

以下は、デバッグできません。

- C11仕様の変数、関数、および型
- GNU C拡張仕様の変数、関数、および型

最適化されたプログラムのデバッグ

- SIMD型の変数はデバッグできません。
- 仮引数と大域変数(グローバル変数)は、関数の入口でデバッグできます。

- ローカル変数のデバッグは、保証されません。
- 以下の型を持つ仮引数はデバッグできません。
  - 複素数型
  - long double型
  - 配列型
  - 構造体型
  - 共用体型
- 関数内で参照されない仮引数はデバッグできません。
- インライン展開された関数はデバッグの対象になりません。
- 行番号の表示は保証されません。

C.2.8 SVEのベクトルレジスタのサイズの変更について

本処理系では、プログラムの実行中にSVEのベクトルレジスタのサイズが変更されないことを仮定して、オブジェクトプログラムを生成します。

SVEを利用したプログラムの実行中に、prctl(2)システムコールなどを利用してSVEの有効なベクトルレジスタのサイズを変更した場合、その動作は保証されません。

SVEのベクトルレジスタのサイズを変更したい場合は、SVEを利用したプログラムの実行前に設定してください。

C.2.9 除数が0の場合の整数除算例外について

Armアーキテクチャである富士通製CPU A64FXでは、除数が0の場合の整数除算例外は検出されません。

翻訳時オプション-NRtrapを指定し、除数が0である場合に、整数除算例外が検出されることを前提としたプログラムでは、演算結果が異なる場合があります。

「整数除算の直前に除数の値を確認する」ことをお勧めします。

例

整数除算の直前に除数の値を確認する例

```
#include <stdio.h>
int main()
{
  int a,b,c;
  a = 1;
  b = 0;
  if (b == 0) {
    fprintf(stderr,"*** Integer divide by zero ***¥n");
    exit(1);
  }
  return 0;
}
```

C.2.10 リンクエラー(undefined reference to)について

プログラムの翻訳時において、configureなどにより以下のような操作をしていた場合、通常行われる本処理系独自オブジェクトのリンクを抑止するため、リンクエラー(undefined reference to)となる場合があります。

- 本処理系がリンカに渡している-Lオプションを、本処理系の翻訳コマンドに直接指定してリンクを行う。

例

リンクエラーの例

```
test.o: In function `__fjc_check_hpctag':
(.text+0x1620): undefined reference to `__jwe_check_hpctag'
```

環境変数FCOMP_LINK_FJOBJを設定することで回避できる場合があります。環境変数FCOMP_LINK_FJOBJについては、“2.3 翻訳コマンドの環境変数”を参照してください。

C.2.11 asmレジスタ宣言のレジスタ名およびインラインアセンブラの破壊(clobber)リストに対するレジスタの指定について

本処理系では、コンパイラで予約されている下記のレジスタを、asmレジスタ宣言のレジスタ名およびインラインアセンブラの破壊(clobber)リストに指定することはできません。

コンパイラで予約されたレジスタ:x19, w19, r19, x29, w29, r29, sp, xzr, xsp, wzr, wsp

付録D　GNU C互換機能

本処理系は、GNU Cコンパイラ仕様(version 6.1)の一部をサポートします。

本処理系がサポートするGNU Cコンパイラの言語仕様(GNU C拡張仕様)および翻訳時オプション(GNU C互換オプション)について説明します。

D.1　GNU C拡張仕様

本処理系がサポートするGNU C拡張仕様は以下のとおりです。

これらのGNU C拡張仕様の詳細については、GCCのウェブサイトをお読みください。

- 式中の複合文(Statements and Declarations in Expressions)
- ローカルスコープのラベル(Locally Declared Labels)
- ラベルのアドレス(Labels as Values)
- typeofによる型の参照(Referring to a Type with typeof)
- 条件演算子の第2オペランドの省略(Conditionals with Omitted Operands)
- 倍精度整数(Double-Word Integers)
- 複素数(Complex Numbers)
  整数複素数型は未サポートです。
- 浮動小数点数の16進表記(Hex Floats)
- 要素数0の配列要素(Arrays of Length Zero)
- 可変長配列(Arrays of Variable Length)
- 可変引数を持つ関数マクロ(Macros with a Variable Number of Arguments)
- 行継続の表記方法規則の緩和(Slightly Looser Rules for Escaped Newlines)
- 左辺値でない配列の添字を取ることが可能(Non-Lvalue Arrays May Have Subscripts)
- void型のポインタと関数型のポインタ間の演算(Arithmetic on void- and Function-Pointers)
- 定数値以外での初期化(Non-Constant Initializers)
- 複合リテラル(Compound Literals)
- 初期化位置指定(Designated Initializers)
- caseラベルでの範囲指定(Case Ranges)
- 共用体型へのキャスト(Cast to a Union Type)
- 属性の指定(Declaring Attributes of Functions/Specifying Attributes of Variables/Specifying Attributes of Types)
  指定できる属性の一覧については、“D.1.1 属性”をお読みください。
- 文字列や文字定数中でのエスケープ文字の使用(The Character ESC in Constants)
- 型や変数の境界値(Inquiring on Alignment of Types or Variables)
- 代替キーワード(Alternate Keywords)
- 不完全なenum型(Incomplete enum Types)
- 関数名文字列(Function Names as Strings)
- ビルトイン関数(Builtins)
- 無名構造体/共用体(Unnamed struct/union fields within structs/unions)
- インラインアセンブラ(Assembler Instructions with C Expression Operands)

- メンバを持たない構造体(Structures With No Members)
- 宣言文と実行文の位置(Mixed Declarations and Code)
- C++ 形式の行コメント(C++ Style Comments)
- 識別子名中のドル記号(Dollar Signs in Identifier Names)
- インライン関数(An Inline Function is As Fast As a Macro)
- GCCが受け付けるプラグマ(Pragmas Accepted by GCC)
  - #pragma redefine_extname
  - #pragma pack
  - #pragma weak
- スレッド・ローカル・ストレージ(Thread-Local Storage)

D.1.1 属性

本処理系がサポートする属性は、以下のとおりです。

これらの属性の詳細については、GCCのウェブサイトをお読みください。

clangモードがサポートする属性は https://releases.llvm.org/7.0.0/tools/clang/docs/AttributeReference.html を参照してください。

- aarch64_vector_pcs (対象は、関数)
- alias
- aligned (対象は、変数、型)
- always_inline
- const
- constructor (対象は、関数)
- destructor (対象は、関数)
- deprecated (対象は、関数、型)
- format
- format_arg (対象は、関数)
- malloc
- may_alias (対象は、型)
- mode
- no_instrument_function (対象は、関数)
- noinline
- nonnull
- noplt (対象は、関数)
- noreturn
- nothrow
- packed (対象は、変数、型)
- pure
- section (対象は、関数、変数)
- sentinel (対象は、関数)

- unused (対象は、関数、変数、型)
- used (対象は、関数、変数)
- transparent_union
- visibility (対象は、関数、型)
- warn_unused_result
- weak (対象は、関数、変数)
- weakref (対象は、関数)

D.1.2 ビルトイン関数

本処理系がサポートするビルトイン関数は、以下のとおりです。

これらのビルトイン関数の詳細については、GCCのウェブサイトをお読みください。

- __atomic_load_n
- __atomic_load
- __atomic_store_n
- __atomic_store
- __atomic_exchange_n
- __atomic_exchange
- __atomic_compare_exchange_n
- __atomic_compare_exchange
- __atomic_add_fetch
- __atomic_sub_fetch
- __atomic_and_fetch
- __atomic_xor_fetch
- __atomic_or_fetch
- __atomic_nand_fetch
- __atomic_fetch_add
- __atomic_fetch_sub
- __atomic_fetch_and
- __atomic_fetch_xor
- __atomic_fetch_or
- __atomic_fetch_nand
- __atomic_test_and_set
- __atomic_clear
- __atomic_thread_fence
- __atomic_signal_fence
- __atomic_always_lock_free
- __atomic_is_lock_free
- __builtin_abort

- __builtin_alloca
- __builtin_bswap16
- __builtin_bswap32
- __builtin_bswap64
- __builtin_calloc
- __builtin_choose_expr
- __builtin_clz
- __builtin_clzl
- __builtin_clzll
- __builtin_constant_p
- __builtin_ctz
- __builtin_ctzl
- __builtin_ctzll
- __builtin_exit
- __builtin_expect
- __builtin_extract_return_addr
- __builtin_ffs
- __builtin_ffsl
- __builtin_ffsll
- __builtin_fpclassify
- __builtin_fprintf
- __builtin_fputc
- __builtin_fputs
- __builtin_fscanf
- __builtin_fwrite
- __builtin_huge_val
- __builtin_huge_valf
- __builtin_huge_vall
- __builtin_index
- __builtin_inf
- __builtin_inff
- __builtin_infl
- __builtin_isfinite
- __builtin_isinf
- __builtin_isinf_sign
- __builtin_isinff
- __builtin_isinfl
- __builtin_isnan

- __builtin_isnanf
- __builtin_isnanl
- __builtin_isnormal
- __builtin_malloc
- __builtin_memchr
- __builtin_memcmp
- __builtin_memcpy
- __builtin_memmove
- __builtin_mempcpy
- __builtin_memset
- __builtin_nan
- __builtin_nanf
- __builtin_nanl
- __builtin_nans
- __builtin_nansf
- __builtin_nansl
- __builtin_offsetof
- __builtin_popcount
- __builtin_popcountl
- __builtin_popcountll
- __builtin_prefetch
- __builtin_printf
- __builtin_putchar
- __builtin_puts
- __builtin_return_address
- __builtin_rindex
- __builtin_scanf
- __builtin_signbit
- __builtin_signbitf
- __builtin_signbitl
- __builtin_sprintf
- __builtin_sscanf
- __builtin_stpcpy
- __builtin_strcat
- __builtin_strchr
- __builtin_strcmp
- __builtin_strcpy
- __builtin_strcspn

- __builtin_strlen
- __builtin_strncat
- __builtin_strncmp
- __builtin_strncpy
- __builtin_strpbrk
- __builtin_strrchr
- __builtin_strspn
- __builtin_strstr
- __builtin_trap
- __builtin_types_compatible_p
- __builtin_vfprintf
- __builtin_vprintf
- __builtin_vsprintf
- __sync_fetch_and_add
- __sync_fetch_and_sub
- __sync_fetch_and_or
- __sync_fetch_and_and
- __sync_fetch_and_xor
- __sync_fetch_and_nand
- __sync_add_and_fetch
- __sync_sub_and_fetch
- __sync_or_and_fetch
- __sync_and_and_fetch
- __sync_xor_and_fetch
- __sync_nand_and_fetch
- __sync_bool_compare_and_swap
- __sync_val_compare_and_swap
- __sync_synchronize
- __sync_lock_test_and_set
- __sync_lock_release

D.2 GNU C互換オプション

本処理系はGNU C互換オプションをサポートします。

これらのGNU C互換オプションの詳細については、GCCのウェブサイトをお読みください。

{--print-file-name|-print-file-name}=include

本処理系が提供するヘッダを格納するディレクトリを表示することを指示します。

{--print-prog-name|-print-prog-name}={as|ld|objdump|ranlib|ar}

翻訳コマンドが呼び出すプログラムを表示することを指示します。

--shared

リンカに対して、共有オブジェクトを作成するよう指示します。

本オプションは、リンカに渡されます。

--version

コンパイラのバージョンと著作権の情報を標準出力に出力します。

-Wp,-MD,*filename*

本オプションは、-MD -MF *filename*を指定した場合と等価です。

-Xlinker *option*

*option*をリンカへのオプションとして渡すことを指示します。

-dM

-Eオプションが有効な場合、すべてのマクロ定義を出力します。

-f{align-loops[=*N*]|no-align-loops}

ループの先頭アライメントを2の累乗バイトの境界に合わせるか否かを指示します。省略時は、-fno-align-loopsが適用されます。

-f{exceptions|no-exceptions}

既定義マクロ__EXCEPTIONSを定義するか否かを指示します。省略時は、-fno-exceptionsオプションが適用されます。

-ffast-math

数学関数において、IEEE 754またはISOの規則/規格に厳密には従わない最適化を行います。

本オプションを指定した場合、実行結果に副作用(計算誤差および実行時の例外発生など)が生じることがあります。

-ffp-contract=fast

Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行うことを指示します。

本オプションを指定した場合、実行結果に副作用(丸め誤差程度の違い)を生じることがあります。

-f{inline-functions|no-inline-functions}

単純な関数をインライン展開するか否かを指示します。インライン展開の対象とする関数は、コンパイラが自動的に決定します。省略時は、-fno-inline-functionsオプションが適用されます。

-floop-parallelize-all

自動並列化を行うことを指示します。ただし、並列化の効果が期待できないものに対しては、自動並列化を行いません。

ファイル名の並びに本オプションを指定して翻訳したオブジェクトプログラムが含まれている場合、リンク時も本オプションを指定してください。

-fno-common

初期値指定のない大域変数を、オブジェクトファイルのデータセクションに割り当てることを指示します。

複数のソースファイルで、同一の名前の変数がextern指定子なしに宣言されている場合に、リンク時にエラーとして検出できるようになります。

-f{omit-frame-pointer|no-omit-frame-pointer}

フレームポインタをレジスタに格納する必要のない関数において、格納を抑止するか否かを指示します。省略時は、-fno-omit-frame-pointerオプションが適用されます。

-fopenmp

OpenMP仕様を有効にすることを指示します。

-fopenmpオプションが指定された場合、-mtオプションが指定されたものとみなされます。

ファイル名の並びに-fopenmpオプションを指定して翻訳されたオブジェクトプログラムが含まれている場合、-fopenmpオプションを指定する必要があります。

-f{openmp-simd|no-openmp-simd}

OpenMP仕様のsimd構文およびdeclare simd構文のみを有効にするか否かを指示します。省略時は、-fno-openmp-simdオプションが適用されます。

-f{optimize-sibling-calls|no-optimize-sibling-calls}

末尾呼出しの最適化を行うか否かを指示します。省略時は、-fno-optimize-sibling-callsが適用されます。

-f{pic|PIC}

位置独立コード(PIC)を生成することを指示します。

-fpicオプションを指定した場合、-fPICオプションに比べて命令シーケンスが短くなるため高速になりますが、リンク時に参照できるユニークな外部シンボル数は少なくなります。-fPICオプションを指定した場合、-fpicオプションに比べて命令シーケンスが長くなるため低速になりますが、リンク時に参照できるユニークな外部シンボル数は多くなります。これらの場合の参照できるユニークな外部シンボル数は、同時に結合されるすべてのライブラリの合計です。

-f{pie|PIE}

位置独立コード(PIC)を生成することを指示します。

-fpieオプションを指定した場合、-fPIEオプションに比べて命令シーケンスが短くなるため高速になりますが、リンク時に参照できるユニークな外部シンボル数は少なくなります。-fPIEオプションを指定した場合、-fpieオプションに比べて命令シーケンスが長くなるため低速になりますが、リンク時に参照できるユニークな外部シンボル数は多くなります。これらの場合の参照できるユニークな外部シンボル数は、同時に結合されるすべてのライブラリの合計です。

-f{pic|PIC}オプションとの違いは、リンク時に-pieオプションを指定することで、実行可能プログラムを生成できることです。

-f{plt|no-plt}

位置独立コード(PIC)での外部シンボルへのアクセスにProcedure Linkage Table(PLT)を使用するか否かを指示します。省略時は、-fpltオプションが適用されます。

-fprofile-dir=*path*

コードカバレッジ機能使用時に必要な.gcdaファイルの格納先ディレクトリを指示します。*path*には、格納先ディレクトリ名を相対パスまたは絶対パスで指定します。

-f{schedule-insns|no-schedule-insns}

レジスタ割付け前に、命令スケジューリングの最適化を行うか否かを指示します。省略時は、-fno-schedule-insnsが適用されます。

-f{schedule-insns2|no-schedule-insns2}

レジスタ割付け後に、命令スケジューリングの最適化を行うか否かを指示します。省略時は、-fno-schedule-insns2が適用されます。

-f{signed-char|unsigned-char}

char型の宣言を符号付きとみなすか否かを指示します。-fsigned-charオプションを指定した場合、符号付きとみなします。省略時は、-funsigned-charオプションが適用されます。

-f{strict-aliasing|no-strict-aliasing}

ポインタからの間接参照を行う場合に、型が異なるときはエイリアスがないとみなすか否かを指示します。-fstrict-aliasingオプションが指定された場合、エイリアスがないとみなします。省略時は、-fno-strict-aliasingオプションが適用されます。

-funroll-loops

ループアンローリングの最適化を行うことを指示します。

-f{unsafe-math-optimizations|no-unsafe-math-optimizations}

flush-to-zeroモードを使用するか否かを指示します。省略時は、-fno-unsafe-math-optimizationsオプションが適用されます。

-funsafe-math-optimizationsオプションを指定した場合、実行結果に副作用を生じることがあります。

本オプションは、プログラムのリンク時に指定する必要があります。

-g{dwarf|dwarf-4}

DWARF Version 4形式のデバッグ情報を出力します。

本オプションは、-gオプションと等価です。

-idirafter *dir*

ヘッダの検索時に、*dir*を標準のディレクトリの後に追加します。

複数の-idirafterオプションでディレクトリが複数指定された場合、指定された順に検索します。

ヘッダの検索順の詳細については、“2.2.2.1 コンパイラ全般に関連するオプション”の-Iオプションの説明をお読みください。

ヘッダが絶対パス名で指定された場合、指定された絶対パス名だけ検索します。ディレクトリが存在しない場合、本オプションは無効となります。

-include *file*

ソースファイルの先頭で*file*をインクルードするよう指示します。

複数の-includeオプションが指定された場合、指定された順にインクルードします。

-isystem *dir*

ヘッダの検索時に、*dir*を標準のディレクトリとして追加します。

複数の-isystemオプションでディレクトリが複数指定された場合、指定された順に検索します。

ヘッダの検索順の詳細については、“2.2.2.1 コンパイラ全般に関連するオプション”の-Iオプションの説明をお読みください。

ヘッダが絶対パス名で指定された場合、指定された絶対パス名だけ検索します。ディレクトリが存在しない場合、本オプションは無効となります。

-march=*archi*[+*features*]

アーテクチャを指定します。*archi*には、armv8-a、armv8.1-a、armv8.2-a、またはarmv8.3-aを指定します。

*features*には、sveまたはnosveを指定します。*features*の省略時は、sveが適用されます。

-marchオプションの省略時は、-march=armv8.3-a+sveが適用されます。

-mcmodel={small|large}

実行可能プログラムおよび共有オブジェクトの、コード領域と静的データ領域の最大値を指示します。省略時は、-mcmodel=smallオプションが適用されます。

-mcmodel=small

リンク後のコード領域と静的データ領域の合計が4GB以内になると仮定し、効率の良いオブジェクトプログラムを生成します。

-mcmodel=large

リンク後のコード領域が4GB以内になると仮定します。静的データ領域の大きさに制約はありません。静的データ領域が大きくリンク時にエラーが発生する場合は、本オプションを指定します。

-mcpu=*cpu*

ターゲットのプロセッサを指定します。*cpu*には、a64fx、thunderx2t99、またはgenericを指定します。省略時は、-mcpu=a64fxが適用されます。

-m{pc-relative-literal-loads|no-pc-relative-literal-loads}

関数内のコード領域が1MB以内であるとして扱い、リテラルプールに1命令でアクセスします。省略時は、-mno-pc-relative-literal-loadsが適用されます。

-mtls-size={12|24|32|48}

スレッド・ローカル・ストレージ(Thread-Local Storage)へのアクセスに必要なオフセットのサイズを指定します。単位はビットです。

-mtune=*cpu*

最適化のターゲットとするプロセッサを指定します。*cpu*には、a64fx、thunderx2t99、またはgenericを指定します。省略時は、-mtune=a64fxが適用されます。

