ホワイトペーパー

インテル® Xeon Phi™ コプロセッサー
MICROSOFT* WINDOWS* ホスト向けクイック・スタート・ガイド

バージョン 1.3

目次

はじめに

このガイドは、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャー・ベースのインテル® Xeon Phi™ コプロセッサーが装着された Microsoft* Windows* システム向けアプリケーションを作成し実行する際に役立つ情報を提供します。さまざまなツールを紹介し、簡単なサンプルを例に C/C++ および Fortran プログラムを作成し、実行する方法を示します。サンプルコードを実際に実行する場合は、このガイドからコードをコピーアンドペーストしてください。

目的

本ガイドに含まれるトピック:

1. システムの登録手順とインテル® MPSS のインストール手順
2. インテル® MIC アーキテクチャー向けソフトウェアのビルド環境
3. インテル® Xeon Phi™ コプロセッサー向けコードの記述例とインテル® Composer XE 2013 SP1 Windows* 版でのビルド方法
4. インテル® マス・カーネル・ライブラリー (インテル® MKL) などのインテルのライブラリーの使用例
5. インテル® Xeon Phi™ コプロセッサーで実行中のプログラムのデバッグ方法とプロファイル方法
6. インテルによって開発された最も一般的な手法 (BKM)

本ガイドに含まれないトピック:

1. 各種ツールの詳細情報 (各ツールのユーザーガイドを参照してください)
2. 詳細なトレーニング

用語

ホスト – PCIe* スロットにインテル® XeonPhi™ コプロセッサーが装着されたインテル® Xeon® プロセッサー・ベースのプラットフォーム。次のオペレーティング・システム (OS) がサポートされています:
Windows* 7 Enterprise SP1 (64 ビット)、Windows* 8 Enterprise (64 ビット)、Windows Server* 2008 R2 SP1 (64 ビット)、Windows Server* 2012 (64 ビット)。

ターゲット – インテル® Xeon Phi™ コプロセッサーおよびコプロセッサー内にインストールされている対応するランタイム環境。

uOS – マイクロオペレーティング・システムの略。Linux* ベースのオペレーティング・システムとインテル® Xeon Phi™ コプロセッサー上で動作するツール。

ISA – 命令セット・アーキテクチャーの略。ネイティブデータ型、命令、レジスター、アドレッシング・モード、メモリー・アーキテクチャー、割り込み/例外処理、外部 I/O など、コンピューター・アーキテクチャーのプログラミングに関連する部分。[1]

VPU – ベクトル・プロセシング・ユニットの略。SIMD (Single Instruction Multiple Data) 命令を実行する CPU 部分。

NAcc – ネイティブ・アクセラレーションの略。処理されるデータとそのデータを処理するインテル® MKL 関数がインテル® Xeon Phi™ コプロセッサー上にある、インテル® MKL のモードまたは形式。

オフロード・コンパイラー – インテル® C/C++ コンパイラー XE 14.0 Windows* 版およびインテル® Visual Fortran コンパイラー XE 14.0 Windows* 版。ホスト上でのみ実行されるバイナリー、インテル® Xeon Phi™ コプロセッサー上でのみ実行されるバイナリー、そしてホストとインテル® Xeon Phi™ コプロセッサーが互いに通信し両方で実行されるバイナリーのペアを生成することができます。

[1] Intel acronyms dictionary, 8/6/2009, http://library.intel.com/Dictionary/Details.aspx?id=5600

SDP – ソフトウェア開発プラットフォームの略。ホスト・プラットフォームとインテル® Xeon Phi™ コプロセッサーの組み合わせ。

KNC – 最初のインテル® Xeon Phi™ 製品である、インテル® Xeon Phi™ コプロセッサー (開発コード名: **Knights Corner**) の略。

インテル® MPSS – インテル® メニーコア・プラットフォーム・ソフトウェア・スタックの略。プログラムがインテル® Xeon Phi™ コプロセッサーと通信し、コプロセッサー上で実行できるようになる、ユーザーレベルおよびシステムレベルのソフトウェア。

SCIF – 対称コミュニケーション・インターフェイスの略。単一プラットフォーム内のノード間通信構造。ノードは、インテル® Xeon Phi™ コプロセッサーまたはインテル® Xeon® プロセッサー・ベースのホストです。特に、SCIF はすべてのノードで対称な API を提供し、PCIe バスを介した通信の詳細 (およびインテル® Xeon Phi™ コプロセッサーのハードウェア関連の制御) を抽象化します。

システム構成

テストには、2 つのインテル® Xeon®プロセッサー、PCIe* x16 バスを介して装着された 1 つまたは 2 つのインテル® Xeon Phi™ コプロセッサー、グラフィックス表示用の GPU が搭載されたインテルのワークステーションで構成されるインテル® ソフトウェア開発プラットフォームを使用しました。インテル® Xeon Phi™ コプロセッサーをサポートするシステムの一覧は、次のサイトで確認できます。
http://www.isus.jp/article/idz/hpc/which-systems-support-the-intel-xeon-phi-coprocessor

インテル® Xeon Phi™ コプロセッサー向けソフトウェア

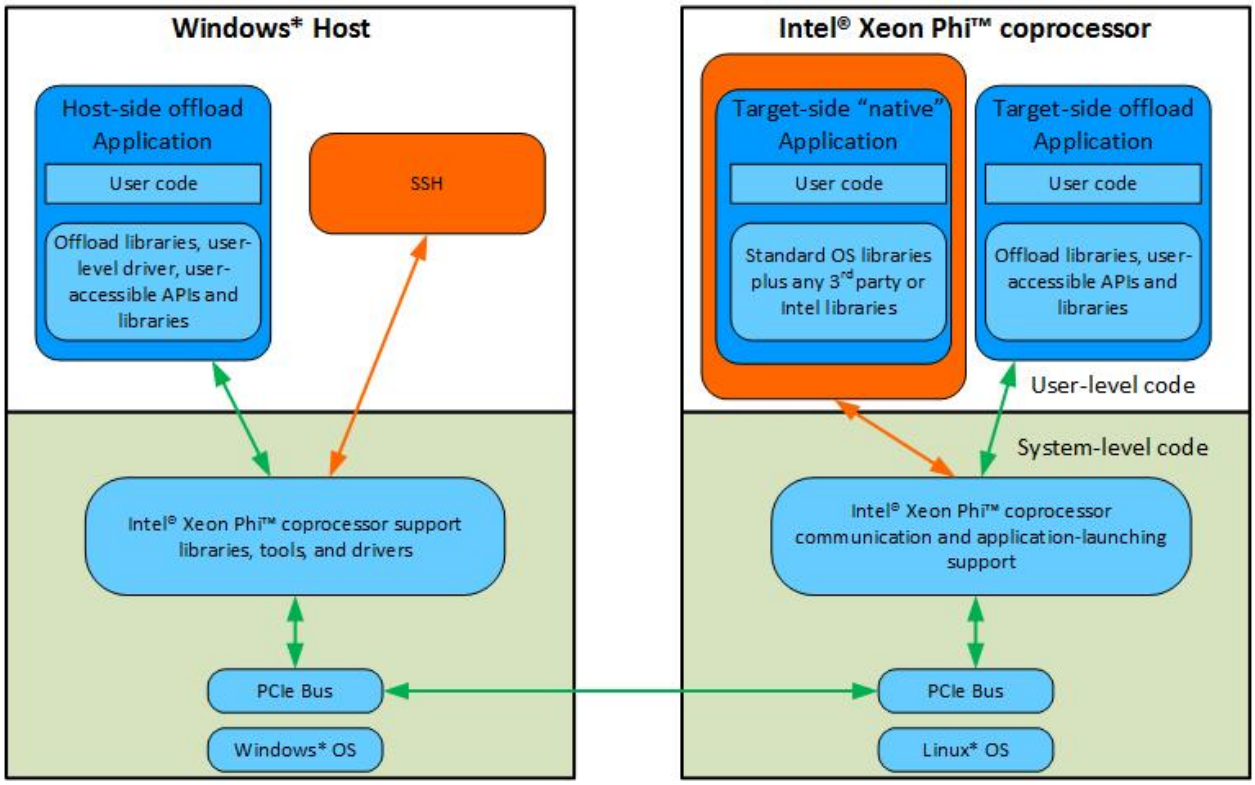

図 1: ソフトウェア・スタック

インテル® Xeon Phi™ コプロセッサーのソフトウェア・スタックは、図 1 と以下の説明に示すように、さまざまなソフトウェア・アーキテクチャーで構成されています。

ドライバースタック:
インテル® Xeon Phi™ コプロセッサー向け Windows* ソフトウェアは、次のコンポーネントで構成されています。

- **デバイスドライバー:** ソフトウェア・スタックの最下層のカーネル空間にあるインテル® Xeon Phi™ コプロセッサーのデバイスドライバー。デバイスの初期化およびホストとターゲットデバイス間の通信を管理します。
- **ライブラリー:** デバイスドライバーの上層のユーザーおよびシステム空間にあるライブラリー。システム内のカードの列挙、バッファー管理、ホストとカード間の通信などの基本的なカード管理機能に加え、インテル® Xeon Phi™ コプロセッサーへのユーザー実行ファイルのロード/アンロード、カード上の実行ファイルからの関数呼び出し、ホストとカード間の双方向通知構造の提供など高度な機能を提供します。バッファー管理と PCIe* バスを介した通信は、ライブラリーによって処理されます。
- **ツール:** ソフトウェア・スタックの保守に役立つ各種ツール。例えば、システム情報を照会する `<MPSS-install-dir> \bin\MicInfo.exe`、カードのフラッシュを更新する `<MPSS-install-dir>\bin\MicFlash.exe`、カードの設定に役立つ `<MPSS-install-dir>\bin\micctrl.exe`、プラットフォームの状態を監視する `<MPSS-install-dir>\bin\micsmc.exe`、RAS イベントを収集および記録する `<MPSS-install-dir>\bin\micras.exe` などがあります。デフォルトでは、`<MPSS-install-dir>` は `c:\Program Files\Intel\MPSS` です。
- **カード OS (uOS)**: インテル® Xeon Phi™ コプロセッサーに搭載されている Linux* ベースのオペレーティング・システム。

注: 最新の uOS バージョンの Linux* ソース、デバイスドライバー、下位レベルの SCIF ライブラリー・インターフェイスについては、http://www.isus.jp/article/mic-article/software-stack-mpss/ をご覧ください。ほかの下位レベルのインターフェイス (COI、MYO) は現在、汎用目的でインテル® ツールでのみ使用されています。これらの下位レベルのインターフェイスは、将来廃止または公開される可能性があります。

インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーの概要

インテル® Xeon Phi™ コプロセッサーは、50 を超えるインオーダーのインテル® MIC アーキテクチャー・ベースのプロセッサー・コアを搭載しており、これらは 1GHz (最大 1.3GHz) で動作します。インテル® MIC アーキテクチャーは x86 ISA をベースに、64 ビットのアドレッシング、新しい 512 ビットの SIMD ベクトル命令とレジスターで拡張されています。各コアは 4 つのスレッドをサポートします。コアに加えて、複数のオンダイ・メモリー・コントローラーやその他のコンポーネントを搭載しています。

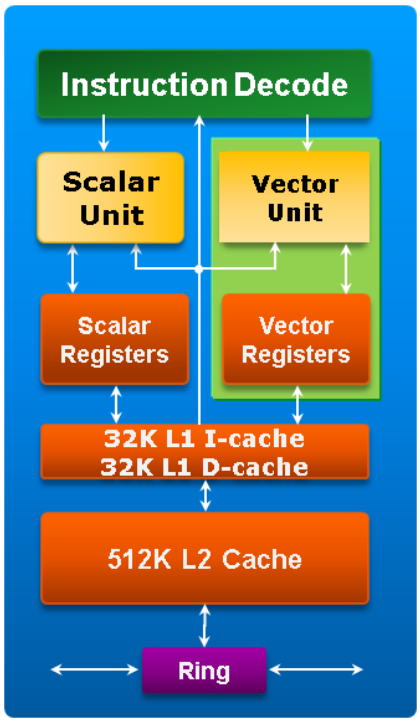

図 2: インテル® MIC アーキテクチャー・ベースのコアの概要

各コアには新しく設計されたベクトル・プロセシング・ユニット (VPU) があり、各 VPU には 32 個の 512 ビット・ベクトル・レジスターがあります。新しいベクトル・プロセシング・モデルをサポートするため、新たに 512 ビットの SIMD ISA が導入されました。

