



インテル® oneAPI ポーティング・ガイド

DPCPP または ICX へ移行する ICC ユーザー向け

IntelCorporation

www.intel.com(英語)

[著作権と商標について](#)

注意事項:

この日本語マニュアルは、インテルコーポレーションのウェブサイト <https://software.intel.com/content/www/us/en/develop/articles/porting-guide-for-icc-users-to-dpcpp-or-icx.html> (英語)で公開されている『Porting Guide for ICC Users to DPCPP or ICX』(更新日:2021/04/27: 2021.2 対応版)の参考訳です。

インテル社の許可を得て iSUS (IA Software User Society)が翻訳版を作成した iSUS の著作物です。

原文は Intel Corporation の Copyright であり、日本語参考訳版にも適用されます。

目次

著作権と商標について

用語

ICX に対する指針

コンパイラーのデフォルトに関する大きな変更

重要な新しいオプション

コンパイラー・バージョン

重要なコンパイラー・オプションの対応付け

プラグマのサポート

事前定義マクロのサポート

ビルトイン関数

プリコンパイルされたヘッダーファイルのサポート

診断オプションと診断メッセージ番号の変更

リンク、IPO、PGO に関する変更

言語機能

組込み関数の使用モデルに関する変更

インテル固有のプロセッサ・ターゲット・プラグマと関数のサポート

浮動小数点結果の再現性の制御

ブルータスマたはバイセクション最適化のサポート

付録:リファレンス

著作権と商標について

インテルのテクノロジーを使用するには、対応したハードウェア、ソフトウェア、またはサービスの有効化が必要となる場合があります。

絶対的なセキュリティを提供できるコンピューター・システムはありません。

実際の費用と結果は異なる場合があります。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

最適化に関する注意事項

インテル® コンパイラーでは、インテル® マイクロプロセッサに限定されない最適化に関して、他社製マイクロプロセッサ用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサ依存の最適化は、インテル® マイクロプロセッサでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサ用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。

注意事項の改訂 #20110804

このポーティング・ガイドは、新しい LLVM ベースのインテル® oneAPI DPC++/C++ コンパイラー (dpcpp および icx)に移行するインテル® C++ コンパイラー・クラシック(icc/icl)ユーザー向けに情報と提案を提供します。

用語

簡潔性と明確性のため、次に示すようにこのドキュメントではいくつかの非公式の用語を使用します。

- ICX - インテル® oneAPI DPC++/C++ コンパイラー
- ICC クラシック - インテル® C++ コンパイラー・クラシック
- DPCPP - インテル®oneAPI DPC++/C++コンパイラー

DPCPP は、ベースとなる C++ コンパイラーとして ICX を使用するため、ICX の大部分の情報は DPCPP にも適用されます。

ICX に対する指針

以下に ICX に対する指針を示します。

- ICX と ICC/ICL は異なるコンパイラー・ドライバーを使用します。ICC クラシック・ドライバーは、icc、icpc、および icl です。ICX ドライバーは、icx であり、C と C++ をコンパイルします。また、従来の icpc ドライバーに対応するドライバー icpx もありますが、これは単純に ICX を呼び出します。

ICX は、コア C++ コンパイラーであるとともにインテル® oneAPI DPC++ コンパイラーと新しい dpcpp ドライバーのベース・コンパイラーでもあります。

ICX は新しいコンパイラーです。このコンパイラーは、ICC/ICL と比較して機能面および動作上の違いがあります。ICC/ICL を使用する既存のアプリケーションを ICX に移行する場合、ある程度の移植作業が必要になると予測されます。

- ICC/ICL クラシックから ICX への移行自体は、スムーズで簡単です。しかし、既存のアプリケーションを ICC/ICL クラシックから ICX に移行してチューニングする必要があります。

コンパイラーのデフォルトに関する大きな変更

- **データ並列 C++ (DPC++) アプリケーション:** ベース・コンパイラーは ICX/ICL で、ドライバーは `dpcpp` です。
- **標準 C++ アプリケーション:** ICC/ICL クラシック・コンパイラーの使用を推奨します。ドライバーは、`icc/icpc/icl` です。

コンパイラーのデフォルトに関する大きな変更

コンパイラーのデフォルトに関する大きな変更を以下に示します。

- ICC/ICL クラシック・コンパイラー製品のユーザーは、引き続き `icc`、`icpc` または `icl` ドライバーを使用できます。
- DPC++ ユーザーは、DPC++ 拡張機能を備えた ICX を起動する `dpcpp` ドライバーを使用できます。
- Clang とは異なり、ICX のデフォルト浮動小数点モデルは ICC/ICL と一致するように設定されており、デフォルトは `-fp:fast` です。

