OpenMP* 4.x における拡張

オフロード

sugatech

内容

- デバイス (アクセラレーター) 拡張
- 入れ子の並列化制御
- プロセッサーバインドとアフィニティーの制御

内容

- デバイス (アクセラレーター) 拡張
  - 基本
  - データ移動
  - 永続性
  - 並行 (非同期) 実行
  - インテル® コンパイラーの
    オフロード向け言語拡張 (LEO) ←→ OpenMP*

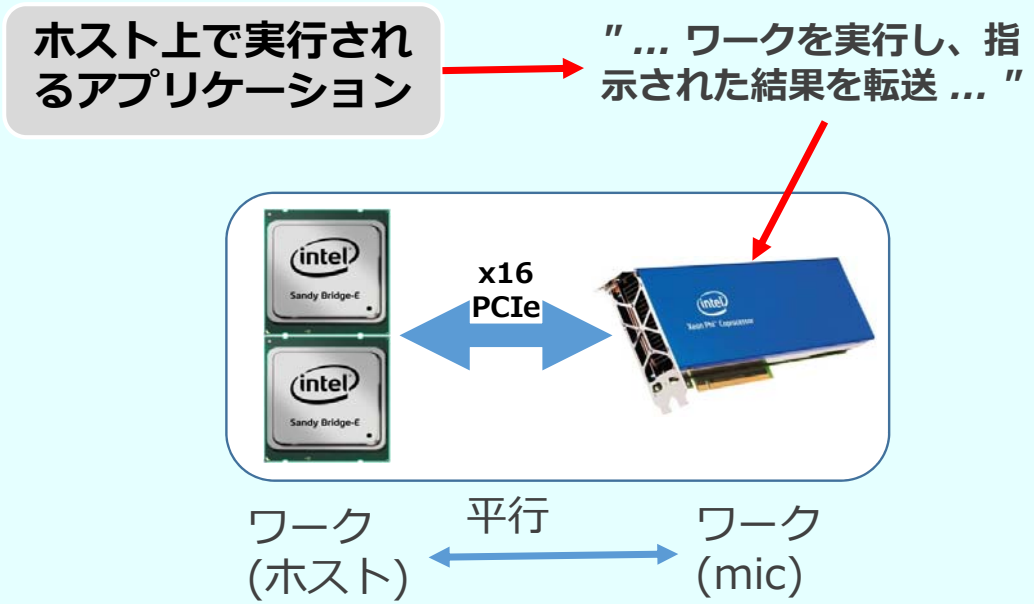
内容

- デバイス (アクセラレーター) 拡張
  - 基本
  - データ移動
  - 永続性
  - 並行 (非同期) 実行
  - インテル® コンパイラーの
    オフロード向け言語拡張 (LEO) ←→ OpenMP*

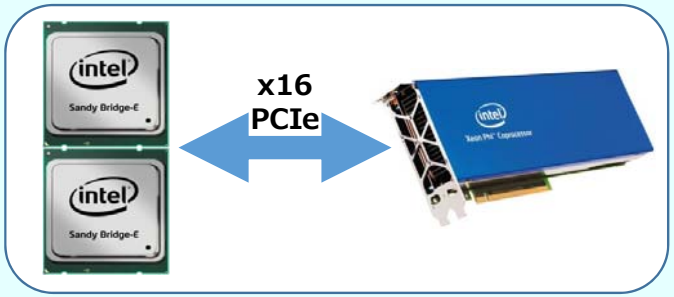
オフロード:他のプロセッサーで実行

ホスト上で動作するプログラムは、コードの特定のブロックを MIC で実行するため、ワークを “オフロード” する。 **また、ホストはホストとデバイス間のデータ転送を指示する**