新しい VPU は、インテル® MIC アーキテクチャー・ベースのコアの主要機能です。インテル® Xeon Phi™ コプロセッサーで最良のパフォーマンスを得るには、VPU を最大限に活用することが重要です。インテル® MIC アーキテクチャー・ベースのコアは、ほかの SIMD ISA (例えば、インテル® MMX® 命令、インテル® SSE 命令、インテル® AVX 命令など) をサポートしないことに注意してください。

各コアには、32KB L1 命令キャッシュ、32KB L1 データキャッシュ、および 512KB L2 キャッシュが装備されています。すべてのコアの L2 キャッシュはインターコネクトにより互いに接続され、双方向リングバスを介してメモリー・コントローラーに接続されているため、最大 32MB の共有 LLC を効率良く利用できます。各コアには、短いインオーダーのパイプラインがあります。スカラー操作はレイテンシーなしで、ベクトル操作は低レイテンシーで実行できます。また、分岐予測ミスのオーバーヘッドも低くなっています。

マシン・アーキテクチャーについての詳細は、http://www.isus.jp/article/idz/mic-developer/ の「ツールとダウンロード」タブにある『インテル® Xeon Phi™ コプロセッサー・ソフトウェア開発者ガイド』(英語) を参照してください。

管理タスク

ドライバーの入手方法は、http://www.isus.jp/article/mic-article/software-stack-mpss/ を参照してください。インテル® Composer XE 2013 SP1 Windows* 版が必要になります。無料評価版および購入については、http://www.isus.jp/article/intel-software-dev-products/intel-composer-xe/ をご覧ください。

初めて使用する前のシステム準備

ドライバーのインストールとコプロセッサーの起動手順

1. インテル® デベロッパー・ゾーンの http://software.intel.com/mic-developer (英語) にアクセスし、「TOOLS & DOWNLOADS」タブにある「Software Drivers:Intel® Manycore Platform Software Stack(Intel® MPSS)」をクリックします。表示されるページの「Downloads」セクションから、Microsoft* Windows* 用の Readme (*readme-windows.pdf*)、リリースノート (*releaseNotes-windows.txt*)、およびインテル® MPSS ユーザーズガイド (*Windows_MPSS_Users_Guide.pdf*) をダウンロードします。
2. 以下のサポートされているオペレーティング・システムのいずれかをインストールします。
    - Microsoft* Windows* 7 Enterprise SP1 (64 ビット)
    - Microsoft* Windows* 8 Enterprise (64 ビット)
    - Microsoft* Windows Server* 2008 R2 SP1 (64 ビット)
    - Microsoft* Windows Server* 2012 (64 ビット)

3. 「管理者」としてログインします。
4. .NET Framework 4.0 以上 (http://msdn.microsoft.com/ja-jp/vstudio/aa496123.aspx) と、カードの uOS へのログインに使用する PuTTY* と PuTTYgen* (詳細は後述) をシステムにインストールします。
5. Readme ファイルのセクション 2.2.1「Preliminary Steps」の手順を実行します。
6. システムを再起動します。
7. ステップ 1 に記載したページから、Windows* 用のドライバーパッケージ (`mpss-3.*-windows.zip`) をダウンロードします。
8. zip ファイルを展開し、Windows* インストーラー・ファイル (Intel(R) Xeon Phi(TM) coprocessor.msi と Intel(R) Xeon Phi(TM) coprocessor essentials.msi) を取得します。
9. Readme ファイルのセクション 2.2.2 の手順に従って、Windows* インストーラー・ファイル Intel(R) Xeon Phi(TM) coprocessor.msi を実行してインストールします。以前のバージョンのインテル® Xeon Phi™ コプロセッサー・スタックがインストールされている場合は、現在のバージョンをインストールする前に、Windows* の [コントロール パネル] から以前のバージョンをアンインストールしてください。デフォルトでは、インテル® MPSS は `c:\Program Files\Intel\MPSS` にインストールされます。インストール時に、[Intel からのソフトウェアを常に信頼する] チェックボックスをオンにします。また、インテル® Xeon Phi™ コプロセッサー用のネイティブ・バイナリー・ユーティリティー (*Intel(R) Xeon Phi(TM) coprocessor essentials.msi*) もインストールします。これは、オフロード・プログラミングまたはクロスコンパイラーを使用する際に必要です。
10. [コントロール パネル] > [プログラム] > [プログラムと機能] で、インテル® MPSS スタックが正常にインストールされたことを確認します。図 3 のように、Intel(R) Xeon Phi(TM) coprocessor と Intel(R) Xeon Phi(TM) coprocessor essentials が表示されます。

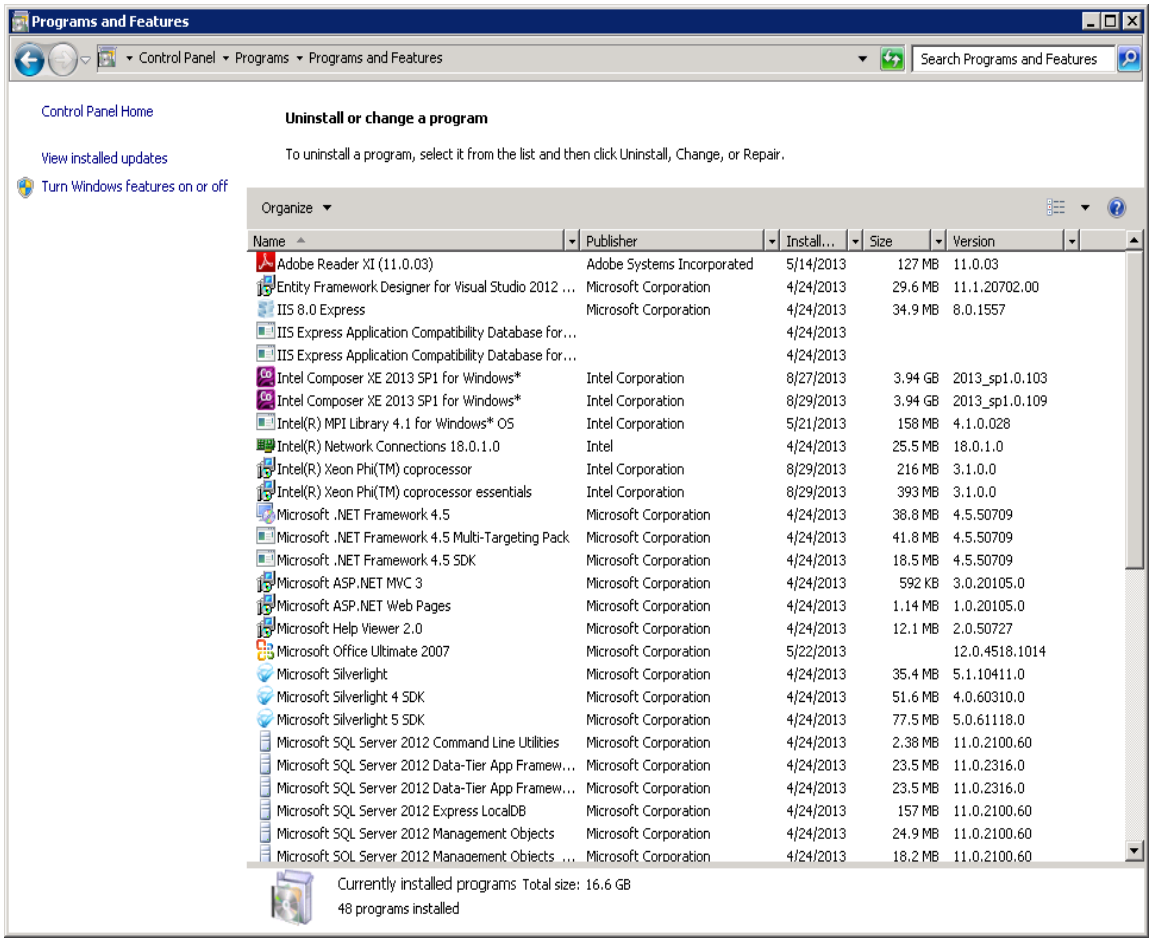

図 3: [プログラムと機能] パネルでインテル® MPSS がインストールされていることを確認

11. Readme ファイルのセクション 2.2.3 の手順に従って、フラッシュをアップデートします。
12. システムを再起動します。
13. ホストにログインして、デバイス マネージャーによってインテル® Xeon Phi™ コプロセッサーが検出されることを確認します ([コントロール パネル] > [ハードウェアとサウンド] > [デバイス マネージャー] を選択し、[システム デバイス] を展開します)。

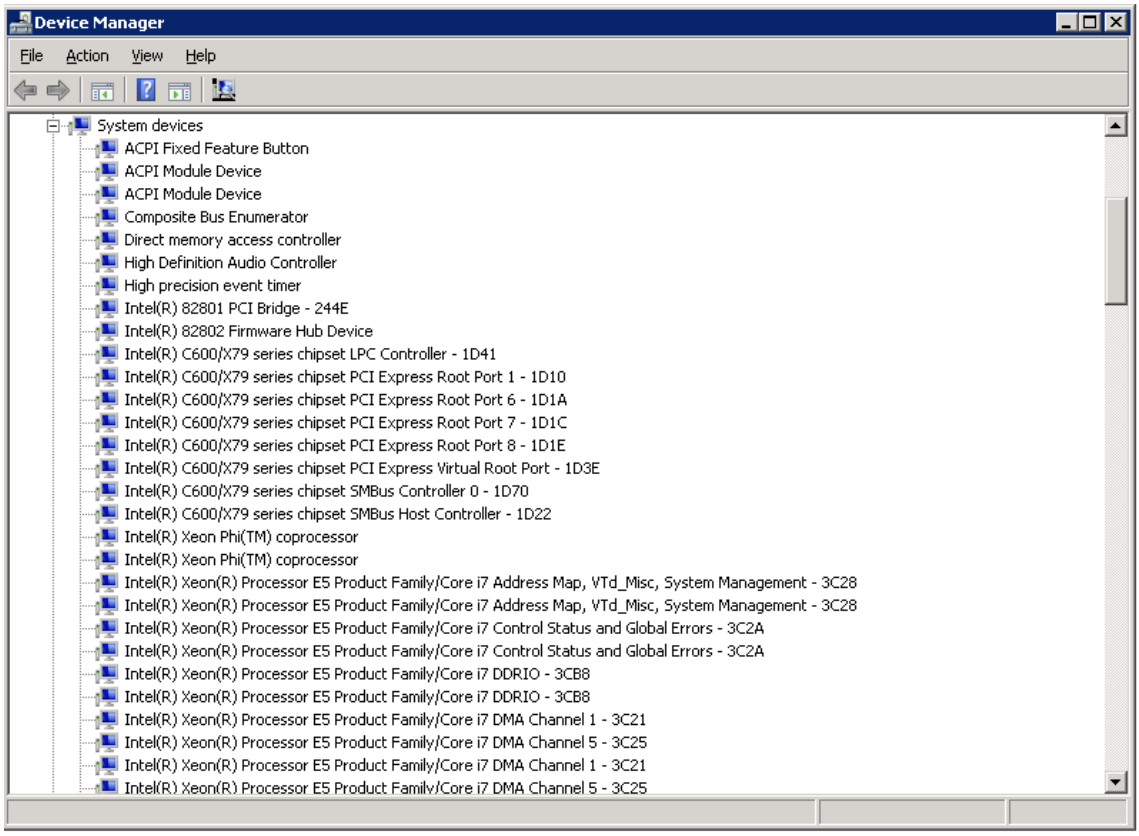

図 4: デバイス マネージャーによって検出されたインテル(R) Xeon Phi(TM) コプロセッサー

14. インテル® Xeon Phi™ コプロセッサーを起動します (ホストシステムの起動時にカードを起動するように設定することもできます。この設定はデフォルトでは行われません)。コマンドプロンプト・ウィンドウを開き、次のコマンドを実行してインテル® MPSS スタックを起動します。

```
prompt> micctrl --start
```

