AOCC ユーザー ガイド

発行番号	**57222**	改訂番号	**4.2**
発行日	**2024 年 2 月**		

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

商標

AMD、AMD の矢印形のロゴ、およびその組み合わせは Advanced Micro Devices, Inc の商標です。

Dolby は Dolby Laboratories の商標です。

ENERGY STAR は米国環境保護庁の登録商標です。

HDMI は HDMI Licensing, LLC の商標です。

HyperTransport は HyperTransport Technology Consortium の許諾商標です。

Linux® は米国およびその他の国における Linus Torvalds 氏の登録商標です。

LLVM™ は LLVM Foundation の商標です。

Microsoft、Windows、Windows Vista、Windows Server、Visual Studio、DirectX は Microsoft Corporation の登録商標です。

MMX は Intel Corporation の商標です。

OpenCL は Apple Inc. の商標であり、Khronos の許可を受けて使用されています。

PCIe は PCI-SIG (PCI-Special Interest Group) の登録商標です。

この資料で使用されているその他の製品名は識別のみを目的としたものであり、各社の商標である可能性があります。

Dolby Laboratories, Inc.

Dolby Laboratories の許諾を受けて製造しています。

Rovi Corporation

このデバイスは、米国特許およびその他の知的所有権によって保護されています。デバイスで Rovi Corporation のコピー防止技術を使用するには Rovi Corporation の認可が必要であり、Rovi Corporation が書面で別途認可しない限り、家庭用およびその他の限定的なペイパービュー用途のみを目的としています。

リバース エンジニアリングまたは分解も禁じられています。

本製品を MPEG-2 規格に準拠した方法で使用することは、MPEG-2 特許ポートフォリオの該当する特許に基づく許諾がない限り明示的に禁止されており、このライセンスは、MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111 より入手できます。

目次

改訂履歴 6
第 1 章 はじめに 7
第 2 章 プログラミング言語のサポート 8
2.1 C、C++、Fortran プログラミング言語 8
2.2 規格との互換性/適合性 10
2.2.1 C99/C11 の付録 F (IEEE-754/IEC 559) のサポート 10
2.2.2 IEEE-754 のサポート 10
第 3 章 AOCC を使用する 11
3.1 Linux へのインストール 11
3.1.1 使用要件 11
3.1.2 インストール 11
3.1.3 SPACK のサポート 12
3.1.4 AOCL-LibM (AMD 数値演算ライブラリ) のアップグレード 13
3.1.5 サポートされるオペレーティング システム (OS) 13
3.1.6 既知の問題と制限 14
3.2 AOCC の起動 14
3.2.1 AOCC オプティマイザー 14
3.2.2 コンパイラの使用 14
3.2.3 ライブラリ 15
第 4 章 プラグマ指示子の使用 17
4.1 Flang 17
4.1.1 NOINLINE 17
4.1.2 FORCEINLINE 17
4.1.3 UNROLL 18
4.1.4 NOUNROLL 18
4.1.5 PREFETCH 19
4.1.6 ベクトル化プラグマ 19
4.1.7 FREEFORM/NOFREEFORM 19
第 5 章 コマンド ライン オプション 20
5.1 Clang と Flang のオプション 20
5.1.1 ターゲットの選択 20
5.1.2 ドライバー 20

5.2 Flang のオプション 21
5.3 コード生成および最適化オプション 23
5.4 ランタイム環境変数 34
5.5 非推奨のオプション 34
第 6 章 デバッグ可能性 35
6.1 OMPD (OpenMP Debugging Support: OpenMP デバッグ サポート) 35
6.2 OMPD コマンド 36
6.3 OMPD サブコマンド 36
第 7 章 診断機能 37
7.1 AOR (AOCC Optimization Report: AOCC 最適化レポート) 37
第 8 章 サポート 39
第 9 章 参考資料 40

表目次

表 1. AOCC の使用条件 11
表 2. AOR の使用条件 11
表 3. OMPD コマンド 36
表 4. OMPD サブコマンド 36

改訂履歴

日付	改訂内容	説明
2024 年 2 月	4.2	リリース固有の更新を追加。
2023 年 8 月	4.1	• 表 2 とセクション 5.4 を追加。 • リリース固有の更新を追加。
2022 年 11 月	4.0	• 7 を追加。 • リリース固有の更新を含め、内容を再構成。
2021 年 12 月	3.2	• Flang サニタイズ コマンドについて第 5 章を更新。 • (NO)FREEFORM プラグマに関連する情報について第 4 章を更新。
2021 年 7 月	3.1	差分更新と、いくつかの主要な/一般的な編集。
2021 年 3 月	3.0	初版。

第 1 章 はじめに

AOCC (AMD Optimizing C/C++ and Fortran Compiler: AMD 最適化 C/C++ および Fortran コンパイラ) は x86 ターゲット、特に AMD “Zen” ベースのプロセッサに高度に最適化されています。このガイドでは、AOCC の使い方について説明します。

AOCC 4.2 は LLVM™ 16.0.3 コンパイラ インフラストラクチャ (llvm.org、2023 年 5 月 3 日) をベースとしており、バグ修正およびその他の新機能のサポートが含まれています。詳細は、AOCC 4.2 のリリース ノートを参照してください。

第 2 章 プログラミング言語のサポート

AOCC は、C、C++、Fortran プログラミング言語用の高性能な x86 CPU コンパイラです。ターゲット (x86 ターゲット、特に AMD プロセッサ) 依存の最適化と、ターゲットに依存しない最適化の両方をサポートしています。

AOCC は、C/C++ プログラムのコンパイラおよびドライバーとして LLVM Clang を利用しています。Flang は、Fortran プログラム向けのコンパイラおよびドライバーです。Clang は 32 ビットと 64 ビットのターゲットをサポートしていますが、Flang は 64 ビットのターゲットのみをサポートしています。

2.1 C、C++、Fortran プログラミング言語

AOCC Clang および AOCC Flang は、前処理、構文解析、最適化、コード生成、アセンブリ、リンクをサポートしています。これらのドライバーを使用すると、コンパイラ、アセンブラー、リンカーなどのその他のツール全体の実行を、渡されたハイレベルなモード設定に応じて制御できます。Clang と Flang は高度に統合されていますが、コンパイルの段階を理解することは重要です。コンパイル段階は次の順番で実行されます。

1. **ドライバー**

   Clang は、プログラムを LLVM 中間表現 (IR) にコンパイルする単なる C および C++ フロントエンドではありません。Clang は、必要な LLVM 最適化パスを確実に使用し、コード生成のターゲットをバイナリの生成に設定するドライバーでもあります。

   Clang と同様、Flang は Fortran フロントエンド コンパイラであり、次の 2 つのコンポーネントで構成されています。

   – **flang1**: Fortran プログラムをトークンに変換するフロントエンド ドライバーによって呼び出されます。パーサーはこれらのトークンを AST (Abstract Syntax Tree: 抽象構文木) に変換します。AST は、ILM コードの生成に使用される標準的な形式に変換されます。
   – **flang2**: flang1 で生成した ILM コードを ILI に変換します。その後、内部オプティマイザーで ILI を最適化します。最適化された ILI は LLVM IR に変換されます。この LLVM IR は、最適化とターゲット コード生成のため、フロントエンド ドライバーによって LLVM オプティマイザーに転送されます。

   簡単に使用する場合、これらの Clang と Flang をエンドツーエンド ドライバーとして使用できます。
   一方、高度なコンパイルを実行する場合、各コンパイル段階を手動で実行できます。

2. **前処理**

   この段階には、入力ソース ファイルのトークン化、マクロ展開、#include 展開、その他のプリプロセッサ指示子の処理が含まれます。この段階の出力は通常、*.i* ファイル (C の場合)、*.ii* ファイル (C++ の場合)、または *.i* ファイル (Fortran の場合) と呼ばれます。

3. **構文解析とセマンティック解析**

   この段階では、入力ファイルを解析し、プリプロセッサ トークンを構文解析木に変換します。構文解析木の形式になっている場合、セマンティック解析を式の演算型に適用し、コードの形式が正しいかチェックします。コンパイラの警告や構文解析エラーのほとんどがこの段階で生成されます。Clang の場合、この段階の出力は AST (抽象構文木) になります。Flang を使用する場合、プログラム トークンを AST に変換し、さらにそれを ILM コードの生成に使用される標準的な形式に変換するために Flang1 が呼び出されます。

4. **LLVM IR コードの生成**

   Clang の場合、この段階では AST を低水準中間コード (LLVM IR) に変換します。

   Flang の場合、Flang2 は Flang1 で生成した ILM コードを受け取り、ILI に変換します。その後、内部オプティマイザーで ILI を最適化し、さらにそれを LLVM IR に変換します。

