インテル® oneAPI ポーティング・ガイド

ICX へ移行する ICC ユーザー向け

2023年3月23日

注意事項:

このドキュメントは、インテル® デベロッパー・ゾーンで公開されている「Porting Guide for ICC Users to DPCPP or ICX」(英語)の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

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

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、特定のソフトウェア、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ー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。

製品および性能に関する情報

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、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。 注意事項の改訂 #20201201

はじめに

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 (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。同様の Fortran (ifx) 向けのポーティング・ガイドも用意されています。

改訂履歴	
2023 年 3 月 23 日	ツールキットの 2023 に対応しました。

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

このドキュメントでは、特に記載がない限り `icx` と明記される内容は `icpx` 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。インテル® C++ コンパイラー・クラシックのドライバーは、`icc`、`icpc`、および `icl` です。インテル® oneAPI DPC++・C++ コンパイラーのドライバーは、`icx` および `icpx` です。C プログラムのコンパイルとリンクには `icx` を使用し、C++ プログラムには `icpx` を使用します。
- `icc` ドライバーとは異なり、`icx` はファイル拡張子から C と C++ のコンパイルを決定することはありません。ユーザーは、C++ ファイルをコンパイルするには `icpx` を起動する必要があります。ICX/ICPX は、コアの C++ コンパイラーに加え、追加のフラグ `-fsycl` を指定すると、インテル® oneAPI DPC++ コンパイラー向けの SYCL*/DPC++ コードをコンパイルできます。

> インテル® oneAPI DPC++/C++ コンパイラーは新しいコンパイラーです。このコンパイラーは、インテル® C++ コンパイラー・クラシックと比較して機能面および動作上の違いがあります。インテル® C++ コンパイラー・クラシックを使用する既存のアプリケーションを新しいコンパイラーに移行する場合、ある程度の移植作業が必要になると予測されます。

インテル® C++ コンパイラー・クラシックからインテル® oneAPI DPC++/C++ コンパイラーへの移行自体は、スムースで簡単です。しかし、既存のアプリケーションを移行した後に、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

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。

- インテル® oneAPI DPC++/C++ コンパイラーのドライバーは、`icx` および `icpx` です。
- インテル® C++ コンパイラー・クラシックは、`icc`、`icpc`、または `icpx` ドライバーを使用しますが、このコンパイラーは次のリリースで非推奨となります。
- SYCL*/DPC++ ユーザーは、`icx`/`icpx` ドライバーに `-fsycl` フラグを指定することで SYCL* 拡張機能を利用できます。
- Clang とは異なり、ICX のデフォルト浮動小数点モデルは ICC と一致するように設定されており、デフォルトは `-fp-model=fast`(Linux*)/`-fp:fast`(Windows*)です。
- マクロ名が変わっています。ICX 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。
- 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。
- インテル® C++ コンパイラー・クラシックとは異なり、コンパイラーの組込み関数は、プロセッサー・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。

パフォーマンス

- ベクトル化
    - ICX 2022.0.0 以降のリリースでは、`-O2` または `-O3` オプションだけではインテル® コンパイラーの高度なループ最適化とベクトル化を有効にできません。追加レベルの最適化とベクトル化を有効にするには、プロセッサー・ターゲット・オプション `-x` または `/Qx` でターゲット・アーキテクチャーを指定します(例:`-xskylake-avx512`)。または、`-xhost` や `/Qxhost` オプションを使用して、コードをコンパイルするプラットフォームのプロセッサーで利用可能なすべてのインテル® コンパイラーの最適化とベクトル化を有効にします。 ベクトル化と実装の詳細については、「インテルの CPU と GPU 向けの LLVM と GCC のベクトル化」を参照してください。

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使用するには、`-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。
    - OpenMP* 4.5 以降の 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

- `icx` で定義される新しいバージョン定義マクロ
    - `__INTEL_LLVM_COMPILER`
- バージョン文字列

  LLVM ベースのコンパイラーのバージョン文字列は新しくなりました。インテル® oneAPI はセマンティクス・バージョニングを使用します。インテル® oneAPI のバージョン管理の詳細については、「インテル® oneAPI ツールキットとコンポーネントのバージョン規則」(英語)を参照してください。