15. micinfo コマンドを実行して適切に設定されていることを確認します。

```
prompt> micinfo.exe
```

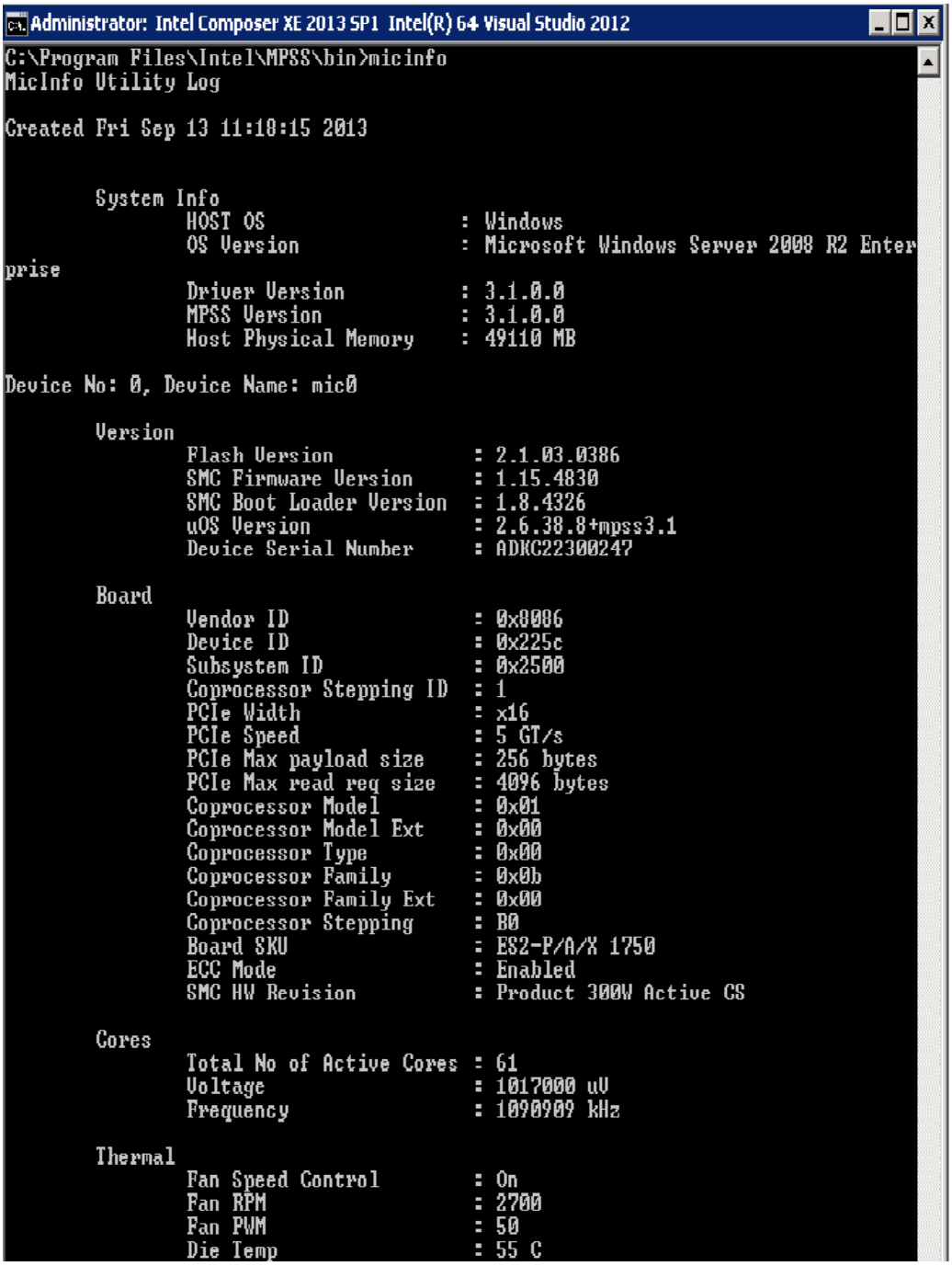

図 5: micinfo.exe コマンドの出力例

- Driver Version が 3.1.* であることを確認します。
- MPSS Version が 3.1.* であることを確認します。
- Flash Version が 2.1.*.* であることを確認します。

ソフトウェア開発ツールのインストール手順

前述のとおり、インテル® C++/Fortran Composer XE 2013 SP1 Windows* 版 (http://www.isus.jp/article/intel-software-dev-products/intel-composer-xe/) (デバッガー、インテル® MKL などが含まれる) が必要です。

- インテル® C++ Composer XE 2013 SP1 Windows* 版またはインテル® Visual Fortran Composer XE 2013 SP1 Windows* 版のドキュメントを参照して、これらのパッケージをインストールしてください。
- 初めてインストールする場合は、製品をアクティベーションするため、インストール時にシリアル番号を入力する必要があります。次回のインストールからは、[既存のライセンスを使用する] オプションを利用できます。
- ドキュメントをよくお読みください。
- 実行ファイル (.EXE) をダブルクリックしてインストールを開始します。インストールする前に、以前のバージョンをアンインストールしたり、アップデートする必要はありません。新しいバージョンは、既存のバージョンをアップデートするか、既存のバージョンと共存できます。

1. ソフトウェア・ツールをインストールします。
2. ホストとインテル® Xeon Phi™ コプロセッサー間の通信内容を表示する "`set OFFLOAD_REPORT=3`" を指定し、サンプルプログラム (`<install-dir>\Samples\ja_JP\C++\mic_samples` または `<install-dir>\Samples\ja_JP\Fortran\mic_samples`) を実行して、カードが正常に動作することを確認します (プロセッサーからのメッセージには、プリフィクス "MIC:" が付けられます)。通信内容が表示されれば、カードは正しく動作しており、使用できる状態です。サンプルの説明は、http://software.intel.com/en-us/articles/offload-programming-fortran-and-c-code-examples (英語) を参照してください。
3. SEP とインテル® Vtune™ Amplifier XE 2013 Update 13 を使用してパフォーマンス・データの収集および解析を行う場合は、コプロセッサーの起動後、データ収集ドライバーをロードする必要があります。SEP をインストールするには、次の操作を行います。
   a) コプロセッサーの起動後、データ収集ドライバーをロードするため、`<VTune-install-dir>\bin64\k1om\` に移動し、次のコマンドを実行します。

      ```
      prompt> .\sep_micboot_install.cmd
      ```

   b) インテル® MIC アーキテクチャー・サービスを開始 (または再起動) します (前のステップでファイルのコピーが終わっている場合、サンプリング・ドライバーも開始されます)。

      ```
      prompt> micctrl --start
      prompt> micctrl -w
      ```

      コプロセッサーが正常に再起動されると、`micctrl –w` コマンドは micx: online を出力します。
   c) 次回から、コプロセッサーが再起動されるたびにサンプリング・ドライバーも起動されます。
   d) サンプリング・ドライバーをアンインストールする必要がある場合は、次のコマンドを実行します。

      ```
      prompt> micctrl --stop
      prompt> .\sep_micboot_uninstall.cmd
      prompt> micctrl --start
      ```

```
prompt> micctrl –w
```

再起動後のインテル® Xeon Phi™ コプロセッサーへのアクセスの確立

インテル® Xeon Phi™ コプロセッサーは、ホストシステムの再起動時に起動されません。そのため、手動でインテル® Xeon Phi™ コプロセッサーを起動し、micinfo コマンドを実行して正常に起動されたかどうかを確認する必要があります。このコマンドを実行するには、管理者権限が必要です。**[スタート]** > **[すべてのプログラム]** > **[アクセサリ]** を選択します。**[アクセサリ]** にある **[コマンド プロンプト]** を右クリックして、**[管理者として実行]** を選択します。管理者のパスワードを入力すると、コマンドライン・ウィンドウが表示されます。

```
prompt> micctrl --start
prompt> micctrl -w
prompt> micinfo
```

インテル® Xeon Phi™ コプロセッサーがハングアップした場合の再起動

インテル® Xeon Phi™ コプロセッサーで、あるプロセスだけがハングアップし、その他の応答には問題がない場合、PuTTY* でカードにログインし、ほかの Linux* プロセスと同様の方法でそのプロセスを強制終了します。

コプロセッサーがハングアップしアクセスできない場合、あるいは PuTTY* でも応答しない場合、コプロセッサーを再起動する方法は 2 つあります。最初に、次のコマンドでハングアップの原因を探ります。

```
prompt> micctrl -s <micx>
```

インテル® MPSS サービスが正しく動作している場合は、次のコマンドを実行することで、装着されたほかのコプロセッサーに影響を与えることなく、問題のコプロセッサーの再起動を試みることができます。

```
prompt> micctrl –r <micx>
prompt> micctrl -w
prompt> micctrl –b <micx>
prompt> micctrl -w
prompt> micinfo
```

インテル® MPSS サービスが正しく動作していない場合は、ドライバーと装着されているすべてのコプロセッサーを再起動する必要があります。

```
prompt> micctrl --stop
prompt> micctrl --start
prompt> micctrl -w
prompt> micinfo
```

インテル® Xeon Phi™ コプロセッサー の uOS 環境での直接操作

デフォルトでは、ホストから見たコプロセッサーの IP アドレスは `192.168.<coprocessor>.100` で、コプロセッサーから見たホストの IP アドレスは `192.168.<coprocessor>.99` です。ホストからコプロセッサーを参照する場合は、エイリアス `mic<coprocessor>` を使用することもできます。例えば、システムに最初に装着したコプロセッサーは “`mic0`” となり、その IP アドレスは `192.168.1.100` になります。このコプロセッサーから見たホストの IP アドレスは `192.168.1.99` です。2 つ目のコプロセッサーは “`mic1`” で、`192.168.2.100` になり、ホストは `192.168.2.99` になります。

コプロセッサーは Linux* が動作している独立したネットワーク・ノードなので、PuTTY* を介して root または root 以外のユーザーとしてログインし、多くの一般的な Linux* コマンドを利用できます。コプロセッサーとのファイルの受け渡しには、WinSCP* ツールやその他の手段を使用します。

- サードパーティー・ソフトウェア PuTTY* は、http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (英語) からダウンロードできます。ダウンロード後、ホストフォルダー "`<MPSS-install-dir>\bin`" に配置します。SSH キーを作成するには、上記のリンクから PuTTYgen* ユーティリティーをダウンロードし、ホストフォルダー `<MPSS-install-dir>\bin` に配置します。