5. **AOCC オプティマイザー**

   この段階には、生成された LLVM IR の最適化とターゲット固有のコード生成の処理が含まれます。この段階の出力は、通常、*.s* (アセンブリ ファイル) と呼ばれます。

6. **機械語の生成**

   この段階では、最適化された LLVM IR からターゲット固有コードの生成を実行します。この段階の出力は、通常、*.s* (アセンブリ ファイル) と呼ばれます。Clang と Flang は、コード ジェネレーターがオブジェクト ファイルを直接生成する統合アセンブラーの使用もサポートしています。これにより、*.s* ファイルを生成してからターゲット アセンブラーを呼び出すオーバーヘッドを削減できます。

7. **アセンブラー**

   この段階では、ターゲット アセンブラーを実行し、コンパイラの出力をターゲット オブジェクト ファイルに変換します。この段階の出力は、通常、*.o* (オブジェクト ファイル) と呼ばれます。

8. **リンカー**

   この段階では、ターゲット リンカーを実行し、複数のオブジェクト ファイルをまとめて 1 つの実行ファイルまたはダイナミック ライブラリにマージします。この段階の出力は、通常、*a.out* ファイル、*.dylib* ファイル、または *.so* ファイルと呼ばれます。

2.2 規格との互換性/適合性

AOCC は、次の言語およびデバッグ規格をサポートしています。

- C:
  - C17 標準 (ISO/IEC 9899:2018) (デフォルト)
  - C11 標準 (ISO/IEC 9899:2011)
  - C99 標準 (ISO/IEC 9899:1999)
- C++:
  - C++20 標準 (ISO/IEC 14882:2020)[1]
  - C++17 標準 (ISO/IEC 14882:2017) (デフォルト)
  - C++14 標準 (ISO/IEC 14882:2014)
  - C++11 標準 (ISO/IEC 14882:2011)
  - C++98 標準 (ISO/IEC 14882:1998)
- Fortran:
  - Fortran-95 (ISO/IEC 1539:1997)
  - Fortran-2003 (ISO/IEC 1539:2004)
  - Fortran-2008 (ISO/IEC 1539:2010) を一部サポート[2]
- C および C++ アプリケーションのための OpenMP 5.0 およびそれ以前の標準[3]
- Fortran アプリケーションのための OpenMP 4.5 仕様
- C、C++、Fortran のデバッグを可能にする DWARF5 およびそれ以前の標準 (デフォルトは DWARF4)

2.2.1 C99/C11 の付録 F (IEEE-754/IEC 559) のサポート

Clang コンパイラは IEC 559 の数値演算機能をサポートしていません。Clang は、__STDC_IEC_559__ macro の定義を制御または遵守しません。Clang を特定のオプション (-Ofast、-ffast-math など) で使用した場合、さまざまな最適化が有効となり、より高速な数値演算が可能になる一方で、IEEE-754 仕様に準拠しないことがあります。マクロ __STDC_IEC_559__ の値は定義できますが、これらの高速な最適化が有効になっている場合は無視されます。

2.2.2 IEEE-754 のサポート

`-Ofast` オプションまたは `-ffast-math` オプションを指定した場合、Flang コンパイラは IEEE-754 仕様に準拠しません。`-Ofast` および `-ffast-math` のコンパイル モードでは、さまざまな最適化が有効となり、より高速な数値演算が可能になります。

注記: AOCC Flang は GitHub バージョン (https://github.com/flang-compiler/flang.git) を拡張し、機能強化と安定化を図ったものです。

1. C++ 20 への適合性については、『Clang C++20 インプリメンテーション ステータス』を参照してください。
2. AOCC は F2008 の Co-array をサポートしていません。
3. OpenMP 5.0 への適合性については、『Clang 16.0.0 OpenMP 資料』を参照してください。

第 3 章 AOCC を使用する

3.1 Linux へのインストール

3.1.1 使用要件

AOCC をインストールする前に、次のソフトウェア パッケージをインストールする必要があります。

表 1. AOCC の使用条件

パッケージ名	バージョン	説明
libstdc++	6 またはそれ以降	GNU 標準 C++ ライブラリ **注記**: C++20 標準には libstdc++9 またはそれ以降が必要です。
libncurses-dev	5.9 またはそれ以降	libtinfo (低水準の terminfo ライブラリ) を提供します
zlib	1.2.7 またはそれ以降	圧縮ライブラリ
Libxml2	2 またはそれ以降	XML ドキュメントの構文解析を実行します
libquadmath	4.8 またはそれ以降	GCC 4 倍精度数値演算ライブラリ
python	3.x	Python ライブラリ

注記:

1. パフォーマンスを向上させたい場合、最新バージョンの Glibc と Binutils を使用することを推奨します。
2. AOCC コンパイラ バイナリは、Glibc バージョン 2.17 またはそれ以降がインストールされた Linux® システムのみに適合します。

AOR の使用条件を次の表に示します。

表 2. AOR の使用条件

パッケージ名	バージョン	説明
Python	3.8	Python ライブラリ
Python3-pip	23.1.2	パッケージ管理ユーティリティ
pip3 パッケージ Pygments	2.15.1	汎用構文ハイライト ユーティリティ
pip3 パッケージ PyYAML	6.0	Python 用 YAML パッケージ

3.1.2 インストール

注記: このインストールには、root 権限も sudo 権限も必要ありません。

aocc-compiler-<ver>.tar をインストールするには、次のコマンドを実行します。

1. `cd <compdir>`
2. `tar -xvf aocc-compiler-<ver>.tar`
3. `cd aocc-compiler-<ver>`

4. `bash install.sh`

   コンパイラがインストールされ、AOCC のセットアップ手順が表示されます。

5. `source <compdir>/setenv_AOCC.sh`

   このコマンドが実行される AOCC C、C++、Fortran コンパイラを使用するためのシェル環境がセットアップされます。

次の点を確認する必要があります。

- bash コマンド `<compdir>/aocc-compiler-<ver>/AOCC-prerequisites-check.sh` を実行し、使用条件がすべて整っており、シェル環境が正しく設定されているかチェックします。
  - チェックに失敗した項目がある場合、修正 (上記の手順のうち、漏れていた可能性のある手順をもう一度実行) し、`prerequisites_check.sh` を再度実行します。
  - `AOCC-prerequisites-check.sh` で **Check:PASSED** と表示されるまで繰り返します。

  注記: チェック失敗時の警告で言及されているパッケージが実行に必要なければ、先に進んでもかまいません。
- コンパイラがインストールされ、環境が最新版の AOCC に設定されています。いつでも、コマンド ソース `<compdir>/setenv_AOCC.sh` を実行すると、インストール済みのコンパイラに環境変数を設定できます。

3.1.3 SPACK のサポート

注記:

1. SPACK は AOCC 2.2 以降でサポートされています。
2. 次の手順において、aocc@<Version Number> の <Version Number> は AOCC のバージョンを指しています。たとえば、AOCC 4.2.0 をインストールする場合、aocc@4.2.0 を使用する必要があります。

SPACK への AOCC のインストール

AOCC コンパイラを SPACK にインストールするには、次の手順を実行します。

1. AOCC をインストールします。
```
$ spack install -v aocc@<Version Number> +license-agreed
```
2. AOCC を SPACK のコンパイラ リストに追加します。
```
$ spack cd -i aocc@<Version Number>
$ spack compiler add $PWD
$ spack cd -i aocc@<Version Number>
$ spack compiler add $PWD
```
3. 利用可能なすべてのコンパイラの一覧を表示します。
```
$ spack compilers
```

SPACK からの AOCC コンパイラのアンインストール

1. AOCC をアンインストールします。
```
$ spack uninstall aocc@<Version Number>
```
2. *compiler.yaml* ファイルからコンパイラを削除します。
```
$ spack compiler remove aocc@<Version Number>
```

SPACK における AOCC の使用方法の詳細は、次の URL を参照してください。

https://www.amd.com/en/developer/zen-software-studio/applications/spack/spack-aocc.html

3.1.4 AOCL-LibM (AMD 数値演算ライブラリ) のアップグレード

これは、AOCL-LibM を次の URL からアップグレードする場合のみに必要になります。

https://www.amd.com/ja/developer/aocl/libm.html

次の手順を実行してアップグレードします。

1. 最新の AOCL-LibM パッケージを展開します。
2. *aocc-compiler-<ver>/lib/libalm.so* を最新バージョンの *libalm.so* で上書きし、*aocc-compiler-<ver>/lib/libalm.a* を最新バージョンの *libalm.a* で上書きします。
3. 同様に、*aocc-compiler-<ver>/include/amdlibm.h* を最新バージョンの *amdlibm.h* で上書きし、*amdlibm_vec.h* を最新バージョンの *amdlibm_vec.h* で上書きします。