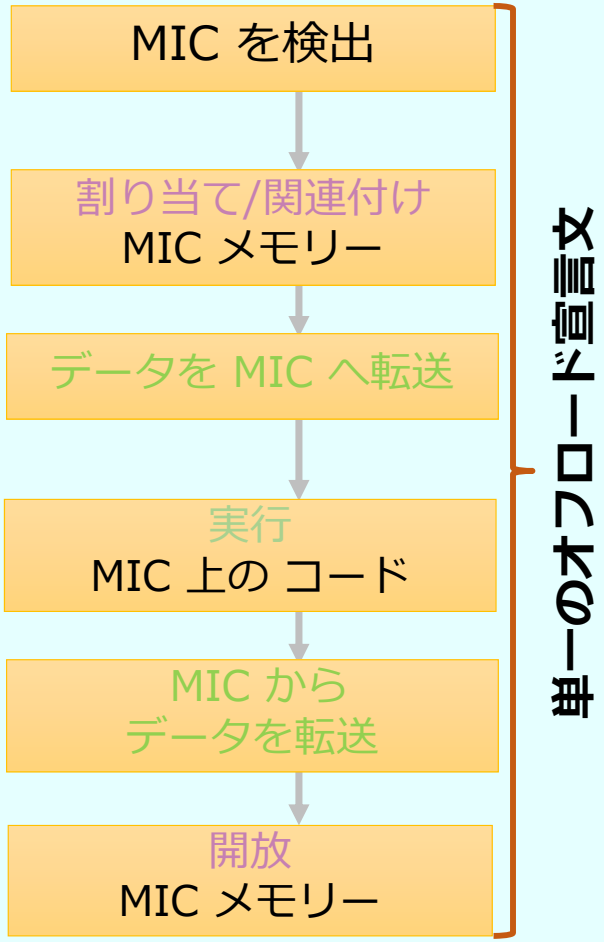
デバイス (別のプロセッサー) が割り当てられたワークを実行している間、**ホストがアクティブ状態を保つのが理想的**



オフロードを制御



データ転送
割り当て/開放
平行性

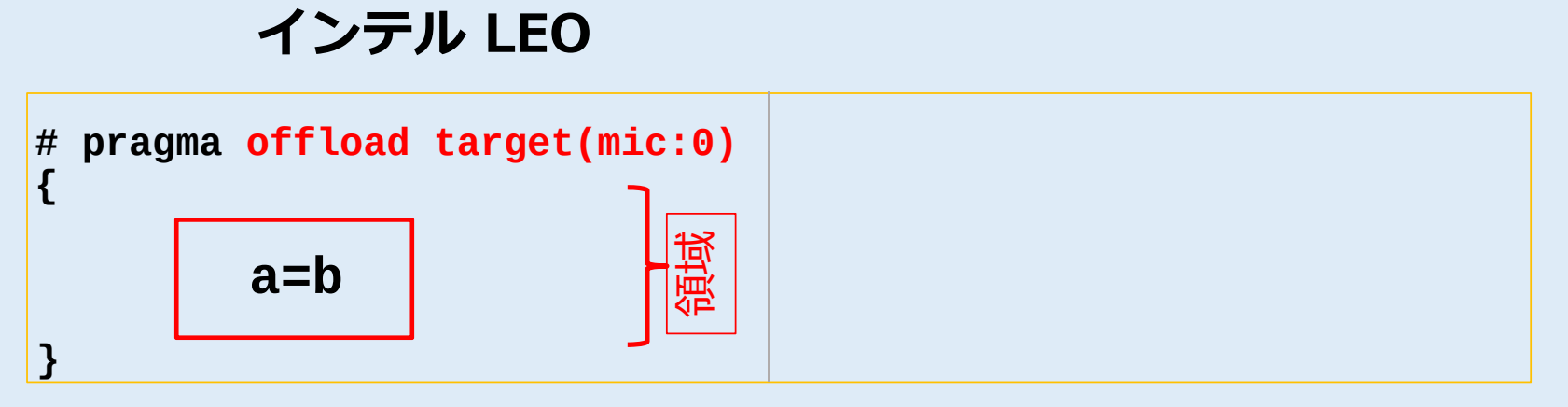


自動オフロードをさらに効率よく行うには、「修飾子」が必要
(句、属性、指定子、キーワード)

sugatech

基本オフロード
同期(デバイス選択、データ転送、およびデバイス上のストレージ)

データが構文の範囲内にある場合、変数はデバイスへ/から転送され、領域の入口/出口で割り当て/開放される



OpenMP* 4.0 への移行

- 基本操作は同じ
- シンタックスが異なる

インテル LEO

```
# pragma offload target(mic:0)
{
        a=b
}
```

OpenMP

```
# pragma omp target device(0)
{
    a=b
}
```

データ転送の方向

- PCIe バスの帯域幅を抑える

インテル LEO

```
#pragma offload target(mic:0) ¥
        in(b),out(a),inout(c)
{
        a=b;c=c*c
}
```

OpenMP* 4.0 への移行

- OpenMP の動作は同じ、シンタックスが異なるだけ

インテル LEO	OpenMP

`#pragma offload target(mic:0) ¥`
`  in(b),out(a),inout(c)`
`{`
`  a=b;c=c*c`
`}`
	`#pragma omp target device(0) ¥`
	`map(to:b),map(from:a),map(tofrom:c)`
	`{`
	`  a=b;c=c*c`
	`}`