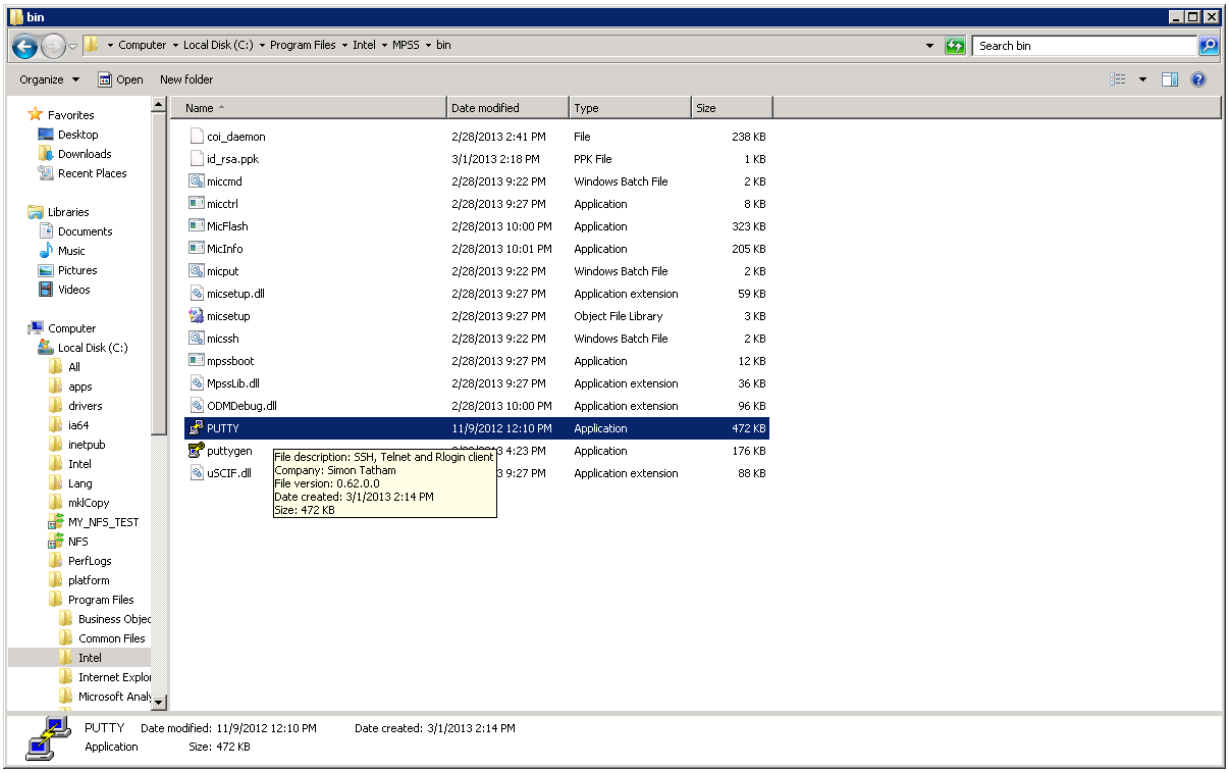

図 6: PuTTY* と PuTTgen* を <MPSS-install-dir>\bin に配置

- PuTTYgen* を起動し、[Generate] ボタンをクリックして、SSH 公開キーと秘密キーを作成します。次の手順に従って公開キーを作成します (図 7)。

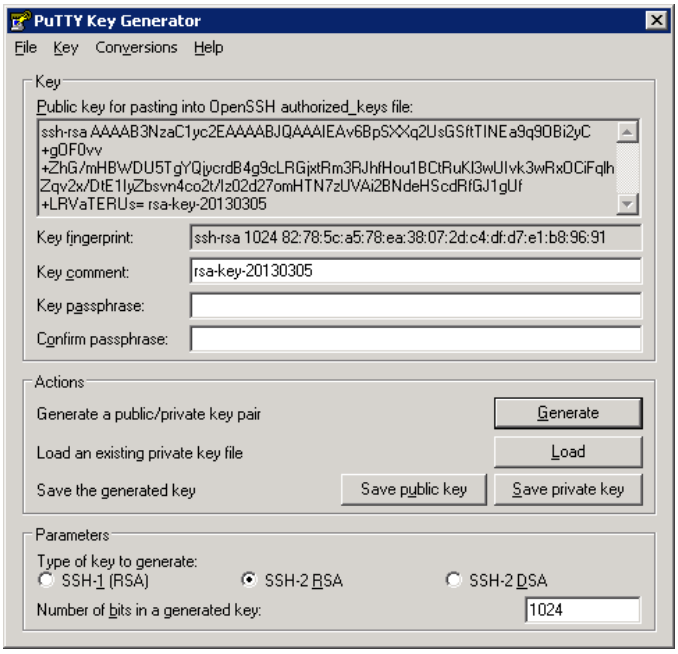

図 7: PuTTYgen*

- メモ帳で `authorized_keys` という名前のファイルを作成し (ファイル拡張子 .txt を含めずに)、`<MPSS-install-dir>\bin` フォルダーに配置します。
- [Public key pasting into OpenSSH authorized_keys file] に表示されているテキストをコピーし、`authorized_keys` ファイルにペーストします。
- [Save private key] をクリックして、`id_rsa.ppk` という名前のファイルに秘密キーを保存します。この `id_rsa.ppk` ファイルを `<MPSS-install-dir>\bin` に移動します。
- コマンドプロンプト・ウィンドウを (**[管理者として実行]** モードで) 開き、<MPSS-install-dir>\bin ディレクトリーに移動し、次のコマンドを実行します。

```
prompt> micctrl --addssh root –f "<MPSS-install-dir>\bin\authorized_keys"
```

- `"micctrl --stop"` コマンド、そして `"micctrl --start"` コマンドを実行して、コプロセッサーを再起動します。

```
prompt> micctrl --stop
prompt> micctrl --start
```

これで、PuTTY* を使用してコプロセッサーにログインできるようになります。

- PuTTY* ツールを起動します。
- [Host Name (or IP address)] ボックスに root@192.168.1.100 (mic0 の場合) と入力します (mic1 の場合は root@192.168.2.100)。

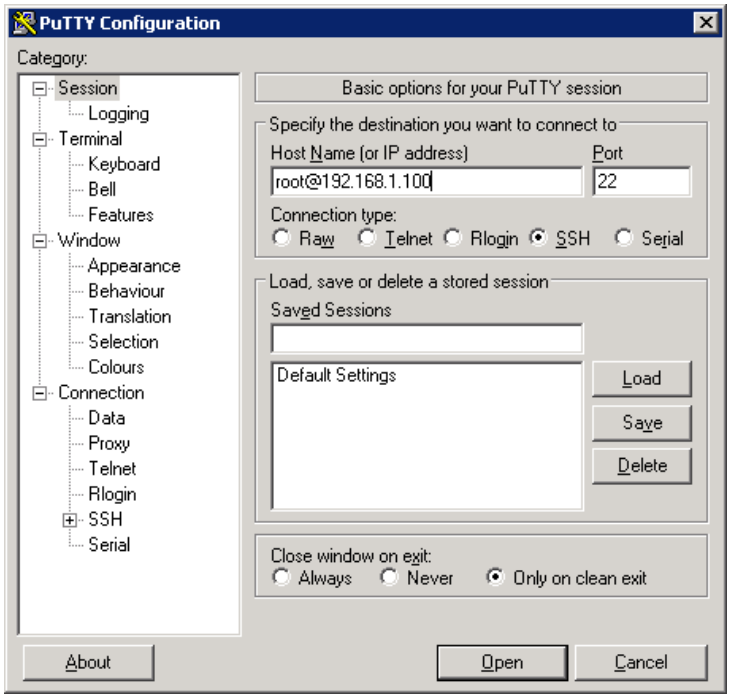

図 8: PuTTY* を使用してコプロセッサーにログイン

- [Connection] > [SSH] を展開して [Auth] をクリックします。[Private key file for authentication] ボックスで、[Browse] ボタンを使って秘密キーファイル `id_rsa.ppk` を選択します。

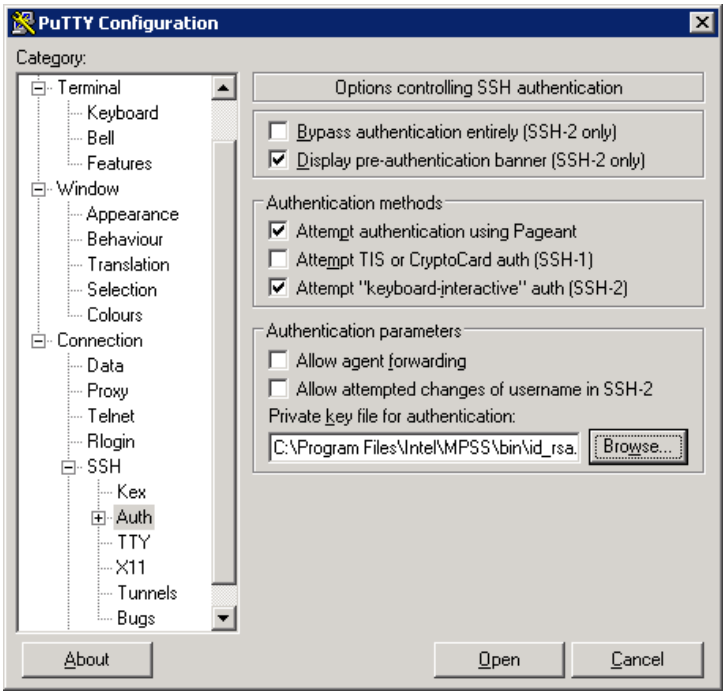

図 9: 秘密キーファイルの選択

- [Open] をクリックしてコプロセッサーに接続します。ウィンドウが表示され、コプロセッサーにログインできます (パスワードの入力は不要です)。

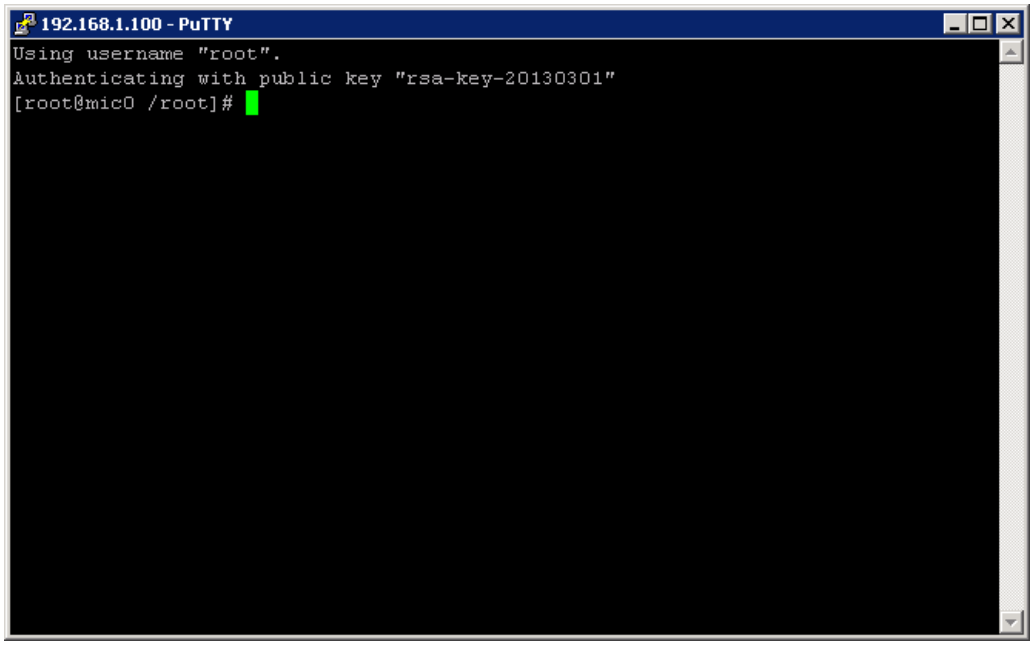

図 10: mic0 のルート・ディレクトリー

- WinSCP* ツールにより、ホストからコプロセッサーへファイルをコピーします。このサードパーティー・ツールは、http://winscp.net (英語) からダウンロードする必要があります。

  例えば、ホストで WinSCP* ツールを起動し、[File protocol] に「SCP」、[Host name] に「192.168.1.100」、[User name] に「root」、そして [Private key file] ボックスに秘密キーファイルの場所を指定し、コプロセッサー向けにコンパイルされたファイル (`application.mic`) をホストから mic0 コプロセッサーにコピーできます。