マクロ名が変わっています。ICX および DPCPP に含まれる将来のマクロについては、リリースノートを参照してください。

ICC/ICL クラシックで使用される `__INTEL_COMPILER` マクロの代わりに、ICX では `__INTEL_LLVM_COMPILER` マクロが定義されています。コンパイラーで定義されているすべてのマクロの値を確認するには、`-E-dM` オプションを使用して `.ii` ファイルを生成するか、`stderr` に値を出力できます。

以下は、コンパイラーが `hello.ii` ファイルを生成するデータ並列 C++ の例です。

```
dpcpp -E -dM ./hello.cpp
more hello.ii
```

ICX の場合 (stderr に出力)

```
icx -E -dM ./hello.cpp
```

- リマーク、警告/メモの診断番号は含まれません。各診断は、その機能を無効にするコンパイラー・オプションとともに出力されます。

- ICC/ICL クラシックとは異なり、コンパイラーの組込み関数は、プロセッサ・ターゲット・オプションを指定しないと自動認識されません。組込み関数を使用する場合、後述の「[組込み関数の使用モデルに関する変更](#)」の説明をお読みください。

重要な新しいオプション

インテルのアナライザーとプロファイル・ツール向けのオプション

アナライザーには以下のすべてのオプションを使用してください。

- `-gline-tables-only`
 - このオプションは、プロファイル・ツールに役立ちます。
 - 行テーブルデバッグ情報のみを生成します。
 - これにより、インライン情報を含むシンボリック・バックトレースが可能ですが、変数とその位置および型に関する情報は含まれません。
- `-fdebug-info-for-profiling`
 - より正確なプロファイルを可能にする情報を追加します。

ICX の OpenMP* オプション

- `-fiopenmp`
 - OpenMP* `parallel` 構造と SIMD プラグマ/ディレクティブを認識してコンパイルし、インテルの OpenMP* ランタイム・ライブラリーを使用します。
- `-fopenmp` (非推奨)
 - OpenMP* `parallel` 構造と SIMD プラグマ/ディレクティブを認識してコンパイルし、オープンソースの OpenMP* ランタイムを使用します。パフォーマンスを目的としない互換性テストのみに使用してください。パフォーマンスと機能を求めるには、`-fiopenmp` を使用します。
- `-fopenmp-targets=spir64`
 - OpenMP* 4.5/5.0 の TARGET プラグマ/ディレクティブを使用する場合、このオプションを指定する必要があります。

コンパイラー・バージョン

- OpenMP* 4.5/5.0 の TARGET ディレクティブは、インテル® oneAPI HPC ツールキットに含まれる ICX コンパイラーのみが認識できます。上記の 2 つのコンパイラー・オプションを同時に指定します。
 - `icx-fiopenmp-fopenmp-targets=spir64`
 - `icpx-fiopenmp-fopenmp-targets=spir64`

コンパイラー・バージョン

データ並列 C++ と ICX のバージョン文字列

2つのLLVMベースのコンパイラー `dpcpp` と `icx` のバージョン文字列は新しいものです。次に例を示します。

```
dpcpp --version
DPC++ Compiler 2021.2 (2021.2.0.20210317)
```

形式は、MAJOR.MINOR.PATCH(ビルド番号)です。

説明:

- MAJOR は、製品バージョン(年)です。
- MINOR は、"1" から始まる 1 桁のマイナーバージョン番号です。"1" は最初のリリースで、新しいマイナーリリースが提供されるたびに 1 ずつ増えます。
- PATCH は "0" から始まり、特定のバグとセキュリティーの問題が解決されるごとに数値が増加します。
- (ビルド番号)は次のように表現されます(YYYYMMDD)。

重要なコンパイラー・オプションの対応付け

次に重要なコンパイラー・オプションの対応状況を示します。

- ICX ドライバー `icx` と `icpx` は、ICC/ICL クラシック・コンパイラーのオプションおよび Clang/LLVM コンパイラーのオプションを受け入れます。