-nostartfiles

リンク時に、標準のシステムスタートアップファイルを使用しないことを指示します。

-nostdinc

ヘッダの検索時に、標準のディレクトリを検索しないことを指示します。

-nostdlib

リンク時に、標準のシステムスタートアップファイルおよび標準ライブラリを使用しないことを指示します。

-pthread

POSIXスレッドライブラリを利用するマルチスレッドオブジェクトを生成することを指示します。

-rdynamic

リンカに対して、すべてのシンボルを動的シンボルテーブルに追加するよう指示します。本オプションは、リンカに-export-dynamicオプションを渡します。

-undef

システム固有およびGNU C固有の既定義マクロを抑止することを指示します。

-v

翻訳コマンドが呼び出すコンパイラ、アセンブラおよびリンカの実行コマンドを表示します。

-w

すべての警告を抑止することを指示します。

-x *language*

本オプションより後に指定された入力ファイルの種別を指示します。*language*には、cまたはassembler-with-cppを指定します。

付録E　データおよびメモリ領域

ここでは、本処理系でのデータおよびメモリ領域の属性について説明します。

E.1　データのサイズおよびアライメント

基本的なデータの型、サイズ、およびアライメントを以下の表に示します。

データの型 (注1)	サイズ(byte)	アライメント(byte)
char	1	1
short int	2	2
int	4	4
long int	8	8
long long int		
float	4	4
double	8	8
long double	16	16
float _Complex	8	4
double _Complex	16	8
long double _Complex	32	16
構造体	-	(注2)
共用体		

注1) 符号付きおよび符号なし共通

注2) メンバ変数の最大アライメントと8とを比較し、大きい方の値

E.2　メモリ領域

メモリ領域の主な用途を以下に示します。

メモリ領域	用途
Text	命令が配置されるメモリ領域
静的データ(.data)	初期化ありの静的データが格納されるメモリ領域
静的データ(.bss)	初期化なしの静的データが格納されるメモリ領域
プロセススタック	プロセス/メインスレッド用のスタック領域
スレッドスタック	子プロセス用のスタック領域
動的メモリ確保領域	動的にメモリの確保を要求する時に割り当てられるメモリ領域 または、 スレッドヒープ領域/スレッドスタックとして割り当てられるメモリ領域
共有メモリ	プロセス間で共有するメモリ領域

E.3　データの割付け

変数を割り付けるメモリ領域は、変数の種別や初期値の有無により異なります。

例

```
int e[2];
int f[2] = {3,3};

int main(){
    int          a[2];
    int          b[2] = {1,1};
    static float c[2];
    static float d[2] = {2.0,2.0};
    double       *g;

    g = (double *)alloca(sizeof(double) * 2);
    ...
}
```

このプログラムでは、各配列を以下のように割り付けます。

配列名	説明
a	局所配列であるため、プロセススタック領域に割り付けます。
b	
c	静的配列であり、初期設定がないため、静的データ領域(.bss)に割り付けます。
d	静的配列であり、初期設定があるため、静的データ領域(.data)に割り付けます。
e	大域配列であり、初期設定がないため、静的データ領域(.bss)に割り付けます。
f	大域配列であり、初期設定があるため、静的データ領域(.data)に割り付けます。
g	動的メモリ確保領域に割り付けます。

E.4 スタック領域へのデータ割付け

スタック領域へのデータ割付け順序を、以下の翻訳時オプションによって制御できます。

-N{reordered_variable_stack|noreordered_variable_stack}

自動変数をスタック領域に割り付ける順序をコンパイラに指示します。

-Nreordered_variable_stackオプションが指定された場合、以下の順序で自動変数の割付け順を決めます。

1. アライメントの降順
2. アライメントが等しい場合は、データサイズの降順
3. アライメントおよびデータサイズが等しい場合は、ソースプログラム内の宣言文の記載順

アライメントの降順に自動変数を割り付けると、プログラム全体のスタック領域を減らすことができます。

-Nnoreordered_variable_stackオプションが指定された場合、ソースプログラム内の宣言文の記載順に自動変数を割り付けます。省略時は、-Nnoreordered_variable_stackオプションが適用されます。

-Nnolineオプション、-g0オプション、または-Koclオプションが有効な場合、割付け順序は保証されません。

例

- ソースプログラム

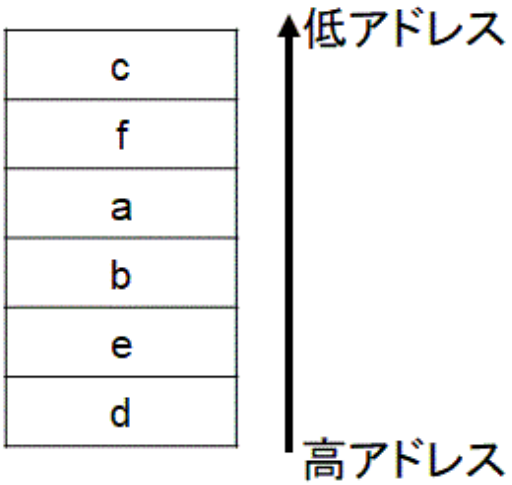
```
int    a;
double b;
char   c;
double  d[2],e[2];
short int f;
```

```
a = 1;
b = 2.0;
c = '3';
d[0] = 4.0;
e[0] = 5.0;
f = 6;
```

- データ割付け順序を指定した場合

  データ割付け順序を指定した場合、自動変数は下図のようにスタック領域に割り付けられます。プログラム全体のスタック領域を減らすことができます。

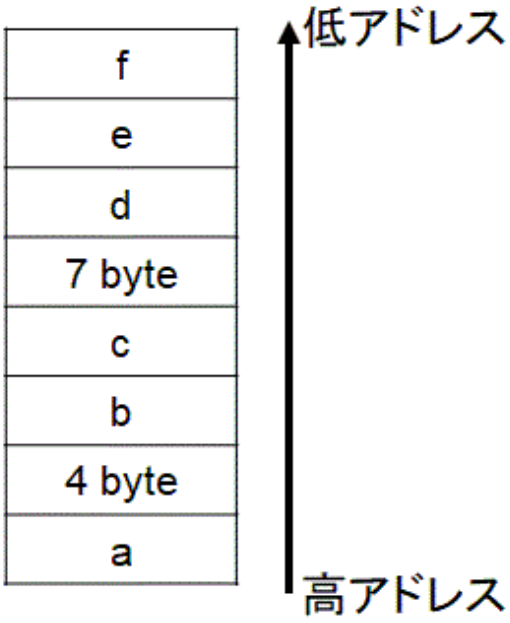
  図E.1 -Nreordered_variable_stackオプション有効時の割付け



- データ割付け順序を指定しない場合

  データ割付け順序を指定しない場合、自動変数は下図のようにスタック領域に割り付けられます。

  図E.2 -Nnoreordered_variable_stackオプション有効時の割付け



付録F　コードカバレッジ機能

ここでは、コードカバレッジ機能について説明します。

コードカバレッジ機能は、実行時に実行文の実行回数を計測し、プログラムのコード網羅率を調べる機能です。

コードカバレッジ機能では、オープンソースソフトウェアであるコンパイラ基盤LLVMに含まれるコードカバレッジツール(以降、llvm-covコマンドと呼びます)を使用します。llvm-covコマンドの使用方法については、manコマンドを使用して、llvm-covコマンドのオンラインマニュアルを参照してください。

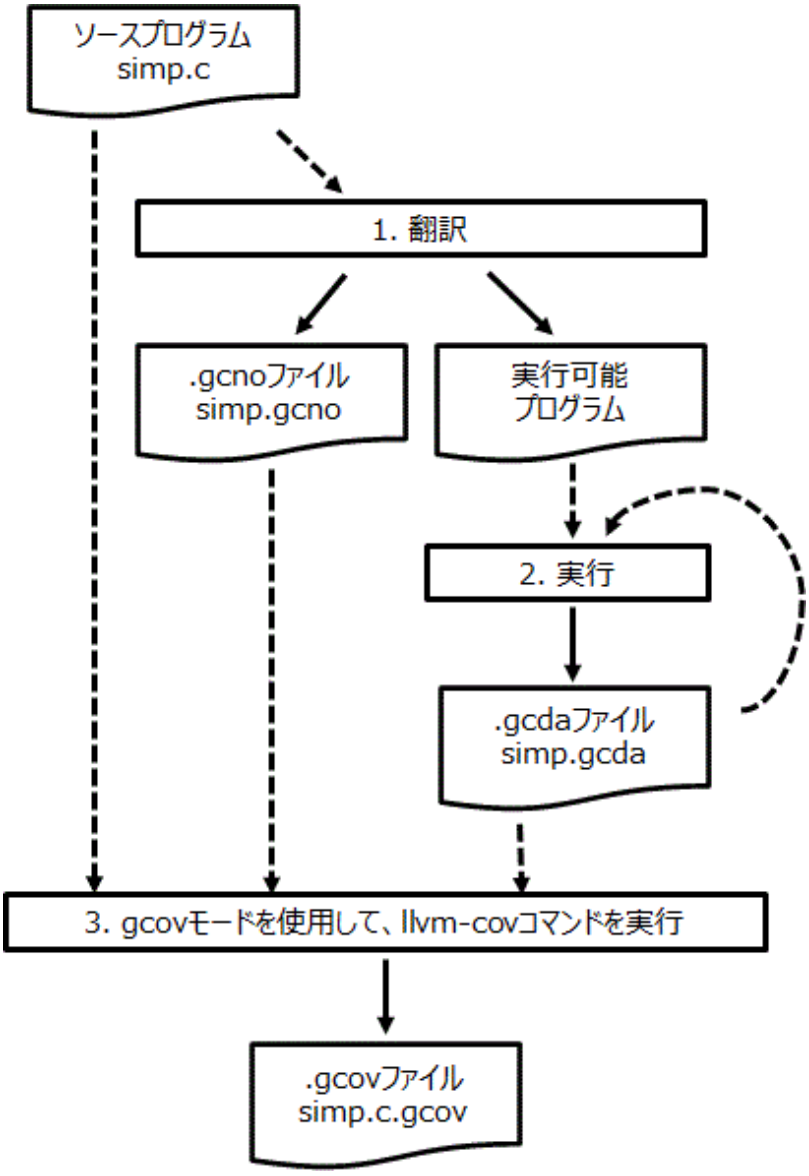
F.1　コードカバレッジ機能の使用方法

コードカバレッジ機能は、以下の手順で使用します。

1. 翻訳
2. 実行
3. gcovモードを使用して、llvm-covコマンドを実行

以下の図に、コードカバレッジ機能の手続きを示します。

図F.1 コードカバレッジ機能の手続き



F.2 コードカバレッジ機能使用時に必要なファイル

コードカバレッジ機能を使用する時に必要なファイルについて説明します。

F.2.1 .gcnoファイル

.gcnoファイルは、翻訳時に得られる情報を含んだバイナリファイルです。

ファイル生成タイミング	翻訳時に生成されます。
ファイル名	ソースプログラムの拡張子をgcnoに変更したファイル名になります。
ファイル生成場所	翻訳を行ったディレクトリに生成されます。

以下に、.gcnoファイルの生成例を示します。

例

```
$ fcc -Ncoverage ../simp.c
$ ls
a.out simp.gcno
```

-oオプションを指定してアセンブラソースファイルおよびオブジェクトファイルを生成した場合は、-oオプションが、.gcnoファイルにも適用されます。

以下に、-oオプションを指定した場合の生成例を示します。

例

```
$ fcc -Ncoverage ../simp.c -c -o www/yyy.o
$ ls www/
yyy.gcno yyy.o
```

F.2.2 .gcdaファイル

.gcdaファイルは、実行時に得られる情報を含んだバイナリファイルです。

ファイル生成タイミング	実行時に生成されます。 同一名のファイルがすでに存在する場合は、そのファイルに情報が累積されます。
ファイル名	ソースプログラムの拡張子をgcdaに変更したファイル名になります。
ファイル生成場所	翻訳を行ったディレクトリに生成されます。

以下に、.gcdaファイルの生成例を示します。

例

```
$ fcc -Ncoverage ../simp.c
$ ls
a.out simp.gcno
$ ./a.out
$ ls
a.out simp.gcda simp.gcno
```

-oオプションを指定してアセンブラソースファイルおよびオブジェクトファイルを生成した場合は、-oオプションが.gcdaファイルにも適用されます。

以下に、-oオプションを指定した場合の生成例を示します。

例

```
$ fcc -Ncoverage ../simp.c -c -o www/yyy.o
$ ls www/
yyy.gcno yyy.o
$ fcc -Ncoverage www/yyy.o
$ ./a.out
$ ls www/
yyy.gcda yyy.gcno yyy.o
```

.gcdaファイルの格納先ディレクトリは-Nprofile_dir=*dir_name*オプションで変更できます。

以下に、-Nprofile_dir=*dir_name*オプションを指定した場合の生成例を示します。

例

```
$ fcc -Ncoverage ../simp.c -c -o www/yyy.o -Nprofile_dir=/tmp
$ ls www/
yyy.gcno yyy.o
$ fcc -Ncoverage www/yyy.o
$ ./a.out
$ ls www/
yyy.gcno yyy.o
$ ls /tmp/www/
yyy.gcda
```

F.3 コードカバレッジ機能の注意事項

コードカバレッジ機能使用時の注意事項について説明します。

- 実行回数を計測する命令がオブジェクトファイル内に追加されるため、実行性能が低下する場合があります。
- 実行時に生成される.gcdaファイルの格納先ディレクトリは、翻訳時に決定されます。
  - 翻訳後に実行可能プログラムを移動しても、.gcdaファイルの格納先ディレクトリは変わりません。
  - .gcdaファイルの格納先ディレクトリを変更したい場合は、翻訳時に-Nprofile_dir=*dir_name*オプションを使用して変更してください。
- -Kcmodel=largeオプションまたは-mcmodel=largeオプションを必要とするプログラムでは、使用できません。
- 以下の場合、実行回数を正しく計測できないことがあります。
  - 1行に複数の実行文が含まれる場合
  - exit(3)関数が呼ばれた場合
  - setjmp関数またはlongjmp関数を呼び出す場合
  - 例外が捕捉された場合
  - ソースプログラムに#line指令が含まれる場合
  - コンパイラの最適化が適用された場合

付録G　富士通拡張関数

ここでは、本処理系で提供する富士通拡張関数について説明します。

G.1 富士通拡張関数の使用方法

G.1.1 ヘッダ

富士通拡張関数を使用するためには、以下の標準ヘッダをインクルードします。

・ fjcex.h

G.1.2 関数一覧

本処理系が提供する富士通拡張関数を以下に示します。

表G.1 富士通拡張関数一覧

関数プロトタイプ	説明
double __gettod(void)	__gettod()関数は現在の高分解能な実時間を返します。実時間は過去のある任意の時間からのマイクロ秒単位の経過時間です。通常、システムブートからの時間を表します。

付録H　実行時情報出力機能

ここでは、実行時情報出力機能について説明します。

実行時情報出力機能は、プログラムの高速化や性能確認の足掛かりとなる情報を、翻訳時オプションおよび実行時環境変数を指定することによって、プログラム実行時に計測して出力する機能です。

H.1　実行時情報出力機能の使用方法

ここでは、実行時情報出力機能の使用方法について説明します。

H.1.1　情報を取得できる範囲

実行時情報出力機能で情報を取得できる範囲を以下に示します。

表H.1 情報を取得することができる範囲

翻訳時オプション	実行時環境変数	情報取得可能な範囲
-Nrt_tune	FLIB_RTINFO	プログラム開始から終了まで
-Nrt_tune_func		利用者定義の関数ごと
-Nrt_tune_loop		ループごと

翻訳時オプション-Nrt_tuneおよび実行時環境変数FLIB_RTINFOを指定することにより、実行時情報の出力機能が有効となり、プログラムの開始から終了までの情報を取得することができます。また、翻訳時オプション-Nrt_tune_funcまたは-Nrt_tune_loopを指定することにより、利用者定義の関数ごとの情報、ループごとの情報も取得することができます。

翻訳時オプションについては、“2.2 翻訳時オプション”を参照してください。

MPIプログラムおよびCOARRAYプログラムの場合は、各プロセスごとの情報が出力されます。

利用者定義の関数ごと

翻訳時オプション-Nrt_tune_funcを指定すると、利用者定義の関数ごとの情報が出力されます。

注意

- 関数の中から関数が呼び出された場合、呼び出された関数の情報は、呼出し元の関数の情報に含まれません。
- 同じ関数が複数箇所で呼び出された場合、合計値がその関数の情報になります。
- 出力される関数は、ユーザが定義した関数のみで、システムコールやライブラリ関数については情報が取得されません。

ループごと

翻訳時オプション-Nrt_tune_loopを指定すると、ループごとに以下の情報が出力されます。

- 逐次ループごとの情報
- OpenMP指示文で並列化された箇所内のループごとの情報

逐次ループのコストは、逐次ループごとの情報として出力されます。OpenMP指示文で並列化された箇所内にループが存在する場合は、OpenMP指示文で並列化された箇所内のループごとの情報として出力されます。OpenMP指示文で並列化された箇所内のループの情報は、マスタースレッドの情報のみが出力されます。

注意

- ループの中から関数が呼び出された場合、呼び出された関数の情報は、呼出し元のループの情報に含まれます。
- ループの中からループが呼び出された場合、呼び出されたループの情報は、呼出し元のループの情報に含まれます。

H.1.2 実行時環境変数

実行時情報出力機能で使用する環境変数について、以下に示します。

実行時環境変数のオペランドに指定される値は、英大文字と英小文字を区別します。オペランドが不要である環境変数については、オペランドに指定した値は無視されます。

表H.2 出力する情報を制御する環境変数

実行時環境変数		意味
変数名	オペランド	
FLIB_RTINFO	なし	実行時情報を出力します。
FLIB_RTINFO_NOFUNC	なし	翻訳時オプション-Nrt_tune_funcを指定した場合でも、利用者定義の関数ごとの情報出力を抑止します。
FLIB_RTINFO_NOLOOP	なし	翻訳時オプション-Nrt_tune_loopを指定した場合でも、ループごとの情報出力を抑止します。
FLIB_RTINFO_CSV	ファイル名	取得した情報を、CSV形式でファイルに出力します。また、オペランドには任意のファイル名が指定可能です。オペランドを省略した場合は、flib_rtinfo.csvというファイルに出力します。
	-	

H.2 実行時情報出力機能が出力する情報

ここでは、実行時情報出力機能が出力する情報について説明します。

H.2.1 出力情報

実行時情報出力機能が出力する情報を以下に示します。

表H.3 出力情報

翻訳時オプション	実行時環境変数	出力情報
-Nrt_tune	FLIB_RTINFO	■プログラムの開始から終了までの以下の情報を出力 ・ 経過時間 ・ ユーザCPU時間 ・ システムCPU時間
-Nrt_tune_func		■-Nrt_tuneの情報に加えて、利用者定義の関数ごとに、以下の情報を出力 ・ 関数のコスト ・ 関数のコストの割合 ・ 関数の呼出し1回当たりのコスト ・ 関数の開始行番号 ・ 関数の終了行番号 ・ 関数の呼出し回数 ・ 関数名
-Nrt_tune_loop		■-Nrt_tuneの情報に加えて、逐次ループごとおよびOpenMP指示文で並列化された箇所内のループごとに、以下の情報を出力 ・ ループのコスト ・ ループのコストの割合 ・ ループの呼出し1回当たりのコスト ・ ループの開始行番号

翻訳時オプション	実行時環境変数	出力情報
		• ループの終了行番号 • ループの呼出し回数 • ループのネストレベル • 関数名

H.2.2 出力形式

実行時情報は、テキスト形式またはCSV形式で出力することができます。

テキスト形式

環境変数FLIB_RTINFO_CSVを指定しない場合、情報はテキスト形式で標準出力ファイルに出力されます。

情報の出力形式の詳細については、“H.2.3 出力例”を参照してください。

CSV形式

環境変数FLIB_RTINFO_CSVを指定することにより、情報をCSV形式で出力することができます。オペランドにファイル名を指定した場合、指定したファイルに取得した情報が出力されます。ファイル名を指定しない場合、flib_rtinfo.csvというファイルに取得した情報が出力されます。すでに指定したファイルが存在する場合は、既存のファイルに情報が上書きされます。ファイルのオープンに失敗した場合は、実行時メッセージjwe1653i-wが出力され、標準出力に情報が出力されます。