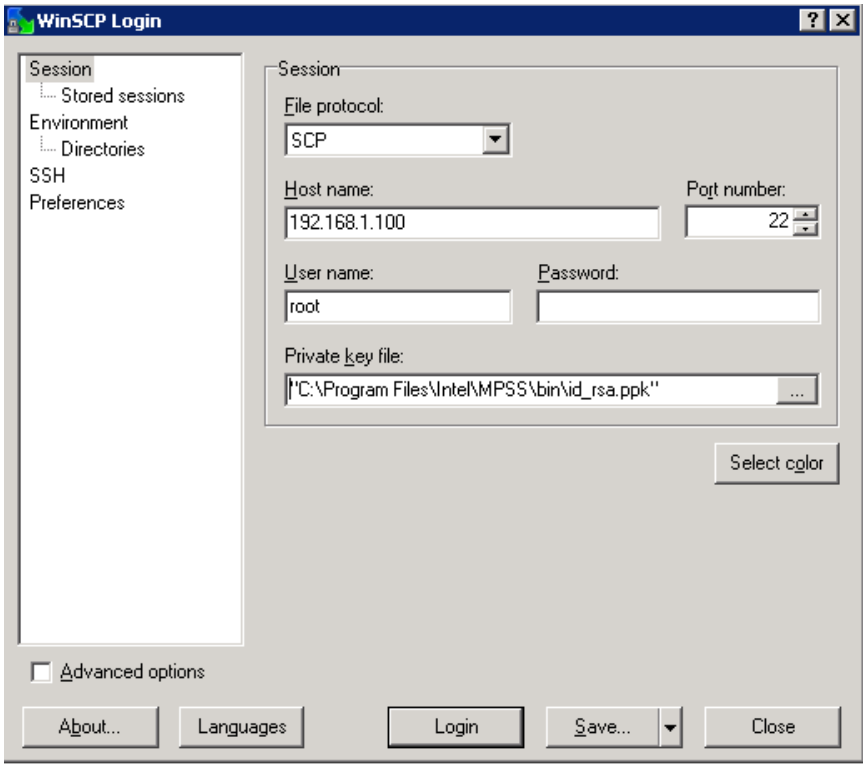

図 11: WinSCP* を使用してファイルを転送

[Login] ボタンをクリックすると、GUI ウィンドウが開き、ホストと mic0 間でファイル転送が可能になります。コプロセッサーにファイルが転送されたかどうかは、コプロセッサーにログインし ls コマンドで確認できます。

ホスト・ファイル・システムの共有

インテル® Xeon Phi™ コプロセッサー用のネイティブ実行ファイルをコプロセッサーに転送する代わりに、ホストとコプロセッサー間で共有ディレクトリーを設定できます。この設定を行うと、ホストとコプロセッサーの両方から共有ディレクトリーのすべてのファイルにアクセスできます。

ホストによってインテル® Xeon Phi™ コプロセッサーへエクスポートされた NFS ファイルシステムのマウントに関する詳細は、Readme ファイルのセクション 11 を参照してください。

ホストと全コプロセッサー間でフォルダーを共有する場合は、**[共有フォルダーの準備ウィザード]** ウィンドウで **[編集]** をクリックし、**[アクセス権の種類]** フィールドを「読み書き」にし、**[ルート アクセスを許可する]** をオンにします。

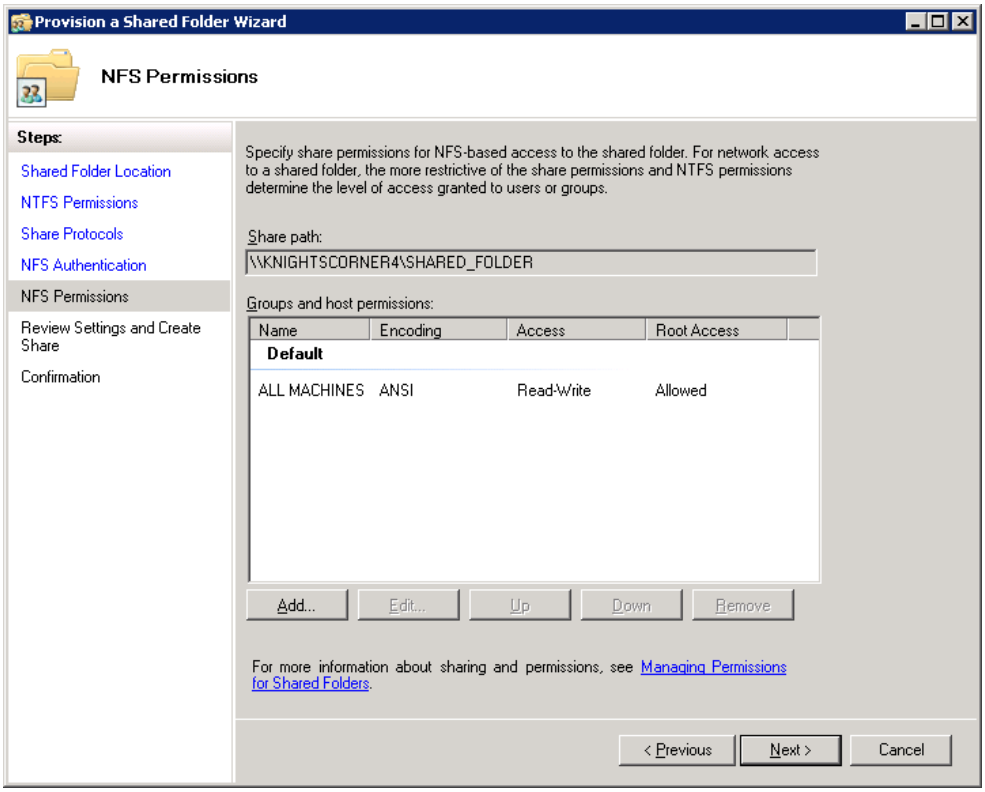

図 12: すべてのコプロセッサーとフォルダーを共有

便利な管理ツール

インテル® MPSS には便利な管理ツールが含まれています。これらは、<MPSS-install-dir>\bin フォルダーにあります。root およびこれらのツールを使用するユーザーは、このフォルダーをデフォルトのパスに追加しておくべきです。

- **micinfo** - ホストとコプロセッサーのシステム構成に関する情報を提供します。
- **micflash** - コプロセッサー上のフラッシュを更新します。フラッシュの各セクションのバージョンおよびその他の情報を保存/取得します。

- **micctrl** – コプロセッサーの設定や再起動などを行うシステム管理ツールです。
- **micras** – ホストで実行されます。インテル® Xeon Phi™ コプロセッサーにより生成される RAS イベントの収集と記録を行います。さらに、致命的な RAS イベントを検出すると、コプロセッサーをメンテナンス・モードに移行させ、テストおよび修復を行います。
- **micsmc** – コアの使用率、温度、メモリー使用量、消費電力統計、エラーログを監視します。グラフィカル・ユーザー・インターフェイス (GUI) またはコマンドライン・インターフェイスの 2 つのモードで利用できます。

これらのツールの詳細と引数は、Readme ファイルのセクション 7、8、9 を参照してください。

インテル® Xeon Phi™ コプロセッサー向けソフトウェアの開発

インテル® MIC アーキテクチャー向けアプリケーションの開発は、マルチコアおよび SIMD プログラミングの既存の知識に基づいて行います。オフロード言語拡張により、インテル® Xeon Phi™ コプロセッサーで実行するため (C/C++ または FORTRAN で記述された) コードの一部を移植したり、あるいはインテル® MIC アーキテクチャー向けにアプリケーション全体を移植することができます。最良のパフォーマンスは、高度に最適化され、ほとんどの実行に (コンパイラーにより生成された、またはコンパイラーの組込み関数を使用して生成された) SIMD 操作を用いるアプリケーションでのみ達成できます。

利用可能なソフトウェア開発ツール/環境

既存の並列プログラミングの知識とホストの並列アプリケーション開発と同じ手法を利用して、インテル® Xeon Phi™ コプロセッサーの開発に取り掛かることができます。インテル® Xeon Phi™ コプロセッサー専用の新しい開発ツールはありませんが、インテル® MIC アーキテクチャーに対応するため、いくつかの標準言語と API の追加により、ホスト用の既存のインテル® ツールが拡張されています。開発ツールを最大限に利用し、インテル® Xeon Phi™ コプロセッサーから最良のパフォーマンスを引き出すには、インテル® MIC アーキテクチャーについて理解することが重要です。

開発環境: コンパイラーとライブラリー

- **コンパイラー**
  - インテル® C++ Composer XE 2013 SP1 Windows* 版。インテル® 64 アーキテクチャーおよびインテル® MIC アーキテクチャーで動作するアプリケーションをビルドできます。
  - インテル® Visual Fortran Composer XE 2013 SP1 Windows* 版。インテル® 64 アーキテクチャーおよびインテル® MIC アーキテクチャーで動作するアプリケーションをビルドできます。
- **ライブラリー** (コンパイラーに付属):
  - インテル® マス・カーネル・ライブラリー (インテル® MKL)。インテル® MIC アーキテクチャー向けに最適化されています。インテル® MKL のインテル® Xeon Phi™ コプロセッサー・サポート機能はデフォルトでは無効です。インテル® C++/Visual Fortran Composer のインストール時に有効にする必要があります。
  - インテル® スレッディング・ビルディング・ブロック (インテル® TBB)
  - インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)

開発環境: ツール

上記のコンパイラーとライブラリーに加えて、次のツールを利用してインテル® Xeon Phi™ コプロセッサーで動作するソフトウェアのデバッグと最適化を行えます。

- **インテル® Debugger Extension**
  - インテル® 64 アーキテクチャーおよびインテル® MIC アーキテクチャーで動作するアプリケーション向け。
- **プロファイル**

- インテル® VTune™ Amplifier XE 2013 Update 13 Windows* 版。Windows* ベースのホストシステムでインテル® Xeon Phi™ コプロセッサーのデータを収集し、収集したデータを確認できます。

開発に関する一般情報

開発環境のセットアップ

- このバージョンでは、Microsoft* Visual Studio* 2008 以降との統合がサポートされています。ここでは、コマンドライン・インターフェイスを介してオフロード向けにコンパイルする手順を示します。
  - **インテル® C++/Visual Fortran Composer XE 2013 SP1 Windows* 版:**

    コマンドプロンプト・ウィンドウを開くには、[スタート] > [すべてのプログラム] > [Intel Parallel Studio XE 2013 (インテル(R) Parallel Studio XE 2013)] > [Command Prompt (コマンドプロンプト)] > [Parallel Studio XE with Intel Compiler XE 14.0 (インテル(R) コンパイラー XE 14.0)] > [Intel 64 Visual Studio XXXX mode (インテル(R) 64 Visual Studio XXXX モード)] を選択します。

ドキュメントとサンプルコード

- 次のような役立つドキュメントが、**<install-dir>\Documentation\ja_JP** フォルダーにインストールされます。
  - **compiler_c\cl\compiler_ug_c** および **compiler_f\cl\compiler_ug_f** - インテル® C++ コンパイラー XE 14.0 Windows* 版およびインテル® Visual Fortran コンパイラー XE 14.0 Windows* 版のドキュメント ([スタート] > [すべてのプログラム] > [Intel Parallel Studio XE 2013 (インテル(R) Parallel Studio XE 2013)] > [Documentation (ドキュメント)] からも利用できます)。
    - インテル® MIC アーキテクチャー向けのビルドに関するほとんどの情報は、「主な機能」>「インテル® MIC アーキテクチャー」>「インテル® MIC アーキテクチャー用にビルド」セクションにあります。
    - インテル® MIC アーキテクチャー向け組込み関数に関する情報は、「コンパイラー・リファレンス」>「組込み関数」>「インテル® MIC アーキテクチャー向け組込み関数」セクションにあります。
  - **Release_Notes-C-2013SP1_W_JA.pdf** および **Release_Notes-F-2013SP1_W_JA.pdf** - インテル® MIC アーキテクチャーをサポートするすべてのツールに関する既知の問題とその回避方法、インストール手順をよくお読みください。インテル® MIC アーキテクチャーに関する情報は、主にセクション 4 にあります。
    - **注**: さまざまな理由から、このドキュメントには最新の情報が含まれていない可能性があります。インテル® C++/Fortran Composer の最新のリリースノート (**Release_Notes-*-2013SP1_W_EN.pdf**) は、インテル® ソフトウェア開発製品レジストレーション・センターからダウンロードできます。
  - **<install-dir>\Documentation\ja_JP\debugger\gdb\pdf\vmigdb_config_guide.pdf** (英語) – インテル® Xeon Phi™ コプロセッサー向けアプリケーションをデバッグするための Visual Studio* 2012 の設定方法に関する情報。Windows* システムでは、Visual Studio* と GDB を介してインテル® Xeon Phi™ コプロセッサー上のオフロード処理をデバッグできます。
- 次のドキュメントにもインテル® Xeon Phi™ コプロセッサーに関するセクションがあります。
  - インテル® MKL ユーザーズガイド。**<install-dir>\Documentation\ja_JP\mkl** 以下の **mkl_documentation.htm** から利用できます (このファイルは、[スタート] > [すべてのプログラム] > [Intel Parallel Studio XE 2013 (インテル(R) Parallel Studio XE 2013)] > [Documentation (ドキュメント)] からも利用できます)。「インテル® Xeon Phi™ コプロセッサーでのインテル® MKL の使