- Clang/LLVM コンパイラー・オプションは直接解釈されます。
- ICX に渡された ICC/ICL クラシック・コンパイラーのオプションは、可能であれば対応する Clang/LLVM オプションに変換されます。
- すべての ICC/ICL クラシック・コンパイラーのオプションが ICX で認識/実装されるわけではありません。
- ICC/ICL クラシック・コンパイラーのドキュメントに明記されていないオプションは、実装されません。また、今後も実装される予定はありません。ICX コンパイラーは ICC/ICL クラシック・コンパイラーとは大きく異なるということを忘れないでください。ドキュメント化されていない内部オプションに必要と思われる機能がある場合は、想定される動作と ICX でそれがサポートされないことを[オンライン・サービス・センター\(OSC\)](#) (英語)に報告してください。「Makefile でこのオプションを使用しており、ICC/ICL ではこのオプションが使用できた」というようなことは、正当な理由とは認められません。ICX コンパイラーの最適化と動作は、ICC/ICL クラシック・コンパイラーとは異なります。オプションを指定せずに ICX を試してみてください。
- ICC/ICL クラシック・コンパイラー・オプション:診断の警告は ICC/ICL クラシック・コンパイラーのオプションでは出力されますが、現時点では ICX で実装される予定はありません。

```
Command line warning #10430: Unsupported command line
options encountered
These options as listed are not supported with the compiler
selected.
For more information, use '-qnextgen-diag'.
```

- ICX オプション `-qnextgen-diag` (Linux*)、`/Qnextgen-diag` (Windows*)を指定すると、ICX で受け入れられない ICC/ICL クラシック・コンパイラー・オプションのリストを表示します。
- サポートされていないオプションがアプリケーションにとって重要であり、そのオプションを ICX のビルドオプションから除くとエラーになる場合は、[オンライン・サービス・センター\(OSC\)](#) (英語)に問題として報告してください。ICX に実装する必要があるオプションについて、皆様のご意見をお聞かせください。
- 実装済みまたは間もなく実装される予定の ICC/ICL クラシック・コンパイラーのオプションに対してメッセージは表示されません。
- ICX の Clang バージョンでサポートされている Clang/LLVM オプションはすべて認識および実装されます。しかし、場合によってはオプションを Clang へ渡す必要があります。Clang へオプションを直接渡すには、次のオプションを使用します。
 - `-Xclang<Clang へのオプション>`

プラグマのサポート

- オプションに引数がある場合は、複数の `-Xclang` オプションを指定します。例えば、`-target-feature +aes` オプションを渡すには、`-Xclang -target-feature -Xclang +aes` のようにします。
- `-Xclang` オプションは、Linux* と Windows* の両方で使用できます。
- GNU* および Microsoft* 互換オプションは、ICC/ICL クラシック・コンパイラーと ICX で認識されます。

プラグマのサポート

ICC/ICL または GCC プラグマが、ICX でサポートされると想定しないでください。

ICC/ICL クラシック・コンパイラーでは、多くのインテル独自プラグマがサポートされています。OpenMP* に関連するプラグマを除き、インテル独自プラグマのサブセットが ICX でサポートされています。そのため、移植作業の最初のステップとして、サポートされないプラグマを特定することを推奨します。

ICX コンパイラーの `-Wunknown-pragmas` オプションを使用して、サポートされていないプラグマをチェックすることができます。

```
icx -Wunknown-pragmas<file>
```

次の例について考えてみます。

```
cat unknown-pragmas.c

int main(void) {
float arr[1000];

#pragma totallybogus
#pragma simd
#pragma vector
    for (int k=0; k<1000; k++) {
        arr[k] = 42.0;
    }
}

-c -Wunknown-pragmas unknown-pragmas.c

unknown-pragmas.c:4:9: warning: unknown pragma ignored [-Wunknown-
pragma]
#pragma totallybogus
    ^
```

```
unknown-pragmas.c:5:9: warning: unknown pragma ignored [-Wunknown-pragmas]
#pragma simd
      ^
2 warnings generated.
```

この例では 2 つのことが警告されています。

- “#pragma totallybogus” は、ICC/ICL クラシック、GCC、および ICX ではサポートされないプラグマです。そのため、警告として示されるのは当然のことでしょう。
- “#pragma simd” は、ICC/ICL クラシックでサポートされていたプラグマですが、ICX ではサポートされません。ICX はこれらのプラグマを無視し、プログラマーが ICC/ICL クラシック・コンパイラーに期待することを行いません。ここでは、pragma SIMD を pragma omp simd に置き換える必要があります。

最後に “#pragma vector” は、ICX で認識され実装されているため警告は出力されません。

事前定義マクロのサポート

マクロは動的に追加されています。次のコマンドを使用して、特定のバージョンの ICX で定義されるマクロを表示できます。

```
icx -x c /dev/null -dM -E (Linux*)
icx /#x /QdM -E (Windows*)
```