CSV形式で出力する場合、データとなる行の情報は以下に示す構成に従います。出力する情報がない場合、“-”が出力されます。

表H.4 CSV形式で出力されるカラム構成

カラム位置	出力される情報
1カラム目	出力された情報の種別を示す識別子 [COST] : プログラム全体の情報 [COST_ROUTINE] : 関数ごとの情報 [COST_LOOP_SERIAL] : 逐次ループごとの情報 [COST_LOOP_PRL] : OpenMP指示文で並列化された箇所内のループごとの情報
2カラム目以降	出力する情報によって異なります。

情報の出力形式の詳細については、“H.2.3 出力例”を参照してください。

H.2.3 出力例

実行時情報出力機能の情報の出力結果の例を以下に示します。

実行時情報の出力例

```
++=============================================================================================++
||                                EXECUTION PERFORMANCE INFORMATION                            ||
++=============================================================================================++

+(1)---------+(2)---------+(3)---------+
| Elapsed    | User       | System     |
|       (sec)|       (sec)|       (sec)|
+------------+------------+------------+
|     2.5691|     2.5636|     0.0020|
+------------+------------+------------+

Routine (4)
+(5)---------+(6)----+(7)---------+(8)-----+(9)-----+(10)----+(11)-------------------------+
| Cost(sec)  |  %    | Once(sec)  | Start  | End    | Count  |          Routine name         |
+------------+-------+------------+--------+--------+--------+-------------------------------+
```

```
|     1.9691| 84.41|     0.0010|     71|     86|   2000|sub2                          | (12)
|     0.3638| 15.59|     0.0364|     38|     68|     10|sub1                          |
+-----------+------+-----------+-------+-------+-------+------------------------------+
|     2.3329|100.00|          -|      -|      -|      -|-                             | (13)
+-----------+------+-----------+-------+-------+-------+------------------------------+

Serial Loop (14)
+-----------+------+-----------+-------+-------+-------+(15)+------------------------------+
| Cost(sec) |  %   | Once(sec) | Start | End   | Count |Nest|         Routine name         |
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
|     2.1979| 45.25|     2.1979|     25|     27|      1|   1|main                          | (16)
|     2.1658| 44.59|     0.0011|     77|     78|   2000|   2|(sub2)                        |
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
|     4.8568|100.00|          -|      -|      -|      -|   -|-                             | (17)
+-----------+------+-----------+-------+-------+-------+----+------------------------------+

Parallel Loop (18)
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
| Cost(sec) |  %   | Once(sec) | Start | End   | Count |Nest|         Routine name         |
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
|     0.2462| 50.04|     0.0246|     58|     64|     10|   2|(sub1._OMP_1)                 |
|     0.2458| 49.96|     0.0002|     59|     60|   1000|   3|(sub1._OMP_1)                 |
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
|     0.4919|100.00|          -|      -|      -|      -|   -|-                             |
+-----------+------+-----------+-------+-------+-------+----+------------------------------+
```

(1) 経過時間

(2) ユーザCPU時間

(3) システムCPU時間

(4) 関数ごとの情報(-Nrt_tune_func指定時)

(5) コスト(単位は秒)

(6) コストが全体に占める割合

(7) 呼出し1回当たりのコスト

(8) 開始行番号

(9) 終了行番号

(10) 呼出し回数

(11) 関数名

(12) 各関数の情報

(13) 各関数の合計の情報

(14) 逐次ループごとの情報(-Nrt_tune_loop指定時)

(15) ネストレベル

(16) 各ループの情報

(17) 各ループの合計の情報

(18) OpenMP指示文で並列化された箇所内のループごとの情報(-Nrt_tune_loop指定時)

備考.ループごとの情報取得時には、関数名として内部関数名が出力されます。ネストされたループの場合には、関数名が括弧で囲まれます。

実行時情報の出力例(FLIB_RTINFO_CSV指定時)

```
========================================
EXECUTION PERFORMANCE INFORMATION
========================================
Type,Elapsed(sec),User(sec),System(sec)
```

```
[COST],2.5525,2.5716,0.0020

Routine
Type,Cost(sec),%,Once(sec),Start,End,Count,Routine name
[COST_ROUTINE],1.9597,84.39,0.0010,71,86,2000,"sub2"
[COST_ROUTINE],0.3626,15.61,0.0363,38,68,10,"sub1"
[COST_ROUTINE],2.3222,100.00,-,-,-,-,-

Serial Loop
Type,Cost(sec),%,Once(sec),Start,End,Count,Nest,Routine name
[COST_LOOP_SERIAL],2.1828,45.18,2.1828,25,27,1,1,"main"
[COST_LOOP_SERIAL],2.1596,44.70,0.0011,77,78,2000,2,"(sub2)"
[COST_LOOP_SERIAL],0.2276,100.00,-,-,-,-,-,-

Parallel Loop
Type,Cost(sec),%,Once(sec),Start,End,Count,Nest,Routine name
[COST_LOOP_PRL],0.2476,50.06,0.0248,58,64,10,2,"(sub1._OMP_1)"
[COST_LOOP_PRL],0.2470,49.94,0.0002,59,60,1000,3,"(sub1._OMP_1)"
[COST_LOOP_PRL],2.3222,100.00,-,-,-,-,-,-
```

H.3 実行時情報出力機能の注意事項

ここでは、実行時情報出力機能の注意事項について説明します。

- -Nrt_tune_funcまたは-Nrt_tune_loopを指定し、関数ごとまたはループごとの情報を取得する場合、取得する情報が増加し、実行時間が増加する可能性があります。また、情報が取得できない場合、診断メッセージjwe1655i-iが出力されることがあります。エラー処理中に出力された場合、診断メッセージの出力される順序が正しくない場合があります。
- プログラム中にgoto文、setjmp()/longjmp()などがある場合、または例外が捕捉された場合には、正しく情報を計測できない場合があります。
- コンパイラの最適化などにより、以下の影響を受ける可能性があります。
  - ー ループごとの情報を取得する場合、計測の対象となるループが変わる場合があります。
  - ー 行番号がソースプログラムの行番号と一致しない場合があります。
  - ー ループのネスト関係がソースプログラムのネストと一致しない場合があります。次のような最適化によって、ループ情報を正しく認識できない可能性があります。
    - 関数のインライン展開
    - ループアンローリング
    - ループブロッキング
    - ループ交換
    - ループ融合
    - ループストライピング
    - ループ分割
    - ループピーリング
    - ループアンスイッチング
    - マルチ演算関数化・SIMD化

付録I　高速化機能の利用

ここでは、FXシステム向けの高速化機能を使用するためのプログラムの翻訳および実行について説明します。

以下の環境変数を使用することにより、FXシステムのCPUで実装されている高速化機能(コア間ハードウェアバリアとセクタキャッシュ)を利用することができます。

環境変数FLIB_HPCFUNC

高速化機能(コア間ハードウェアバリア、セクタキャッシュ)を利用するかどうかを指示します。TRUEを指定した場合、高速化機能を利用することができます。FALSEを指定した場合、高速化機能は利用できません。省略値はFALSEです。

環境変数を指定しない場合、高速化機能は利用できません。

値	説明
TRUE	高速化機能(コア間ハードウェアバリア、セクタキャッシュ)の利用を可能にします。 NUMAノードを1プロセスで占有可能である場合に、利用することができます。 CPUの上限値は1ノードに割り当てられたCPU数です。
FALSE (省略値)	高速化機能(コア間ハードウェアバリア、セクタキャッシュ)は利用できません。 以下の環境変数を指定している場合、エラーメッセージ(jwe1047i-w, jwe1050i-w)が出力されます。 ・ FLIB_SCCR_CNTL=TRUE ・ FLIB_BARRIER=HARD

環境変数FLIB_HPCFUNC_INFO

実行開始時のプロセスのCPU affinityを表示するかどうかを指示します。省略値はFALSEです。

環境変数を指定しない場合、実行開始時のプロセスのCPU affinityは表示されません。

値	説明
TRUE	実行開始時のプロセスのCPU affinityを表示します。
FALSE (省略値)	実行開始時のプロセスのCPU affinityを表示しません。

I.1　コア間ハードウェアバリア

コア間ハードウェアバリア(以降、ハードウェアバリアと呼びます)とは、スレッド並列処理における同期処理を高速にし、実行性能を向上させるハードウェア機能です。ここでは、LLVM OpenMPライブラリを使用して、ハードウェアバリアを使用するためのプログラムの翻訳および実行について説明します。

I.1.1　翻訳

ハードウェアバリアをスレッド間バリアとして利用するプログラムを作成するためには、-Kparallelオプションまたは-Kopenmpオプションを指定します。

I.1.2　実行

環境変数FLIB_BARRIER

OpenMPまたは自動並列を使用する場合のスレッド間バリアの種類を制御できます。指定できる値とその意味は、以下のとおりです。デフォルトは、SOFTです。

HARD

ハードウェアバリアを使用します。

使用できない場合、診断メッセージjwe1050i-wを出力し、ソフトウェアバリアを使用して実行を続けます。

ハードウェアバリアを使用する場合、以下のOpenMP機能のすべてまたは一部が使用できません。これらの機能を使用した場合、診断メッセージjwe1051i-wを出力し、機能を無効化して実行を続けます。

- スレッド数の制御

  スレッド数は、以下の優先順位に従って決まります。

  1. 環境変数OMP_NUM_THREADSの指定値

  2. CPU数の上限値

  この優先順位で決まるスレッド数がCPU数の上限値を超える場合、スレッド数はCPU数の上限値になります。

  num_threads指示節およびomp_set_num_threads関数に指定する値は、上記で決まるスレッド数と一致していなければなりません。環境変数OMP_DYNAMICおよびomp_set_dynamic関数の設定に関わらず、スレッド数はCPU数の上限値を超えることはできません。

- スレッドアフィニティの制御

  プログラム開始時にプロセスに割り当てられたCPUを使用してスレッドアフィニティを設定します。

  環境変数OMP_PROC_BINDおよびparallel構文のproc_bind指示節の設定に関わらず、OpenMPのスレッドアフィニティ機能は無効化されます。他の方法でスレッドアフィニティを設定した場合、動作は保証されません。

- ネスト並列

  環境変数OMP_NESTEDおよびomp_set_nested関数の設定に関わらず、ネストしたparallelリージョンは常に逐次化されます。

- タスク

  常に即時に実行するタスクを生成します。

- キャンセレーション

  環境変数OMP_CANCELLATIONの設定に関わらず、常にcancel構文とキャンセレーションポイントの効果は無効化されます。

SOFT

ソフトウェアバリアを使用します。(省略値)

注意

- スレッド数が1の場合、ハードウェアバリアは使用できません。
- ハードウェアバリアは同じNUMAノード内でのみ使用できます。したがって、プロセスが複数のNUMAノードに跨る場合は、NUMAノード内はハードウェアバリアが使用されますがNUMAノード間はソフトウェアバリアが使用されるためNUMAノード内にのみと比べて性能は大きく低下します。バリア性能を優先する場合は、1プロセスが1NUMAノード内に収まる使い方を推奨します。
- 以下の場合、1つのプロセスがコア間ハードウェアバリア資源を占有できません。そのためハードウェアバリアの動作は不定になります。診断メッセージjwe1044i-uまたはjwe1045i-uを出力しプログラムを終了する、または、異常終了することがあります。

  ー マルチプロセスで実行している場合

  例:

  - Fortran FORK/SYSTEM/SHサービス関数などを使用する場合
  - C forkシステムコール/system関数などを使用する場合

例

例: コア間ハードウェアバリアを使用するプログラムを実行

```
$ cat exec.sh
#!/bin/sh
export FLIB_HPCFUNC=TRUE
export FLIB_BARRIER=HARD
```

```
./a.out
$ ./exec.sh
```

I.2 セクタキャッシュ

セクタキャッシュの制御方法については、“3.5 セクタキャッシュのソフトウェア制御”をお読みください。

ここでは、実行上の注意を説明します。

実行上の注意

1次キャッシュと2次キャッシュでセクタキャッシュを利用できる実行環境は以下の通りです。

実行環境	1次キャッシュでセクタキャッシュ利用	2次キャッシュでセクタキャッシュ利用
NUMAノードを1プロセスで占有できる	利用可	利用可
NUMAノードを1プロセスで占有できない	利用可 (注)	利用不可

注) 環境変数FLIB_L1_SCCR_CNTL=FALSEを指定した場合、利用しません。環境変数FLIB_L1_SCCR_CNTLについては、“3.5.2.2 環境変数および最適化制御行によるソフトウェア制御”を参照してください。

また以下の場合、NUMAノードを1プロセスで占有できる場合を識別できません。そのためセクタキャッシュの動作は不定になります。性能が大幅に劣化する、診断メッセージjwe1048i-uを出力しプログラムを終了する、またはプログラムが異常終了することがあります。

- マルチプロセスプログラムの場合

  例:

  - Fortran FORK/SYSTEM/SHサービス関数などを使用する場合
  - C forkシステムコール/system関数などを使用する場合

- スレッドを富士通の自動並列/OpenMP機能以外で生成する場合

  例:

  - pthread関数などでスレッドを制御する場合

実行環境に関わらず無条件にセクタキャッシュ制御機能を無効化したい場合は、環境変数FLIB_SCCR_CNTLにFALSEを指定します。詳細は、“3.5.2.2 環境変数および最適化制御行によるソフトウェア制御”をお読みください。

例

例: セクタキャッシュを使用するプログラムを実行

```
$ cat exec.sh
#!/bin/sh
export FLIB_HPCFUNC=TRUE
./a.out
$ ./exec.sh
```

付録J　マルチプロセスによる実行

ここでは、マルチプロセスでプログラムを実行する方法および注意事項について説明します。

J.1　CPU affinityの指定

マルチプロセスでプログラムを実行する場合、環境変数FLIB_PROCESS_CPU_AFFINITYを使って、プロセスのCPU affinityを設定することができます。

プログラム開始時に有効になっているスレッドアフィニティポリシーに従い、システムは指定されたCPUにスレッドをバインドします。

環境変数FLIB_PROCESS_CPU_AFFINITY

プロセスに割り当てるcpuidを指定します。

各cpuidは、コンマ(',')または空白(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

cpuid1[-*cpuid2*[:*inc*]]

cpuid1: 範囲指定の最初のcpuid (0<=*cpuid1*<CPU_SETSIZE)
cpuid2: 範囲指定の最後のcpuid (0<=*cpuid2*<CPU_SETSIZE)
inc: 増分値 (1<=*inc*<CPU_SETSIZE)

なお、以下の条件を満たす必要があります。

cpuid1<=*cpuid2*

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべてを指定した場合と等価になります。

cpuidは、上記のように指定はできますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)を参照してください。

注意

- cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーを出力しプログラムが終了する、プログラムが終了しない、など、プログラムが正しく動作しないことがあります。設定値を見直してください。実行開始時のプロセスのCPU affinityについては、FLIB_HPCFUNC_INFO=TRUE、またはtasksetやnumactlなどのシステムのコマンドで、確認することができます。環境変数FLIB_HPCFUNC_INFOについては、“付録I 高速化機能の利用”を参照してください。システムのコマンドについては、各manマニュアルを参照してください。
- 環境変数FLIB_HPCFUNC=TRUEを指定していない場合、環境変数FLIB_PROCESS_CPU_AFFINITYは無効になります。環境変数FLIB_HPCFUNCについては、“付録I 高速化機能の利用”を参照してください。
- Fortran FORK、SYSTEM、SHサービス関数、C forkシステムコール、system関数などを使用してマルチプロセスでプログラムを実行する場合は、コア間ハードウェアバリアは使用できません。ソフトウェアバリアを使用してください。コア間ハードウェアバリアについては“I.1 コア間ハードウェアバリア”を参照してください。
- 環境変数FLIB_PROCESS_CPU_AFFINITYに指定していない値を、環境変数GOMP_CPU_AFFINITYに指定した場合、エラーメッセージが出力されてプログラムが終了します。環境変数GOMP_CPU_AFFINITYについては、自動並列化機能を使用している場合は“4.2.2.4 スレッドのCPUバインド”を、OpenMP仕様に基づいた並列化機能を使用している場合は“4.3.2.1 実行時の環境変数”を参照してください。

例

環境変数FLIB_PROCESS_CPU_AFFINITYの指定例

— 例1:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0,1,2,3,4,5,6,7"
```

0,1,2,3,4,5,6,7をプロセスに割り当てます。

— 例2:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-7"
```

0,1,2,3,4,5,6,7をプロセスに割り当てます。

— 例3:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-7:2"
```

0,2,4,6をプロセスに割り当てます。

— 例4:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-5:2,3,8"
```

0,2,3,4,8をプロセスに割り当てます。

付録K　富士通OpenMPライブラリ

ここでは、tradモードにおいて富士通OpenMPライブラリを使用して、C言語プログラムを並列処理する方法について説明します。

LLVM OpenMPライブラリを使用する場合は、“第4章 並列化機能”をお読みください。

注意

以下の条件をすべて満たしている場合、本処理系は富士通OpenMPライブラリを使用します。

- プログラムのリンク時に、-Nfjomplibオプションが有効である。
- プログラムの翻訳時およびリンク時に、tradモードを使用している。

参考

並列処理用ライブラリ(LLVM OpenMPライブラリおよび富士通OpenMPライブラリ)と結合可能なオブジェクトファイルの組合せを下表に示します。

	Fortran オブジェクト ファイル	C/C++オブジェクトファイル	
		tradモード (-Nnoclangオプション)	clangモード (-Nclangオプション)
LLVM OpenMPライブラリ (-Nlibompオプション)	結合可	結合可	結合可
富士通OpenMPライブラリ (-Nfjomplibオプション)	結合可	結合可	**結合不可**

K.1　並列処理の概要

“4.1 並列処理の概要”をお読みください。

K.2　自動並列化

ここでは、本処理系で提供している自動並列化について説明します。

K.2.1　翻訳の方法(自動並列化)

自動並列化の機能を使用するには、翻訳コマンドに以下のオプションを指定します。

- 翻訳時

```
-Kparallel
```

- リンク時

```
-Kparallel -Nfjomplib
```

自動並列化のための翻訳時オプション

自動並列化機能に関連するオプションを以下に示します。各オプションの詳細については、“2.2 翻訳時オプション”をお読みください。

```
-K{parallel|parallel_strong|visimpact}
[,array_private,dynamic_iteration,instance=N,loop_part_parallel,ocl,optmsg=2,parallel_fp_precision,parallel_iteration=N,
reduction,region_extension] -Nfjomplib
```

K.2.2 実行の方法(自動並列化)

自動並列化されたプログラムを実行させるときは、環境変数PARALLELまたはOMP_NUM_THREADSで、並列に動作させるスレッドの数を指定することができます。また、環境変数THREAD_STACK_SIZEまたはOMP_STACKSIZEで、スレッドごとのスタック領域の大きさを指定することができます。さらに、環境変数FLIB_SPINWAITまたはOMP_WAIT_POLICYで、スレッドの同期待ちの方法を変更することができます。

その他の実行に関する手続きは、逐次実行のときと同じです。

環境変数PARALLEL

環境変数PARALLELで、並列に動作させるスレッドの数を指定することができます。指定できる値は、1～2147483647の整数値です。

スレッド数の決定方法の詳細については、“K.2.2 実行の方法(自動並列化)”の“スレッド数”をお読みください。

スレッド数

並列に動作するスレッドの数は、以下の優先順位に従って決まります。

1. 環境変数PARALLELの指定値
2. 環境変数OMP_NUM_THREADSの指定値
3. 環境変数FLIB_HPCFUNC=TRUEを指定している場合はシステムで利用できるCPU数、環境変数FLIB_HPCFUNC=TRUEを指定していない場合は1スレッド