用」セクションにインテル® MKL 関数の「自動オフロード」と「コンパイラーによるオフロード支援」の説明があります。
    - インテル® VTune™ Amplifier XE 2013 Windows* 版によるインテル® Xeon Phi™ コプロセッサーでのパフォーマンス・データ収集に関する情報は、**<VTune-install-dir>\documentation\en\tutorials\find_hotspots\C++\index.htm** (英語) にあります。
- (Linux* ホスト向けだが役立つ) Web ドキュメント:
    - http://www.isus.jp/article/idz/mic-developer/ から、さまざまなドキュメントをダウンロードできます。特に「ツールとダウンロード」タブにある「インテル® Xeon Phi™ コプロセッサー・ソフトウェア開発者ガイド」(英語) が役立ちます。このページには、ツールに関する情報もあります。
    - http://www.isus.jp/article/performance-special/knights-corner-open-source-software-stack/ の RESOURCES (including downloads) リンクから、インテル® Xeon Phi™ コプロセッサーの uOS のソースコード、GDB のネイティブ・インテル® MIC アーキテクチャー・バージョン、そして「インテル® Xeon Phi™ コプロセッサー命令セット・アーキテクチャー・リファレンス・マニュアル」(英語)、ABI ドキュメント「System V Application Binary Interface K1OM Architecture Processor Supplement」(英語)、「インテル® Xeon Phi™ コプロセッサーのパフォーマンス・モニタリング・ユニット」などのドキュメントを入手できます。
- 明示的なメモリー・コピー・モデルを使用するサンプル・オフロード・コード:
    - **C++**: `<install-dir>\Samples\ja_JP\C++\mic_samples`
    - **Fortran**: `<install-dir>\Samples\ja_JP\Fortran\mic_samples\`
    - **インテル® MKL**:
        - 次のサンプルは、コンパイラーによるオフロード支援を利用して、インテル® MKL を使用する方法を示しています。
          `<install-dir>\mkl\examples\mic_samples\mic_offload`
- 暗黙的なメモリー・コピー・モデルを使用するサンプル・オフロード・コード:
    - **C**: `<install-dir>\Samples\ja_JP\C++\mic_sample\LEO_tutorial`

ビルドに関する情報

- オフロード・コンパイラーは、ホスト用のコードとインテル® Xeon Phi™ コプロセッサー用のコードを含む「ファット」バイナリーと .dll ファイルを生成します。
- 利用可能なインテル® Xeon Phi™ コプロセッサーがあるかどうか、ランタイム実行環境を確認するコードも生成します。利用可能なコプロセッサーがない場合、オフロードは失敗し、プログラムはエラーメッセージを出力して終了します。
- *release-notes-*-2013-w-JA.pdf* に多くの回避方法とヒントが掲載されています。

コンパイラー・オプション

一部のコードをインテル® Xeon Phi™ コプロセッサーにオフロードするアプリケーションをビルドする場合、ホストコードとオフロードコードで異なるコンパイラー・オプションを指定できます。各コンパイラー・オプションの指定方法は、コンパイラー・ドキュメントの「コンパイラー・リファレンス」>「コンパイラー・オプション」>「コンパイラー・オプションのカテゴリーと説明」セクションに記載されています。ここで、/Qoffload-option オプションと /Qoffload-attribute-target オプションの説明を確認してください。場合によっては、ソースファイルを変更する代わりに /Qoffload-attribute-target オプションを利用できます (このオプションは、プラグマベースのオフロード手法に適用されます)。また、/Qoffload- (あるいは /Qoffload:none) を指定すると、コンパイラーはオフロード言語拡張を無視します (その結果、デフォルトでヘテロジニアス・バイナリーが生成されます)。

実行中のオフロード・アクティビティーのデバッグ

オフロード・アクティビティーをデバッグするには、OFFLOAD_REPORT 環境変数を利用します。

- プログラムのオフロード領域がホストで実行されているか、コプロセッサーで実行されているかを検出し、デバッグ情報を取得するには、次のように設定します。

```
prompt> set OFFLOAD_REPORT=3
```

- 1 に設定するとホストにより測定されたオフロード時間とコプロセッサーによる計算時間のみ出力されます。2 に設定すると、ホストとコプロセッサー間のデータ転送量も出力されます。

```
prompt> set OFFLOAD_REPORT=<1 または 2>
```

詳細は、コンパイラー・ドキュメントの「コンパイル」>「サポートされている環境変数」セクションを参照してください。

オフロード・コンパイラーの使用 – 明示的なメモリー・コピー・モデル

このセクションでは、リダクションを例に、オフロード・コンパイラーでインテル® Xeon Phi™ コプロセッサー向けアプリケーションを生成する方法を説明します。オフロード・コンパイラーは、ホスト CPU とターゲット・コンパイル環境に対応するヘテロジニアス [2]・コンパイラーです。ホスト CPU 用のコードとインテル® Xeon Phi™ コプロセッサー用のコードはどちらもホスト環境でコンパイルされ、オフロードコードは自動的にターゲット環境で実行されます。オフロード動作は、コンパイラー・プラグマ (C/C++) とコンパイラー宣言子 (Fortran) により制御されます。

インテル® マス・カーネル・ライブラリー (インテル® MKL) のような一部の一般的なライブラリーには、CPU バージョンとターゲットバージョンがあります。アプリケーションで最初のオフロードを実行する際にターゲットが利用可能であれば、ランタイムはインテル® Xeon Phi™ コプロセッサーにターゲット用の実行ファイルをロードし、ターゲットコードとリンクされているライブラリーを初期化します。ホストプログラムが終了するまで、ロードしたターゲット用の実行ファイルはターゲットのメモリーに残ります。そのため、ライブラリーによって維持されるグローバル状態はすべてのオフロード・インスタンスで維持されます。

注: プログラマーがターゲットで実行するコード領域を指定した場合であっても、そのコード領域がインテル® Xeon Phi™ コプロセッサーで実行される保証はありません。ターゲット・ハードウェアが利用可能かどうか、あるいは実行がオフロード領域に到達したときにインテル® Xeon Phi™ コプロセッサーで利用可能なリソースに応じて、そのコードをインテル® Xeon Phi™ コプロセッサーで実行するか、ホストで実行するかが決定されます。

次の例は、オフロードプラグマを使って、リダクション・コードをインテル® Xeon Phi™ コプロセッサー向けに移植する方法を示します。

リダクション

次の式を計算します。

```
ans = a[0] + a[1] + … + a[n-1]
```

ホストバージョン:

次のサンプル C コードは、このリダクション操作を実装します。

[2] http://dictionary.reference.com/browse/heterogeneous

```
float reduction_serial(float *data, int size)
{
    float ret = 0.f;
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

サンプルコード 1: リダクション・コードの実装 (C/C++)

オフロードバージョンの作成

オフロードを使用するシリアル・リダクション

プログラマーは、(次のサンプルコードに示すように) **#pragma offload target(mic)** を用いてインテル® Xeon Phi™ コプロセッサーで実行する文 (オフロード構造) を指定できます。オフロード領域は、オフロード構造と関数呼び出しによりターゲットで実行される追加のコード領域で定義されます。ホスト上の文の実行は、ターゲット上の文の実行が完了すると再開され、ターゲットで実行された処理の結果はホストで利用できます (つまり、このプラグマには非同期実行が可能なバージョンがあるにもかかわらず、オフロードによりホストの実行がブロックされます)。**in**、**out**、**inout** 節は、ホストとターゲット間のデータ転送の方向を示します。

オフロード構造の有効範囲外 (ファイルの有効範囲外も含む) で宣言され、オフロード構造内で使用される変数は、実行前にホストからターゲットにコピーされ (デフォルト)、実行後にターゲットからホストに戻されます。

例えば、次のコードで変数 **ret** は、実行前にホストからターゲットに自動的にコピーされ、実行後にターゲットからホストに戻されます。次のオフロードコードは、1 つのインテル® MIC アーキテクチャー・ベースのコアで 1 つのスレッドによって実行されます。

```
float reduction_offload(float *data, int size)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(size))
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

サンプルコード 2: オフロードを使用するシリアル・リダクション

オフロードを使用するベクトル・リダクション

インテル® Xeon Phi™ コプロセッサーの各コアには VPU が装備されています。オフロード・コンパイラーでは、デフォルトで自動ベクトル化オプションが有効になります。さらに、次のコードのように、インテル® Cilk™ Plus の配列表記でベクトル化を最大限にし、インテル® MIC アーキテクチャー・ベースのコアにある 32 個の 512 ビット・レジスターを利用することができます。このオフロードコードは、1 つのコアで 1 つのスレッドによって実行されます。スレッドは、ビルトイン

のリダクション関数 *__sec_reduce_add()* により、コアの 32 個の 512 ビット・ベクトル・レジスターを使用し、一度に配列の 16 個の要素をレデュースします。

```
float reduction_vectorreduction(float *data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(data:length(size))
    ret = __sec_reduce_add(data[0:size]); // インテル® Cilk™ Plus の配列表記

    return ret;
}
```

サンプルコード 3: オフロードを使用するベクトル・リダクション (C/C++)

非同期オフロードとデータ転送

ホストとインテル® Xeon Phi™ コプロセッサー間では、非同期のオフロードおよびデータ転送が可能です。詳細は、『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』の「主な機能」>「インテル® MIC アーキテクチャー」>「インテル® MIC アーキテクチャー向けプログラミング」以下にある、「非同期計算について」と「非同期データ転送について」を参照してください。

非同期のオフロードおよび転送の使用例は、次のサンプルを参照してください。

`<install-dir>\Samples\ja_JP\C++\mic_samples\ intro_sampleC\sampleC13.c`

C/C++ の明示的なメモリー・コピー・モデルでは、配列要素がスカラー型か、ビット単位でコピーできる構造体またはクラスの場合のみ、配列を利用できます。ポインターの配列はサポートされません。C/C++ の複雑なデータ構造では、暗黙的なメモリー・コピー・モデルを使用してください。詳細は、『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』の「主な機能」>「インテル® MIC アーキテクチャー」>「インテル® MIC アーキテクチャー向けプログラミング」>「プラグマを使用したオフロード」>「プラグマを使用したオフロードの制約事項」を参照してください。

オフロード・コンパイラーの使用 – 暗黙的なメモリー・コピー・モデル

インテル® Composer XE 2013 では、リンクリストやバイナリーツリーなどの複雑なポインター・ベースのデータ構造を処理するため、「共有メモリー」オフロード・プログラミング・モデルを提供する C/C++ 向けの 2 つのキーワード拡張 (`_Cilk_shared` および `_Cilk_offload`) が新たに追加されています (これらのキーワードは Fortran では利用できません)。「共有メモリー」オフロード・プログラミング・モデルは、ホストとコプロセッサーで共有する (`_Cilk_shared` キーワードで示された) 変数をそれぞれのマシンの同じ仮想アドレスに配置し、`_Cilk_offload` キーワードで示されたオフロード関数の開始時と終了時にその値を同期します。また、同期するデータは、それぞれのマシンで同じ仮想アドレスにメモリーが割り当てられることを保証する、特殊な割り当て/解放関数を用いて動的に割り当てられます。