ビルトイン関数

Clang のビルトイン関数については、オープンソースの Clang ドキュメントで説明されています。

プリコンパイルされたヘッダーファイルのサポート

ICX は、“リロケータブルな” プリコンパイル・ヘッダーの生成をサポートします。これらは、指定されるビルド・ディレクトリーのパスでビルドされ、後に生成された場所から使用されます。

診断オプションと診断メッセージ番号の変更

--relocatable-pch オプションは、この機能を有効にします。詳細については、「[リロケートブル PCH ファイル](#)」(英語)を参照してください。

ICC/ICL クラシック・コンパイラーではプリコンパイル済みヘッダーの利用に制限がありましたが、ICX では大幅に改善されています。ICC/ICL クラシック・コンパイラーの制限に関する詳細は、「[マップされたメモリーを取得できない](#)」(英語)をご覧ください。

ICX は、プリコンパイル済みヘッダー(PCH)の作成と参照に Clang のアプローチを使用します。これは、2つのステップで行われます。

1. PCH を作成します(Linux* の icc の例を示しますが、Windows* でも同様です)。

```
icx -x c-header file.h // file.h.gch を作成
```

2. PCH を参照します。

```
icx -include-pch file.h.gch file.c //file.c をコンパイルする際に  
PCH ファイルを参照します
```

診断オプションと診断メッセージ番号の変更

以下に現在サポートされているコンパイラーの診断オプションを示します。

Linux* オプション	Windows*オプション	新しい名前
-diag-<type>	/Qdiag	未サポート、詳細は以下を参照
-diag-dump	/Qdiag-dump	未サポート
-diag-enable=power	/Qdiag-enable:power	未サポート、検討中
-diag-error-limit	/Qdiag-error-limit	-fmax-errors=<arg>
-diag-file	/Qdiag-file	-serialize-diagnostics <filenamearg>
-diag-file-append	/Qdiag-file-append	未サポート

-diag-id-numbers	/Qdiag-id-numbers	未サポート
-diag-once	/Qdiag-once	未サポート

diag-<type> オプションと診断メッセージ番号はサポートされません。LLVM テクノロジーベースのインテル® oneAPI DPC++/C++ コンパイラーは、説明的なフレーズを使用して診断メッセージを分類します。Clang のマニュアルには、診断を有効/無効にするのに利用できる説明的なフレーズの一覧が示されます。詳細については、「[Clang の診断フラグ](#)」(英語)を参照してください。

Linux* と Windows* コンパイラーの両方に同等の診断制御オプションが用意されています。ここでは、Linux* オプションを使用して説明します。前述の表に示す Windows* オプションに置き換えて、Windows* で診断を制御できます。

例えば、次の行が含まれる unknown-pragma.c ファイルのテストケースについて考えてみます。

```
#pragma unknown_pragma
```

icc を使用してコンパイルすると、次の警告メッセージが出力されます。

```
icc- c unknown-pragma.c
unknown-pragma.c(1): warning #161: unrecognized #pragma
      #pragma unknown_pragma
```

警告 #161 は、認識できないプリAGMAの診断を無効にする ICC クラシックの -diag-disable:161 オプションで無効にすることで出力しないようにできます。

ICX では番号付きの診断メッセージが出力されませんが、診断を制御するために使用できる診断オプションに関連するヒントが示されます。-Wall オプションを使用して、プログラムに関連するすべての警告メッセージを出力できます。警告メッセージには、診断を有効/無効にするオプションが示されます。

ICX では、不明なプリAGMA診断はデフォルトで出力しないように設定されています。有効にするには、-Wunknown-pragmas オプションを、無効にするには、-Wno-unknown-pragmas オプションを使用します。-Wno- プリフィクスを使用して、特定の診断を無効にできます。

リンク、IPO、PGO に関する変更

```
icx -Wall -c ~/unknown-pragma.c
unknown-pragma.c:1:9: warning: unknown pragma ignored
[-Wunknown-pragmas]
#pragma unknown_pragma
```

ICC クラシック・プロジェクトからアプリケーションの診断制御オプションを移行するには、ICX を使用してソースをビルドし、診断の出力を読み取る必要があります。表示したくない診断を無効にする `-Wno-` オプションに指定できるキーワードを探し、それらのオプションを使用するようにビルド手順を変更します。

診断の重要度を警告からエラーに上げるには、`-Werror=unknown-pragmas` オプションを使用します。これは、ICC クラシックの `-diag-error:161` オプションに相当します。ICX では、エラーメッセージの重要度を下げる方法はありません。