環境変数PARALLELの指定値

リンク時に-Kopenmpオプションが指定され、かつ環境変数FLIB_FASTOMPにTRUEが設定されている場合、環境変数OMP_NUM_THREADSの指定値があれば、その値と環境変数PARALLELの指定値は一致していなければなりません。一致していない場合には、スレッド数には小さい方の値が採用されます。この時にjwe1042i-wのメッセージが出力されます。

上記の優先順位で決まったスレッド数と、利用可能なCPU数の上限値を比較して、小さい方を最終的なスレッド数として採用します。

環境変数OMP_NUM_THREADSについては、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“OpenMP仕様の環境変数”をお読みください。

環境変数THREAD_STACK_SIZEおよびOMP_STACKSIZE

THREAD_STACK_SIZE

環境変数THREAD_STACK_SIZEで、スレッドごとのスタック領域の大きさをKバイト単位で指定することができます。指定できる値は、1～18446744073709551615の整数値です。

環境変数OMP_STACKSIZEが指定されている場合、大きい方の指定値がスレッドごとのスタック領域の大きさの値になります。スタック領域の詳細については、“K.2.2 実行の方法(自動並列化)”の“実行時の領域”をお読みください。

OMP_STACKSIZE

環境変数OMP_STACKSIZEで、スレッドごとのスタック領域の大きさをバイト、Kバイト、Mバイト、Gバイト単位で指定することができます。

環境変数THREAD_STACK_SIZEが指定されている場合、大きい方の指定値がスレッドごとのスタック領域の大きさの値になります。スタック領域の詳細については、“K.2.2 実行の方法(自動並列化)”の“実行時の領域”をお読みください。

実行時の領域

自動並列化されたループ内でローカルな変数は、スタック領域に割り付けられます。これらの変数が多い場合には、スタック領域を十分な大きさに拡張する必要があります。

スレッドごとのスタック領域を特定の大きさで確保したい場合には、環境変数THREAD_STACK_SIZEまたはOMP_STACKSIZEで指定してください。

環境変数の指定がない場合、スレッドごとのスタック領域は、プロセスのスタック領域と同じ大きさで確保されます。

ただし、「プロセスのスタック領域の制限値」が「使用可能な実装メモリと仮想メモリの大きさの小さい方をスレッド数で割った値」より大きい場合、スレッドごとのスタック領域は下記の仮定値で確保されます。

```
仮定値(byte) = ( min(使用可能な実装メモリの大きさ, 仮想メモリ大きさ) / スレッド数 ) / 5
```

なお、この仮定値は動作を保証するものではありません。プロセススタック領域の制限値に、適切な値を設定することをお勧めします。プロセスのスタック領域の制限値は、ulimit(bash組込みコマンド)などで設定が可能です。

仮想メモリの大きさとは、ulimit(bash組込みコマンド)などで仮想記憶の最大サイズとして表示される値です。

スレッド数については、“K.2.2 実行の方法(自動並列化)”の“スレッド数”をお読みください。

注意

一部のpthreadライブラリでは、fixed stackになっているためスレッドスタックサイズを上記のように変更できないことがあります。同一環境であってもスタティック版は、fixed stackであることが多いため、できるだけ共有版のpthreadライブラリを使用するようにしてください。

同期待ち処理

環境変数FLIB_SPINWAITまたはOMP_WAIT_POLICYで、同期待ち処理の動作を指定することができます。

FLIB_SPINWAITの値:

unlimited	同期が獲得できるまでスピン待ちを行います。 デフォルトです。
0	スピン待ちを行わず、サスペンド待ちを行います。
*n*s	*n*秒間スピン待ちし、その後サスペンド待ちに移ります。 *n*は0以上の整数です。*n*に続けて、単位の"s"を指定します。
*n*ms	*n*ミリ秒間スピン待ちし、その後サスペンド待ちに移ります。 *n*は0以上の整数です。*n*に続けて、単位の"ms"を指定します。

スピン待ちとは、スレッド間で同期を待ち合うとき、CPU時間を消費する方法で待たせる方式です。逆に、サスペンド待ちはCPU時間を消費せずに待たせます。スピン待ちを行う方が並列処理のオーバヘッドが少ないため、経過時間を重視する場合には、unlimitedを選択してください。CPU時間を重視する場合は、0を選択してください。

OMP_WAIT_POLICYの値:

ACTIVE	同期が獲得できるまでスピン待ちを行います。 デフォルトです。
PASSIVE	スピン待ちを行わず、サスペンド待ちを行います。

経過時間を重視する場合には、ACTIVEを選択してください。CPU時間を重視する場合は、PASSIVEを選択してください。

環境変数FLIB_SPINWAITが指定されている場合は、環境変数FLIB_SPINWAITの設定を有効とします。

fork(2)系機能の制限

fork(2)系機能を使用した場合、動作は保証されません。

スレッドのCPUバインド

環境変数FLIB_HPCFUNC=TRUEを設定していない場合、環境変数FLIB_CPU_AFFINITYを使用することによりスレッドの固定のCPUへのバインドを制御することができます。環境変数FLIB_HPCFUNCについては、“K.4 高速化機能の利用”の“環境変数FLIB_HPCFUNC”をお読みください。

環境変数FLIB_CPU_AFFINITY

スレッドを指定されたcpuid順にCPUバインドします。

スレッド数が、指定したcpuidの数を超える場合は、先頭から繰り返して使用します。

各cpuidは、コンマ(',')または空白文字(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

```
cpuid1[-cpuid2[:inc]]
```

cpuid1:範囲指定の最初のcpuid(0 <= *cpuid1* < CPU_SETSIZE)
cpuid2:範囲指定の最後のcpuid(0 <= *cpuid2* < CPU_SETSIZE)
inc:増分値(1 <= *inc* < CPU_SETSIZE)

なお、以下の条件を満たす必要があります。

```
cpuid1 <= cpuid2
```

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべて指定した場合と等価になります。

cpuidは、上記のように指定できますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)をお読みください。

注意

cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーが出力されますので、設定値を見直してください。実行開始時のプロセスのCPU affinityについては、tasksetやnumactlなどのシステムのコマンドで、確認することができます。システムのコマンドについては、各manマニュアルを参照してください。

例

例1:

```
$ export FLIB_CPU_AFFINITY="12,14,13,15"
```

12,14,13,15の順でスレッドにバインドします。
スレッド数5以上の場合は、先頭から繰り返して使用します。

例2:

```
$ export FLIB_CPU_AFFINITY="12-19"
```

12,13,14,15,16,17,18,19の順でスレッドにバインドします。
スレッド数9以上の場合は、先頭から繰り返して使用します。

例3:

```
$ export FLIB_CPU_AFFINITY="12-19:2"
```

12,14,16,18の順でスレッドにバインドします。
スレッドが5以上の場合は、先頭から繰り返して使用します。

例4:

```
$ export FLIB_CPU_AFFINITY="12-16:2,13,19"
```

12,14,16,13,19の順でスレッドにバインドします。
スレッドが6以上の場合は、先頭から繰り返して使用します。

K.2.3 翻訳・実行の例

例1:

```
$ fccpx -Kparallel,reduction,ocl -Nfjomplib test1.c
$ ./a.out
```

リダクションの最適化と最適化制御行を有効にして、ソースプログラムを翻訳します。このプログラムが実行時に使用するスレッド数は、実行時環境によって決まります。スレッド数の詳細については、“K.2.2 実行の方法(自動並列化)”の“スレッド数”をお読みください。

例2:(shの場合)

```
$ fccpx -Kparallel -Nfjomplib test2.c
$ PARALLEL=2
$ export PARALLEL
$ ./a.out
$ PARALLEL=4
$ export PARALLEL
$ ./a.out
```

環境変数PARALLELに値2を設定して、2個のスレッドを使用して動作させます。

次に、環境変数PARALLELに値4を設定して、4個のスレッドを使用して動作させます。

K.2.4 並列化プログラムのチューニング

“4.2.4 並列化プログラムのチューニング”をお読みください。

K.2.5 自動並列化

“4.2.5 自動並列化機能の詳細”をお読みください。

K.2.6 最適化制御行

“4.2.6 最適化制御行”をお読みください。

K.2.7 自動並列化機能を使うときの留意事項

“4.2.7 自動並列化機能を使うときの留意事項”をお読みください。

K.2.8 他のマルチスレッドプログラムとの結合

自動並列化プログラム以外の、マルチスレッドプログラムのオブジェクトプログラムとの結合について、制限事項を示します。

本処理系のOpenMPプログラム

本処理系で-Kopenmpオプションを指定し作成したOpenMPプログラムのオブジェクトプログラムについては、本処理系で-Kparallelオプションを指定し作成したオブジェクトプログラムと結合可能です。

他のマルチスレッドプログラム

本処理系で作成した以外の、他のマルチスレッドプログラムのオブジェクトプログラムとは、結合できません。

ただし、環境変数FLIB_PTHREADを使用する場合、pthreadプログラムを結合することができます。

環境変数FLIB_PTHREAD

環境変数FLIB_PTHREADで、pthreadプログラムとの結合を制御できます。指定できる値は、以下のとおりです。省略値は、0になります。

値	説明
0 (省略値)	pthreadスレッドを制御スレッドとしてのみ使用するpthreadプログラムを結合できます。 ただし以下の制限があります。 ・ pthreadスレッドはサスペンド待機する必要があります。 ・ pthreadプログラムは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳してはなりません。(注) ・ pthreadプログラムでは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳したルーチンを使用してはなりません。(注)

値	説明
	・ pthreadプログラムでは、Fortranルーチンを使用してはなりません。(注)
1	pthreadスレッドで並列処理を行うpthreadプログラムを結合できます。 pthreadスレッドを制御スレッドとして使用するpthreadプログラムとの結合もできます。 ただし以下の制限があります。 ・ pthread並列処理とOpenMP/自動並列処理を順番に実行する必要があります。(注) pthread並列処理からOpenMP/自動並列処理を実行してはなりません。 また、OpenMP/自動並列処理からpthread並列処理を実行してはなりません。 ・ pthreadスレッドはサスペンド待機する必要があります。 ・ pthreadプログラムは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳してはなりません。(注) ・ pthreadプログラムでは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳したルーチンを使用してはなりません。(注) ・ pthreadプログラムでは、Fortranルーチンを使用してはなりません。(注) ・ pthreadプログラムでは、スレッド並列版の数学ライブラリを使用してはなりません。(pthreadプログラムを、-SSL2BLAMPオプションを指定して翻訳してはなりません)(注) ・ pthreadスレッドの生成前にチームスレッド数2以上のOpenMP並列処理を実行する必要があります。また、このスレッド数をあとで変更できません。 また本機能を使用する場合、以下の機能が有効になります。 ・ FLIB_SPINWAIT=0 ・ FLIB_CPUBIND=off 上記の環境変数に異なる値を指定した場合、動作は保証されません。 環境変数FLIB_SPINWAITについては、“K.2.2 実行の方法(自動並列化)”の“同期待ち処理”を参照してください。 環境変数FLIB_CPUBINDについては、“K.4.1 スレッドのCPUへのバインド”を参照してください。

注) 制限機能を使用した場合、動作は保証されません。

K.3 OpenMP仕様による並列化

ここでは、本処理系で提供しているOpenMP仕様による並列化について説明します。OpenMP仕様については、“OpenMP Architecture Review Board”のウェブサイトをお読みください。

本処理系では、以下の仕様をサポートしています。

サポートする仕様
OpenMP 3.1
OpenMP 4.0の一部 (注)
OpenMP 4.5の一部 (注)

注) 以下の機能を使用できます。

・ simd構文

・ declare simd構文

K.3.1 翻訳の方法(OpenMP仕様による並列化)

ソースプログラムの翻訳およびリンクを行うには、翻訳コマンドに以下のオプションを指定します。

指定するオプションは、OpenMP仕様による並列化を行うか否かで異なります。

OpenMP仕様による並列化	指定するオプション	
	翻訳時	リンク時
OpenMP仕様による並列化およびSIMD化を行う場合	-Kopenmp	-Kopenmp -Nfjomplib
OpenMP仕様によるSIMD化のみを行い、並列化を行わない場合	-Kopenmp_simd	-

-: 指定するオプションはありません。

OpenMPプログラムのための翻訳時オプション

ここでは、OpenMPプログラムを翻訳するための翻訳時オプションについて説明します。

-Kopenmp

-Kopenmpオプションは、OpenMP指示文を有効にし、ソースプログラムを翻訳します。

-Kopenmpオプションは、リンク時にも指定する必要があります。-Kopenmpオプションを指定して翻訳されたオブジェクトプログラムが含まれる場合は、リンク時に-Kopenmpオプションを指定する必要があります。

例

```
$ fccpx a.c -Kopenmp -c
$ fccpx a.o -Kopenmp -Nfjomplib
```

-Kopenmp_simd

-Kopenmp_simdオプションは、OpenMP仕様のsimd構文およびdeclare simd構文のみを有効にし、ソースプログラムを翻訳します。

-Nfjomplib

OpenMPライブラリとして、富士通OpenMPライブラリを使用することを指示します。

-Nfjomplibオプションは、リンク時に指定する必要があります。

OpenMPプログラムの最適化情報を出力するための翻訳時オプション

“4.3.1.2 OpenMPプログラムの最適化情報を出力するための翻訳時オプション”をお読みください。

OpenMPプログラムの制限事項

“4.3.1.3 OpenMPプログラムの制限事項”をお読みください。

K.3.2 実行の方法(OpenMP仕様による並列化)

OpenMPプログラムを実行させる手続きは、逐次実行のときと同じです。

実行時の環境変数

以下の環境変数を使用できます。

環境変数FLIB_FASTOMP

環境変数FLIB_FASTOMPで、高速実行時ライブラリの使用を制御することができます。指定できる値とその意味は、以下のとおりです。デフォルトはTRUEです。

- TRUE

  高速実行時ライブラリを使用します。

ー FALSE

高速実行時ライブラリを使用しません。

高速実行時ライブラリは、OpenMP仕様の一部を制限することにより、実行時の高速化を実現しています。高速実行時ライブラリには、以下のような特徴があります。

ー ネストした並列化がない、および、スレッドがCPUに1対1に対応付けられていることを前提とすることにより、高速化を図っています。

ー 実行時環境変数FLIB_HPCFUNC=TRUEを指定した場合、高速なコア間ハードウェアバリアを使用することができます。環境変数FLIB_HPCFUNCについては“K.4 高速化機能の利用”の“環境変数FLIB_HPCFUNC”をお読みください。

ただし、高速実行時ライブラリを使用できるプログラムには、以下の制限があります。

ー num_threads指示節およびomp_set_num_threads関数で指定するスレッド数は、実行時環境によって決まるスレッド数と一致していなければなりません。

実行時環境によって決まるスレッド数については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

高速実行時ライブラリを使用するとき、OpenMP仕様の環境変数とOpenMP仕様による実行の制御は、以下のように制限を受けます。

ー 環境変数OMP_NESTEDとomp_set_nested関数による設定に関わらず、ネストしたparallelリージョンは常に逐次化されます。

ー 環境変数OMP_DYNAMICとomp_set_dynamic関数による設定に関わらず、スレッド数は使用可能なCPU数を超えることはできません。

使用可能なCPU数については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

環境変数FLIB_SPINWAITおよびOMP_WAIT_POLICY

環境変数FLIB_SPINWAITおよびOMP_WAIT_POLICYで、同期待ち処理の動作を指定することができます。

FLIB_SPINWAITの値:

unlimited	同期が獲得できるまでスピン待ちを行います。 デフォルトです。
0	スピン待ちを行わず、サスペンド待ちを行います。
*n*s	*n*秒間スピン待ちし、その後サスペンド待ちに移ります。 *n*は0以上の整数です。*n*に続けて、単位の"s"を指定します。
*n*ms	*n*ミリ秒間スピン待ちし、その後サスペンド待ちに移ります。 *n*は0以上の整数です。*n*に続けて、単位の"ms"を指定します。

スピン待ちとは、スレッド間で同期を待ち合うとき、CPU時間を消費する方法で待たせる方式です。逆に、サスペンド待ちはCPU時間を消費せずに待たせます。スピン待ちを行う方が並列処理のオーバヘッドが少ないため、経過時間を重視する場合には、unlimitedを選択してください。CPU時間を重視する場合は、0を選択してください。

OMP_WAIT_POLICYの値:

ACTIVE	同期が獲得できるまでスピン待ちを行います。 デフォルトです。
PASSIVE	スピン待ちを行わず、サスペンド待ちを行います。

経過時間を重視する場合には、ACTIVEを選択してください。CPU時間を重視する場合は、PASSIVEを選択してください。

環境変数FLIB_SPINWAITが指定されている場合は、環境変数FLIB_SPINWAITの設定を有効とします。

環境変数OMP_PROC_BIND

スレッドのCPUへのバインドを制御することができます。指定できる値とその意味は、以下のとおりです。

値	説明
TRUE	スレッドは、実行開始時のプロセスのCPU affinityの範囲内のcpuid順にCPUバインドされます。

値	説明
FALSE	スレッドはCPUにバインドされません。

環境変数FLIB_CPUBINDまたはFLIB_CPU_AFFINITYの指定は、環境変数OMP_PROC_BINDよりも優先されます。

ただし、以下の場合には環境変数OMP_PROC_BINDの指定が有効になります。

－ 環境変数FLIB_CPUBINDがoff、かつ、環境変数FLIB_CPU_AFFINITYの指定なし

環境変数FLIB_CPUBINDについては、“K.4.1 スレッドのCPUへのバインド”の“FLIB_CPUBIND”をお読みください。

環境変数FLIB_CPU_AFFINITYについては、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

環境変数THREAD_STACK_SIZE

スレッドごとのスタック領域の大きさをKバイト単位で指定することができます。環境変数OMP_STACKSIZEが指定されている場合、大きい方の指定値がスレッドごとのスタック領域の大きさの値になります。詳細については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

環境変数OMP_STACKSIZE

スレッドごとのスタック領域の大きさをバイト、Kバイト、Mバイト、Gバイト単位で指定することができます。

環境変数THREAD_STACK_SIZEが指定されている場合、大きい方の指定値がスレッドごとのスタック領域の大きさの値になります。詳細については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

OpenMP仕様の環境変数

以下のOpenMP仕様の環境変数を使用できます。OpenMP仕様の環境変数の詳細については、OpenMP仕様書をお読みください。

環境変数OMP_SCHEDULE

schedule指示節にruntimeを持つ、for指示文またはparallel for指示文に対して、実行すべきスケジュールタイプとチャンクサイズを指定します。

環境変数OMP_NUM_THREADS

実行中に使用するスレッドの数を指定します。

環境変数OMP_DYNAMIC

動的スレッド調整機能の有効または無効を指定します。

環境変数OMP_PROC_BIND

スレッドをCPUにバインドするかどうかを指定します。

環境変数OMP_NESTED

parallelリージョンのネスト機能の有効または無効を指定します。

環境変数OMP_STACKSIZE

スレッドごとのスタック領域の大きさを指定します。

環境変数OMP_WAIT_POLICY

同期待ち処理の動作を指定します。

環境変数OMP_MAX_ACTIVE_LEVELS

ネストしている活動状態のparallelリージョンの最大数を指定します。

環境変数OMP_THREAD_LIMIT

OpenMPプログラムで実行するスレッド数の最大数を指定します。

実行時の注意事項

本処理系のOpenMP仕様による並列化機能を使用する場合の注意事項を、以下に示します。

実行時の変数割付け

本処理系のOpenMP仕様による並列化機能を使用したプログラムでは、関数内でローカルな変数およびprivate変数はスタック領域に割り付けられます。これらの変数が多い場合には、スタック領域を十分な大きさに拡張する必要があります。

スレッドごとのスタック領域を特定の大きさで確保したい場合には、環境変数THREAD_STACK_SIZEまたはOMP_STACKSIZEで指定してください。

環境変数の指定がない場合、スレッドごとのスタック領域は、プロセスのスタック領域と同じ大きさで確保されます。ただし、「プロセスのスタック領域の制限値」が「使用可能な実装メモリと仮想メモリの大きさの小さい方をスレッド数で割った値」より大きい場合、スレッドごとのスタック領域は下記の仮定値で確保されます。

```
仮定値(byte) = ( min(使用可能な実装メモリの大きさ, 仮想メモリ大きさ) / スレッド数 ) / 5
```