  例:

```
$ icx --version
Intel(R) oneAPI DPC++ Compiler 2023.0.0 (2023.0.0.20221201)
```

  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。これは、常にカレンダーの西暦と一致するわけではありません。
    - `MINOR` は、1 桁のマイナーバージョン番号です。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` から始まるパッチ番号です。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(`YYYYMMDD`)のように表現されます。

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。

- インテル® oneAPI DPC++/C++ コンパイラーのドライバー `icx` と `icpx` は、インテル® C++ コンパイラー・クラシックのオプションまたは Clang/LLVM コンパイラーのほとんど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 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。
- すべてのインテル® C++ コンパイラー・クラシックのオプションが ICX で認識/実装されるわけではありません。
- インテル® C++ コンパイラー・クラシック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。インテル® oneAPI DPC++/C++ コンパイラーは、インテル® C++ コンパイラー・クラシックとは大きく異なる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 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。インテル® oneAPI DPC++/C++ コンパイラーの最適化と動作は、インテル® C++ コンパイラー・クラシックとは異なります。オプションを指定せずに ICX を試してみてください。
- インテル® C++ コンパイラー・クラシック・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。
- 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`
    - 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(例:`icx -Wunknown-pragmas`)。

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;
    }
}

$ icx -c -Wunknown-pragmas unknown-pragmas.c

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

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

2 warnings generated.
```

この例では 2 つの警告が報告されています。

- `#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。

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、`stdout` 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 SYCL*/データ並列 C++ の例です。

```
$ icx -fsycl -E –dM ./hello.cpp
$ more hello.ii
```

C/C++場合（`stdout` に出力）

```
$ icpx -E -dM ./hello.cpp
```

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
```

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(PCH)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。`--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、「リロケータブル 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
```

「リロケータブルな」PCH を使用して、`pch` に明示的に名前を付けるには、次のようにします。

```
$ icx --relocatable-pch -isysroot /path/to/build /path/to/build/file.h
file.h.pch
// file.c のコンパイルに PCH ファイルを使用します
```

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-	/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=
-diag-file	/Qdiag-file	-serialize-diagnostics
-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-` 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
```

認識できないプラグマの診断 #161 は、ICC クラシックの `-diag-disable:161` オプションで無効にできます。

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、不明なプラグマ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。特定の診断を無効にする場合は、常に `-Wno-` プリフィクスを使用します。

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

ICC/ICL 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/ICL 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力してきました。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。サンプリング・プロファイラーは、非常に低いオーバーヘッドでプロファイルを行います。また、より詳細なプロファイル情報を収集するコードのインストルメント・バージョンをビルドすることもできます。どちらのタイプのプロファイルでも、コードの命令回数、分岐や関数呼び出しに関する情報を取得できます。詳細は、「プロファイルに基づく最適化」(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。
    - 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。
- 初期化されていないグローバル変数は、デフォルトで `.bss`(block started symbol)に配置されます。ICX では、初期化されていないグローバル変数は `.bss` に配置され、シンボルが `.bss` にある場合、リンカーは複数の定義を持つことができません。

  ICC/ICL とは異なり、GCC バージョン 10 以下を使用したり、初期されていない古い clang グローバル変数を使用すると、それらは共通セクションに配置されます。シンボルが共通セクションにある場合、リンカーは複数の定義を持つことができます。次の例を考えてみます。

  ここで、`my_struct` は `test.h` を定義する構造体であり、`test.h` は `test1.c` と `test2.c` でインクルードされています。

```
$ cat test.h
struct my_struct
{     char my_structid[8];     int  temp2[6]; };

#if _EXTERN
extern struct  my_struct my_struct;
#else
   struct  my_struct my_struct;
#endif

$ cat test1.c
#include"test.h"

$ cat test2.c
```

```
#include"test.h"
int main ()

{      return 0; }
```