データ転送属性は省略可能

サンプルコード

Pi を求めるプログラムをインテル® コンパイラーのオフロード向け拡張と OpenMP 4.0 の機能を使用して記述

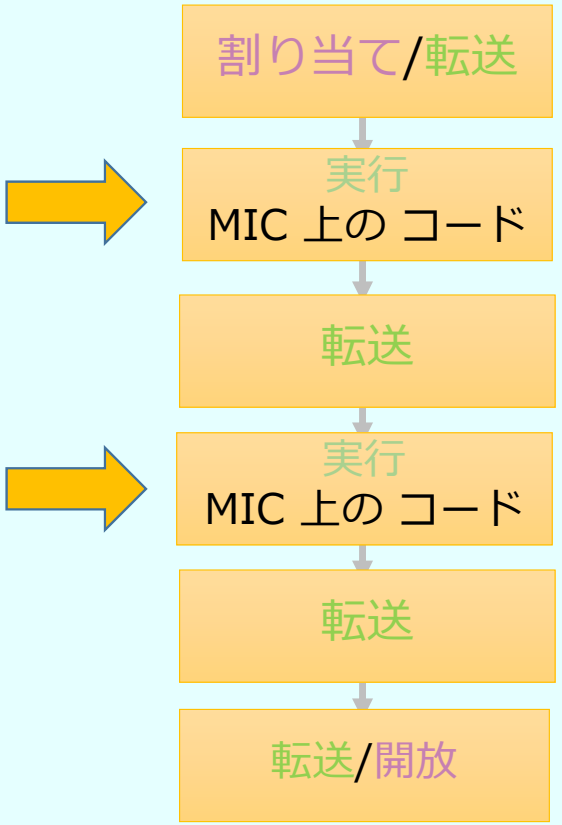
演習と手順

内容

- デバイス (アクセラレーター) 拡張
  - 基本
  - データ移動
  - 永続性
  - 並行 (非同期) 実行
  - インテル® コンパイラーの
    オフロード向け言語拡張 (LEO) ←→ OpenMP*

データの永続性

割り当て/開放　→ 永続性
データ転送



sugatech

データの永続性 -- LEO と OpenMP* 4.0

インテル LEO

新しい永続性マッピングは、
オフロード宣言文を含むどこ
にでも作成できる

非構造化

OpenMP

新しい永続性マッピングは、
この領域内で作成できない

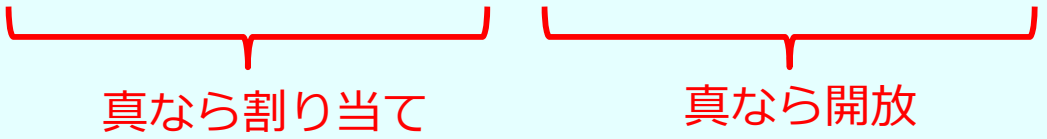
構造化

*OpenMP の非構造化は、
OpenMP 4.5 で利用可能

LEO のデータ永続性

非構造化転送：

```
#pragma offload_transfer ¥
in/out(vars:alloc_if(logical) free_if(logical))
```



オフロード宣言文でも指定可能：

```
#pragma offload ¥
in/out(vars:alloc_if(logical) free_if(logical) )
```


sugatech

OpenMP* のデータ永続性

構造化転送：

```
#progma target data device(0) ¥
        map([to|from|alloc]: vars)
{
   ...
}
```

デフォルトのマッピングは、
変数 vars の "tofrom"

OpenMP* 4.0 への移行

データの永続性

インテル LEO

```
#pragma offload_transfer target(mic:0)&
  in(c:alloc_if(1) free_if(0))
    !データは、割り当てられ、コピーされ、開放されない
...
#pragma offload target(mic:0) nocopy(c)
        a=b;c=c*c;   ! a & b, 自動; c 永続
...
 ! Offload_transfer は、データをデバイスからコピーし開放
#pragma offload_transfer target(mic:0)&
  out(c:alloc_if(0) free_if(1))
```

OpenMP*

```
#pragma omp target data device(0) &
        map(c)        !デフォルトは tofrom
!データは、割り当てられ、コピーされ、開放されない
...
{
#pragma omp target device(0)
        a=b;c=c*c;    ! a & b, 自動;
                      ! c 永続
...
                      !データ構造化ブロックの最後
                      !デフォルトは from
}
```