Clang で強化された診断について

Clang コンパイラーの開発者は、より有用な診断メッセージの作成に努めてきました。Clang の診断はいくつかの点で改善されています。

- 可読性を向上する色付けされた診断。プログラムのソースコードと診断テキストを明確に区別できます。
- 行番号と列番号を含む正確なソース位置情報と関連テキスト範囲のハイライト表示。
- 報告された問題の修正方法に関するヒント。
- 構文エラーからの回復が強化され、問題を正確に報告できるようになりました。コンパイルを続行して他の問題をさらに検出できます。

詳細については、「[Clang の表現豊かな診断](#)」(英語)を参照してください。

リンク、IPO、PGO に関する変更

ICXコンパイラーは、プロシージャー間の最適化(IPO)とプロファイルに基づく最適化(PGO)に関して、クラシック・コンパイラーとは異なるアプローチを採用しています。これらの機能を使用している場合は、次の点に注意してください。

- PGO:LLVM は、PGO に関して全く異なるアプローチを採用しています。詳細は、「[マルチステージ PGO](#)」(英語)を参照してください。

- IPO:LLVM は、ICC/ICL クラシック・コンパイラーの「プロシージャー間の最適化 (IPO)」に相当するリンク時の最適化 (LTO - Link Time Optimization) テクノロジーを採用しています。
 - LLVM LTO の詳細については、「[LLVM リンク時の最適化の設計と実装](#)」(英語)を参照してください。
 - インテルの「xi*」リンカーツール(xilink、xild、xiar)は、インテル® コンパイラーが使用する独自のオブジェクト・ファイル形式向けであるため、ICX からは削除されました。
 - Makefile やプロジェクトの設定で「xilink」や「xild」を使用している場合は、同等のネイティブリンカーに置き換えてください。同様に、「xiar」は「ar」などのアーカイバーに置き換えます。

言語機能

インテル® Cilk™ Plus は、ICX コンパイラーではサポートされません。そのため、インテル® Cilk™ Plus プログラムは、OpenMP* またはインテル® TBB へ移行してください。

詳細については、「[インテル® Cilk™ Plus アプリケーションを OpenMP* もしくはインテル® TBB へ移行する](#)」を参照してください。これには、ICC/ICL クラシック・コンパイラー向けにチューニングされた多くのコードで使用される `#pragma simd` が含まれます。`pragma simd` は、OpenMP* SIMD プラグマ(`pragma omp simd`)に置き換える必要があります。OpenMP* SIMD プラグマは、`o2` または `o3` オプションが指定された場合、もしくは `-fopenmp-simd` オプションが指定される場合に認識されます。

組込み関数の使用モデルに関する変更

- ICX は、インライン展開時に組込み関数への引数の型チェックを行います。ICC/ICL クラシック・コンパイラーでは行いません。そのため、ICX では ICC/ICL クラシック・コンパイラーでは表示されなかった組込み関数の引数に関連する警告やエラーが表示されることがあります。
- ICC/ICL クラシック・コンパイラーでは、`__INTEL_COMPILER_USE_INTRINSIC_PROTOTYPES` マクロを定義すれば、`immintrin.h` ヘッダーファイルを使用する必要はありません。

組込み関数の使用モデルに関する変更

- また、ICC/ICL クラシック・コンパイラーでは、特定の組込み関数を使用するため、対応するプロセッサ/アーキテクチャー固有のコンパイラー・オプションを有効にする必要はありません。

ICC/ICL クラシック・コンパイラーが提供するこの 2 つの機能は、ICX では提供されません。

- ICX で組込み関数を使用するには、次の手順に従ってください。
 - コンパイラーがプロセッサ/アーキテクチャー固有の組込み関数を認識できるように、`-march` または `-m<arch>`、あるいは `-x<arch>/-ax<arch>` コンパイラー・オプションを使用する必要があります。