3.1.5 サポートされるオペレーティング システム (OS)

このリリースでは、次の OS がサポートされています。

- RHEL 8.6 および 9.0
- SLES 15 SP3
- Ubuntu 22.04 LTS
- CentOS 8
- glibc 2.17 以降がインストールされたその他の Linux フレーバー /バージョン

RHEL システムと SLES システムにおいて、複数のバージョンの devtoolset がインストールされており、最新バージョンに GCC のインストールが提供されていない場合、コンパイラ オプション `-gcc-install-dir` を使用して、使用する正しい GCC バージョンを指定します。また、RHEL システムでは次のコマンドを使用して devtoolset バージョン 9 を有効にすることもできます。
```
scl enable devtoolset-9 'bash'
```

詳細は、Red Hat の資料を参照してください。

https://access.redhat.com/documentation/en-us/red_hat_developer_toolset/9/html/user_guide/chap-gcc#sect-GCC-CPP

3.1.6 既知の問題と制限

本リリースには、次のような既知の問題と制限があります。

- AOCC バイナリは、glibc バージョン 2.17 またはそれ以降がインストールされた Linux システムでのみ最適な実行が可能です。
- 現在、Flang では 64 ビット ターゲットのみがサポートされています。

3.2 AOCC の起動

コンパイラ ドライバーを起動する前に必要な環境を設定する場合、次のコマンドを実行します。

```
$ source <compdir>/setenv_AOCC.sh
```

3.2.1 AOCC オプティマイザー

AOCC には、依存/非依存のターゲットに対応する多くの最適化が含まれています。最適化レベル *O3* またはそれ以上を使用すると、特定の最適化がデフォルトに設定されます。詳細は、コマンド ライン オプションのセクションを参照してください。最適化の中には、プログラム全体の解析を必要とするため、-flto を使用した LTO (Link Time Optimization: リンク時最適化) 使用時に有効になるものもあります。AOCC に推奨されるリンカーは LLD です。コンパイラ ドライバーで LLD を使用する方法は、LLD リンカーを参照してください。

3.2.2 コンパイラの使用

3.2.2.1 Clang と Clang++

C または C++ のプログラムをビルドして実行するには、次のコマンドを実行します。

```
$ clang [command line flags] xyz.c -o xyz.out
$ ./xyz.out

$ clang++ [command line flags] xyz.cpp -o xyz.out
$ ./xyz.out
```

3.2.2.2 Flang

Fortran のプログラムをビルドして実行するには、次のコマンドを実行します。

```
$ flang [command line flags] xyz.f90 -o xyz.out
$ ./xyz.out
```

3.2.2.3 LLD リンカー

LLD リンカーを使用するには、次のコマンドを実行します。

```
$ clang [command line flags] -fuse-ld=lld xyz.c abc.c -o xyz.out [here -fuse-ld=lld is optional
as this option is default]
$ ./xyz.out
```

3.2.3 ライブラリ

アプリケーションによっては、AOCL (AMD Optimizing CPU Libraries: AMD 最適化 CPU ライブラリ) を使用した方がパフォーマンスが向上する場合があります。AOCC はこれらのライブラリとシームレスに連携します。AOCC でアプリケーションを構築する際は、これらのライブラリを評価検討することを推奨します。AOCL (AMD 最適化 CPU ライブラリ) の詳細は、次の URL を参照してください。

https://www.amd.com/ja/developer/aocl.html

3.2.3.1 ライブラリ パスの設定

次のコマンドを実行し、ライブラリ パスを設定します。

- 64 ビット ライブラリ:
  ```
  export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/lib:$LD_LIBRARY_PATH
  ```
- 32 ビット ライブラリ:
  ```
  export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/lib32:$LD_LIBRARY_PATH
  ```
- その他の AMD 最適化 CPU ライブラリの場合
  ```
  export LD_LIBRARY_PATH=<Path to AMD optimizing CPU Libraries>:$LD_LIBRARY_PATH
  ```

3.2.3.2 AMD ライブラリのリンク

AMDLIBM

リンカーで AMDLIBM をリンクするには、次のコマンドを実行します。

```
$ clang [command line flags] xyz.c -L<compdir>/aocc-compiler-<ver>/lib -lamdlibm -lm -o xyz.out
```

次のコマンドを実行し、リンカーでその他の AMD 最適化 CPU ライブラリをリンクします。

```
$ clang [command line flags] xyz.c -L<Path to AMD optimizing CPU Libraries> -l<library name> -o
xyz.out
```

AMD ベクトル ライブラリ

次のいずれかのコマンドを実行し、AOCC からのベクトル ライブラリの呼び出しをリンクします。

```
$ clang [command line flags] xyz.c -fveclib=AMDLIBM -lamdlibm -lm -o xyz.out
$ clang [command line flags] xyz.c -mllvm -vector-library=AMDLIBM -lamdlibm -lm -o xyz.out
```

AMD FASTMATH ライブラリ

`-fsclrlib=AMDLIBM` を `-Ofast` を付けて使用すると、amdlibm の fastmath 関数が有効になります。リンクに正しいライブラリを使用しない場合、未定義のシンボルが原因でリンクが失敗することがあります。

リンカーで AMDLIBMFAST をリンクするには、次のコマンドを実行します。

```
$clang -Ofast [other command line flags] xyz.c -fsclrlib=AMDLIBM -lamdlibmfast -lamdlibm -lm -o
xyz.out
```

amdalloc

amdalloc は、jemalloc をベースとする AMD のメモリ アロケーターです。AOCC バイナリ パッケージでは、amdalloc と amdalloc-ext という 2 つのバージョンの amdalloc を利用できます。amdalloc-ext はアロケーション サイズの小さいワークロードに最適化されています。

リンカーで amdalloc をリンクするには、次のコマンドを実行します。

```
$ clang [command line flags] xyz.c -L<Path to AMD optimizing CPU Libraries> -lamdalloc -o
xyz.out
```

3.2.3.3 pSTL のサポート

AOCC は、C++ の STL (標準テンプレートライブラリ) 並列アルゴリズムをサポートしています。

次に例を示します。

```
clang++ -O3 -pthread -std=c++17 -ltbb pstl1.cpp
```

使用要件

pSTL をサポートするには、インテル® TBB (スレッディング ビルディング ブロック) ライブラリが必須です。TBB パッケージがインストールされていない場合、pSTL 関数が使用されていても、コンパイラは STL のシリアル インプリメンテーションに切り換えます。

第 4 章 プラグマ指示子の使用

4.1 Flang

ここでは、Flang 専用のプラグマ指示子について述べます。

4.1.1 NOINLINE

この指示子は、指定したルーチンをインライン展開しないようコンパイラに指示します。

!DIR$ NOINLINE

この指示子を使用するには、コンパイラ最適化レベルを -O0 ～ -O3 にする必要があります。NOINLINE 指示子は、コンパイラ オプション `-finline-functions` と `-fno-inline-functions` より優先されます。

例:

```
!DIR$ NOINLINE
SUBROUTINE func_noinline
    INTEGER :: i
    do i = 0, 5
            WRITE(*, *) "Hello World"
    end do
END SUBROUTINE func_noinline

PROGRAM test_inline
    IMPLICIT NONE
    call func_noinline
END PROGRAM test_inline
```

4.1.2 FORCEINLINE

この指示子は、指定したルーチンを常にインライン展開するようコンパイラに指示します。

!DIR$ FORCEINLINE

この指示子を使用するには、コンパイラ最適化レベルを -O0 ～ -O3 にする必要があります。FORCEINLINE 指示子は、コンパイラ オプション `-finline-functions` と `-fno-inline-functions` より優先されます。

例:

```
!DIR$ FORCEINLINE
SUBROUTINE func_forceinline
    INTEGER :: i
    do i = 0, 5
            WRITE(*, *) "Hello World"
    end do
END SUBROUTINE func_forceinline

PROGRAM test_inline
    IMPLICIT NONE
    call func_forceinline
END PROGRAM test_inline
```

4.1.3 UNROLL

この指示子は、ループを展開する回数をコンパイラに指示します。

!DIR$ UNROLL [(n)]

- **n** – オプションのパラメーターで、整数定数を 1 ～ 512 の範囲で指定します。
- **n** = 0 のとき、展開するかどうかはコンパイラが決定します。

この指示子を使用するには、コンパイラ最適化レベルを -O1 またはそれ以上にする必要があります。

n を指定した場合、オプティマイザーはループを n 回展開します。

n を指定しない場合、または範囲外の場合、オプティマイザーは収益性に基づいてループを展開します。

例:

```
例 1:
subroutine func1(a, b)
  integer :: m = 10
  integer :: i, a(m), b(m)

  !dir$ unroll
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func1

例 2:
subroutine func2(m, a, b)
  integer :: i, m, a(m), b(m)

  !dir$ unroll(4)
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func2
```