OpenMP* 4.5 における新しいデータの永続性

非構造化転送 :

```
#progma omp target enter data device(0) ¥
        map(to|alloc: vars)
```

非構造化転送 :

```
#progma omp target exit data device(0) ¥
        map(from|release|delete: vars)
```

delete = 完全に開放　　release = 参照カウントを減らす

サンプルコード

プログラム内に複数のオフロード領域がある場合、デバイス上のデータをどのように継続利用、もしくは廃棄するか？

オフロードのサンプルコード

内容

- デバイス (アクセラレーター) 拡張
  - 基本
  - データ移動
  - 永続性
  - 並行 (非同期) 実行
  - インテル® コンパイラーの
    オフロード向け言語拡張 (LEO) ←→ OpenMP*

非同期オフロード

同期実行：

- ホストのスレッド/プロセスは、オフロード構文で完了を待機

非同期実行：

- ホストのスレッド/プロセスは、オフロード構文を実行後、すぐに実行を継続し、wait で指示される場所で待機

平行実行 -- 簡単

LEO

- target … signal(id)

- signal 句
  - 非同期に行う
  - 追加のスレッドは必要なし
  - signal の引数は待機する "プロセス" を特定
    wait(id) でプロセスを待機
    (単独もしくは target 構文)

OpenMP*

- target … nowait
  インテル® コンパイラー 16 (OpenMP 4.5) でサポート
- nowait 句
  - 非同期に行う
  - 追加のスレッドが必要

  - task 識別子なし
    taskwait を使用

同時実行
LEO (ホストと MIC -- 簡単)

```
#pragma offload target(mic:0) signal(&isig)
#pragma omp parallel num_threads(60)
      work(noff1, nend1)      !mic で実行
```

!T1

```
#pragma omp parallel num_threads(16)
      work(noff2, nend2)      !ホストで実行
```

!T2

```
#pragma offload_wait target(mic:0) wait(&isig)
```

!T0

MIC

オフロードが完了するまでホストはポーズ

CPU

MIC への新たなワークも開始しない

インテルの非同期実装では、オフロード後即座にホストのほかの並列領域を実行することを許可

時間 (秒) は、ホスト上で計測

	Total	T0	T1	T2
非同期 off	2.11117	1.12804	0.98314	
非同期 on	1.08984	0.00002	1.01347	

同時実行
OpenMP* (ホストと MIC -- 簡単)

```
#pragma omp target parallel num_threads(60) nowait
    work(noff1, nend1)      ! mic で実行
```

MIC

```
#pragma omp parallel num_threads(16)
    work(noff2, nend2)     !ホストで実行

#pragma omp taskwait
```

CPU

- 複合構文に注意 (target と parallel)

LEO のまとめ

操作

関数定義、グローバル
コード実行

領域割り当て
データ転送
同時実行

同期

宣言文
句もしくは修飾子

attributes/declspec

offload

offload_transfer

alloc_if　free_if
in, out, inout, length
signal、wait

offload_wait

データ転送
永続性
非同期実行

OpenMP* まとめ

操作

関数定義、グローバル
コード実行

領域割り当て
データ転送
同時実行

同期

宣言文
句もしくは修飾子

declare target

target

target update

alloc, release, delete
map(to,from,tofrom) length
nowait、depend

taskwait

データ転送
永続性
非同期実行

OpenMP* 4.x への移行

- OpenMP 4.0 は、LEO (Language Extensions for Offload) と同様のデバイス Target 宣言を持っている
- インテル® コンパイラー 15.0 は、OpenMP 4.0 に準拠している Update 版は、4.5 のコンポーネントを組み込んでいる
- OpenMP 4.0 では、明示的な非同期句と非構造化データマッピングがサポートされていない。これらは、OpenMP 4.5 でサポートされる
- データの永続性はより簡単になるが、非構造化マッピングの制御はさらに必要となる。インテル® コンパイラー 16.0 は、非同期句 (nowait) をサポートしている

非同期オフロードの例

素数を求める計算をホスト(CPU)とデバイス(MIC) で非同期に同時実行する

オフロードのサンプルコード

内容

- デバイス (アクセラレーター) 拡張
  - 基本
  - データ移動
  - 永続性
  - 並行 (非同期) 実行
  - インテル® コンパイラーの
    オフロード向け言語拡張 (LEO) ←→ OpenMP*

LEO ←→ OpenMP 4.x (Fortran)