組込み関数に関する ICC/ICL クラシック・コンパイラーとの互換性は引き続き評価中です。最新のアップデートについては、リリースノートを確認してください。

- 組込み関数を定義する `immintrin.h` ヘッダーファイルをインクルードします。

LLVM の組込み関数の処理の違いの例

型チェックの方法は次の例を参照してください。ICX は、引数の型チェックを行います。ICC/ICL クラシック・コンパイラーでは引数エラーを見逃します。

```
cat sample mm prefetch.c
#include <immintrin.h>
#define CACHE_LINE_SIZE 64
__attribute__((always inline))
inline void Prefetch_Block(const void* addr, size_t sz, int hint)
{
    char* pref_addr = (char*)addr;
    size_t pref_iters = (sz + CACHE_LINE_SIZE - 1) / CACHE_LINE_SIZE;
    for (int i = 0; i < pref_iters; i++)
    {
        mm_prefetch(pref_addr, hint /* MM_HINT_T1 */);
        pref_addr += CACHE_LINE_SIZE;
    }
}
$ icc -c sample_mm_prefetch.c
$ icx -c sample_mm_prefetch.c
sample mm prefetch.c:13:9: error: argument to ' builtin prefetch'
must be a constant integer
    mm_prefetch(pref_addr, hint /* MM_HINT_T1 */);
    ^~~~~~
```

```

/nfs/pdx/disks/cts2/tools/compiler/cpro/Compiler/19.1/initial/compiler
s_and_libraries_2020.0.166/linux/lib/clang/10.0.0/include/xmmintrin.h:
2103:31: note:
    expanded from macro '_mm_prefetch'
#define _mm_prefetch(a, sel) ( __builtin_prefetch((void *) (a), \
    ^
1 error generated.
compilation aborted for sample_mm_prefetch.c (code 1)

```

この場合、`_mm_prefetch` の 2 番目の引数は `const` でなければなりません。組込み関数 `mm_prefetch` のドキュメントでは明記されていません。組込み関数は `const` 引数に対して定義されています。

ICC/ICL クラシックは型チェックを行いませんが、ICX は型チェックを(正しく)行っていることに注目してください。

この違いを以下に示します。

現在、エラー診断の ISA に関する推奨事項は正しくありません。この問題は、オープンソース・コミュニティに報告済みであり、修正待ちです。

```

$ cat intrinsic.cpp

#include<iostream>
#include<immintrin.h> //ICX ではインクルードする必要があります
using namespace std;

void add_sse(float *a, int N){
    __m128 x, y;
    y = _mm_set_ps(1.f);
    for(int i = 0; i < N/4; i++)
    {
        x = _mm_load_ps(a);
        x = _mm_add_ps(x, y);
        _mm_store_ps(a, x);
        a+=4;
    }
}

void add_avx(float *a, int N){
    __m256 x, y;
    y = _mm256_set_ps(1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f);
    for(int i = 0; i < N/8; i++)
    {
        x = _mm256_load_ps(a);
        x = _mm256_add_ps(x, y);
        _mm256_store_ps(a, x);
        a+=8;
    }
}

void add_avx512(float *a, int N){

```


組み込み関数の使用モデルに関する変更

```
        m512 x, y;
        y = _mm512_set_ps(1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f,
1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f);
        for(int i = 0; i < N/16; i++)
        {
            x = _mm512_load_ps(a);
            x = _mm512_add_ps(x, y);
            _mm512_store_ps(a, x);
            a+=16;
        }
}

int main(){
    float a[32];
    for(int i = 0; i < 32; i++)
        a[i] = i;
    #ifdef SSE
        add_sse(a,32);
    #elif AVX
        add_avx(a,32);
    #else
        add_avx512(a,32);
    #endif
    std::cout<<"a[15] = "<<a[15]<<"\n";
    return 0;
}
```

上記のコードは、ICC/ICL クラシックでは問題なくコンパイルされますが、ICX コンパイラーでは異なります。次に ICX の例を示します。

```
$ icpx intrinsic.cpp -DSSE
intrinsic.cpp:19:13: error: always inline function '_mm256_set_ps'
requires target feature 'sse4.2', but would be inlined into function
'add_avx' that is compiled without support for 'sse4.2'
        y = ^_mm256_set_ps(1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f);
            ^
intrinsic.cpp:22:21: error: always inline function '_mm256_load_ps'
requires target feature 'sse4.2', but would be inlined into function
'add_avx' that is compiled without support for 'sse4.2'
            x = ^_mm256_load_ps(a);
                ^
intrinsic.cpp:23:21: error: always inline function '_mm256_add_ps'
requires target feature 'sse4.2', but would be inlined into function
'add_avx' that is compiled without support for 'sse4.2'
            x = ^_mm256_add_ps(x, y);
                ^
intrinsic.cpp:24:17: error: always inline function '_mm256_store_ps'
requires target feature 'sse4.2', but would be inlined into function
'add_avx' that is compiled without support for 'sse4.2'
            ^_mm256_store_ps(a, x);
            ^
4 errors generated.
compilation aborted for intrinsic.cpp (code 1)
```