4.1.4 NOUNROLL

この指示子は UNROLL の逆で、ループを展開する前に、ループの展開を無効にします。

!DIR$ NOUNROLL

例:

```
subroutine func1(a, b)
  integer :: m = 10
  integer :: i, a(m), b(m)

  !dir$ nounroll
  do i = 1, m
    b(i) = a(i) + 1
  end do
end subroutine func1
```

4.1.5 PREFETCH

この指示子は、サポートされている場合、メモリ参照の命令をプリフェッチするためのヒントをコード ジェネレーターに挿入するために使用されます。これにより、コードの特性のパフォーマンスが向上します。詳細は、llvm.prefetch 組み込み関数の資料を参照してください。

!$MEM PREFETCH

制約: この指示子を使用するには、コンパイラ最適化レベルを -O0 ～ -O3 にする必要があります。

例:

```
subroutine prefetch_dir(a1, a2)
  integer :: a1(4096)
  integer :: a2(4096)

  do i = 128, (4096 - 128)
    !$mem prefetch a1, a2(i + 256)
    a1(i) = a2(i - 127) + a2(i + 127)
  end do
end subroutine prefetch_dir
```

4.1.6 ベクトル化プラグマ

ループのベクトル化を制御するコンパイラ指示子を次に示します。

- !DIR$ VECTOR
- !DIR$ NOVECTOR
- !DIR$ VECTOR ALWAYS

これらのプラグマを適用するには、最適化レベルを -O1 ～ -O3 にする必要があります。

AOCC 3.1 で導入された (ベクトル化プラグマを制御する) `-Menable-vectorize-pragmas` は、AOCC 3.2 では非推奨です。

4.1.7 FREEFORM/NOFREEFORM

FREEFORM はコンパイラが自由形式でソースをコンパイルするように指示し、NOFREEFORM は固定形式でコンパイルするよう指示します。これらはファイル内の指示子が記述された箇所以降の部分に適用されますが、同じファイル内でコンパイラが反対の指示子が見つけた場合、指示は逆転します。

例:

```
!DIR$ FREEFORM
! This is free-form
temp = a; a = b; b = temp ! Swap a and b
write(6,*) 'Swapped a and b values are =',  &
           a,b           ! Print a and b
!DIR$ NOFREEFORM
C---This-is-fixed-form
```

第 5 章 コマンド ライン オプション

5.1 Clang と Flang のオプション

5.1.1 ターゲットの選択

次のリストは、すべてのターゲット選択オプションを示しています。

- **-march**=<cpu>

  Clang/Flang でファミリの特定のプロセッサ用のコードを生成できるようにします。たとえば、-march=znver1 を指定した場合、コンパイラは AMD 17h またはそれ以降のプロセッサで有効な命令を生成できます。

- **-march**=znver1

  このアーキテクチャ オプションは、AMD “Zen” ベースの x86 アーキテクチャに最適なコード生成と調整を有効にするために使用します。すべての x86 AMD “Zen” ISA と関連する組み込み関数がサポートされます。

- **-march**=znver2

  このアーキテクチャ オプションは、AMD “Zen2” ベースの x86 アーキテクチャに最適なコード生成と調整を有効にするために使用します。すべての x86 AMD “Zen2” ISA と関連する組み込み関数がサポートされます。

- **-march**=znver3

  このアーキテクチャ オプションは、AMD “Zen3” ベースの x86 アーキテクチャに最適なコード生成と調整を有効にするために使用します。すべての x86 AMD “Zen3” ISA と関連する組み込み関数がサポートされます。

- **-march**=znver4

  このアーキテクチャ オプションは、AMD “Zen4” ベースの x86 アーキテクチャに最適なコード生成と調整を有効にするために使用します。すべての x86 AMD “Zen4” ISA と関連する組み込み関数がサポートされます。

5.1.2 ドライバー

- **-mllvm <options>**

  オプションがコンパイラのフロント エンドを通過し、この最適化が実装されるオプティマイザーに適用されるようにするには、-mllvm を指定する必要があります。

  例: `-mllvm -enable-strided-vectorization`

- **-fuse-ld=lld**

  コンパイラ ドライバーから lld リンカーを優先リンカーとして呼び出すために使用します。

注記: Clang オプションの詳細は、Clang 16.0.0 のコンパイラ オプションを参照してください。

5.2 Flang のオプション

コンパイラ オプションの一覧を表示するには、次のコマンドを使用します。

```
$flang -help
$flang --help-hidden
```

Flang コンパイラは、Clang 16.0.0 のコンパイラ オプションすべてと、次の Flang 固有のコンパイラ オプションをサポートしています。

- `-Kieee`

  AOCC 2.2.0 ではデフォルトで有効に設定されています。

  IEEE-754 仕様に準拠するようコンパイラに指示します。コンパイラは IEEE 754 仕様に厳密に準拠して浮動小数点演算を実行するようになります。このオプションを指定すると、一部の最適化は無効になります。

- `-no-flang-libs`

  Flang ライブラリにリンクしません。

- `-mp`

  OpenMP を有効にし、OpenMP ライブラリ *libomp* をリンクします。

- `-nomp`

  OpenMP ライブラリ *libomp* とリンクしません。

- `-Mbackslash`

  クォーテーションで囲んだ文字列内のバックスラッシュをほかの文字と同じように扱います。

- `-Mnobackslash`

  クォーテーションで囲んだ文字列内のバックスラッシュを C スタイルのエスケープ文字として扱います (デフォルト)。

- `-Mbyteswapio`

  書式なしの入出力のバイト順をスワップします。

- `-Mfixed`

  ソースを固定形式と想定します。

- `-Mextend`

  ソース行を最大 132 文字まで許可します。

- `-Mfreeform`

  ソースを自由形式と想定します。

- `-Mpreprocess`

  Fortran ファイルからプリプロセッサを実行します。

- `-Mstandard`

  標準に適合しているかチェックします。

- `-Msave`

  すべての変数が SAVE 属性を持つと想定します。

- `-module`

  モジュール ファイルへのパスです (-I でも可)。

- `-Mallocatable=95`

  割り当て可能オブジェクトへの割り当てに Fortran 95 セマンティクスを選択します (デフォルト)。

- `-Mallocatable=03`

  割り当て可能オブジェクトへの割り当てに Fortran 03 セマンティクスを選択します。

- `-static-flang-libs`

  静的 Flang ライブラリを使用してリンクします。

- `-M[no]daz`

  非正規化数を 0 として扱います。

- `-M[no]flushz`

  SSE をゼロフラッシュ モードに設定します。

- `-Mcache_align`

  大きなオブジェクトをキャッシュ ライン境界にアライメントします。

- `-M[no]fprelaxed`

  このオプションは無視されます。

- `-fdefault-integer-8`

  INTEGER と LOGICAL を、INTEGER*8 と LOGICAL*8 として扱います。

- `-fdefault-real-8`

  REAL を REAL*8 として扱います。

- `-i8`

  INTEGER と LOGICAL を、INTEGER*8 と LOGICAL*8 として扱います。

- `-r8`

  REAL を REAL*8 として扱います。

- `-fno-fortran-main`

  Fortran main ではリンクしません。

- `-Mrecursive`

  ローカル変数をスタック上に割り当てます (これにより、再帰が可能になります)。このスイッチの設定にかかわらず、SAVEd、data-initialized、または namelist のメンバーは常に静的に割り当てられます。

5.3 コード生成および最適化オプション

Clang と Flang の両方が、LLVM IR を変換してターゲットの x86 プラットフォームに最適なコードを生成するために、AOCC オプティマイザーとコード ジェネレーターの段階に依存しています。

次のリストは最適化オプションをタイプ別に分類したものです。

最適化レベルのオプション

- -O0

  最適化なし: 最もコンパイル速度が速く、最もデバッグしやすいコードを生成します。

- -O1

  -O0 レベルと -O2 レベルの中間です。

- -O2 (デフォルト)

  ほとんどの最適化を有効にします。

- -O3

  すべての最適化を有効にします。実行に時間がかかったり、(プログラムを高速で実行しようとするために) コードのサイズが大きくなったりすることがあります。

  AOCC の -O3 レベルには、基本の LLVM バージョンよりも多くの最適化が含まれています。これらの最適化には、間接呼び出しの処理の改善と、高度なベクトル化が含まれます。