汎用		
デバイス実行	OFFLOAD	TARGET
デバイス指示	target(MIC:#)	device(#)
デバイスメモリー割り当て、転送	IN( ) OUT( ) INOUT( )	map(TO:) map(FROM:) map(TOFROM:)
非構造化データ転送	OFFLOAD_TRANSFER	UPDATE
非同期	SIGNAL(S) WAIT(S)	NOWAIT* TASKWAIT

例

```
!dir$ offload begin target(MIC:0) in(x) out(y) signal(sync)
    <mic work>
!dir$ end offload
    <host work>
!dir$ offload_wait target(mic:0) wait(sync)
```

```
!$omp target device(0) map(to:x) map(from:y) nowait
   <device work>
!$omp end target
   <host work>
!$omp taskwait
```

データの永続性

```
!dir$ attributes offload:mic :: x, y
!dir$ offload offload_transfer target(MIC:0) in(a: alloc/free...)
                    out(b: alloc/free…)  nocopy(c: alloc/free…)
更新:
!dir$ offload offload_transfer target(MIC:0) in(a:…)out(b:…)
!dir$ offload begin target(mic:0)                in(a:…)out(b:…)
```

```
!$omp target declare (x, y)                                  グローバル
!$omp target data device(0) map(to: a)
                map(from: b) map(alloc: c)                   構造化*

!$omp target update device(0) to(a)  from(b)
!$omp target device(0) map(always,to:     a)
                                map(always,from: b)
```

*nowait は、インテル® コンパイラー 16 の 4.5 の機能、*非構造化マッピング (target enter/exit data) は OpenMP 4.5 でサポート

LEO ←→ OpenMP 4.x (C/C++)

汎用		
デバイス実行	OFFLOAD	TARGET
デバイス指示	target(MIC:#)	device(#)
デバイスメモリー割り当て、転送	IN( ) OUT( ) INOUT( )	map(TO:) map(FROM:) map(TOFROM:)
非構造化データ転送	OFFLOAD_TRANSFER	UPDATE
非同期	SIGNAL(S) WAIT(S)	NOWAIT* TASKWAIT

例

```
#pragma offload target(MIC:0) in(x) out(y) signal(sync)
    { <mic work> }

    { <host work> }
#pragma offload_wait target(mic:0) wait(sync)
```

```
#pragma omp target device(0) map(to:x) map(from:y) nowait
  {  <device work> }

  {  <host work> }
#pragma omp taskwait
```

データの永続性

```
#pragma offload_attribute( target(mic)) int x,y
#pragma offload offload_transfer target(MIC:0) in(a: alloc/free...)
                    out(b: alloc/free…)  nocopy(c: alloc/free…)
更新:
#pragma offload offload_transfer target(MIC:0) in(a:…)out(b:…)
#pragma offload target(mic:0)  in(a:…)out(b:…)
```

```
#pragma omp target map(x, y)                                  グローバル
#pragma omp target data device(0) map(to: a)
                map(from: b) map(alloc: c)                    構造化*

#pragma omp target update device(0) to(a)  from(b)
#pragma omp target device(0) map(always,to: a)
                             map(always,from: b)
```

*nowait は、インテル® コンパイラー 16 の 4.5 の機能、*非構造化マッピング (target enter/exit data) は OpenMP 4.5 でサポート

内容

- デバイス (アクセラレーター) 拡張
- 入れ子の並列化制御
- プロセッサーバインドとアフィニティーの制御

OpenMP* 3.1 における並列領域の入れ子

```
#pragma omp parallel
#pragma omp parallel
```

OpenMP* 3.1 では、入れ子になった並列領域の内側は、デフォルトでシングルスレッドで実行される

OMP_NESTED 環境変数を true に設定すると、内側の領域もマルチスレッドで実行できるが、最大スレッド数は (外側のスレッド x 内側のスレッド) となり、オーバーサブスクライブとなる。スレッド数とアフィニティーの制御は困難

OpenMP* 4.0 では、数百スレッドを実行できるデバイスでの入れ子になった並列領域を制御するため、teams と distribute 句が追加された

サンプルコード: nest.c

teams 構文

- 複数レベルの並列デバイスをサポート

- 構文 (C/C++):
  #pragma omp teams *[節[[,]節],…]*
  　　構造化ブロック

- 構文 (Fortran):
  !$omp teams *[節[[,]節],…]*
  　　構造化ブロック

このプラグマの直後は、各チームのマスタースレッドのみが実行し、ほかのチームメンバーは次の (入れ子構造の) 並列領域からのみ実行を開始します。そのため、実行中のスレッド数は **num_teams** のみで、それぞれのスレッドは **omp_get_thread_num**() == 0 になります