`-mavx` を指定して Intel® AVX ISA を有効にすると、Intel® AVX-512 組込み関数のエラーが表示されます。

```
$ icpx intrinsic.cpp -DSSE -mavx
intrinsic.cpp:31:13: error: always inline function '_mm512_set_ps'
requires target feature 'avx2', but would be inlined into function
'add_avx512' that is compiled without support for 'avx2'
    y = _mm512_set_ps(1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f,
1.f, 1.f, 1.f, 1.f, 1.f, 1.f, 1.f);
    ^
intrinsic.cpp:34:21: error: always inline function '_mm512_load_ps'
requires target feature 'avx2', but would be inlined into function
'add_avx512' that is compiled without support for 'avx2'
    x = _mm512_load_ps(a);
    ^
intrinsic.cpp:35:21: error: always inline function '_mm512_add_ps'
requires target feature 'avx2', but would be inlined into function
'add_avx512' that is compiled without support for 'avx2'
    x = _mm512_add_ps(x, y);
    ^
intrinsic.cpp:36:17: error: always inline function '_mm512_store_ps'
requires target feature 'avx2', but would be inlined into function
'add_avx512' that is compiled without support for 'avx2'
    _mm512_store_ps(a, x);
4 errors generated.
compilation aborted for intrinsic.cpp (code 1)
```

`-mavx512f` オプションで Intel® AVX-512 ISA を有効にして問題を解決します。

`__attribute__((target()))` 関数定義による組込み関数の使用

上記の例では、コンパイラー・オプション (`-mavx512f`) を指定して特定の命令セット向けにコンパイルしました。これは、ソースファイルに命令セットが 1 つだけである場合に使用できます。多くの場合、ソースファイルでは、組込みデータ宣言や組込み関数命令で複数の命令セットが使用されています。これは、ランタイム時のプロセッサ検出に基づいて特定の関数やコード領域を呼び出すことで実現されます。通常、これらの関数やコード領域は、特定のターゲット・アーキテクチャー向けに `#ifdef` で保護されており、ユーザーコードによってプロセッサ・ディスパッチが行われます。

Clang/LLVM コミュニティーでは、デフォルトのプロセッサ・ターゲットに依存せずに、特定のターゲット・アーキテクチャー向けの組込み関数を含む関数の定義を `gcc` 形式のターゲット属性でマークすることを強く推奨しています。

```
__attribute__((target(<required target>)))
```

組み込み関数の使用モデルに関する変更

デフォルトのプロセッサをターゲットとせず、特定のターゲット・アーキテクチャーで実行することを目的とした組み込み関数を使用する関数をマークします。この属性を使用したほうが、コンパイル時のエラーチェックが大幅に向上します。これには、特定のターゲット・アーキテクチャーごとに関数内にコードを配置して、ターゲット属性を関数定義に適用する必要があります。属性には、関数の組み込みレベルと関数内で許可される組み込み関数のセットを指定します。ターゲット属性と gcc 形式の関数のマルチバージョンの詳細は、次の資料を参照してください。

- ターゲット属性: [Clang 属性ターゲット](#) (英語)
- 関数のマルチバージョン: gcc 形式のマルチバージョンに関する [Wiki](#)(英語)と [ドキュメント](#)(英語)

マルチバージョンの例:

```
#include <stdio.h>

__attribute__((target("avx2")))
void dispatch_func() {
    printf("\nCode for Intel Core processors supporting Intel AVX2 goes here\n");
}

__attribute__((target("sse4.2")))
void dispatch_func() {
    printf("\nCode for Intel Core processors supporting SSE4.2 goes here\n");
}

__attribute__((target("sse3")))
void dispatch_func() {
    printf("\nCode for Intel Core 2 Duo processors supporting SSSE3 goes here\n");
}

__attribute__((target("default")))
void dispatch_func() {
    printf("\nCode for default implementation goes here\n");
};

int main() {
    dispatch_func();
    printf("Return from dispatch_func\n");
    return 0;
}
```

intrinsic-promote オプションを使用したレガシー組込み関数の拡張

このオプションは一般的な使用には推奨されません。ICC/ICL クラシック形式の組込み関数を使用するレガシー・アプリケーション向けに、ICX コンパイラーは新しいオプションを用意しています。このオプションはエラーが発生しやすいため非推奨です。このオプションは、組込み関数を含む関数をその関数内の組込み関数の最上位のターゲット・アーキテクチャーに自動的に昇格しようとします。