上記のコードは、ICC および GCC 10.0 以下ではエラーなしでコンパイルできますが、ICX では次のエラーが報告されます。

```
/usr/bin/ld: /tmp/test2-e6091a.o:(.bss+0x0): multiple definition of
`my_struct'; /tmp/test1-b4c494.o:(.bss+0x0): first defined here

clang: error: linker command failed with exit code 1 (use -v to see
invocation)
```

これに対処するには、次の 2 つの方法があります。

- `-fcommon` オプションを指定してコンパイルして、`.bss` ではなく共通セクションにシンボルを配置するように ICX に指示します。

  ```
  $ icx test1.c test2.c -fcommon
  ```

- グローバル変数を extern として宣言します。

  ```
  $ icx test1.c test2.c -D_EXTERN
  ```

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* の SIMD プラグマ (`#pragma omp simd`) に置き換える必要があります。OMP 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が、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明示的にインクルードする必要はありません。

- ICC/ICL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ー/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。
- 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ー/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`、あるいは `-x` コンパイラー・オプションを使用します。
    - 組込み関数に対する 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が、ICC/ICL クラシック・コンパイラーでは引数エラーを確認しません。

```
$ 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/compilers_a
nd_libraries_2020.0.166/linux/lib/clang/10.0.0/include/xmmintrin.h:2103:3
1: 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が、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。

次の例では、特定のプロセッサー/アーキテクチャー固有のコンパイルオプションを指定する必要がある ICX の動作の違いを示しています。

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・コミュニティーに報告済みであり、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_ps1(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){
        __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);
        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` を指定してインテル® AVX ISA を有効にすると、インテル® 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)
```

`-march=skylake-avx512` オプションでインテル® 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(`-march=skylake-avx512`)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ー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ー・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ー・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>)))
```

デフォルトのプロセッサーではなく、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、関数の組込みレベルと関数内で許可される組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ニングの詳細は、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ニング：gcc 形式のマルチバージョニングに関する Wiki（英語）とドキュメント（英語）

マルチバージョニングの例：

```
#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ー・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ー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ー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()))` 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ー・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_parameter target_arch=<CPU>
  #pragma [intel] optimization_parameter inline-max-total-size=n
  #pragma [intel] optimization_parameter inline-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、関数属性として ICX で使用できます。

  例: `__attribute__(allow_cpu_features())`

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` はサポートされませんが、同等の機能は `-fp-model=precise -fimf-arch-consistency=true -no-fma` で提供されます。
- `-fp-model=strict` は、値の安全性を厳密に維持する最適化をコンパイラーに指示します。
- `-fp-speculation=safe/fast/strict/off` は、すべてサポートされます。`#pragma fenv_access` はサポートされていません。
- 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)最適化」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・コミュニティーは、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`」(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

  https://www.isus.jp/products/c-compilers/compiler2022-japanese-docs/

- インテル® oneAPI プログラミング・ガイド

  https://www.isus.jp/products/oneapi/oneapi-programming-guide-released/

- インテル® C++ コンパイラーでサポートされる C++20 の機能

  https://www.isus.jp/products/c-compilers/c20-features-supported-by-cpp-compiler/

- インテル® C++ コンパイラーでサポートされる C23 の機能

  https://www.intel.com/content/www/us/en/developer/articles/technical/c23-features-supported-by-intel-compiler.html (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

  https://www.isus.jp/products/c-compilers/openmp-features-and-extensions-supported-in-icx/

- SYCL* 2020 仕様とインテル® oneAPI DPC++ コンパイラー(icx)の DPC++ 言語拡張でサポートされる機能対応

  https://www.isus.jp/products/oneapi/sycl-2020-features-dpc-language-oneapi-c/

- インテル® DPC++/C++ コンパイラー・リリースノート

  https://www.xlsoft.com/jp/products/intel/tech/documents.html?tab=1

- インテル® oneAPI ポーティング・ガイド(ifx)

  https://www.isus.jp/products/fortran-compilers/oneapi-porting-guide_ifx/