なお、この仮定値は、動作を保証するものではありません。プロセススタック領域の制限値に、適切な値を設定することをお勧めします。プロセスのスタック領域の制限値は、ulimit(bash組込みコマンド)などで設定が可能です。

仮想メモリの大きさとは、ulimit(bash組み込みコマンド)などで仮想記憶の最大サイズとして表示される値です。

スレッドごとのスタック領域の大きさを一意にするために動的にスレッド数を変更できる機能を考慮しません。したがって、この式では“FLIB_FASTOMP=TRUE”相当のスレッド数を使用します。

動的にスレッド数を変更する場合は、あらかじめ環境変数OMP_NUM_THREADSに最大スレッド数を設定しておくことをお勧めします。

環境変数THREAD_STACK_SIZEおよびOMP_STACKSIZEについては、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“OpenMP仕様の環境変数”をお読みください。

CPU数の上限

CPU数の上限値は、システムのCPU数です。

スレッド数

並列実行のスレッド数は、高速実行時ライブラリを使用するときOpenMPと自動並列化で共通となり、そうでないときそれぞれ独立に決定されます。高速実行時ライブラリの使用は、環境変数FLIB_FASTOMPの設定によって制御することができます。詳細については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“OpenMP仕様の環境変数”をお読みください。

高速実行時ライブラリを使用するとき(FLIB_FASTOMPがTRUEのとき)

スレッド数は、以下の優先順位に従って決まります。

1. 環境変数OMP_NUM_THREADSの指定値
2. 環境変数PARALLELの指定値
3. 環境変数FLIB_HPCFUNC=TRUEを指定している場合はシステムで利用できるCPU数、環境変数FLIB_HPCFUNC=TRUEを指定していない場合は1スレッド

この優先順位で決まるスレッド数がCPU数の上限値を超えるとき、スレッド数はCPU数の上限値となります。

num_threads指示節およびomp_set_num_threads関数によって指定する値は、ここで決まるスレッド数と一致していなければなりません。異なる値を指定した場合には、プログラムの実行が終了されます。この時にjwe1041i-sのメッセージが出力されます。

高速実行時ライブラリを使用しないとき(FLIB_FASTOMPがFALSEのとき)

OpenMPのスレッド数は、以下の優先順位に従って決まります。自動並列化のスレッド数は、“K.2 自動並列化”の記述に従います。

1. parallel指示文のnum_threads指示節の指定値
2. omp_set_num_threads関数の指定値
3. 環境変数OMP_NUM_THREADSの指定値
4. 環境変数PARALLELの指定値
5. 環境変数FLIB_HPCFUNC=TRUEを指定している場合はシステムで利用できるCPU数、環境変数FLIB_HPCFUNC=TRUEを指定していない場合は1スレッド

動的スレッド調整機能が有効な場合、上記の優先順位で決まったスレッド数と、利用可能なCPU数の上限値を比較して、小さい方をスレッド数として採用します。

動的スレッド調整機能が無効な場合、上記の優先順位で決まったスレッド数となり、CPU数の上限を超えることができます。1CPU当たりのスレッド数が1を超える場合には、並列に実行されることを意図したスレッドが時分割で実行されることになります。このような場合、スレッド間の同期処理のオーバヘッドが大きくなり、実行性能が低下することがあります。システムによる負荷も考慮して、1CPU当たりのスレッド数が1以下になるようにスレッド数を設定することをお勧めします。

fork(2)系機能の制限

fork(2)系機能を使用した場合、動作は保証されません。

スレッドのCPUバインド

環境変数FLIB_HPCFUNC=TRUEを設定していない場合、環境変数FLIB_CPU_AFFINITYを使用することによりスレッドの固定のCPUへのバインドを制御することができます。環境変数FLIB_HPCFUNCについては、“K.4 高速化機能の利用”の“環境変数FLIB_HPCFUNC”をお読みください。

環境変数FLIB_CPU_AFFINITY

スレッドを指定されたcpuid順にCPUバインドします。

スレッド数が、指定したcpuidの数を超える場合は、先頭から繰り返して使用します。

各cpuidは、コンマ(',')または空白文字(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

```
cpuid1[-cpuid2[:inc]]
```

cpuid1:範囲指定の最初のcpuid(0 <= *cpuid1* < CPU_SETSIZE)
cpuid2:範囲指定の最後のcpuid(0 <= *cpuid2* < CPU_SETSIZE)
inc:増分値(1 <= *inc* < CPU_SETSIZE)

なお、以下の条件を満たす必要があります。

```
cpuid1 <= cpuid2
```

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべて指定した場合と等価になります。

cpuidは、上記のように指定できますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)をお読みください。

注意

cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーが出力されますので、設定値を見直してください。実行開始時のプロセスのCPU affinityについては、tasksetやnumactlなどのシステムのコマンドで、確認することができます。システムのコマンドについては、各manマニュアルを参照してください。

例

例1:

```
$ export FLIB_CPU_AFFINITY="12,14,13,15"
```

12,14,13,15の順でスレッドにバインドします。
スレッド数5以上の場合は、先頭から繰り返して使用します。

例2:

```
$ export FLIB_CPU_AFFINITY="12-19"
```

12,13,14,15,16,17,18,19の順でスレッドにバインドします。
スレッド数9以上の場合は、先頭から繰り返して使用します。

例3:

```
$ export FLIB_CPU_AFFINITY="12-19:2"
```

12,14,16,18の順でスレッドにバインドします。
スレッドが5以上の場合は、先頭から繰り返して使用します。

例4:

```
$ export FLIB_CPU_AFFINITY="12-16:2,13,19"
```

12,14,16,13,19の順でスレッドにバインドします。
スレッドが6以上の場合は、先頭から繰り返して使用します。

複数のスレッドからの出力情報

プログラムの実行時に同一プログラム内で複数のスレッドからエラーが検出された場合、トレースバックマップは、スレッド単位に出力されます。したがって、parallelリージョン内では、スレッド数分の情報が出力されます。

K.3.3 処理系依存の仕様

OpenMP仕様において、処理系の実現に任されている仕様については、本処理系では以下のように実現しています。各項目の詳細については、OpenMP仕様書をお読みください。

メモリモデル

4バイトを超えるか4バイト境界を跨ぐ変数に対して、

- 2つのスレッドからの書き込みが同時に起こるとき

  明示的な排他制御が行われなければ、変数の値はどちらの値にもならず不定となることがあります。

- 2つのスレッドから書き込みと読み出しが同時に起こるとき

  明示的な排他制御が行われなければ、読み出される変数の値は書き込み前の値でも書き込み後の値でもなく不定となることがあります。

内部制御変数

各内部制御変数の初期値は以下のとおりです。

内部制御変数	初期値
nthreads-var	1
dyn-var	TRUE
run-sched-var	チャンクサイズなしのstatic
def-sched-var	チャンクサイズなしのstatic
bind-var	FALSE
stacksize-var	“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”のスレッドごとのスタック領域の大きさ
wait-policy-var	ACTIVE
thread-limit-var	2147483647
max-active-levels-var	2147483647

動的スレッド調整機能

本処理系のOpenMP仕様による並列化機能では、動的スレッド調整機能を実現しています。この機能の効果については、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。

デフォルトの状態では、動的スレッド調整機能は有効です。

ループ指示文

一重化したループの繰返し数の計算に使用する変数の型は、long int型です。

内部制御変数run-sched-varにautoが設定されたときのschedule(runtime)指示節の効果は、schedule(static)となります。

sections構文

sections構文内の構造化ブロックのスレッドへの割当ては、dynamicスケジュールと同じ方法で行われます。スレッドは、sectionsリージョンに到達するか、その中の1つの構造化ブロックの実行を終了したとき、その順序で次の構造化ブロックと対応付けられます。

single構文

singleリージョンは、最初にそのリージョンに到達したスレッドによって実行されます。

simd構文

一重化したループの繰返し数の計算に使用する変数の型は、long int型です。

SIMD長はコンパイラが自動的に決定する値になります。

aligned指示節に*alignment*パラメタが指定されていないときは、*alignment*パラメタに次の値が指定されたものとみなされます。

翻訳時に-KSVEオプションが有効な場合	リストアイテムの型のアライメント
翻訳時に-KNOSVEオプションが有効な場合	16

declare simd構文

simdlen指示節が指定されていないときのSIMD長は、次のようになります。

- 翻訳時に-KSVEオプションが有効な場合

  SIMD長は、実行時に決定されます。

- 翻訳時に-KNOSVEオプションが有効な場合

  SIMD長は、ベクトル化対象の引数と戻り値の中で最も小さい型の大きさで決定されます。型の大きさにより、SIMD長は以下のようになります。

型の大きさ (byte)	SIMD長
1	16と8
2	8と4
4	4と2
8	2
16	

aligned指示節に*alignment*パラメタが指定されていないときは、*alignment*パラメタに次の値が指定されたものとみなされます。

翻訳時に-KSVEオプションが有効な場合	リストアイテムの型のアライメント
翻訳時に-KNOSVEオプションが有効な場合	16

タスクスケジューリングポイント

アンタイドtaskリージョンのタスクスケジューリングポイントはタイドtaskリージョンと同じ位置にあります。すなわち、task構文、taskwait構文、暗黙/明示的なbarrier構文、およびタスクの完了ポイントにあります。

atomic構文

2つのatomicリージョンは、更新する変数の型が異なっていれば、独立に(排他的でなく)実行されます。型が一致しているとき、以下の場合にはアドレスが異なっていても排他的に実行される場合があります。

- 論理型、複素数型、1バイト整数型、2バイト整数型、または4倍精度(16バイト)実数型の変数の更新
- 配列要素の更新であり、添字式が字面上一致していないとき
- 対象の式が明示的または暗黙の型変換を伴うとき

omp_set_num_threads関数

omp_set_num_threads関数の呼び出しは、引数が0以下の値のとき、何の効果もありません。この値として、システムがサポートするスレッド数を超える数値を設定してはなりません。

omp_set_schedule関数

実装依存のスケジュールタイプはありません。

omp_set_max_active_levels関数

omp_set_max_active_levels関数が明示的なparallelリージョンから呼ばれた場合、呼び出しは無効になります。omp_set_max_active_levels関数の引数が0以上の整数ではない場合、呼び出しは無効になります。

omp_get_max_active_levels関数

omp_get_max_active_levels関数はプログラム内のどの場所からでも呼び出されることができ、内部制御変数max-active-levels-varの値を返します。

環境変数OMP_SCHEDULE

OMP_SCHEDULEに指定されたスケジュールタイプが有効なスケジュールタイプではない場合、指定は無効となり、デフォルト値(チャンクサイズなしのstatic)が適用されます。

OMP_SCHEDULEに指定されたスケジュールタイプがstatic、dynamic、またはguidedであり、指定されたチャンクサイズが正ではない場合、チャンクサイズは次のようになります。

スケジュールタイプ	チャンクサイズ
static	チャンクサイズなし
dynamic	1
guided	

環境変数OMP_NUM_THREADS

OMP_NUM_THREADSのリストに0以下の値を設定すると、1を設定した場合と同じ効果をもちます。この値として、システムがサポートするスレッド数を超える数値を設定してはなりません。

環境変数OMP_PROC_BIND

OMP_PROC_BINDにTRUEでもFALSEでもない値を設定すると、指定は無効となり、デフォルト値(FALSE)が適用されます。

環境変数OMP_DYNAMIC

OMP_DYNAMICにTRUEでもFALSEでもない値を設定すると、指定は無効となり、デフォルト値(TRUE)が適用されます。

環境変数OMP_NESTED

OMP_NESTEDにTRUEでもFALSEでもない値を設定すると、指定は無効となり、デフォルト値(FALSE)が適用されます。

環境変数OMP_STACKSIZE

OMP_STACKSIZEに設定された値が定義された形式を満たしていない場合、指定は無効となり、デフォルト値が適用されます。

環境変数OMP_WAIT_POLICY

ACTIVEの振舞いはスピン待ちです。PASSIVEの振舞いはサスペンド待ちです。

環境変数OMP_MAX_ACTIVE_LEVELS

OMP_MAX_ACTIVE_LEVELSに設定された値が0以上の整数ではない場合、指定は無効となり、デフォルト値(2147483647)が適用されます。

環境変数OMP_THREAD_LIMIT

OMP_THREAD_LIMITに設定された値が正の整数ではない場合、指定は無効となり、デフォルト値(2147483647)が適用されます。

K.3.4 プログラミングの注意事項

ここでは、OpenMP仕様のプログラミングにおける注意事項について説明します。

parallelリージョンおよび明示的なtaskリージョンの実現

parallel構文またはtask構文の構造化ブロックは、翻訳処理により、内部関数化されます。

内部関数の名前を以下に示します。

内部関数の種別	内部関数名
parallel構文から生成された内部関数	"._OMP_*識別番号A*"という名前が付加されます
task構文から生成された内部関数	"._TSK_*識別番号B*"という名前が付加されます

識別番号A: ソースファイル内でparallel構文を識別する一意の数字

識別番号B: ソースファイル内でtask構文を識別する一意の数字

threadprivate変数の実現

"4.3.4.2 threadprivate変数の実現"をお読みください。

OpenMPプログラムの自動並列化

"4.3.4.3 OpenMPプログラムの自動並列化"をお読みください。

K.3.5 他のマルチスレッドプログラムとの結合

OpenMPプログラム以外の、マルチスレッドプログラムのオブジェクトプログラムとの結合について説明します。

本処理系の自動並列化プログラム

本処理系で-Kparallelオプションを指定して作成した自動並列化プログラムのオブジェクトプログラムは、本処理系で-Kopenmpオプションを指定して作成したオブジェクトプログラムと結合可能です。

他のマルチスレッドプログラム

本処理系で作成した以外の、他のマルチスレッドプログラムのオブジェクトプログラムとは、結合できません。

ただし、環境変数FLIB_PTHREADを使用する場合、pthreadプログラムを結合することができます。

環境変数FLIB_PTHREAD

環境変数FLIB_PTHREADで、pthreadプログラムとの結合を制御できます。指定できる値は、以下のとおりです。省略値は、0になります。

値	説明
0 (省略値)	pthreadスレッドを制御スレッドとしてのみ使用するpthreadプログラムを結合できます。 ただし以下の制限があります。 ・ pthreadスレッドはサスペンド待機する必要があります。 ・ pthreadプログラムは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳してはなりません。(注) ・ pthreadプログラムでは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳したルーチンを使用してはなりません。(注) ・ pthreadプログラムでは、Fortranルーチンを使用してはなりません。(注)
1	pthreadスレッドで並列処理を行うpthreadプログラムを結合できます。 pthreadスレッドを制御スレッドとして使用するpthreadプログラムとの結合もできます。 ただし以下の制限があります。 ・ pthread並列処理とOpenMP/自動並列処理を順番に実行する必要があります。(注) pthread並列処理からOpenMP/自動並列処理を実行してはなりません。 また、OpenMP/自動並列処理からpthread並列処理を実行してはなりません。

値	説明
	・ pthreadスレッドはサスペンド待機する必要があります。 ・ pthreadプログラムは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳してはなりません。(注) ・ pthreadプログラムでは、-Kopenmpオプションまたは-Kparallelオプションを指定して翻訳したルーチンを使用してはなりません。(注) ・ pthreadプログラムでは、Fortranルーチンを使用してはなりません。(注) ・ pthreadプログラムでは、スレッド並列版の数学ライブラリを使用してはなりません。(pthreadプログラムを、-SSL2BLAMPオプションを指定して翻訳してはなりません)(注) ・ pthreadスレッドの生成前にチームスレッド数2以上のOpenMP並列処理を実行する必要があります。また、このスレッド数をあとで変更できません。 また本機能を使用する場合、以下の機能が有効になります。 ・ FLIB_SPINWAIT=0 ・ FLIB_CPUBIND=off 上記の環境変数に異なる値を指定した場合、動作は保証されません。 環境変数FLIB_SPINWAITについては、“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の環境変数”を参照してください。 環境変数FLIB_CPUBINDについては、“K.4.1 スレッドのCPUへのバインド”を参照してください。

注) 制限機能を使用した場合、動作は保証されません。

K.3.6 OpenMPプログラムのデバッグ

本処理系が用意しているデバッグ機能については、“第8章 プログラムのデバッグ”を参照してください。

OpenMPプログラムのデバッグには、以下の制限事項があります。

・ declare simd構文によって生成された関数は、デバッガを用いてデバッグできません。

K.4 高速化機能の利用

ここでは、FXシステム向けの高速化機能を使用するためのプログラムの翻訳および実行について説明します。

以下の環境変数を使用することにより、FXシステムのCPUで実装されている高速化機能(コア間ハードウェアバリアとセクタキャッシュ)を利用することができます。

環境変数FLIB_HPCFUNC

高速化機能(コア間ハードウェアバリア、セクタキャッシュ)を利用するかどうかを指示します。TRUEを指定した場合、高速化機能を利用することができます。FALSEを指定した場合、高速化機能は利用できません。省略値はFALSEです。

環境変数を指定しない場合、高速化機能は利用できません。

値	説明
TRUE	高速化機能(コア間ハードウェアバリア、セクタキャッシュ)の利用を可能にします。 NUMAノードを1プロセスで占有可能である場合に、利用することができます。 CPUの上限値は1ノードに割り当てられたCPU数です。
FALSE (省略値)	高速化機能(コア間ハードウェアバリア、セクタキャッシュ)は利用できません。 以下の環境変数を指定している場合、エラーメッセージ(jwe1047i-w)が出力されます。 ・ FLIB_SCCR_CNTL=TRUE

環境変数FLIB_HPCFUNC_INFO

実行開始時のプロセスのCPU affinityを表示するかどうかを指示します。省略値はFALSEです。

環境変数を指定しない場合、実行開始時のプロセスのCPU affinityは表示されません。

値	説明
TRUE	実行開始時のプロセスのCPU affinityを表示します。
FALSE (省略値)	実行開始時のプロセスのCPU affinityを表示しません。

K.4.1 スレッドのCPUへのバインド

環境変数FLIB_HPCFUNC=TRUEを指定した場合、自動並列化またはOpenMPを使用したプログラムを実行する時に、スレッドをCPUにバインドすることができます。

自動並列化については“K.2 自動並列化”を、OpenMP仕様による並列化については“K.3 OpenMP仕様による並列化”をお読みください。

FLIB_CPUBIND

環境変数FLIB_CPUBINDでスレッドのCPUへのバインドを制御できます。FLIB_CPUBINDには、以下のいずれかを指定します。指定がない場合は、chip_packと同じになります。

chip_pack	1つのCPUチップに集中するようにスレッドをバインドします。
chip_unpack	多くのCPUチップに分散するようにスレッドをバインドします。
off	スレッドをCPUへバインドしません。独自にバインドする場合などに使用します。

1CPUチップしか使用できない環境では、chip_packとchip_unpackは等価になります。

1CPUチップあたり1CPUしか使用できない環境では、chip_packとchip_unpackは等価になります。

環境変数FLIB_HPCFUNC=TRUEを指定していない場合、FLIB_CPUBINDの設定は無効になります。

K.4.2 コア間ハードウェアバリア

環境変数FLIB_HPCFUNC=TRUEを指定した場合、FXシステムのCPUで実装されているコア間ハードウェアバリアをスレッド間バリアとして使用することができます。コア間ハードウェアバリアとは、スレッド並列処理における同期処理を高速にし、実行性能を向上させるハードウェア機能です。

翻訳

コア間ハードウェアバリアをスレッド間バリアとして利用するプログラムを作成するためには、-Kparallelオプションまたは-Kopenmpオプションを指定します。

コア間ハードウェアバリアを使用できる環境でプログラムを実行すると、自動的にコア間ハードウェアバリアが使用されます。使用できない場合は、ソフトウェアバリアが使用されます。

実行

コア間ハードウェアバリアは、環境変数FLIB_HPCFUNC=TRUEを指定したときのみ利用することができます。コア間ハードウェアバリアは、コア間のスレッド間同期専用機能です。