- 節: num_teams(整数式)、thread_limit(整数式)、default(shared | none)、
  private(リスト)、firstprivate(リスト)、shared(リスト)、
  reduction(演算子 :リスト)

distribute 構文

- ループ反復を複数のチームのマスタースレッドで分割

- 構文 (C/C++):
  #pragma omp distribute *[節[[,]節],…]*
  　　構造化ブロック

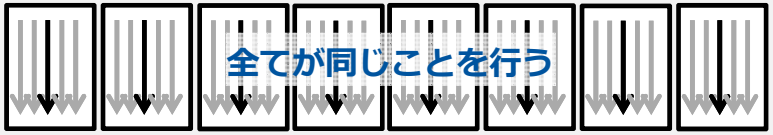
- 構文 (Fortran):
  !$omp distribute *[節[[,]節],…]*
  　　構造化ブロック

- 節:　collapse(n)、 private(リスト)、firstprivate(リスト)、
  　　dist_schedule(static [, chunk_size])

このプラグマは、teams 構造内の緊密な入れ子構造の 1 つ以上のループに関連付けられます。**collapse** を使用すると、omp for プラグマで **collapse** 節を指定した場合と同様に、複数のループを 1 つの反復シーケンスに結合できます

コプロセッサーへ SAXPY をオフロードする

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
   // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
 {
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
```

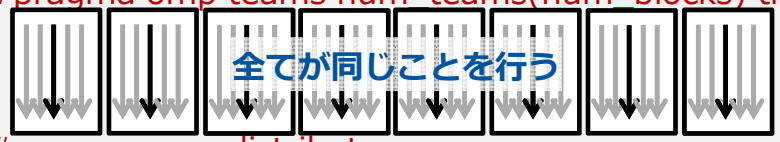



```
  for (int i = 0; i < n; i += num_blocks){
    for (int j = i; j < i + num_blocks; j++) {
        y[j] = a*x[j] + y[j];
 } }
 }
  free(x); free(y); return 0;
}
```

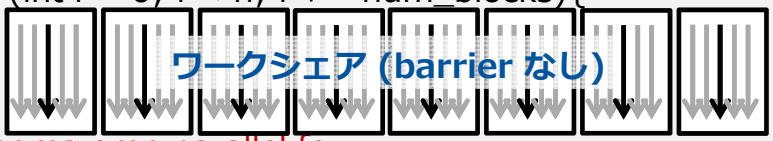

sugatech

コプロセッサーへ SAXPY をオフロードする

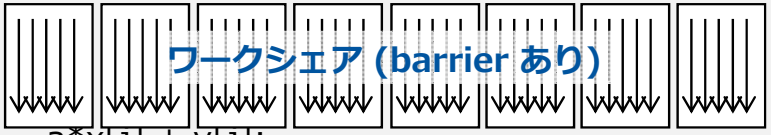
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
   // Define scalars n, a, b & initialize x, y


#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
```




```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```




```
#pragma omp parallel for
for (int j = i; j < i + num_blocks; j++) {
```




```
      y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```


sugatech

コプロセッサーへ SAXPY をオフロードする

- SAXPY複合構文

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
   // Define scalars n, a, b & initialize x, y

#pragma omp target map(to:x[0:n]) map(tofrom:y)
  {
#pragma omp teams distribute parallel for ¥
        num_teams(num_blocks) thread_limit(nthreads)
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }

  free(x); free(y); return 0;
}
```

入れ子並列とデバイスへの割り当て

オフロードのサンプルコード

内容

- デバイス (アクセラレーター) 拡張
- 入れ子の並列化制御
- プロセッサーバインドとアフィニティーの制御

スレッド・アフィニティー: プロセッサーのバインド

- バインドの方針は、マシンとアプリケーションに依存する

- スレッドを離して配置、例、異なるパッケージ
  - (おそらく) メモリー帯域幅を向上させる
  - (おそらく) 統合されたキャッシュサイズを改善
  - (おそらく) 同期構文のパフォーマンスを低下させる

- スレッドを近づけて配置、例、キャッシュを共有する可能性がある 2 つのコアに隣接
  - (おそらく) 同期構文のパフォーマンスを向上させる
  - (おそらく) 利用可能なメモリー帯域幅とキャッシュサイズ (スレッドごとの) を低下させる

OpenMP* 4.0 におけるスレッド・アフィニティー

- OpenMP* 4.0 で、アフィニティーのコンセプトを導入 ...
  - 1 つ以上のプロセッサー上で動作する一連のスレッド
  - ユーザーによって定義される
  - 事前定義された配置:
    - スレッド ハイパースレッドごとに 1 つの位置
    - コア　　物理コアごとに 1 つの位置
    - ソケット プロセッサー・パッケージごとに 1 つの位置

... そしてアフィニティーのポリシーは ....

  - spread　　OpenMP* スレッドをすべての位置に広く配置
  - close　　OpenMP* スレッドをマスタースレッドの近辺にパック
  - master　　OpenMP* スレッドをマスタースレッドを併置

- ... そしてこれらの設定を制御する
  - 環境変数 OMP_PLACES と OMP_PROC_BIND
  - 並列領域向けに proc_bind 節

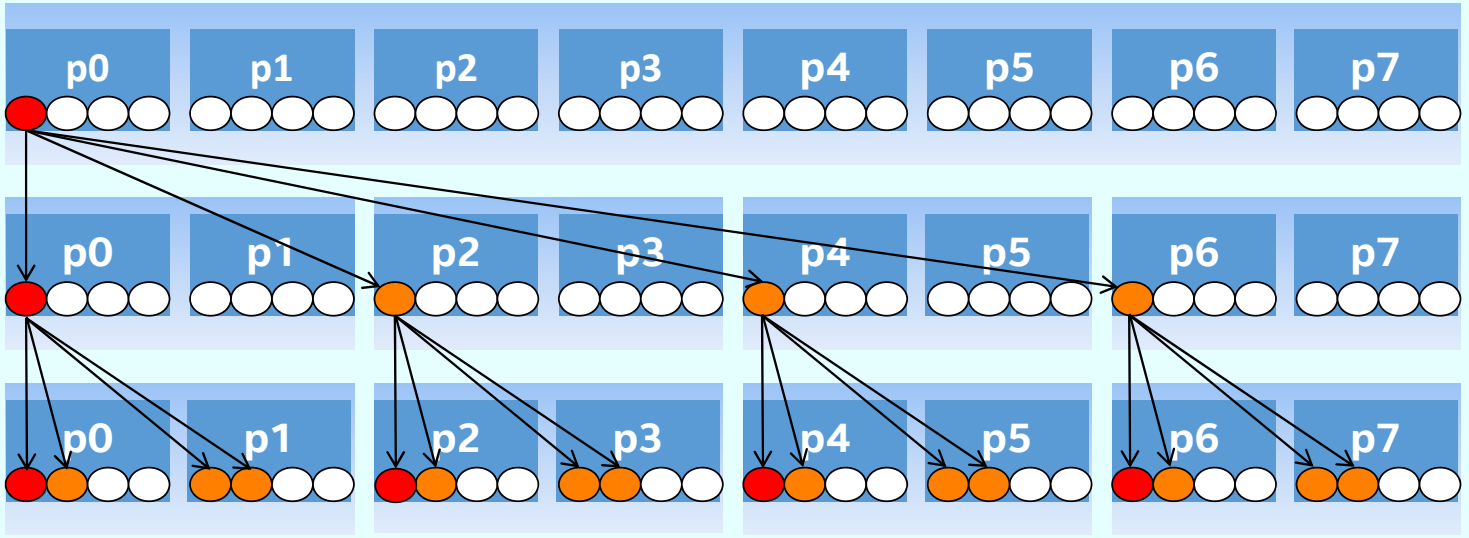
スレッド・アフィニティーの例

- 例 (インテル® Xeon Phi™ コプロセッサー): 外部領域を分配し、内部領域を近く保つ

  OMP_PLACES=cores(8)

#pragma omp parallel proc_bind(spread)

#pragma omp parallel proc_bind(close)



アフィニティー制御の例

オフロードのサンプルコード

環境変数を使用してテスト
set MIC_ENV_PREFIX=MIC
set MIC_OMP_PROC_BIND=[master, close, spread]

$> matmul_dist_para を実行

内容

- デバイス (アクセラレーター) 拡張
- プロセッサーバインドとアフィニティーの制御
- GFX コンパイラーと GFX へのオフロード

Gfx コンパイラー
インテル® グラフィックス・テクノロジー

- プログラミング・モデル機能
  - 共有仮想メモリー
  - OpenMP* 4.0 の一部
  - 非同期プログラミング・サポートの改善
- パフォーマンスの改善
  - 共有ローカルメモリー
  - 第 5 世代インテル® Core™ プロセッサー向けにチューニング
  - Gen ターゲット向けのベクトル化機能の改善
- 利用法
  - Gfx_sys_check ツール
  - デバッグサポートの改善

Gfx コンパイラー
OpenMP* 4.0 offload サポートへの追加機能