- -Ofast

  -O3 から引き継ぐすべての最適化を有効にした上で、言語仕様への厳密な準拠に違反する可能性のあるその他のアグレッシブな最適化を有効にします。

  AOCC の -Ofast レベルには、基本の LLVM バージョンよりも多くの最適化が含まれています。これらの最適化には、部分的な条件分岐排除、インライン展開の改善、展開などがあります。

- -flto

  リンク時最適化に適した LLVM 形式で出力ファイルを生成します。このオプションを -S と合わせて使用すると、LLVM 中間言語アセンブリ ファイルが生成されます。それ以外の場合、LLVM ビットコード形式のオブジェクト ファイルが生成されます (段階選択オプションによっては、これらのファイルはリンカーに渡される可能性があります)。

- -m32

  32 ビット環境用のコードを生成します。32 ビット環境では、int、long、ポインターが 32 ビットに設定され、任意の i386 システムで実行可能なコードが生成されます。コンパイラは x86 または IA32 32 ビット ABI を生成します。32 ビット ホストにおけるデフォルトは 32 ビット ABI です。

  注記: SUSE Linux Enterprise Server は、SP1 からの 32 ビット アプリケーションのコンパイルをサポートしていません。32 ビット バイナリのランタイム サポートのみを提供しています。

- -m64

  64 ビット環境用のコードを生成します。64 ビット環境では、int を 32 ビットに、long とポインターを 64 ビットに設定し、x86-64 アーキテクチャ向けのコードを生成します。64 ビット ホストにおけるデフォルトは 64 ビット ABI です。

- -ffast-math

  高速だが低精度で IEEE-754 仕様に準拠しないこともある数値演算を提供するさまざまな最適化を有効にします。このオプションを指定すると、システム ヘッダーに __STDC_IEC_559__ マクロが設定されていても無視されます。

- -fopenmp

  OpenMP 指示子の処理を有効にし、並列コードを生成します。リンクする OpenMP ライブラリは、オプション -fopenmp=library で指定できます。

- -Os

  レベル -O2 と似ていますが、コード サイズを削減するための追加の最適化が含まれています。

- -Oz

  レベル -Os に (したがって -O2 にも) 似ていますが、コード サイズをさらに削減します。

- -O

  レベル -O2 と同等です。

- -O4 およびそれ以上

  レベル -O3 と同等です。

- -zopt

  ループ不変量コード移動、SLP およびループのベクトル化、ループ融合、ループ変換、ループ内条件分岐排除、ループ タイル化、ループ分割などの改良版を含む、スカラー、ベクトル、ループ変換のサブセットを有効にします。

- -fPIC

  このオプションはデフォルトでオンです。共有および静的ライブラリ用の PIC (位置独立コード) を生成します。PIC の生成を無効にするには `-fno-PIC` を使用します。

- -fPIE

  このオプションはデフォルトでオンです。実行ファイルと静的ライブラリ用の PIC を生成します。PIC の生成を無効にするには `-fno-PIE` を使用します。

注記: AOCC 4.1 からは -fPIC オプションと -fPIE オプションがデフォルトで設定されているため、デフォルトで位置独立コードが生成されます。AOCC 4.1 以降では、これまでに静的にリンクされ、-fPIC オプションを使用していない静的ライブラリは、AOCC でこれらのデフォルトのオプションを使用して再ビルドする必要があります。そうしないと、リンカー エラーが表示されることがあります。または、AOCC で -fno-PIC/-fno-PIE オプションを使用して、位置独立コードの生成を無効にします。

浮動小数点オプション

- `-ffp-model`

  浮動小数点の動作を指定します。`-ffp-model` は、その他の単一用途の浮動小数点オプションの機能を含む包括的なオプションです。

  有効な値: `precise` と `fast`

  説明:

  – `precise` は、浮動小数点データの値にとって安全でない最適化を無効にします。浮動小数点の縮約 (FMA) を無効にします。

  – fast は `-ffast-math` を指定した場合と同じような動作です--。

  この `-ffp-model` の動作は、AOCC Flang のみに適用されます。Clang の場合、Clang の浮動小数点の資料を参照してください。

  使用法: `-ffp-model={precise, fast}`

ベクトル最適化オプション

- **`-enable-strided-vectorization`**

  LLVM にあるインターリーブされたベクトル化フレームワークの拡張として、メモリ ストライドのベクトル化を有効にします。ギャザー /スキャッター命令パターンをより効果的に使用できるようになります。このオプションは、インターリーブ ベクトル化オプションと組み合わせて使用する必要があります。

  使用法: `-mllvm -enable-strided-vectorization`

- **`-global-vectorize-slp={true,false}`**

  基本ブロック内の直線的なコードを、データ並べ替えベクトル演算でベクトル化します。このオプションはデフォルトで **false** に設定されています。

  使用法: `-mllvm -global-vectorize-slp={true,false}`

- **`-enable-loop-vectorization-with-conditions`**

  ループ本体を平坦化してベクトル化する代わりに、ベクトル命令を条件付きで実行することで、条件付きループの効率的なベクトル化を可能にします。ベクトル化されたコードは、比較命令のベクトル バージョンを使用してループ本体内の命令を保護し、マスクされた命令を使用して安全でないメモリ操作から保護します。デフォルトでは、このオプションは O3 で有効に設定されます。

  使用法: `-mllvm -enable-loop-vectorization-with-conditions`

- **`-legalize-vector-library-calls`**

  ベクトル ライブラリ呼び出しのサポートされていないベクトル係数の大きいバージョンを、ベクトル ライブラリ呼び出しのサポートされているベクトル係数のバージョンに分割し、ベクトル化を有効にします。デフォルトでは、このオプションは O3 で有効に設定されます。

  使用法: `-mllvm -legalize-vector-library-calls`

- **`-vectorize-non-contiguous-memory-aggressively`**

  複数のロード/ストア、挿入を生成することにより、非連続なメモリ割り当てを伴うベクトル化を有効にします。デフォルトでは、このオプションは O3 で有効に設定されます。

  使用法: `-mllvm -vectorize-non-contiguous-memory-aggressively`

- `-fvector-transform`

  ベクトル変換のサブセット (SLP およびループのベクトル化の改良版を含む) を有効にします。`-zopt` で有効です。

  使用法: `-fvector-transform`

- `-fepilog-vectorization-of-inductions`

  ループ帰納変数のベクトル化を必要とするループの epilog ベクトル化を有効にします。

  使用法: `-fepilog-vectorization-of-inductions`

ループ最適化オプション

- **`-enable-partial-unswitch`**

  部分的な条件分岐排除を有効にします。これは、LLVM に既に存在するループ内条件分岐排除最適化を拡張したものです。オリジナルのループ内条件分岐排除が完全にループ不変な条件で動作するのに対し、部分的な条件分岐排除では、実行条件が不変のパスからループ内の条件を外に巻き上げます。不変のパスからはループ内の条件を外に巻き上げ、条件が変化するパスでは元のループを保持します。

  使用法: `-mllvm -enable-partial-unswitch`

- **`-aggressive-loop-unswitch`**

  分岐条件の値に基づいて、アグレッシブなループ内条件分岐排除ヒューリスティック (`-mllvm -enable-partial-unswitch` を含む) を有効にします。ループ内条件分岐排除はコードの肥大化につながります。巻き上げた条件を頻繁に実行する場合、コードの肥大化は最小限に抑えられます。このヒューリスティックは、ループ内で条件が使用される回数に基づいて条件を優先順位付けします。ヒューリスティックはオプション `-unswitch-identical-branches-min-count=<n>` で制御できます。

  ある分岐条件値 (B) について、B が少なくとも **<n>** 回の比較に出現する場合、ループの条件分岐排除を有効にします。このオプションは `-aggressive-loop-unswitch` で有効になります。デフォルト値は 3 です。

  使用法: `-mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-min-count=<n>`

  ここで、**n** は正の整数で、**<n>** の値が小さいほど多くの条件分岐排除が実行されます。

- **`-lv-function-specialization`**

  関数内のループがベクトル化可能で、引数が互いにエイリアス化されていない場合、特殊化されたバージョンの関数を生成します。

  使用法: `-mllvm -lv-function-specialization`

- **`-loop-splitting`**

  ループを複数のループに分割し、ループ帰納変数と不変式または定数式を比較する分岐をなくします。デフォルトでは、このオプションは -O3 で有効に設定されます。この最適化を無効にするには、`-loop-splitting=false` を使用します。

  使用法: `-mllvm -loop-splitting`

- **`-enable-ipo-loop-split`**

  ループを複数のループに分割し、ループ帰納変数と定数式を比較する分岐をなくします。この定数式は、プロシージャー間の分析によって導出できます。デフォルトでは、このオプションは -O3 で有効に設定されます。この最適化を無効にするには、`-enable-ipo-loop-split=false` を使用します。