FLIB_CNTL_BARRIER_ERR

コア間ハードウェアバリアを使用できない場合には、以下の診断メッセージを出力しソフトウェアバリアを使用して実行を続けます。

```
jwe1050i-w The hardware barrier couldn't be used and continues processing using the software barrier.
```

環境変数FLIB_CNTL_BARRIER_ERRを使用することにより診断メッセージjwe1050i-wのエラーを制御することができます。指定できる値とその意味は、以下のとおりです。デフォルトは、TRUEです。

- TRUE

  診断メッセージjwe1050i-wのエラーを検出します。

  コア間ハードウェアバリアを使用できない場合、診断メッセージjwe1050i-wのメッセージを出力し、ソフトウェアバリアを使用して実行を続けます。

- FALSE

  診断メッセージjwe1050i-wのエラーを検出しません。

  コア間ハードウェアバリアを使用できない場合は、ソフトウェアバリアを使用して実行を続けます。

FLIB_NOHARDBARRIER

環境変数FLIB_NOHARDBARRIERを設定した場合、コア間ハードウェアバリアを使用せず、ソフトウェアバリアを使用します。常に、診断メッセージjwe1050i-wを出力します。

詳細は、“K.4.2 コア間ハードウェアバリア”の“FLIB_CNTL_BARRIER_ERR”をお読みください。

注意

- -Kopenmpオプションを指定して作成したプログラムでコア間ハードウェアバリアを使用する場合には、環境変数FLIB_FASTOMPがFALSEであってはいけません。FLIB_FASTOMPの意味と使用上の注意については、“K.3 OpenMP仕様による並列化”をお読みください。
- コア間ハードウェアバリアを使用する場合、スレッドのCPUへのバインドが必須になります。したがって、環境変数FLIB_CPUBINDにoffを指定した場合は、コア間ハードウェアバリアは、使用できません。スレッドのCPUへのバインドについては、“K.4.1 スレッドのCPUへのバインド”をお読みください。
- スレッド数が1の場合、コア間ハードウェアバリアは使用できません。スレッド数の詳細については、“K.2.2 実行の方法(自動並列化)”の“スレッド数”または“K.3.2 実行の方法(OpenMP仕様による並列化)”の“実行時の注意事項”をお読みください。
- 1プロセスのCPU数が3以下の場合、コア間ハードウェアバリアを使用できないことがあります。1プロセスのCPU数を4以上に設定することをお勧めします。
- FXシステムのコア間ハードウェアバリアは同じNUMAノード内でのみ使用できます。したがって、プロセスが複数のNUMAノードに跨る場合は、NUMAノード内はコア間ハードウェアバリアが使用されますがNUMAノード間はソフトウェアバリアが使用されるためスレッド間のバリア性能は劣化します。スレッド間のバリア性能を優先する場合は、1プロセスが1NUMAノード内に収まる使い方を推奨します。
- 以下の場合、1つのプロセスがコア間ハードウェアバリア資源を占有できません。そのためハードウェアバリアの動作は不定になります。診断メッセージjwe1044i-uまたはjwe1045i-uを出力しプログラムを終了する、または、異常終了することがあります。

  - マルチプロセスプログラムの場合

    例:

    - Fortran FORK/SYSTEM/SHサービス関数などを使用する場合
    - C forkシステムコール/system関数などを使用する場合

例

例: コア間ハードウェアバリアを使用するプログラムを実行

```
$ cat exec.sh
#!/bin/sh
export FLIB_HPCFUNC=TRUE
./a.out
$ ./exec.sh
```

K.4.3 セクタキャッシュ

環境変数FLIB_HPCFUNC=TRUEを指定した場合、FXシステムのCPUで実装されているセクタキャッシュを制御することができます。

セクタキャッシュの制御方法については、“3.5 セクタキャッシュのソフトウェア制御”をお読みください。

ここでは、実行上の注意のみを説明します。

実行上の注意

1次キャッシュと2次キャッシュでセクタキャッシュを利用できる実行環境は以下の通りです。

実行環境	1次キャッシュでセクタキャッシュ利用	2次キャッシュでセクタキャッシュ利用
NUMAノードを1プロセスで占有できる	利用可	利用可
NUMAノードを1プロセスで占有できない	利用可 (注)	利用不可

注) 環境変数FLIB_L1_SCCR_CNTL=FALSEを指定した場合、利用しません。環境変数FLIB_L1_SCCR_CNTLについては、“3.5.2.2 環境変数および最適化制御行によるソフトウェア制御”を参照してください。

また以下の場合、NUMAノードを1プロセスで占有できる場合を識別できません。そのためセクタキャッシュの動作は不定になります。性能が大幅に劣化する、診断メッセージjwe1048i-uを出力しプログラムを終了する、またはプログラムが異常終了することがあります。

- マルチプロセスプログラムの場合

  例:

  ー Fortran FORK/SYSTEM/SHサービス関数などを使用する場合

  ー C forkシステムコール/system関数などを使用する場合

- スレッドを富士通の自動並列/OpenMP機能以外で生成する場合

  例:

  ー pthread関数などでスレッドを制御する場合

実行環境に関わらず無条件にセクタキャッシュ制御機能を無効化したい場合は、環境変数FLIB_SCCR_CNTLにFALSEを指定します。詳細は、“3.5.2.2 環境変数および最適化制御行によるソフトウェア制御”をお読みください。

例

例: セクタキャッシュを使用するプログラムを実行

```
$ cat exec.sh
#!/bin/sh
export FLIB_HPCFUNC=TRUE
./a.out
$ ./exec.sh
```

K.5 マルチプロセスによる実行

ここでは、マルチプロセスでプログラムを実行する方法および注意事項について説明します。

K.5.1 CPU affinityの指定

マルチプロセスでプログラムを実行する場合、環境変数FLIB_PROCESS_CPU_AFFINITYを使って、プロセスのCPU affinityを設定することができます。

プログラム開始時に有効になっているスレッドアフィニティポリシーに従い、システムは指定されたCPUにスレッドをバインドします。

環境変数FLIB_PROCESS_CPU_AFFINITY

プロセスに割り当てるcpuidを指定します。

各cpuidは、コンマ(',')または空白(' ')で区切る必要があります。

cpuidは、以下のような形式で増分値付き範囲指定ができます。

```
cpuid1[-cpuid2[:inc]]
```

cpuid1: 範囲指定の最初のcpuid (0<=*cpuid1*<CPU_SETSIZE)
cpuid2: 範囲指定の最後のcpuid (0<=*cpuid2*<CPU_SETSIZE)
inc: 増分値 (1<=*inc*<CPU_SETSIZE)

なお、以下の条件を満たす必要があります。

```
cpuid1<=cpuid2
```

*cpuid1*から*cpuid2*の範囲のcpuidを増分値*inc*ごとにすべてを指定した場合と等価になります。

cpuidは、上記のように指定はできますが、実際に使用できるcpuidは、実行開始時のプロセスのCPU affinityの範囲内のcpuidのみになります。

CPU_SETSIZEについては、CPU_SET(3)を参照してください。

注意

- cpuidが実行開始時のプロセスのCPU affinityの範囲外である場合、エラーを出力しプログラムが終了する、プログラムが終了しない、など、プログラムが正しく動作しないことがあります。設定値を見直してください。実行開始時のプロセスのCPU affinityについては、FLIB_HPCFUNC_INFO=TRUE、またはtasksetやnumactlなどのシステムのコマンドで、確認することができます。環境変数FLIB_HPCFUNC_INFOについては、“K.4 高速化機能の利用”を参照してください。システムのコマンドについては、各manマニュアルを参照してください。
- 環境変数FLIB_HPCFUNC=TRUEを指定していない場合、環境変数FLIB_PROCESS_CPU_AFFINITYは無効になります。環境変数FLIB_HPCFUNCについては、“K.4 高速化機能の利用”を参照してください。
- Fortran FORK、SYSTEM、SHサービス関数、C forkシステムコール、system関数などを使用してマルチプロセスでプログラムを実行する場合は、コア間ハードウェアバリアは使用できません。ソフトウェアバリアを使用してください。コア間ハードウェアバリアについては“K.4.2 コア間ハードウェアバリア”を参照してください。

例

環境変数FLIB_PROCESS_CPU_AFFINITYの指定例

- 例1:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0,1,2,3,4,5,6,7"
```

0,1,2,3,4,5,6,7をプロセスに割り当てます。

- 例2:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-7"
```

0,1,2,3,4,5,6,7をプロセスに割り当てます。

- 例3:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-7:2"
```

0,2,4,6をプロセスに割り当てます。

- 例4:

```
$ export FLIB_PROCESS_CPU_AFFINITY="0-5:2,3,8"
```

0,2,3,4,8をプロセスに割り当てます。

付録L　ラージページライブラリ

ここでは、HPC拡張機能のうち、ラージページライブラリについて説明します。

HPC分野における大規模なメモリを使用するアプリケーションプログラムでは、OSがメモリを管理するコストがアプリケーションプログラムの実行性能に影響を与えます。このコストを減らすために、HPC拡張機能では独自に拡張したラージページ機能を提供しています。

ラージページライブラリの理解に必要な基本的なメモリ割り当ての考え方についても説明します。

L.1　メモリ割り当て機能の概要

本節では、FXシステムの拡張機能としてのメモリ割り当て機能の概要として、以下の機能と処理を説明します。

- ラージページ機能
- ページング方式

L.1.1　ラージページ機能

ラージページ機能は、Linuxの標準機能であるHugeTLBfsを拡張し、大規模なデータを扱うアプリケーションプログラムに対して、通常のページ(ノーマルページ)より大きなページサイズのメモリ(ラージページ)を割り当てることで、OSのアドレス変換処理によるコストを低減し、メモリアクセス性能を向上させる機能です。

メモリアドレス変換とTLB

アプリケーションプログラムがメモリアクセスをする際には、仮想メモリアドレスを使います。そのため、仮想メモリアドレスから物理メモリアドレスへ変換する必要があります。このアドレス変換には、主記憶上に存在するアドレス変換テーブルを使用します。その際、さらに高速アクセスするため、CPU内に存在するロード・ストア機構がTLB(Translation Look-aside Buffer)というアドレス変換バッファを使用して物理アドレスを求めます。

CPUのロード・ストア機構は、動作中のアプリケーションプログラムからメモリへのロード・ストア要求を受け取ります。ロード・ストア機構によるメモリアドレス変換の概要を“図L.1 仮想メモリアドレスから物理メモリアドレスへの変換”に示します。

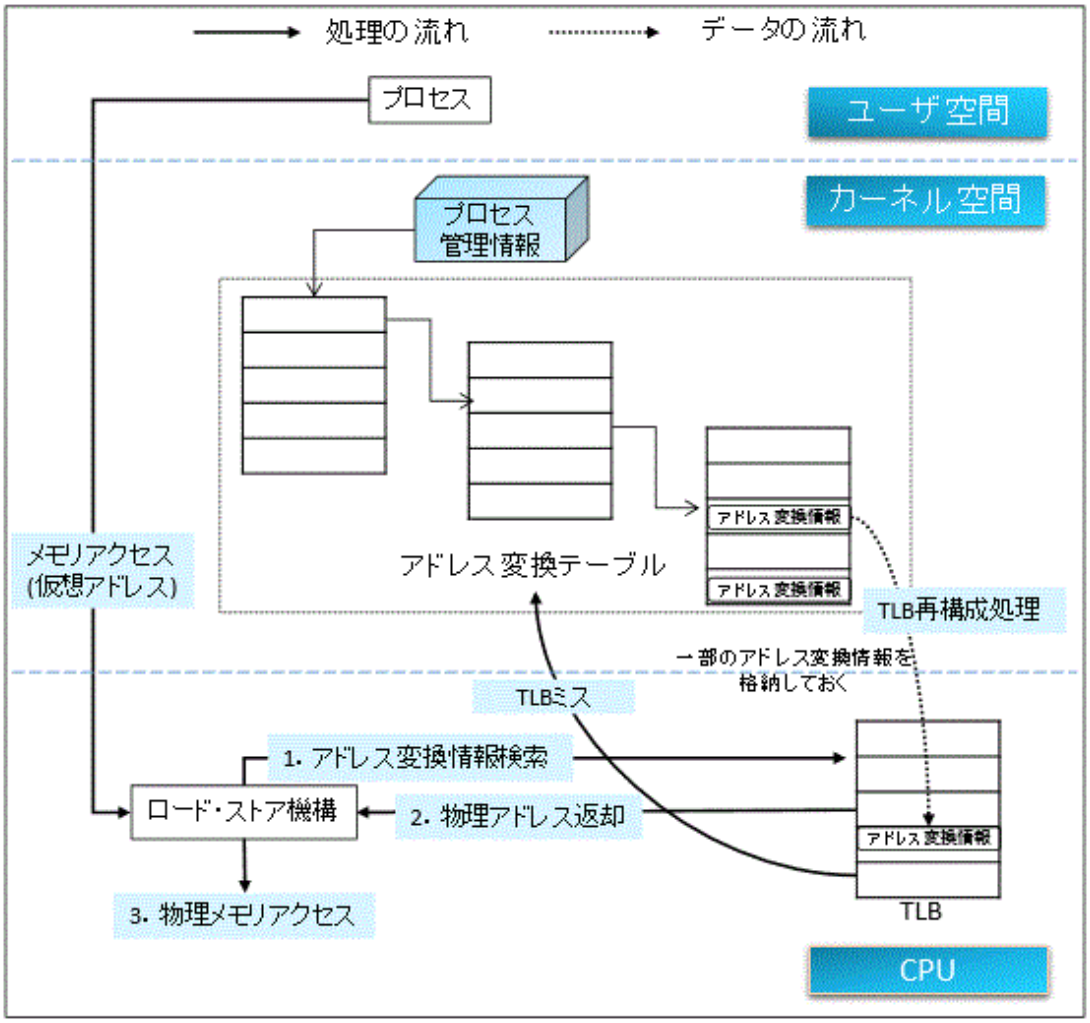
1. ロード・ストア先として指定された仮想メモリアドレスに対応する物理メモリアドレスをTLBから求めます。

   TLBに対応するアドレス変換情報がなければTLBミスを発生させ、アドレス変換テーブルからアドレス変換情報を求め、TLBへ読み込みます。

2. TLBから該当するアドレス変換情報により物理メモリアドレスを入手します。
3. ロード・ストア先に対して、物理メモリアクセスを開始します。

TLBから物理メモリアドレスが求められなかったときには、TLBミスが発生します。TLBミスが発生した場合にはTLB再構成処理(アドレス変換テーブルの再読み込み)が必要となり、一般的にその処理には大きなコスト(時間)がかかります。

図L.1 仮想メモリアドレスから物理メモリアドレスへの変換



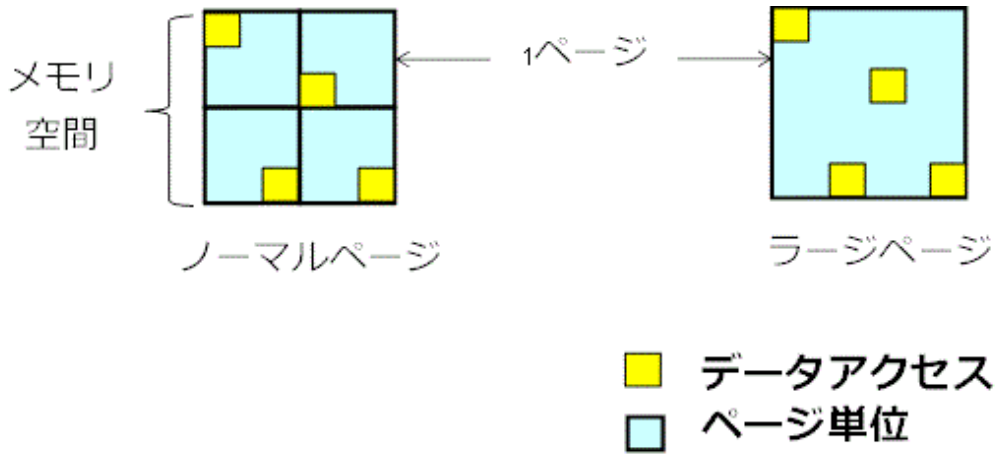
ノーマルページ・ラージページ

FXシステムのOSではページサイズとして、ノーマルページで64KiB、ラージページで2MiBを利用可能です。ラージページ機能ではデフォルトでラージページ化が有効となっています。“L.3 ラージページライブラリ設定用環境変数”の節に記載した環境変数(XOS_MMM_L_HPAGE_TYPE)によってラージページの有効化/無効化を選択できます。

大規模なメモリを扱うアプリケーションプログラムにおいては、ノーマルページではアクセスするページ数が多くなり、TLB再構成処理の頻度が高くなる傾向があります。ラージページを使用した場合、ノーマルページと比較して、アクセスするページが少なくなるため、TLB再構成処理の頻度は低くなります。

つまり、大量のメモリを使用するアプリケーションプログラムに対して、ラージページを割り当ててTLBミスを削減することで、OSのTLB再構成処理によるコストを低減し、メモリアクセス性能を向上させることができます。

図L.2 ノーマルページとラージページのデータアクセスイメージ



アプリケーションプログラムが初めてアクセスするメモリアドレスに対しては、アドレス変換情報がTLBに存在しないため、TLBミスが発生します。“図L.2 ノーマルページとラージページのデータアクセスイメージ”を例に取ると、ノーマルページの場合、あるメモリ空間に対し4つのページへのデータアクセスにより4回のTLBミスがするのに対して、ラージページの場合は、同じメモリ空間に対して同じページ内のデータアクセスとなるため、TLBミスが1回だけで済みます。

ページサイズの違いによるメリット・デメリット

ページサイズに 64KiB、2MiB を利用した際のそれぞれのメリット・デメリットを以下に示します。

表L.1 ページサイズによるメリット・デメリット

評価項目	64KiB	2MiB
TLBミス率	高い	低い
メモリ初期化コスト	小さい	大きい
メモリ使用効率	高い	低い

1. ページサイズ 64KiBは、ノーマルページです。TLBミス率は高いものの、メモリ初期化コストが小さく、メモリ使用効率が高いため、メモリ使用量が少ないアプリケーションプログラムに対しては有効な場合があります。
2. ページサイズ 2MiBは、ラージページです。64KiBよりメモリ初期化コストが大きくなり、メモリ使用効率が低くなりますが、TLBミス率が低くなります。そのため、FXシステムのラージページ機能では、ラージページ(2MiB)利用をデフォルトとしています。

アプリケーションプログラムは、典型的な通信で使用する通信バッファなどの小規模メモリから計算用の大規模メモリまで多彩なサイズのメモリを使用します。

アプリケーションプログラムのメモリ使用量が少ないときにラージページを利用すると、メモリ使用効率が悪くなる可能性があります。例えば、ヒープ領域のメモリ消費量が 1MiB のみの場合でも、ラージページを利用した場合、2MiB のメモリが確保されることになるため、利用されないメモリ領域が 1MiB 発生します。

逆に、アプリケーションプログラムのメモリ使用量が多いときにノーマルページを使用すると、TLB ミス率が高くなり、実行性能が向上しない可能性が高くなります。

L.1.2 ページング方式

ページング方式には、デマンドページング方式とプリページング方式の2種類があります。FXシステムのラージページ機能では、“L.3 ラージページライブラリ設定用環境変数”の節に記載した環境変数(XOS_MMM_L_PAGING_POLICY)によって、静的データ(.bss領域)、スタック/スレッドスタック領域、および動的メモリ確保領域の各メモリ領域にページング方式を設定できます。

1. デマンドページング方式とは、アプリケーションプログラム実行中に必要なページが主記憶に存在しない場合に、必要に応じてページを主記憶に割り当てる方式のことです。最初にメモリ領域にアクセスしたタイミングで物理ページを割り当てます。
2. プリページング方式とは、あらかじめ主記憶にページを割り当てておく方式のことです。メモリ領域を割り当てたタイミングで物理ページを割り当てます。

表L.2 ページング方式の違いによるメモリアクセスコスト

ページング方式	NUMA内/外のメモリアクセス	初回メモリアクセス
デマンドページング方式	できる限りNUMA内のメモリを使用 (コスト:低い)	メモリアクセス時にページを物理メモリにロード (コスト:高い)
プリページング方式	NUMA内外のメモリに関係なく使用 (コスト:高い)	ページをあらかじめ物理メモリにロード (コスト:低い)

“表L.2 ページング方式の違いによるメモリアクセスコスト”に示すように、メモリアクセス性能に影響する要因は2つあります。1つ目は、ある演算コアからアクセスする物理メモリが、同じNUMA内にあるか否かです。演算コアと同じNUMAに属する物理メモリへのアクセスよりも、異なる物理メモリへのアクセスは処理コストがかかります。2つ目は、初回メモリアクセス時のコストです。デマンドページング方式を選択した場合、メモリアクセス時に物理メモリを割り当てるため、初回メモリアクセス時にページロード処理コストがかかります。

例えば、4コアのスレッド並列アプリケーションプログラムで個々のスレッドで動的にメモリ領域を獲得する場合には、プリページング方式からデマンドページング方式にすると、演算コアと同じNUMA内の物理メモリアクセスの頻度が増え、メモリアクセス性能の向上が期待できます。

L.2 FXシステムのラージページ

FXシステムの拡張機能としてラージページライブラリ(libmpg.so)を提供します。本節では、FXシステムのラージページライブラリを利用する際の、各領域のページサイズおよびアプリケーションプログラムのデータの配置について説明します。

L.2.1 FXシステムのメモリ領域とラージページ化対象

FXシステムのラージページライブラリは、領域のうち、静的データ(.bss領域、.data領域)、動的メモリ確保領域(mmap領域)、スレッドヒープ領域、スタック領域、およびスレッドスタック領域をラージページ(2MiBページ)化します。テキスト領域、動的メモリ確保領域(ヒープ領域)、および共有メモリ領域についてはラージページ化の対象外です。

表L.3 FXシステムのメモリ領域とラージページ化対象

領域	ラージページ化対象	領域の使用用途
テキスト(.text)領域	×(64KiBページ)	アプリケーションプログラム(a.out)の命令列が配置されるメモリ領域
静的データ(.data)領域	○(2MiBページ)	アプリケーションプログラム(a.out)の静的データが格納されるメモリ領域(初期化有り)
静的データ(.bss)領域	○(2MiBページ)	アプリケーションプログラム(a.out)の静的データが格納されるメモリ領域(初期化無し)
動的メモリ確保領域(ヒープ領域)	×(64KiBページ)	プロセスヒープ領域/メインスレッド用ヒープ領域 ラージページライブラリでは、本メモリ領域はラージページ化されません。 メモリ領域は、以下の時に割り当てられます。 brk(2)/sbrk(2)システムコールで獲得するか、または、ラージページライブラリをリンクしない状態で、環境変数MALLOC_MMAP_THRESHOLD_で指定したサイズより小さいサイズの動的メモリ確保要求(malloc(3))を発行する時に割り当てられる。
スレッドヒープ領域	○(2MiBページ)	サブスレッド用のヒープ領域
スタック領域	○(2MiBページ)	プロセススタック領域/メインスレッド用スタック領域 (ラージページ化には環境変数XOS_MMM_L_LPG_MODEによる明示指定が必要)
スレッドスタック領域	○(2MiBページ)	サブスレッド用のスタック領域 (ラージページ化には環境変数XOS_MMM_L_LPG_MODEによる明示指定が必要)

領域	ラージページ化対象	領域の使用用途
動的メモリ確保領域(mmap領域)	○(2MiBページ)	mmap(2)を発行する時に割り当てられるメモリ領域 メモリ領域は、以下の時に割り当てられます。 ラージページライブラリをリンクして動的メモリ確保要求(malloc(3))を発行する時に割り当てられる。 または、ラージページライブラリをリンクしない状態で、環境変数MALLOC_MMAP_THRESHOLD_で指定したサイズより大きいサイズの動的メモリ確保要求(malloc(3))を発行する時に割り当てられる、または、スレッドヒープ領域/スレッドスタック領域としても割り当てられる。
共有メモリ	×(64KiBページ)	プロセス間でのメモリ共有のために使用

ラージページライブラリlibmpg.soはFXシステム向けに最適化されたラージページ機能で、HugeTLBfsのオーバーコミット機能を利用することで、事前のメモリプールを予約せず、ユーザI/Fの使い勝手を改善します。翻訳時に -Klargepage オプションを指定することで、アプリケーションプログラムが本ライブラリを利用するようになります。libmpg.soを使わずLinux標準のラージページ機能を利用する場合、翻訳時に -Knolargepage オプションを指定します。ノーマルページを利用する場合には、環境変数XOS_MMM_L_HPAGE_TYPE=noneを明示的に指定してください。

本項で説明した環境変数の詳細については“L.3 ラージページライブラリ設定用環境変数”の節をご覧ください。

注意

- ラージページライブラリのリンクの順序について

  ラージページライブラリの指定(コンパイラで指定する-lmpgオプション)は、エンドユーザが明示的に指定するライブラリ(例:-lc や -lpthread)より常に前に来るよう指定してください。この順序を守らない場合、正常にラージページが割り当てられない場合があります。

- brk(2)/sbrk(2)で獲得されたメモリ領域について

  カーネルの仕様によりbrk(2)/sbrk(2)で獲得されたメモリ領域はラージページ化されません。そのため、ラージページライブラリlibmpg.soをリンクした状態で、brk(2)/sbrk(2)とmalloc(3)(環境変数MALLOC_MMAP_THRESHOLD_より大きいサイズ要求時)を混在利用すると、ノーマルページとラージページが混在することになり、メモリ獲得の量とタイミングによっては、メモリが獲得できずに、brk(2)/sbrk(2)がENOMEMで終了することがあります。

  ラージページライブラリlibmpg.soをリンクする場合は、malloc(3)を利用することを推奨します。

  ラージページライブラリlibmpg.soをリンクしたアプリケーションプログラムでbrk(2)/sbrk(2)を利用したい場合は、環境変数XOS_MMM_L_HPAGE_TYPE=noneを明示的に指定してください。

- 頻繁にシグナルを発行するような処理を行うアプリケーションについて

  ラージページライブラリ(libmpg)をリンクし、プリページング方式を設定した状態で、timer_create(2)システムコールを使用し、頻繁にSIGALRM/SIGVTALRMなどの任意のシグナルを発生させるような処理を行うと、アプリケーションから発行されたfork(2)(clone(2))がシグナルを受けて、システムコールがERESTARTNOINTRで復帰し続けます。その結果、fork(2)(clone(2))の再実行が頻発し、終了しない現象が発生することがあります。

  このような場合は、ページング方式をデマンドページング方式に設定することで現象を回避できます。

  環境変数XOS_MMM_L_PAGING_POLICY=demand:demand:demandを明示的に指定してアプリケーションを実行してください。

参考

一般的なコンパイラ(gccなど)でも、本ラージページライブラリlibmpg.soをリンクすることでラージページ化したアプリケーションプログラムを作成できます。ほかのライブラリ(下記例では"-lc -lpthread")を明示的にリンクする場合、ラージページライブラリはそれらのライブラリより常に前に来るよう指定してください。

ラージページ化するにあたり、本ラージページライブラリは、以下のパスでラージページ用のリンカスクリプトも提供しています。これは、静的データ(.data)、および静的データ(.bss)をラージページ化するためのものです。このリンカスクリプトもコンパイラに適切に指定してください。