共有メモリーの動的割り当て API:

```
void *_Offload_shared_malloc(size_t size);
_Offload_shared_free(void *p);
```

アライメントされた共有メモリーの動的割り当て API:

```
void *_Offload_shared_aligned_malloc(size_t size, size_t alignment);
_Offload_shared_aligned_free(void *p);
```

これは、実際には「共有メモリー」でないことに注意してください。インテル® Xeon Phi™ コプロセッサーの一部をホストシステムに割り当てることができるハードウェアはありません。コプロセッサーとホストのメモリー・サブシステムは完全に独立しており、このプログラミング・モデルは、適切に定義された同期ポイントで 2 つのメモリー・サブシステム間のデータをコピーする方法の 1 つにすぎません。コピーは暗黙的に行われます。同期ポイント (_Cilk_offload で示されたオフロード呼び出し) でコピーするデータを指定しません。代わりに、ホストとコプロセッサー間で変更のあったデータをランタイムが特定し、差分のみをオフロード呼び出しの開始時と終了時にコピーします。

次のコードは、_Cilk_shared および _Cilk_offload キーワードの使用法と「共有」メモリーを動的に割り当てる方法を示します。

```
float * _Cilk_shared data; //「共有」メモリーへのポインター

_Cilk_shared float MIC_OMPReduction(int size)
{
    float Result;
    #ifdef __MIC__
    int nThreads = 60;
    omp_set_num_threads(nThreads);

    #pragma omp parallel for reduction(+:Result)
    for (int i=0; i<size; ++i)
    {
        Result += data[i];
    }
    return Result;
    #else
    printf("Intel(R) Xeon Phi(TM) Coprocessor not available\n");
    #endif
    return 0.0f;
}

int main()
{
    size_t size = 1*1e6;
    int n_bytes = size*sizeof(float);
    float Result;

    data = (_Cilk_shared float *)_Offload_shared_malloc (n_bytes);
    for (int i=0; i<size; ++i)
    {
        data[i] = i%10;
    }

    Result = _Cilk_offload MIC_OMPReduction(size);
    Printf("Cilk Offload Result=%.0f\n",Result);

    _Offload_shared_free(data);
    return 0;
}
```

サンプルコード 4: _Cilk_shared および _Cilk_offload キーワードと動的割り当ての使用 (C/C++)

このほかにも、暗黙的なメモリー・コピー・モデルの使用例として次のサンプルがあります。

C: <install-dir>\Samples\ja_JP\C++\mic_samples 以下の shrd_sampleC および LEO_tutorial

C++: <install-dir>\Samples\ja_JP\C++\mic_samples\shrd_sampleCPP

また、『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』、『インテル® Fortran コンパイラー・ユーザー・リファレンス・ガイド』も参考にしてください。

『インテル® C++ コンパイラー・ユーザー・リファレンス・ガイド』の「主な機能」>「インテル® MIC アーキテクチャー」>「インテル® MIC アーキテクチャー向けプログラミング」>「共有仮想メモリーを使用したオフロード」>「共有仮想メモリーを使用したオフロードコードの制約」セクションに、このプログラミング・モデルに関するいくつかの制約が記載されています。

ネイティブコンパイル

アプリケーションをインテル® Xeon Phi™ コプロセッサーでネイティブ実行することもできます。その場合、コプロセッサーはスタンドアロンのマルチコア・コンピューターと見なされます。ホストシステムでバイナリーをビルドしたら、バイナリーと関連バイナリーおよびデータをインテル® Xeon Phi™ コプロセッサーのファイルシステムにコピーします (または、コプロセッサーが NFS を介して必要なファイルにアクセスできるようにします)。

例:

1. `<install-dir>\Samples\ja_JP\C++\openmp_samples.zip` を展開し、`openmp_sample.c` をホーム・ディレクトリーにコピーします。

2. `/Qmic` オプション を指定してアプリケーションをビルドします。

```
icl /Qmic -openmp openmp_sample.c
```

3. WinSCP* ツールを使って、バイナリー `a.out` をコプロセッサー `mic0` のディレクトリー (ここでは `/tmp` を使用) にアップロードします。

4. アプリケーションで必要なすべての共有ライブラリーをコプロセッサーのディレクトリーにコピーします。ここでは、`c:\Program Files (x86)\Common Files\Intel\Shared Libraries\compiler\lib\mic\lib\libiomp5.so` にある OpenMP* ランタイム・ライブラリーをコプロセッサーの `/tmp` ディレクトリーにコピーします。

5. PuTTY* を使ってコプロセッサーに接続し、アプリケーションが必要な共有ライブラリー (ここでは、OpenMP* ランタイム・ライブラリー) にアクセスできるように、ローカル・ディレクトリーをエクスポートします。

```
export LD_LIBRARY_PATH=/tmp
```

6. 適切なスタックサイズが設定されていない場合、このアプリケーションはセグメンテーション違反になります。スタックサイズを変更するには、次のコマンドを実行します。

```
ulimit –s unlimited
```

7. `/tmp` ディレクトリーに移動し、`a.out` を実行します。

```
cd /tmp
./a.out
```

インテル® Xeon Phi™ コプロセッサーで利用可能な並列プログラミング・モデル

ホストシステムで利用可能な並列プログラミング・モデルのほとんどは、以下を含めインテル® Xeon Phi™ コプロセッサーでも利用できます。

1. インテル® スレッディング・ビルディング・ブロック (インテル® TBB)
2. OpenMP*
3. インテル® Cilk™ Plus
4. Pthreads*

後続のセクションでは、オフロード拡張により、コードでこれらの並列プログラミング・モデルを使用する方法を述べます。インテル® Xeon Phi™ コプロセッサーでネイティブ実行するコードでは、スレッド数が多いことを除き、特に問題なくホストと同様にこれらの並列プログラミング・モデルを使用できます。

インテル® Xeon Phi™ コプロセッサーで利用可能な並列プログラミング・モデル: OpenMP*

ホスト CPU の OpenMP* スレッドとインテル® Xeon Phi™ コプロセッサーの OpenMP* スレッド間で通信は発生しません。オフロード/プラグマ内の OpenMP* 並列領域は 1 つの単位としてオフロードされ、オフロード・コンパイラーは、インテル® Xeon Phi™ コプロセッサーで利用可能なリソースに応じてスレッドチームを作成します。OpenMP* 構文全体がインテル® Xeon Phi™ コプロセッサーで実行されるため、構文内では、共有データおよびプライベート・データに対し通常の OpenMP* セマンティクスが適用されます。

いつでも、複数のホスト CPU スレッドがインテル® Xeon Phi™ コプロセッサーにオフロードできます。CPU スレッドがインテル® Xeon Phi™ コプロセッサーへのオフロードを試み、コプロセッサーに利用可能なリソースがない場合、オフロードコードはホストで実行されます。コプロセッサー上のスレッドは omp parallel 宣言子に到達すると、コプロセッサーで利用可能なリソースに応じてスレッドチームを作成します。作成可能なハードウェア・スレッドの理論的な最大数は、インテル® Xeon Phi™ コプロセッサーのコア数の 4 倍です。実際には、1 つのコアが uOS とそのサービス用に予約されるため、これよりも 4 つ少なくなります (オフロードコードの場合)。

次のサンプルコードは、オフロード構造で OpenMP* を使用し、1 つのホスト CPU スレッドでリダクション・コードをインテル® Xeon Phi™ コプロセッサーにオフロードします。

```
float OMP_reduction_OMP(float *data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(size) in(data:length(size))
    {
        #pragma omp parallel for reduction(+:ret)
        for (int i=0; i<size; ++i)
        {
            ret += data[i];
        }
    }
    return ret;
}
```

サンプルコード 5: オフロード・リダクション・コードでの OpenMP* の使用 (C/C++)

オフロード・リダクション・コードでの OpenMP* の使用例を示す Fortran サンプルは、`<install-dir>\Samples\ja_JP\Fortran\mic_samples\LEO_Fortran_intro` を参照してください。

```
real function FTNReductionOMP(data, size)
    implicit none
    integer :: size, i
    real, dimension(size) :: data
    real :: ret = 0.0

!dir$ omp offload target(mic) in(size) in(data:length(size))
!$omp parallel do reduction(+:ret)
    do i=1,size
        ret = ret + data(i)
    enddo

!$omp end parallel do

    FTNReductionOMP = ret
    return
end function FTNReductionOMP
```

サンプルコード 6: オフロード・リダクション・コードでの OpenMP* の使用 (Fortran)

インテル® Xeon Phi™ コプロセッサーで利用可能な並列プログラミング・モデル: OpenMP* + インテル® Cilk™ Plus の配列表記

次のサンプルは、OpenMP* とインテル® Cilk™ Plus の配列表記を併用する方法を示します。各スレッドは、インテル® Cilk™ Plus 配列表記のビルトイン・リダクション関数 **__sec_reduce_add()** により、インテル® MIC アーキテクチャーの 32 個の 512 ビット・ベクトル・レジスターをすべて使って、配列要素をレデュースします。

```
float OMPnthreads_CilkPlusEAN_reduction(float *data, int size)
{
    float ret=0;
    #pragma offload target(mic) in(data:length(size))
    {
        int nthreads = omp_get_max_threads();
        int ElementsPerThread = size/nthreads;
        #pragma omp parallel for reduction(+:ret)
        for(int i=0;i<nthreads;i++)
        {
            ret =__sec_reduce_add(
                  data[i*ElementsPerThread:ElementsPerThread]);
        }
        // 配列の残りの要素
        for(int i=nthreads*ElementsPerThread; i<size; i++)
        {
            ret+=data[i];
        }
    }
    return ret;
}
```

サンプルコード 7: Open MP* とインテル® Cilk™ Plus を併用する配列のリダクション (C/C++)

インテル® Xeon Phi™ コプロセッサーで利用可能な並列プログラミング・モデル: インテル® Cilk™ Plus

デフォルトでは、インテル® Cilk™ Plus のヘッダーファイルはターゲット環境で利用できません。インテル® Cilk™ Plus を使うインテル® MIC アーキテクチャー向けアプリケーションでこれらのヘッダーファイルを利用するには、次のように *#pragma offload_attribute(push,target(mic))* と *#pragma offload_attribute(pop)* でヘッダーファイルをラップします。

```
#pragma offload_attribute(push,target(mic))
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#pragma offload_attribute(pop)
```

サンプルコード 8: ヘッダーファイルをラップ (C/C++)

次のサンプルでコンパイラーは、効率良い分割統治法により *cilk_for* ループを再起呼び出し関数に変換します。

```
float ReduceCilk(float*data, int size)
{
    float ret = 0;
    #pragma offload target(mic) in(data:length(size))
    {
        cilk::reducer_opadd<int> total;
        cilk_for (int i=0; i<size; ++i)
        {
            total += data[i];
        }
        ret = total.get_value();
    }
    return ret;
}
```

サンプルコード 9: *cilk_for* ループを再起呼び出し関数に変換

インテル® Xeon Phi™ コプロセッサーで利用可能な並列プログラミング・モデル: インテル® TBB

インテル® Cilk™ Plus と同様に、デフォルトでは、インテル® TBB のヘッダーファイルはターゲット環境で利用できません。インテル® Cilk™ Plus と同様の方法で、これらのヘッダーファイルをインテル® MIC アーキテクチャー・ベースのターゲット環境で利用できるようにします。

```
#pragma offload_attribute (push,target(mic))
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "tbb/task.h"
#pragma offload_attribute (pop)

using namespace tbb;
```

サンプルコード 10: インテル® TBB ヘッダーファイルをラップ (C/C++)

オフロード構造内で呼び出される関数とインテル® Xeon Phi™ コプロセッサーで必要なグローバルデータには __declspec(target(mic)) プリフィクスを追加します。

例えば、*parallel_reduce* は、分割コンストラクターを使用して再帰的に配列をサブ範囲に分割し、各スレッドに 1 つ以上のコピー (作業) を割り当てます。そして、各分割ごとに、join メソッドを呼び出して結果を集計します。