  使用法: `-mllvm -enable-ipo-loop-split`

- **`-enable-loop-fusion`**

  このオプションでは、複数のループ ネストの本体を 1 つのループ ネストに融合する、古典的なループ融合を有効にします。この変換は、関連するループ ネストの境界、ループ ネストのネスト制御フローなど、さまざまな正当性条件をチェックします。この変換はデフォルトでオフになっており、このオプションを使用することでユーザーが有効にできます。ループ融合は、ループ ネスト全体でメモリ アクセス操作の再利用を可能にし、キャッシュのパフォーマンスにもメリットをもたらします。この変換では収益性チェックの一環で、作成される融合ループの本体のサイズを制御するコード サイズしきい値を使用します。

  使用法: `-mllvm -enable-loop-fusion`

- **`-enable-loopinterchange`**

  このオプションは、ループ ネストに対して古典的なループ交換またはループ並べ替えを有効にします。多次元ループ ネスト内のループを並べ替え、その間にさまざまな正当性条件をチェックします。この変換はデフォルトでオフになっており、このオプションを使用することで有効にできます。ループ交換は、内側のループから上位のループに巻き上げられるループ不変式の数が最大になるような多次元ループ ネスト内のループの並べ替えを見つけようとします。

  使用法: `-mllvm -enable-loopinterchange`

- **`-fuse-tile-inner-loop`**

  ループ タイル化変換の一環として、隣接するタイル化ループの融合を有効にします。このオプションはデフォルトで **false** に設定されています。

  使用法: `-mllvm -fuse-tile-inner-loop`

- **`-enable-loop-distribute-adv`**

  ループのベクトル化に影響する部分を分離することによってループのベクトル化を改良する、高度なループ分割を有効にします。このフラグはデフォルトでは無効になっています。

  使用法: `-mllvm -enable-loop-distribute-adv`

- `-floop-transform`

  ループ融合、ループ交換、ループ ブロッキング、ループ分割の改良版を含む、ループ変換のサブセットを有効にします。`-zopt` で有効です。

  使用法: `-floop-transform`

- `-faggressive-loop-transform`

  ループ内条件分岐排除、ループ タイル化、ループ バージョン管理 licm の改良版を含む、ループ変換のサブセットを有効にします。`-zopt` で有効です。

  使用法: `-faggressive-loop-transform`

- `-fstrip-mining`

  ループ ストリップ マイニング最適化を有効にします。この最適化は、大きなループを小さいセグメントに分割するかストリップして、時間的/空間的局所性を向上させます。このオプションは、`-flto.`と合わせて呼び出す必要があります。

  使用法: `-flto -fstrip-mining`

- `-ldist-scalar-expand`

  ベクトル化を向上するため、スカラー展開によるループ分割を有効にします。`-Wl,-mllvm -Wl,-ldist-scalar-expand` で呼び出すことができます。

  使用法: `-Wl,-mllvm -Wl,-ldist-scalar-expand`

数値演算オプション

- **`-convert-pow-exp-to-int={true,false}`**

  浮動小数点指数を整数に変換できる場合、浮動小数点指数バージョンの pow への呼び出しを整数指数バージョンに変換します。このオプションはデフォルトで **true** に設定されています。

  使用法: `-mllvm -convert-pow-exp-to-int={true, false}`

- **`-freciprocal-math`**

  除算 x/y を、逆数を用いた乗算 x * (1/y) に置き換えられます。これにより、複数の式で逆数 (1/y) を共有できるようになります。`-mrecip` の場合とは異なり、逆数 (1/y) を計算するときは追加の近似は使用できません。それでも、x/y を直接計算する場合と比較すると丸め誤差が大きくなります。

  使用法: `-freciprocal-math`

インライン最適化オプション

- **`-inline-recursion=[1,2,3,4]`**

  経験則に基づいて再帰関数のインライン展開を有効にします。レベル 4 が最もアグレッシブです。デフォルトのレベルは 2 です。レベルを高くすると、呼び出し元で再帰関数が拡張されるため、コードの肥大化につながることがあります。

  – レベル 1 ～ 2: インライン展開の深さを 1 として、ヒューリスティックを使用した再帰関数のインライン展開を有効にします。レベル 2 では、よりアグレッシブなヒューリスティックが使用されます。
  – レベル 3: インライン展開の深さが 1 のすべての再帰関数のインライン展開を有効にします。
  – レベル 4: インライン展開の深さが 10 のすべての再帰関数のインライン展開を有効にします。

  この最適化を実行するにはプログラム全体の解析が必要であるため、`-flto` と合わせて使用するとさらに効果的です。

  使用法: `-flto -finline-recursion=[1,2,3,4]`

メモリ レイアウト最適化オプション

- **`-fstruct-layout`**`={1,2,3,4,5,6,7,8,9}`

  プログラム全体を解析し、コードの構造体をピーリング可能か、デッド フィールドや冗長なフィールドを削除可能か、構造体に含まれるポインターまたは整数フィールドを圧縮可能かを判断します。実行可能な場合、この最適化はコードを変換し、これらの改善を有効にします。この変換はキャッシュ使用率とメモリ帯域幅を改善する可能性があります。複数のコアで実行されるプログラムのスケーラビリティ向上が見込まれます。

  この最適化を実行するにはプログラム全体の解析が必要であるため、`-flto` と組み合わせた場合のみ有効になります。アプリケーションにこの最適化を適用する際のアグレッシブさは、1 を最もアグレッシブでない、7 を最もアグレッシブなレベルとしてユーザーが選択できます。

  – **`-fstruct-layout`**`=0` は構造体のピーリングを無効にします (デフォルト)。
  – **`-fstruct-layout`**`=1` は構造体のピーリングを有効にします。
  – **`-fstruct-layout`**`=2` は、構造体のピーリングを有効にし、安全である場合に限り、これらの構造体に含まれる自己参照ポインターを 32 ビット ポインターに選択的に圧縮します。
  – **`-fstruct-layout`**`=3` は、構造体のピーリングを有効にし、安全である場合に限り、これらの構造体に含まれる自己参照ポインターを 16 ビット ポインターに選択的に圧縮します。
  – **`-fstruct-layout`**`=4` は、レベル 2 と同様に構造体のピーリングとポインター圧縮を有効にし、さらに 64 ビットの構造体フィールドを 32 ビットの整数型に圧縮します。これは厳しい安全性チェックの下で実行されます。
  – **`-fstruct-layout`**`=5` は、レベル 3 と同様に構造体のピーリングとポインター圧縮を有効にし、さらに 64 ビットの構造体フィールドを 32 ビットの整数型に圧縮します。これは厳しい安全性チェックの下で実行されます。
  – **`-fstruct-layout`**`=6` は、レベル 2 と同様に構造体のピーリングとポインター圧縮を有効にし、さらに 64 ビットの構造体フィールドを 32 ビットの整数型に圧縮します。これは厳しい安全性チェックの下で実行されます。

– **-fstruct-layout**=7 は、レベル 3 と同様に構造体のピーリングとポインター圧縮を有効にし、さらに 64 ビットの構造体フィールドを 16 ビットの整数型に圧縮します。
– -fstruct-layout=8 は、レベル 6 と同様に構造体のピーリング、ポインターの圧縮、64 ビットの整数型の圧縮を有効にし、ピーリングした構造体フィールドの最適な並び順を作成して実行時パフォーマンスを向上させます。
– -fstruct-layout=9 は、レベル 7 と同様に構造体のピーリング、ポインターの圧縮、64 ビットの整数型の圧縮を有効にし、ピーリングした構造体フィールドの最適な並び順を作成して実行時パフォーマンスを向上させます。

使用法: -flto -fstruct-layout={1,2,3,4,5,6,7,8,9}

注記:

1. **-fstruct-layout**=4 は **-fstruct-layout**=2 から、**-fstruct-layout**=5 は、**-fstruct-layout**=3 から派生し、構造体の 64 ビット整数フィールドを 32 ビット整数フィールドに安全に圧縮する機能が追加されています。**-fstruct-layout**=4 から **-fstruct-layout**=5 に変更することにより、ポインターの値が 16 ビットに圧縮可能な場合、パフォーマンスが向上する可能性があります。
2. **-fstruct-layout**=6 は **-fstruct-layout**=2 から、**-fstruct-layout**=7 は、**-fstruct-layout**=3 から派生し、構造体の 64 ビット整数フィールドを 32 ビット整数フィールドに圧縮する機能が追加されています。**-fstruct-layout**=4 や **-fstruct-layout**=5 と似ていますが、安全性を保証することなく、構造体の整数フィールドは常に 64 ビットから 32 ビットに圧縮されます。
3. この最適化は、現在 C/C++ 入力のみに制限されています。また、C/C++ オブジェクトを Fortran オブジェクトにリンクする際はこのオプションを使用しないでください。