```
/opt/FJSVxos/mmm/util/bss-2mb.lds
```

以下にgccでコンパイルする場合の例を示します。

```
例) gcc -Wl,-T/opt/FJSVxos/mmm/util/bss-2mb.lds -L/opt/FJSVxos/mmm/lib64 -lmpg -lc -lpthread test_program.c
```

なお、アプリケーションプログラムがPIE(Position Independent Executable、位置独立実行形式)でコンパイルされている場合、.data/.bss領域はラージページ化されません。この場合、ラージページライブラリは警告ログを出力するだけで、.data/.bss領域にはノーマルページが使用され、アプリケーションプログラムは実行を継続します。例えばgccで明示的にPIE形式とならないようにコンパイルするには、-no-pie オプションを使用してください。

L.2.2 アプリケーションプログラムのデータが配置されるメモリ領域

アプリケーションプログラム内の変数は、その定義の仕方によって配置されるメモリ領域が異なります。そのため、各メモリ領域のページサイズを指定する場合は、変数がどのメモリ領域に配置されるのかを意識する必要があります。

本項では、Fortran、C、C++のソースコードを例に、変数の宣言の仕方でどのメモリ領域に配置されるかを説明します。各領域に対するページサイズは、“L.2.1 FXシステムのメモリ領域とラージページ化対象”の“表L.3 FXシステムのメモリ領域とラージページ化対象”のように割り当てられます。

- Fortran

```
program main
integer*8, parameter::N=(1024_8)
real*8 a(N)                    !a は初期値なしローカル配列
real*8 :: b(N)=1.0             !b は初期値あり配列
real*8,allocatable::c(:)       !c は割付配列
allocate(c(N))
...
end
```

ローカル配列 a は静的データ領域(.bss)に配置されます。ただし、-Kautoまたは-Kthreadsafeが有効なオプションで翻訳したアプリケーションプログラムの場合にはプロセススタック領域に配置されます。配列 b は静的データ領域(.data)に配置されます。配列 c は動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。

- C言語

```
#include <stdlib.h>
#include <pthread.h>
#define N 1024
double a[N];                          //a は初期値なしグローバル変数
double b[N]={1.0};                    //b は初期値ありグローバル変数
double *c;
double *d;                            //c,d はポインタ変数
void *func_thread(void *args) {
  double f[N];                        //f はローカル変数
  d=(double *)malloc(sizeof(double)*N);
  ...
}
int main(void){
  double e[N];                        //e はローカル変数
  c=(double *)malloc(sizeof(double)*N);
  ...

  pthread_t pthread;
  pthread_create( &pthread, NULL, &func_thread, (void *)NULL);
  ...
}
```

グローバル変数 a は静的データ領域(.bss)に配置されます。グローバル変数 b は静的データ領域(.data)に配置されます。ポインタ変数 c,d は動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。ローカル変数 e はプロセススタック領域に配置されます。ローカル変数 f はスレッドスタック領域に配置されます。

注意

各プログラミング言語でスレッド並列アプリケーションプログラムを実行した場合に、各スレッドのスタック領域として専用のスレッドスタック領域を用意します。

- C++

```
#include <vector>
const int N = 1024;
struct Klass {
  double k;
  Klass() : k (0.0) {}
  Klass(double K) : k (K) {}
};
std::vector<Klass> a(N);          //vectorクラスによる領域確保
int main(){
    Klass* b = new Klass[N];      //new演算子による領域確保
    std::vector<double> c(N);     //vectorクラスによる領域確保
    return 0;
}
```

変数 a, b, c はすべて動的メモリ確保領域(ヒープ領域またはmmap領域)に配置されます。

注意

C++の変数に割り当てられるメモリ領域はC言語と同じですが、vectorクラスなどにより動的に獲得されるメモリ領域は動的メモリ確保領域(ヒープ領域またはmmap領域)として割り当てられます。

L.3 ラージページライブラリ設定用環境変数

ラージページライブラリの動作を調整するために、本節で示す環境変数が利用できます。

L.3.1 ラージページライブラリの基本設定

本項ではラージページライブラリの基本的な設定をするための環境変数を示します。

表L.4 ラージページライブラリ基本設定用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_HPAGE_TYPE	hugetlbfs|none	hugetlbfs	ラージページライブラリによるラージページ割り当て動作の有効化/無効化を選択する設定です。 「hugetlbfs」の場合、HugeTLBfsによるラージページ化をします。 「none」の場合、ラージページライブラリによるラージページ化は行いません。この指定の場合、"XOS_MMM_L_" で始まるラージーライブラリの環境変数の指定はすべて無効です。 指定値以外の値を指定した場合は、「hugetlbfs」を指定したものとみなします。

変数名	指定値	デフォルト値	詳細
			ラージページ化については注意事項があります。表の下部にある注意(“ラージページ化したときの /proc/*pid*/maps のスタック領域の表示について”)を参照してください。
XOS_MMM_L_LPG_MODE	base+stack|base	base+stack	スタック領域およびスレッドスタック領域のラージページ割り当て動作の有効化/無効化を選択する設定です。 「base+stack」の場合、静的データおよび動的メモリ確保領域だけではなく、スタック領域およびスレッドスタック領域もラージページ化します。 「base」の場合、静的データおよび動的メモリ確保領域のみラージページ化します。スタック領域およびスレッドスタック領域はラージページ化しません。 指定値以外の値を指定した場合は、「base +stack」を指定したものとみなします。 スタック領域のラージページ化については注意事項があります。表の下部にある注意(“スタック領域のラージページ化に伴うアライメントについて”)を参照してください。
XOS_MMM_L_PRINT_ENV	on|off|1|0	0	アプリケーションプログラムのデバッグ向けの設定です。 本環境変数に「1」または「on」を指定した場合、ラージページライブラリが提供する性能チューニング用環境変数の一覧を標準エラー出力に出力します。出力は、アプリケーションプログラムの開始時(main関数の実行より前)に1度だけ行われます。「0」または「off」を指定した場合、性能チューニング用環境変数の一覧を標準エラー出力に出力しません。 指定値以外の値を指定した場合、「0」を指定したとみなします。 なお、main関数の実行後、mallopt(3)によって変更される設定は出力に反映されません。

注意

- ラージページ化したときの /proc/*pid*/maps のスタック領域の表示について(*pid*:プロセスID)

  ラージページ化にあたり、/proc/*pid*/maps の表示内容が一部書き換わります。以下に例を示します。(/proc/*pid*/numa_maps も同様です。)

  ー 静的データ(.bss領域/.data領域)、動的メモリ確保領域(ヒープ領域/mmap領域)

  (ラージページ化する前)

```
00430000-00890000 rw-p 00000000 00:00 0                                  [heap]
```

  (ラージページ化した後)

```
aaaae2400000-aaaae2600000 rw-p 00000000 00:0e 452989                     /anon_hugepage (deleted)
```

ー スタック領域(プロセススタック領域/メインスレッド用スタック領域)

(ラージページ化する前)

```
fffffffd0000-1000000000000 rw-p 00000000 00:00 0                  [stack]
```

(ラージページ化した後)