ターゲットが異なるセクションを含む関数では、ランタイムエラーが発生する可能性があります。
例: ユーザーによるプロセッサ・ディスパッチ。

そのため、このオプションの使用は推奨されません。現在、ICC/ICL クラシックの組込み関数の動作に対する長期的なソリューションに取り組んでいます。

Windows* オプション: /Qintrinsic-promote

Linux* オプション: -mintrinsic-promote

このオプションを使用すると、特定の CPU 機能を必要とする組込み関数呼び出しを含む関数は、必要な機能が利用できるようにターゲット・アーキテクチャーが自動的に昇格されます。関数内のすべてのコードは、昇格されたターゲット・アーキテクチャーでコンパイルされ、生成されるコードは必要な機能をサポートしていないプロセッサでは正しく動作しません。必要な機能をサポートしていないプロセッサでプログラムが実行された場合、関数に動的に到達できないようにユーザーは実行時に実行パスを保護する責任があります。

このオプションは、従来のコードを容易にコンパイルできるように提供されています。代わりに、`__attribute__((target(<requiredtarget>)))` を使用して特定のターゲット・アーキテクチャー向けの関数をマークすることを強く推奨します。この属性を使用したほうが、コンパイル時のエラーチェックが大幅に向上します。

インテル固有のプロセッサ・ターゲット・プラグマと関数のサポート

- “optimization_parameter*”は、インテル固有のプラグマです。

```
#pragma[intel]optimization_parametertarget_arch=<CPU>
#pragma[intel]optimization_parameterinline-max-total-size=n
#pragma[intel]optimization_parameterinline-max-per-routine=n
```

これらのプラグマは、ICX ではサポートされないため、前述のように `__attribute__((target(<requiredtarget>)))` に置き換えます。

- インテル固有の `_may_i_use_cpu_feature()` 組込み関数はサポートされます。
- インテル固有の `_allow_cpu_features()` 組込み関数は今後のリリースで追加される可能性があります。

浮動小数点結果の再現性の制御

以下は、ICX でサポートされる浮動小数点(FP)モデルサポートの状況です。

- GOLD リリースにおけるデフォルトの FP モデルは、`-fp-model fast=1-fma` と同等です。ベータ版では `-fp-model precise-no-fma` でした。

これは、ICC/ICL クラシックと ICX の主な動作の違いです。これは、ユーザーにとってメリットのある変更であると信じています。

- `-fp-model fast` がサポートされました。
- `-fp-model consistent` は 2021.2 ではサポートされません。同様の機能は `-fp-model=precise -fimf-arch-consistency=true -no-fma` で提供されます。
- FP の厳密性: デフォルトより厳密なものはサポートされていません。`-fp-modelstrict`、`-fp-speculation=safe`、`#pragma fenv_access` などはサポートされていません。現在、これらの実装をサポートするためオープンソース・コミュニティで取り組みが行われています。
- ICC/ICL クラシックの数学ライブラリー関連の機能は、現在 ICX への移行が進められています。IMF(インテル® マス・ライブラリー)は ICX に実装済みです。

ブルータスまたはバイセクション最適化のサポート

ICC/ICL クラシックの「Brutus(ブルータス)」または Clang/LLVM の「Bisectional(バイセクション)最適化」に興味のない方は、このセクションをスキップしてください。

ICX は、Clang/LLVM のバイセクション最適化デバッグ用の `-opt-bisect-limit=N` オプションをサポートしています。これは、ICC/ICL クラシックのブルータスオプションに類似しています。

オープンソース・コミュニティは、現在 Clang/LLVM の最適化デバッグ機能を強化するため取り組んでいます。

詳細については、「[-opt-bisect-limit を使用して最適化のエラーをデバッグ](#)」(英語)を参照してください。

付録:リファレンス

次のリファレンスをご覧ください。

- [インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス \(英語\) \(2021.1 日本語版はこちら\)](#)
- [インテル® oneAPI プログラミング・ガイド](#)
- [インテル® C++ コンパイラーでサポートされる C++20 の機能 \(英語\)](#)
- [インテル® oneAPI DPC++/C++ コンパイラーにおける OpenMP* 機能と拡張のサポート](#)
- [SYCL* 2020 仕様とインテル® oneAPI DPC++ コンパイラー \(dpcpp\) の DPC++ 言語拡張でサポートされる機能対応](#)
- [インテル® oneAPI DPC++/C++ コンパイラー・リリースノート \(英語\)](#)
- [インテル® oneAPI ポーティング・ガイド ~ ifort から ifx へ移行するユーザー向け \(英語\)](#)

製品とパフォーマンス情報

¹性能は、使用状況、構成、その他の要因によって異なります。詳細については、www.intel.com/PerformanceIndex (英語)をご覧ください。