1. コプロセッサー向けのバージョンを生成する場合は、プリフィクスとして各クラスに __MIC__ マクロを、クラス名に __declspec(target(mic)) を追加します。

```
#ifdef __MIC__
class __declspec(target(mic)) ReduceTBB
{
private:
    float *my_data;
public:
    float sum;

    void operator()( const blocked_range<size_t>& r )
    {
        float *data = my_data;
        for( size_t i=r.begin(); i!=r.end(); ++i)
        {
            sum += data[i];
        }
    }

      ReduceTBB( ReduceTBB& x, split) : my_data(x.my_data), sum(0) {}

      void join( const ReduceTBB& y) { sum += y.sum; }

      ReduceTBB( float data[] ) : my_data(data), sum(0) {}
};
#endif
```

サンプルコード 11: インテル® MIC アーキテクチャー向けコードを生成するため インテル® TBB クラスにプリフィクスを追加 (C/C++)

2. インテル® Xeon Phi™ コプロセッサーへオフロードする関数に、プリフィクス __declspec(target(mic)) を追加します。

```
__declspec(target(mic))
float MICReductionTBB(float *data, int size)
{
    ReduceTBB redc(data);
    // ライブラリーの初期化
    task_scheduler_init init;
    parallel_reduce(blocked_range<size_t>(0, size), redc);
    return redc.sum;
}
```

サンプルコード 12: インテル® MIC アーキテクチャー向けコードを生成するため インテル® TBB 関数にプリフィクスを追加 (C/C++)

3. `#pragma offload target(mic)` を指定して、インテル® TBB の並列コードをコプロセッサーにオフロードします。

```
float MICReductionTBB(float *data, int size)
{
    float ret(0.f);
    #pragma offload target(mic) in(size) in(data:length(size)) out(ret)
    ret = _MICReductionTBB(data, size);
    return ret;
}
```

サンプルコード 13: インテル® TBB コードをコプロセッサーへオフロード (C/C++)

注: オフロードで使用するインテル® TBB コードは、`/Qtbb` オプションを指定してビルドします。

インテル® MKL の使用

オフロードする場合、インテル® MKL はよくネイティブ・アクセラレーション (NAcc) モードが使用されます。NAcc では、すべてのデータとバイナリーがインテル® Xeon Phi™ コプロセッサー上に配置されます。データは、オフロード・コンパイラー・プラグマとオフロード領域内またはオフロード関数内のインテル® MKL 呼び出しで使用されるセマンティクスを用いて、プログラマーによって転送されます。NAcc では、BLAS、LAPACK、FFT、VML、VSL、(スパース行列ベクトル) と、必要なインテル® MKL サービス関数を利用できます。最適化されている関数、サポートされていない関数などの詳細は、インテル® MKL のリリースノートを参照してください。

NAcc モードは、インテル® MIC アーキテクチャー向けのネイティブコードでも使用できます。その場合、実行前にインテル® MKL 共有ライブラリーをインテル® Xeon Phi™ コプロセッサーにコピーする必要があります。

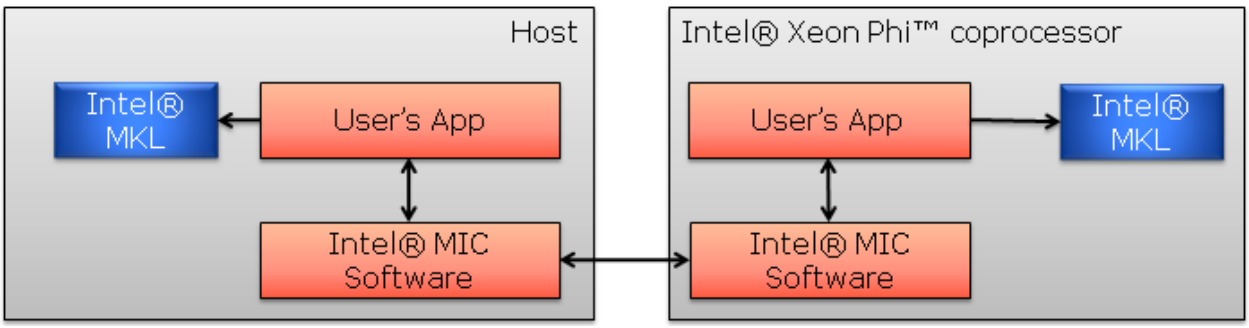

図 13: オフロードを使用するインテル® MKL のネイティブ・アクセラレーション

SGEMM サンプル

BLAS ライブラリーから SGEMM ルーチンを使用します。

サンプルコード – sgemm

ステップ 1: 行列を初期化します。このサンプルでは、データが保持されるように、行列をグローバル変数にする必要があります。

ステップ 2: `#pragma offload` を指定して、インテル® Xeon Phi™ コプロセッサーにデータを転送します。このサンプルでは、`free_if(0)` 修飾子を使って、インテル® Xeon Phi™ コプロセッサーでデータが保持されるようにします。

```
#define PHI_DEV 0
#pragma offload_transfer target(mic:PHI_DEV) \
    in(A:length(matrix_elements) free_if(0)) \
    in(B:length(matrix_elements) free_if(0)) \
    in(C:length(matrix_elements) free_if(0))
```

サンプルコード 14: インテル® Xeon Phi™ コプロセッサーへのデータ転送

ステップ 3: オフロード領域内で sgemm を呼び出し、インテル® Xeon Phi™ コプロセッサーでインテル® MKL の NAcc バージョンを使用します。`nocopy( )` 修飾子により、ステップ 2 でコピーしたデータを再利用します。

```
#pragma offload target(mic:PHI_DEV) \
    in(transa, transb, N, alpha, beta) \
    nocopy(A: alloc_if(0) free_if(0)) nocopy(B: alloc_if(0) free_if(0)) \
    out(C:length(matrix_elements) alloc_if(0) free_if(0)) // output data
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
            &beta, C, &N);
    }
```

サンプルコード 15: オフロード領域内での sgemm の呼び出し

ステップ 4: ステップ 2 でカードにコピーしたメモリーを解放します。オフロード領域の開始時に `alloc_if(0)` 修飾子でカードにあるデータを再利用し、終了時に `free_if(1)` 修飾子でカード上のデータを解放します。

```
#pragma offload_transfer target(mic:PHI_DEV) \
        nocopy(A:length(matrix_elements) alloc_if(0) free_if(1)) \
        nocopy(B:length(matrix_elements) alloc_if(0) free_if(1)) \
        nocopy(C:length(matrix_elements) alloc_if(0) free_if(1))
```

サンプルコード 16: コピーしたメモリーの解放

ほかのプラットフォームでインテル® MKL を使用する場合と同様に、オフロードコード内でインテル® MKL 関数を実行する前に、許容する OpenMP* スレッドの数を設定することで使用するスレッドの数を制限できます。

```
#pragma offload target(mic:PHI_DEV) \
    in(transa, transb, N, alpha, beta) \
    nocopy(A: alloc_if(0) free_if(0)) nocopy(B: alloc_if(0) free_if(0))
    out(C:length(matrix_elements) alloc_if(0) free_if(0)) // output data
    {
        omp_set_num_threads(64); // set num threads in openmp
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
            &beta, C, &N);
    }
```

サンプルコード 17: omp_set_num_threads() を使用してインテル® Xeon Phi™ コプロセッサーのスレッド数を制御

インテル® MKL の自動オフロードモデル

ホスト用のインテル® MKL 関数のいくつかは自動オフロードに対応しており、ホストで通常通り呼び出すことができます。しかし、ライブラリー呼び出しの前に `mkl_mic_enable()` 呼び出しがある場合、インテル® MKL は実行時に自動で問題サイズ、プロセッサーとコプロセッサーの負荷、その他のメトリックを考慮して、呼び出しを完了するのに必要な作業の一部またはすべてを、ホストとインテル® Xeon Phi™ コプロセッサー間で分配すべきかどうかを決定します。この機能は、`mkl_mic_disable()` で無効にできます。

自動オフロードは、`_Cilk_offload` または `#pragma offload` によりインテル® Xeon Phi™ コプロセッサーで実行されるコードとは**別に**、選ばれたインテル® MKL ライブラリー呼び出しにのみ適用されます。そのため、自動オフロード呼び出しと `_Cilk_offload` または `#pragma offload` によりコプロセッサーで実行されるコードの両方で、同じデータの転送を最小限に抑える必要があります。現在、自動オフロードと、プログラマーによって (`_Cilk_offload` または `#pragma offload` を介して) 制御される明示的なオフロード間の共通データをコプロセッサー上に保持する方法はありません。

自動オフロードの制御方法を示すサンプルは、`<install-dir>\mkl\examples\examples_mic\mic_ao\blasc` (C コード) と `<install-dir>\mkl\examples\examples_mic\mic_ao\blasf` (Fortran コード) があります。

インテル® Xeon Phi™ コプロセッサー上のデバッグ

インテル® MIC アーキテクチャー向けアプリケーションのデバッグに関する情報は、**`<install-dir>\Documentation\ja_JP\debugger\gdb\pdf\vsmigdb_config_guide.pdf`** を参照してください。

インテル® Xeon Phi™ コプロセッサー上のパフォーマンス解析

インテル® VTune™ Amplifier XE 2013 Windows* 版を使って、インテル® Xeon Phi™ コプロセッサーでアプリケーションを最適化する方法は、http://www.isus.jp/article/idz/mic-developer/ の「プログラミング」タブの「最適化」セクションにある「最適化とパフォーマンス・チューニング – パート 2: ハードウェア・イベントの理解と使用」を参照してください。

付録 A: Linux* の基本コマンド

ここでは、インテル® Xeon Phi™ コプロセッサーでよく使用されるいくつかの Linux* コマンドを紹介します。

1. exit: コプロセッサーからログアウトします。
   コプロセッサーのターミナルでこのコマンドを実行すると、ユーザーをログアウトしターミナルを閉じることができます。
   ```
   > exit
   ```

2. ls: 現在のディレクトリーにあるファイルとサブディレクトリーのリストを表示します。

   ls –l コマンドを実行すると、すべてのファイルとサブディレクトリーおよびその属性が表示されます。
   ```
   > ls
   a.out
   libioomp5.so
   ……
   ```

3. pwd: 現在のディレクトリーのパスを表示します。
   ```
   > pwd
   /root
   ```

4. cd <*path*>: 特定のディレクトリーに移動します。
   ```
   > cd /tmp
   ```

5. ps: 現在実行中のプロセスをすべて表示します。
   ```
   > ps
   5847 root    0:00 /sbin/sshd
   5914 micuser 2:47 /bin/coi_daemon –coiuser=micuser
   ……
   ```

6. kill -9 <*pid*>: プロセス ID を指定してプロセスを強制終了します。
   ```
   > kill -9 4555
   ```

7. top: 最も CPU 時間を費やしているプロセスを表示します。
   ```
   > top
   ```

8. cp <*sourcefilename*> <*destinationfilename*>: ファイルをコピーします。
   ```
   > cp file1 file2
   ```

9. rm <*filename*>: ファイルを削除します。
   ```
   > rm file1
   ```

10. less <*filename*>: ファイルの内容を確認します。
    ```
    > less file2
    ```

11. grep <*pattern*>: 指定したパターンと一致する行を検索します。例えば、coi という名前のプロセスを検索する場合、grep と ps を併用できます。
```
> ps | grep coi
5914 micuser 2:47 /bin/coi_daemon –coiuser=micuser
```

12. export: 特定の環境変数を設定します。

```
> export LD_LIBRARY_PATH=/tmp
```

13. Ctrl キー + C: 現在実行中のタスクを終了します。

著者紹介

Loc Q Nguyen

ダラス大学で MBA を、マギル大学で電気工学の修士号を、モントリオール理工科大学で電気工学の学士号を取得しています。現在は、インテル コーポレーションのソフトウェア & サービスグループのソフトウェア・エンジニアで、コンピューター・ネットワーク、コンピューター・グラフィックス、並列処理を研究しています。

著作権と商標について

最適化に関する注意事項

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。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® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットに関する詳細は、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。

改訂 #20110804