```
fffaeb800000-1000000000000 rw-p 00000000 00:0e 274404  /memfd: [stack] by libmpg (deleted)
```

- スタック領域のラージページ化に伴うアライメントについて

  ラージページ化したスタック領域のサイズは、基本的に ulimit -s に設定されたsoft limit値を元に、割り当てるHugeTLBfsページのサイズにアラインします。

  また、割り当てるHugeTLBfsページの開始・終了アドレスも同様にアラインアップまたはダウンします。

  マップするアドレスとサイズをアラインした結果、スタック領域が隣接する領域とオーバーラップする可能性があります。オーバーラップした場合、ラージページ化は行いません。警告メッセージを出力し、通常ページのままアプリケーションプログラムを続行します。

L.3.2 ページング方式の設定

本項ではラージページライブラリのページング方式を設定するための環境変数を示します。

表L.5 ページング方式設定用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_PAGING_POLICY	[demand|prepage]:[demand|prepage]:[demand|prepage]	prepage:demand:prepage	各メモリ領域のページング方式(ページの割り当て契機)を選択する設定です。 「demand」はデマンドページング方式、「prepage」はプリページング方式を意味します。 本変数はコロン(:)区切りで3つのメモリ領域のページング方式を指定します。 第1指定は、静的データの.bss領域です。(静的データの.data領域はページング方式指定の対象外で常にprepageとなります。) 第2指定は、スタック領域およびスレッドスタック領域です。 第3指定は、動的メモリ確保領域です。 指定値以外の値を指定した場合は、「prepage:demand:prepage」を指定したものとみなします。 プリページング方式では、その後のアプリケーションプログラムのページフォールトの発生を抑えられ、性能改善および性能ブレが小さくなる傾向があります。ただし、アプリケーションプログラムが自前でプリページング(例えばゼロクリアなど)を実装している場合や、メモリの一部の領域にだけアクセスする場合などには、プリページング方式を選択しないほうが良いケースもあります。

L.3.3 チューニング用の設定

本項ではラージページ割り当てに関するチューニングをするための環境変数を示します。

これらの環境変数のうち、"XOS_MMM_L_"で始まるものは、ラージページライブラリで独自に追加したものです。そのほかの環境変数はglibcの変数です。

表L.6 チューニング用環境変数

変数名	指定値	デフォルト値	詳細
XOS_MMM_L_ARENA_FREE	1|2	1	free(3)で解放されるヒープ領域の扱いに関する設定です。 「1」を指定した場合は、解放可能なメモリを即時に解放します。「2」を指定した場合は、メモリを一切解放せず、全メモリをプールして再利用します。「1」、「2」以外の値を指定した場合、「1」を指定したものとみなします。 「2」の場合、MALLOC_MMAP_THRESHOLD_で指定した値以上のサイズのメモリ要求もヒープ領域から割り当て、解放後も空きメモリ領域として保持し続けます。ヒープ領域の解放処理は行いません。メモリ使用効率低下防止のため単一のヒープ領域ですべてのサイズのメモリ割り当てをする必要があるため、スレッドヒープ領域は生成しません。すなわち「2」は以下の設定の組み合わせと等価です。 XOS_MMM_L_ARENA_LOCK_TYPE=1 XOS_MMM_L_MAX_ARENA_NUM=1 MALLOC_MMAP_THRESHOLD_=ULONG_MAX MALLOC_TRIM_THRESHOLD_=ULONG_MAX
XOS_MMM_L_ARENA_LOCK_TYPE	0|1	1	メモリ割り当てポリシーに関する設定です。「0」はメモリ獲得性能優先、「1」はメモリ使用効率優先を意味します。「0」、「1」以外の値を指定した場合、「1」を指定したものとみなします。 「0」の場合、メインアリーナが競合すると、新たにスレッドヒープ領域を生成します。「1」の場合、メインアリーナが競合すると、最大アリーナ数(XOS_MMM_L_MAX_ARENA_NUM)以内であればスレッドごとにスレッドヒープ領域を生成し、そうでなければロックが獲得できるまで生成を待ちます。 マルチスレッドアプリケーションプログラムでメモリ獲得要求を同時に呼び出した場合、「0」は並列処理が可能ですが、メモリ使用効率が低下します。「1」はメモリ獲得が逐次処理となりますが、メモリ使用効率が向上します。 マルチスレッドアプリケーションプログラムで各スレッドのメモリ獲得要求が競合し、かつ、競合が頻発するようなケースでは、設定を「0」にすることで性能が改善する可能性があります。
XOS_MMM_L_MAX_ARENA_NUM	1以上INT_MAX以下の整数値 [10進数]	1	XOS_MMM_L_ARENA_LOCK_TYPE=1のときのみ有効な変数で、生成可能なアリーナ(プロセスヒープとスレッドヒープ領域の総和)の数を設定できます。生成するスレッドヒープ領域の数を制限したい場合に使用してください。 デフォルト設定(「1」)の場合、プロセスヒープ領域のみを使用し、スレッドヒープ領域は生成されま

変数名	指定値	デフォルト値	詳細
			せん。この場合、XOS_MMM_L_ARENA_LOCK_TYPE=1と等価です。設定をn(≧2)とした場合、プロセスヒープ領域のほか、(n-1)個のスレッドヒープ領域が生成される可能があります。
XOS_MMM_L_HEAP_SIZE_MB	MALLOC_MMAP_THRESHOLD_の2倍以上ULONG_MAX以下の整数値<MiB単位> [10進数]	MALLOC_MMAP_THRESHOLD_の2倍	スレッドヒープ領域を使用する場合に、スレッドヒープ領域の生成時および拡張時に獲得するメモリサイズを設定します。 デフォルト値はMALLOC_MMAP_THRESHOLD_の2倍で、スレッドのメモリ獲得の初回時にデフォルト値のスレッドヒープ領域が生成されます。1スレッド当たりの総獲得メモリ量がデフォルト値を超えるとさらに指定値のスレッドヒープ領域を生成(拡張)します。 各スレッドが獲得するメモリ量が少ないと予想される場合、本環境変数の値を小さく設定することでメモリ使用効率が向上する可能性があります。
XOS_MMM_L_COLORING	0|1	1	キャッシュカラーリングの有無の設定です。 キャッシュカラーリングをすると、プロセッサのL1キャッシュのコンフリクトを軽減します。 「0」の場合、キャッシュカラーリングを行いません。 「1」の場合、MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズで行われるmmap(2)によるメモリ獲得時にはキャッシュカラーリングをします。MALLOC_MMAP_THRESHOLD_未満のサイズで行われるヒープ領域からのメモリ獲得時にはキャッシュカラーリングを行いません。 キャッシュカラーリングが必ず有効になる条件は下記です。 1) MALLOC_MMAP_THRESHOLD_ 以上のサイズのmalloc(3)要求をする、かつ、 2) XOS_MMM_L_FORCE_MMAP_THRESHOLD=1を指定する。(これにより必ずmmap(2)によりメモリ領域を割り当てます。) 「0」、「1」以外の値を指定した場合は、「1」を指定したものとみなします。 一般的にはキャッシュカラーリングをおこなったほうが性能を改善する傾向にありますが、アプリケーションプログラムが自前でカラーリングを実装している場合は、本ライブラリのキャッシュカラーリングを無効にしたほうが良いといえます。
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0|1	0	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)を優先するかどうかの設定です。 「0」の場合、mmap(2) は優先しません。まずヒープ領域の空きを検索し、空きがあればヒープ領域の空きメモリを返します。ヒープ領域の空きが

変数名	指定値	デフォルト値	詳細
			見つからないときにのみ mmap(2) でメモリを獲得します。 「1」の場合、mmap(2) を優先します。ヒープ領域の空きは検索せず、(例え空きがあっても) mmap(2) でメモリを獲得します。 「0」、「1」以外の値を指定した場合、「0」を指定したものとみなします。
MALLOC_CHECK_	0|1|2|3|5|7 [10進数]	3	プログラミングエラー(メモリ破壊や二重解放など)の検出に関する設定です。設定に従い、エラー検出時に以下のアクションをします。 0 : 無視して処理を継続する。 1 : 詳細なエラーメッセージを出力し、処理を継続する。 2 : アプリケーションプログラムをabortする。 3 : 詳細なエラーメッセージ、スタックトレース、メモリマッピングを出力し、アプリケーションプログラムをabortする。 5 : 簡単なエラーメッセージを出力し、処理を継続する。 7 : 簡単なエラーメッセージ、スタックトレース、メモリマッピングを出力し、アプリケーションプログラムをabortする。 これらの値以外の値を指定すると、2進数で下位3ビットの値に対応した動作をします。 なお、すべてのエラーを検出できるわけではなく、メモリリークなどは検出できません。
MALLOC_TOP_PAD_	0以上 ULONG_MAX以下の整数値<byte単位> [10進数]	131072 (=128KiB)	ヒープ領域を伸長する際の1回当たりの伸長サイズを設定します。ページサイズで切り上げた値が使用されます。 アプリケーションプログラムが一度に獲得・解放するメモリ量より大きい値を設定することで、システムコールの発行回数が軽減し、メモリ獲得性能が向上することがあります (ただし、メモリ使用効率は低下する可能性があります)。
MALLOC_PERTURB_	INT_MIN以上 INT_MAX以下の整数値 [10進数]	0	アプリケーションプログラムのデバッグ向けの設定です。メモリ獲得(calloc(3)を除く)およびメモリ解放時に本環境変数に指定された値に基づいてメモリ領域を埋めます。メモリ獲得時は指定された値の最下位バイトの補数、メモリ解放時には最下位バイトを書き込みます。 メモリ領域を初期化せずに使用している、または、解放済みのメモリ領域を参照しているといった問題を検出する際に使用します。
MALLOC_MMAP_MAX_	INT_MIN以上 INT_MAX以下の整数値 [10進数]	2097152 (=2*1024*1024)	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)でメモリ獲得する回数の上限値の設定です。

変数名	指定値	デフォルト値	詳細
			mmap(2)によるメモリ獲得時に現在値を1カウントアップします。free(3)で解放すると現在値を1カウントダウンします。 上限値はプロセス単位で設定されます。 「0」に設定すると、メモリ獲得サイズやMALLOC_MMAP_THRESHOLD_の設定値に関係なくmmap(2)ではなくヒープ領域からメモリを獲得します。
MALLOC_MMAP_THRESHOLD_	0以上ULONG_MAX以下の整数値<byte単位> [10進数または16進数]	134217728 (=128MiB)	本環境変数で指定したサイズ以上のサイズのメモリ獲得要求は、mmap(2)によってメモリを獲得します。本環境変数で指定したサイズ未満のメモリ獲得要求はヒープ領域からメモリを獲得します。 mmap(2)で獲得したメモリをfree(3)すると、メモリは即時に解放されます。一方、ヒープ領域から獲得したメモリをfree(3)すると、ヒープ領域のトップにMALLOC_TRIM_THRESHOLD_以上の連続した空き領域ができない限り、メモリを即時に解放せずプールします。プールしたメモリは再利用が可能です。 本設定値を大きくするとヒープ領域で管理する最大メモリサイズが増大し、メモリ領域の再利用が促進されメモリ獲得性能が向上することがあります(ただし、メモリ使用効率は低下する可能性があります)。
MALLOC_TRIM_THRESHOLD_	0以上ULONG_MAX以下の整数値<byte単位> [10進数または16進数]	134217728 (=128MiB)	ヒープ領域から獲得したメモリをfree(3)する場合に、メモリを即時に解放するかどうかの閾値の設定です。指定した閾値以上の連続した空きメモリがヒープ領域のトップにできる場合、メモリを即時に解放します。連続した空きメモリが閾値未満の場合、メモリを解放せずにプールします。 本設定値を大きくすると一度獲得したメモリを解放する頻度が低下し、メモリ領域の再利用が促進されメモリ獲得性能が向上することがあります(ただし、メモリ使用効率は低下する可能性があります)。

例

環境変数XOS_MMM_L_ARENA_FREEを使ったチューニングによる効果の確認例

- **アプリケーションプログラムの説明**

  1. 8MiBサイズのヒープ領域のメモリを1024回獲得(malloc(3))して、解放(free(3))します。
  2. 1の処理を2回ループ実行します。
  3. 各回のmalloc(3)/free(3)の時間を測定します。

- **アプリケーションプログラム**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
```

```
#include <time.h>

#define N 1024
#define MALLOC_CNT 1024
#define DATA_CNT (1024*1024*1024)

clock_t time_start;
clock_t time_end;
double *c[MALLOC_CNT];  //heap memory
double a[DATA_CNT];  //data memory

int main(int argc, char *argv[]){
    int i;
    long sec;
    long nsec;
    int loop=0;
    struct timespec time1 ={0,0};
    struct timespec time2 ={0,0};

    while(loop <2){
        printf("malloc start.¥n");
        clock_gettime(CLOCK_REALTIME, &time1);
        for(i=0;i<MALLOC_CNT;i++){
            c[i]=(double *)malloc(sizeof(double)*N*N);
            if (c[i] == NULL) {
                fprintf(stderr, "malloc error: cnt=%d, errno=%d¥n", i, errno);
                exit(1);
            }
        }
        clock_gettime(CLOCK_REALTIME, &time2);
        printf("malloc end.¥n");
        sec = (time2.tv_sec - time1.tv_sec);
        nsec= (time2.tv_nsec-time1.tv_nsec);
        if(nsec<0){
            sec--;
            nsec += 1000000000L;
        }
        printf("MALLOC TIME:%d:%010d¥n", sec, nsec);
        sleep(10);

        printf("free start.¥n");
        clock_gettime(CLOCK_REALTIME, &time1);
        for(i=0;i<MALLOC_CNT;i++){
            free(c[i]);
        }
        clock_gettime(CLOCK_REALTIME, &time2);
        printf("free end.¥n");
        sec = (time2.tv_sec - time1.tv_sec);
        nsec= (time2.tv_nsec-time1.tv_nsec);
        if(nsec<0){
            sec--;
            nsec += 1000000000L;
        }
        printf("FREE TIME:%d:%010d¥n", sec, nsec);
        loop++;
    }
    return EXIT_SUCCESS;
}
```

・ チューニング方法

環境変数 XOS_MMM_L_ARENA_FREE を以下の2つのパターンで設定し、性能を比較します。

1. free(3)時に解放可能なメモリページを即時解放するように設定

```
export XOS_MMM_L_ARENA_FREE=1
```

2. free(3)時に解放可能なメモリページを即時解放しないように設定

```
export XOS_MMM_L_ARENA_FREE=2
```

・ **解説・性能予測**

XOS_MMM_L_ARENA_FREE=1を設定した場合、1回目のfree(3)を実行後にmmappedチャンクに確保されたメモリ領域は即時解放します。その後、2回目のmalloc(3)処理では、mmappedチャンクにもう一度メモリ領域を確保するために1回目のmalloc(3)処理とほぼ同じ時間がかかると考えられます。

XOS_MMM_L_ARENA_FREE=2を設定した場合、1回目のfree(3)を実行後にヒープ領域に確保されたメモリ領域は解放されません。2回目のmalloc(3)処理では、ヒープ領域に確保されているメモリ領域を再利用するため、メモリ割り当て処理コストが削減され、malloc(3)処理時間が短くなると考えられます。

例

環境変数XOS_MMM_L_ARENA_LOCK_TYPEを使ったチューニングによる効果の確認例

・ **アプリケーションプログラムの説明**

1. 複数のスレッドを作成し、各スレッドを別々のCPUに割り当てます。
2. 各スレッドはそれぞれmalloc(3)を発行しており、ヒープ領域を確保します。
3. 1,2の内容をループ10回で実行し、毎回のmalloc(3)/free(3)の時間を測定します。

・ **アプリケーションプログラム**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <sched.h>
#include <errno.h>
#include <sys/time.h>

#define KBYTES (1024)
#define CPU_NUM (16)
#define MAX_MALLOC_CNT (5*64*KBYTES)

pthread_t thread[CPU_NUM];
pthread_barrier_t barrier;
int malloc_size = 64*KBYTES;
int malloc_cnt = MAX_MALLOC_CNT/CPU_NUM;
int loop_cnt = 10;
char *strp[MAX_MALLOC_CNT];
int cpuid[CPU_NUM]={};
int cpunum[CPU_NUM]={};

void* thread_main(void *arg) {
    int cpuid = *(int*)(arg);
    int i;
    pthread_barrier_wait(&barrier);
    for (i=0;i<malloc_cnt;i++) {
        strp[i+cpuid*malloc_cnt] = (char*)malloc(malloc_size);
    }
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    int i, ret, loop;
    cpu_set_t cpu;
    struct timeval st, et, rt;

    if ( 2 == argc ) {
        malloc_size = atoi(argv[1]);
    } else if ( 3 == argc ) {
        malloc_size = atoi(argv[1]);
        malloc_cnt = atoi(argv[2]);
    } else if ( 4 == argc ) {
        malloc_size = atoi(argv[1]);
        malloc_cnt = atoi(argv[2]);
        loop_cnt = atoi(argv[3]);
    }

    loop = 0;
    while ( loop < loop_cnt ) {
        ret = pthread_barrier_init(&barrier, NULL, CPU_NUM+1);
        for (i=0;i<CPU_NUM;i++) {
            cpuid[i] = i;
            cpunum[i] = i;
            ret = pthread_create(&thread[i], NULL, thread_main, (void*)(&cpunum[i]));
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_create: errno=%d¥n", ret);
                exit(1);
            }

            CPU_ZERO(&cpu);
            CPU_SET(cpuid[i], &cpu);
            ret = pthread_setaffinity_np(thread[i], sizeof(cpu_set_t), &cpu);
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_setaffinity_np: errno=%d¥n", ret);
                exit(1);
            }
        }

        pthread_barrier_wait(&barrier);
        gettimeofday(&st, NULL);
        for (i=0;i<CPU_NUM;i++) {
            ret = pthread_join(thread[i], NULL);
            if ( 0 != ret ) {
                fprintf(stderr, "pthread_join: errno=%d¥n", ret);
                exit(1);
            }
        }

        gettimeofday(&et, NULL);
        timersub(&et, &st, &rt);
        printf("%d %d %ld.%ld¥n", malloc_size, malloc_cnt, rt.tv_sec, rt.tv_usec);
        fflush(NULL);
        pthread_barrier_destroy(&barrier);
        for (i=0;i<malloc_cnt*CPU_NUM;i++) {
            free(strp[i]);
        }
        loop++;
    }
    return 0;
}
```

・ **チューニング方法**

環境変数(XOS_MMM_L_ARENA_LOCK_TYPE)を以下の2つのパターンで設定し、性能を比較します。

1. メモリ獲得性能優先(動的メモリ確保要求を並列処理)

```
export XOS_MMM_L_ARENA_LOCK_TYPE=0
```

2. メモリ使用効率優先(動的メモリ確保要求を逐次処理)

```
export XOS_MMM_L_ARENA_LOCK_TYPE=1
```

各パターンでは、環境変数XOS_MMM_L_ARENA_FREE=2も合わせて設定します。これにより、物理ページ割り当ての処理コストが小さくなり、複数スレッドによる動的メモリ確保要求が競合した場合の処理コストの比較が行いやすくなります。

上記のサンプルプログラムでは、1スレッドあたり、サイズが64KiBのメモリを20480回獲得させています。それを16スレッドで競合させ10回計測します。

- **解説・性能予測**

  複数のスレッドが同時にmalloc(3)を発行すると、malloc(3)によるメモリ獲得処理が競合する可能性が高くなります。

  XOS_MMM_L_ARENA_LOCK_TYPE=0に設定した場合、malloc(3)競合の発生時にスレッドヒープ領域を生成することで動的メモリ確保要求を並列処理でき、メモリ獲得にかかる時間が短くなると考えられます。

  XOS_MMM_L_ARENA_LOCK_TYPE=1に設定した場合、malloc(3)競合の発生時にスレッドヒープ領域を生成せずプロセスヒープ領域を共有します。そのため、現在処理中のmalloc(3)処理の完了を待ち合わせて逐次処理をしなければならず、メモリ獲得にかかる時間が長くなると考えられます。

L.4 メッセージ

本節では、ラージページライブラリが出力するメッセージについて説明します。メッセージは標準エラー出力に表示します。

L.4.1 メッセージの読み方

ラージページライブラリが出力するメッセージ形式を以下に示します。

```
メッセージ種別 コンポーネント名 - メッセージ本文
```

ラージページライブラリのメッセージの構成内容を以下に示します。

表L.7 メッセージの構成内容

構成内容	意味
メッセージ種別	以下のメッセージ種別を表示します。 [WARN] ワーニングメッセージ
コンポーネント名	「xos LPG 番号」を表示します。番号はメッセージ固有の識別番号です。
'-'	区切り文字です。
メッセージ本文	発生したイベントの内容を表示します。

L.4.2 メッセージ

警告メッセージ

[WARN] xos LPG 2001 - Failed to allocate HugeTLBfs pages, going to try allocating normal pages.

意味

ラージページ(HugeTLBfs)の獲得に失敗したため、ノーマルページを獲得しました。

対処

ラージページの獲得の失敗要因は、空きメモリ不足/断片化などが挙げられます。ジョブの使用メモリ量および空きメモリ量をご確認ください。必要に応じて、ジョブで使用できるメモリ量をシステム管理者にお問い合わせください。アプリケーションプログラムが使用す

るメモリ量やシステムの設定が適切であっても現象が再発するようであれば、コアダンプを採取し、担当保守員(SE)、または当社 Support Desk に連絡してください。

[WARN] xos LPG 2002 - Failed to map HugeTLBfs for data/bss: *a.out* The e_type of elf header must be ET_EXEC when using libmpg. You can check it on your load module by readelf -h command.

意味

アプリケーションプログラムのバイナリ形式がPIE(Position Independent Executable)であるため、.data/.bss領域をラージページ(HugeTLBfs)化できません。.data/.bss領域には通常ページを使用して実行を継続しました。

a.out : アプリケーションプログラム

対処

.data/.bss領域をラージページ化する場合、アプリケーションプログラムのバイナリ形式がPIEとならないようにコンパイルしてください。(readelf -hコマンドでアプリケーションプログラムのe_typeがET_EXECになるように作成してください。)

[WARN] xos LPG 2003 - Failed to map HugeTLBfs for data/bss: Layout problem with segments *seg_index1* and *seg_index2*: Segments would overlap.

意味

アプリケーションプログラムの.data/.bss領域のアドレス範囲が他セグメントと衝突するため、.data/.bss領域をラージページ(HugeTLBfs)化できません。.data/.bss領域にはノーマルページを使用して実行を継続しました。

seg_index1, seg_index2 : セグメントインデックス値

対処

.data/.bss領域をラージページ化する場合、アプリケーションプログラムのコンパイル時に適切なリンカスクリプト(リンカオプション)を指定し、.data/.bss領域の仮想アドレスをラージページサイズでアラインしてください。

[WARN] xos LPG 2004 - Failed to map HugeTLBfs for thread stack: specified stack address in pthread_attr: addr=*stack_addr* size=*stack_size*

意味

pthread_create(3)の第二引数のattr(属性)にスレッドスタックの仮想アドレスが指定されているため、スレッドスタックをラージページ(HugeTLBfs)化できません。スレッドスタックにはノーマルページを使用して実行を継続しました。

stack_addr : 指定されたスタックアドレス

stack_size : 指定されたスタックサイズ

対処

スレッドスタックをラージページ化する場合、pthread_create(3)の第二引数のattr(属性)にスレッドスタックの仮想アドレスを指定しないでください。

[WARN] xos LPG 2005 - Failed to map HugeTLBfs for process stack: confirmed the existence of multiple threads.

意味

プロセススタックのラージページ化処理中にマルチスレッドを検出したため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックはノーマルページを使用して実行を継続しました。

対処

プロセススタックのラージページ化処理中はシングルスレッドである必要があります。自身のプログラムでスレッドを生成しているかどうか確認してください。またラージページライブラリ以外の他ライブラリでスレッドを生成している可能性も考えられます。

[WARN] xos LPG 2006 - Failed to map HugeTLBfs for process stack: The bottom of the process stack is invading the next vma: bottom=*stack_bottom_addr* next-vma=*next_vma_addr*

意味

プロセススタックの後方(アドレス高位)のメモリ領域と衝突するため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックにはノーマルページを使用して実行を継続しました。

stack_bottom_addr: スタックボトムアドレス

next_vma_addr: 後方メモリ領域の先頭アドレス

対処

プロセススタックをラージページ化する場合、他のメモリ領域とアドレス範囲が衝突しないことを保証する必要があります。自身のプログラムで明示的にmmap(2)を使用してプロセススタックの付近にメモリ領域をマップしているかどうか確認してください。また、ラージページライブラリ以外の動的ライブラリでメモリ領域をマップしている可能性も考えられます。

[WARN] xos LPG 2007 - Failed to map HugeTLBfs for process stack: The top of the process stack is invading the previous vma: top=*stack_top_addr* prev-vma=*prev_vma_addr*

意味

プロセススタックの前方(アドレス低位)のメモリ領域と衝突するため、プロセススタックをラージページ(HugeTLBfs)化できません。プロセススタックにはノーマルページを使用して実行を継続しました。

stack_top_addr: スタック先頭アドレス

prev_vma_addr: 前方メモリ領域ボトムアドレス

対処

プロセススタックをラージページ化する場合、他のメモリ領域とアドレス範囲が衝突しないことを保証する必要があります。自身のプログラムで明示的にmmap(2)を使用してプロセススタックの付近にメモリ領域をマップしているかどうか確認してください。また、ラージページライブラリ以外の動的ライブラリでメモリ領域をマップしている可能性も考えられます。