- **-fremap-arrays**

  1 次元配列のデータ レイアウトを変換し、キャッシュの局所性を改善します。この最適化を実行するにはプログラム全体の解析が必要であるため、-flto と組み合わせた場合に有効になります (-flto -fremap-arrays として呼び出せます)。

  使用法: -flto -fremap-arrays

- **-reduce-array-computations={1,2,3}**

  配列データフロー解析を実行し、未使用の配列計算を最適化します。

  – -reduce-array-computations=1: 未使用の配列要素に対する計算を除去します。
  – -reduce-array-computations=2: 値が 0 の配列要素に対する計算を除去します。
  – -reduce-array-computations=3: 未使用かつ値が 0 の配列要素に対する計算を除去します (1 と 2 の組み合わせ)。

  この最適化を実行するにはプログラム全体の解析が必要であるため、-flto と組み合わせた場合に有効になります (-flto -reduce-array-computations={1,2,3} として呼び出せます)。

  使用法: -flto -mllvm -reduce-array-computations={1,2,3}

命令レベル最適化オプション

- **`-enable-X86-prefetching`**

  ループ内/ループ ネストの最も内側のループ内のメモリ参照に対して、x86 プリフェッチ命令の生成を有効にし、多次元配列の 2 次元目/ループ ネストの最も内側のループ内のメモリ参照をプリフェッチできるようにします。

  使用法: `-mllvm -enable-X86-prefetching`

- **`-suppress-fmas`**

  FMA の削減パターンを特定し、削減パターンへの収益性が低い場合、FMA の生成を抑制します。

  使用法: `-mllvm -suppress-fmas`

- **`-fnt-store`**

  トリップ数の多いループ内の配列アクセスについて、非一時的なストア命令を生成します。

  使用法: `-fnt-store`

- **`-fnt-store=aggressive`**

  コンパイル時に反復回数を特定できないループ内の配列アクセスについて、非一時的なストア命令を生成します。この場合、コンパイラは反復回数が大きいものとみなします。

  使用法: `-fnt-store=aggressive`

- **`-enable-redundant-movs`**

  メモリからの冗長なロード、メモリへの冗長なストアを含め、冗長な mov 操作をすべて削除します。`-Wl,-plugin-opt=-enable-redundant-movs` を使用して呼び出せます。

  使用法: `-Wl,-plugin-opt=-enable-redundant-movs`

- **`-merge-constant`**

  使用頻度の高い定数をレジスタで使用するように試みます。その目的は、定数を使用する命令の命令エンコーディングのサイズを削減し、パフォーマンスを向上させることです。

  使用法: `-mllvm -merge-constant`

- **`-mrecip={all,none,sqrt,vec-sqrt,'!'sqrt,'!'vec-sqrt,div,vec-div,'!'div,'!'vec-div}`**

  - `-mrecip={all,none}`
    all/none を使用することで、逆数演算命令の生成を有効/無効にできます。
    使用法: `-march=znver4 -mrecip=all`
    `-march=znver4 -mrecip=none`

  - `-mrecip=sqrt,vec-sqrt`
    float と double に対して逆数平方根命令を生成します。
    先頭に `'vec-'` を付けると、ベクトル型に対して逆数平方根命令を生成します。
    -march=znver4 のとき、このオプションはデフォルトでオンです。
    使用法: `-march=znver4 -mrecip=sqrt,vec-sqrt`

- `-mrecip=div,vec-div`
  float と double に対して逆数除算命令を生成します。
  先頭に `'vec-'` を付けると、ベクトル型に対して逆数除算命令を生成します。
  使用法: `-march=znver4 -mrecip=div,vec-div`
- `-mrecip='!'sqrt,'!'vec-sqrt,'!'div,'!'vec-div`
  先頭に「!」を付けると、対応する逆数演算命令が生成されなくなります。
  使用法: `-march=znver4 -mrecip='!'sqrt,'!'vec-sqrt,'!'div,'!'vec-div`

これらの近似命令は、`-funsafe-math-optimizations` と `-ffinite-math-only` が有効になっている (これらは `-ffast-math` または `-Ofast` によって暗黙的に有効になる) ときのみ生成されます。

- **`-optimize-strided-mem-cost`**

  ストライドによるメモリ アクセスに対してコスト モデルを最適化します。

  使用法: `-mllvm -optimize-strided-mem-cost`

- **`-fenable-aggressive-gather`**

  このオプションは、収益性が高い場合にギャザー命令の生成を有効にします。

  使用法: `-fenable-aggressive-gather`

スカラー最適化オプション

- **`-enable-licm-vrp`**

  ループ不変コードを移動する前に、仮想レジスタ プレッシャーを推定できるようにします。この推定値は、ループ不変コードの移動中に巻き上げられループ不変条件の数を制御するために使用されます。

  使用法: `-mllvm -enable-licm-vrp`

- **`-do-block-reorder={none,simple,aggressive}`**

  制御述語を外側から内側に向かって複雑さが増すように並べ替えます。このオプションはデフォルトで **`simple`** に設定されています。**`simple`** モードではシンプルな式を並べ替えますが、aggressive モードでは例外処理を扱うコードがあったとしても、関数呼び出しを含む述語を並べ替えます。

  この最適化には、基本ブロックを並べ替えても安全かどうかをチェックする安全解析も含まれています。ただし、この最適化が有効になると、安全解析は、並べ替え対象のブロックのコンテキストに関係する呼び出しチェーン内で発生した例外を特に無視します。これはほとんどの入力プログラムでは許容されますが、プログラム内のすべてのポイントで例外のスローを厳密にサポートする必要のあるプログラムでは、このオプションの使用は避けることを推奨します。

  使用法: `-mllvm -do-block-reorder={none, simple, aggressive}`

- **`-fscalar-transform`**

  巻き上げおよび不変コードの移動など、さまざまなコード移動最適化の改良版を含む、スカラー変換のサブセットを有効にします。`-zopt` で有効です。

  使用法: `-fscalar-transform`

プロファイルに基づく最適化オプション

- `-fprofile-instr-generate`

  実行回数を収集して <file> (`LLVM_PROFILE_FILE` 環境変数によってオーバーライドされる) に書き込む計測用コードを生成します。

- `-fprofile-instr-use=<profiled data file>`

  `-fprofile-instr-generate` を使用してコンパイルしたプログラムで生成されるプロファイリング ファイルを、最適化に関する決定の指針として使用します。

使用法:

1. ```
   $clang -fprofile-instr-generate code.c
   $LLVM_PROFILE_FILE=pfname.raw ./code.out
   $llvm-profdata merge -output=pfname.data pfname.raw
   $clang -fprofile-instr-use=pfname.data code.c -o code.out
   ```
2. ```
   $clang -fprofile-instr-generate=pfname.raw code.c -o code.out
   $llvm-profdata merge -output=pfname.data pfname.raw
   $clang -fprofile-instr-use=pfname.data code.c -o code.out
   ```

その他のオプション

- **`-fitodcalls`**

  条件付き呼び出しを配置することにより、間接呼び出しを直接呼び出しに昇格させます。この最適化は、呼び出しパラメーターとして受け渡される少数の確定的なターゲット関数または関数ポインターのセットを持つアプリケーションまたはベンチマークで有益です。間接呼び出しから直接呼び出しへの昇格では、ランタイム チェック時は利用可能なすべての決定済みターゲットを使用し、それ以外のすべてのケースでは元のコードにフォール バックするようにコードを変換します。ランタイム チェックは、これらの利用可能な関数ポインター ターゲットのそれぞれについてコンパイラによって導入され、その後、ターゲットの直接呼び出しが実行されます。

  これは、`-flto -fitodcalls` で呼び出されるリンク時最適化です。

  使用法: `-flto -fitodcalls`

- **`-fitodcallsbyclone`**

  引数として渡される関数ポインターを持つ関数について、値の特殊化を実行します。この特殊化は関数のクローンを生成することによって実行されます。関数のクローン作成は、間接関数呼び出しから直接呼び出しへの変換を可能にするために、必要に応じて呼び出しチェーン内で実行されます。これは、`-fitodcalls` 最適化を補完するもので、リンク時最適化でもあります (`-flto -fitodcallsbyclone` として呼び出せます)。

  使用法: `-flto -fitodcallsbyclone`

- **`-function-specialize`**

  コンパイル時定数仮引数によって関数を最適化します。-O3 では、このオプションはデフォルトで有効になっています。

  使用法: `-mllvm -function-specialize`

- **`-favoid-fpe-causing-opt`**

  浮動小数点例外につながるいくつかの最適化を制限します。

  使用法: `-favoid-fpe-causing-opt`

診断オプション

- **`-fsanitize`**

  診断のためにサニタイザーを実行します。

  使用法: `-fsanitize={address,thread,memory,safe-stack}`

5.4 ランタイム環境変数

AOCC_SUPPRESS_IEEE_WARNINGS

Fortran プログラム終了時に、IEEE 754 浮動小数点例外に関連する警告メッセージを表示するランタイム環境変数です。警告メッセージはデフォルトで表示されます。

変数を設定するには、次を実行します。

```
$ export AOCC_SUPPRESS_IEEE_WARNINGS=1
```

変数の設定を解除するには、次を実行します。

```
$ unset AOCC_SUPPRESS_IEEE_WARNINGS
```

この環境変数を使用して抑制可能な IEEE 警告は次のとおりです。

- ieee_invalid のシグナル通知
- ieee_denorm のシグナル通知
- ieee_divide_by_zero のシグナル通知
- ieee_overflow のシグナル通知
- ieee_underflow のシグナル通知
- ieee_inexact のシグナル通知

5.5 非推奨のオプション

非推奨のオプションは次のとおりです。

- `-vectorize-memory-aggressively` (AOCC 2.2.0 以降)
- `-Menable-vectorize-pragmas=<value>` (AOCC 3.2.0 以降)

第 6 章 デバッグ可能性

6.1 OMPD (OpenMP Debugging Support: OpenMP デバッグ サポート)

注記: これは、AOCC 2.3 またはそれ以降で利用できます。

AOCC インストールには、機能が制限された gdb プラグインを使用して C/C++ OpenMP プログラムをデバッグするための OMPD が含まれています。

注記: オフロード デバイス上で実行されるコードのデバッグはサポートされていません。

OMPD を使用して gdb プラグイン経由で C/C++ OpenMP プログラムをデバッグするには、次の手順を実行します。

注記: OMPD プラグインを使用するには、Python 3.5 またはそれ以降が必要です。

1. 次のコマンドを使用して、フォルダー **ompd** と **lib** を **LD_LIBRARY_PATH** に追加します。
   ```
   $ export LD_LIBRARY_PATH=<compdir>/aocc-compiler-<ver>/ompd:<compdir>/aocc-compiler-<ver>/
   lib:$LD_LIBRARY_PATH
   ```
2. OMP_DEBUG を enabled に設定します。
   ```
   $ export OMP_DEBUG=enabled
   ```
3. 次のサンプル C ソース ファイル *xyz.c* の例に示すとおり、**-g** オプションと **-fopenmp** オプションを付けてデバッグ用にプログラムをコンパイルします。
   ```
   $ <compdir>/aocc-compiler-<ver>/bin/clang -g -fopenmp xyz.c -o xyz.out
   ```
   注記: OpenMP 固有のデバッグが正しく動作するには、デバッグ対象のプログラムに <compdir>/aocc-compiler-<ver>/lib 配下の libomp.so へのダイナミック リンク依存性が必要です。これは、生成されたバイナリ、xyz.out に対して ldd を使用することでチェックできます。
4. 次のようにプラグインで gdb を起動し、バイナリ xyz.out をデバッグします。
   ```
   $ gdb -x <compdir>/aocc-compiler-<ver>/ompd/__init__.py ./xyz.out
   ```
   注記: プラグイン <compdir>/aocc-compiler-<ver>/ompd/__init__.py を使用する必要があります。

6.2 OMPD コマンド

次の表に OMPD コマンドの説明を示します。

表 3. OMPD コマンド

コマンド	説明
`help ompd`	OpenMP 固有のデバッグに利用可能なサブコマンドの一覧を表示します。
`ompd init`	• $LD_LIBRARY_PATH 環境変数で利用可能な libompd.so を読み込み、OMPD ライブラリを初期化するために、最初に実行します。 • プログラムの実行が開始され、OpenMP 内部位置 ompd_dll_locations_valid() の一時ブレークポイントで停止します。 • 一時的ブレークポイントからデバッグを再開できます。 • OpenMP 内部位置 ompd_bp_thread_begin および ompd_bp_thread_end にブレークポイントを配置し、開始イベントと終了イベントを捕捉できます。 • ompd_bp_task_begin および ompd_bp_task_end ブレークポイントを使用すると、イベントの開始と終了を捕捉できます。 • ompd_bp_parallel_begin と ompd_bp_parallel_end を使用すると、並列イベントの開始と終了を捕捉できます。

6.3 OMPD サブコマンド

次の表に、GDB 内の有効な OpenMP 領域内で使用できる OMPD サブコマンドの一覧を示します。

表 4. OMPD サブコマンド

サブコマンド	説明
`ompd init`	OMPD ライブラリを見つけ、初期化します。
`ompd bt`	bt の出力のオン/オフについてフィルターを *on* または *off* にするために使用します。 ワーカー スレッドをマスター スレッドにトレースするには、*on continued* オプションを指定する必要があります。
`ompd icvs`	内部制御変数の値を表示します。
`ompd parallel`	現在の並列領域とそれを囲む並列領域の詳細を表示します。
`ompd step`	ステップ実行し、ランタイム フレームを可能な限りスキップします。
`ompd threads`	現在のスレッドの詳細を提供します。

注記: OMPD コマンドを OMPD 領域外で実行した場合、その動作は未定義となります。

第 7 章 診断機能

7.1 AOR (AOCC Optimization Report: AOCC 最適化レポート)

AOR ツールは、AOCC でコンパイルされた任意の C、C++、Fortran アプリケーション プロジェクトについて最適化レポートを生成します。アプリケーション開発者は最適化レポートを調査することで、コンパイルされたアプリケーションで実行された/されなかった最適化のリストを確認し、今後の対策のヒントを得ることができます。現在、AOR には loop-vectorize 最適化の合格情報に関連するメッセージレポートが含まれており、次のコマンドで生成できます。

- `-fgen-aor`

  コンパイル時の AOR 生成 (CFLAGS、CXXFLAGS、FCFLAG)。

- `-fgen-aor -flto -fuse-ld=lld`

  リンク時の AOR 生成 (CFLAGS、CXXFLAGS、FCFLAG)。

- `-fgen-aor-screen-listing`

  コンパイル時とリンク時の AOR 画面表示 (CFLAGS、CXXFLAGS、FCFLAG)。これをフラグ `-fgen-aor` と合わせて使用すると、AOR レポートが表示されます。

例:

```
$ git clone https://github.com/UoB-HPC/TSVC_2
$ cd TSVC_2/src
$ clang -O3 -march=znver2 -fgen-aor -flto -fuse-ld=lld *.c -I. -o tsvc -lm
```

上記のコマンドは、それぞれコンパイル時レポート *optimization_report.aor* とリンク時レポート *optimization_ld_report.aor* をアプリケーション プロジェクト ビルド ディレクトリに生成します。

次のコマンドは、それぞれコンパイル時レポート *optimization_report.aor* とリンク時レポート *optimization_ld_report.aor* をアプリケーション プロジェクト ビルド ディレクトリに生成し、AOR レポートを表示します。

```
$ git clone https://github.com/UoB-HPC/TSVC_2
$ cd TSVC_2/src
$ clang -O3 -march=znver2 -fgen-aor -fgen-aor-screen-listing -flto -fuse-ld=lld *.c -I. -o tsvc
-lm
```

AOR アーティファクトの内容例

```
*******************************************************************************************

*******************************************************************************************

Source location : sv.c:470:5

Optimizer Passname : loop-vectorize

Function Name : Perl_reg_temp_copy

loop-vectorize Optimization Status : Failed

Reason for loop-vectorize Optimization Failure : the cost-model indicates that interleaving is
not beneficial

*******************************************************************************************

*******************************************************************************************

Source location : mg.c:396:2

Optimizer Passname : loop-vectorize

Function Name : Perl_regexec_flags

loop-vectorize Optimization Status : Failed

Reason for loop-vectorize Optimization Failure :  value that could not be identified as
reduction is used outside the loop

*******************************************************************************************

*******************************************************************************************
```

使用要件

使用要件の詳細は、表 2 を参照してください。

AOCC コンパイラがインストールされており、環境変数が設定されている必要があります。詳細は、3.1.2 インストールを参照してください。

別の方法として、AOCC コンパイラと AOR ツール (opt-viewer.py) パスをエクスポートし、AOR アーティファクトの生成に正しいコンパイラと関連ツールが使用されているか確認することもできます。

第 8 章 サポート

サポート オプション、最新の資料、ダウンロードの詳細は、次の URL を参照してください。

https://www.amd.com/ja/developer/aocc.html

第 9 章 参考資料

本資料の参考として、LLVM 16.0.0 の資料を使用しました。