```
bool Sobel::execute_offload()
{
          int w = COLOR_CHANNEL_NUM * image_width;
          float *outp = this->output;
          float *img = this->image;
          int iw = image_width;
          int ih = image_height;
#pragma omp target map(to: ih, iw, w) ¥
                    map(tofrom: img[0:iw*ih*COLOR_CHANNEL_NUM], ¥
                                        outp[0:iw*ih*COLOR_CHANNEL_NUM])
#pragma omp parallel for collapse(2)
     for (int i = 1; I < ih – 1; i++) {
          for (int k = COLOR_CHANNEL_NUM; k < (iw - 1) * COLOR_CHANNEL_NUM; k++) {
                    float gx = 1 * img[k + (i - 1) * w -1 * 4]
                              + 2 * img[k + (i - 1) * w +0 * 4]
                              + 1 * img[k + (i - 1) * w +1 * 4]
                              - 1 * img[k + (i + 1) * w -1 * 4]
                              - 2 * img[k + (i + 1) * w +0 * 4]
                              - 1 * img[k + (i + 1) * w +1 * 4];
                    float gy = 1 * img[k + (i - 1) * w -1 * 4]
                              - 1 * img[k + (i - 1) * w +1 * 4]
                              + 2 * img[k + (i + 0) * w -1 * 4]
                              - 2 * img[k + (i + 0) * w +1 * 4]
                              + 1 * img[k + (i + 1) * w -1 * 4]
                              - 1 * img[k + (i + 1) * w +1 * 4];
                    outp[i * w + k] = sqrtf(gx * gx + gy * gy) / 2.0;
          }
     }
     return true;
}
```

利用方法：
- サブセットのみのサポート
- “tofrom” と “to” を “pin” へマップ
- “-qopenmp-offload=gfx” を指定

まとめ

OpenMP* 4.0 / 4.5 は、OpenMP における大きな飛躍

- 新しいレベルの並列性を導入
  - デバイス（MIC、GPU) への演算のオフロード
  - データの永続性を制御
  - 非同期実行を制御
- デバイスによる異種システム構成をサポート

参考サイト

インテル® ソフトウェア・フォーラム、ナレッジベース、記事、ツールのサポート
(http://software.intel.com 参照、http://isus.jp 翻訳版)

記事の例:
- http://www.isus.jp/article/parallel-special/requirements-for-vectorizable-loops/
  (ループをベクトル化するための条件)

OpenMP* 4.5 Specification
- http://www.openmp.org 5 /mp-documents/openmp-4.5.pdf

OpenMP* 3.1 仕様をカバーするオンライン・トレーニング
- http://www.isus.jp/online-training/

関連書籍

Structured Parallel Programming: Patterns for Efficient Computation

著者 Michael McCool, James Reinders ,Arch Robison 出版日: 2012 年 7 月 9 日 | ISBN: 978-0-124159938

『構造化並列プログラミング: 効率良い計算を行うためのパターン』

著者　マイケル・マックール／アーク・D・ロビソン／ジェームス・レインダース（共著）

訳者　菅原 清文／エクセルソフト株式会社（共訳） | ISBN　978-4-87783-305-3

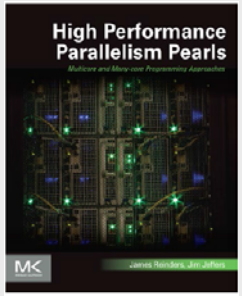
Intel® Xeon Phi™ Coprocessor High Performance Programming

著者 Jim Jeffers , James Reinders 出版日: 2013 年 3 月 | ISBN: 978-0-124104143

『インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング』

著者　ジェームス・レインダース/アーク・D・ロビソン（共著）

訳者　菅原 清文／エクセルソフト株式会社（共訳） | ISBN　978-4-87783-332-9



High Performance Parallelism Pearls

著者 Jim Jeffers , James Reinders　出版日: 2014 年 11 月

簡単にインテル® Xeon Phi™ コプロセッサー・ファミリーの優れた並列性を利用してコードを実行できるため、最適化に集中し、ハイパフォーマンスを実現することが可能です。並列処理を細かくチューニングすることで、正しいアプリケーションを正しく効率良いアプリケーションにすることができます。インテル コーポレーションの並列プログラミング・エバンジェリストである James Reinders とインテル コーポレーションのエンジニアである Jim Jeffers により執筆された最新の書籍は、69 人の専門家の実際の経験を基に、インテルのマルチコアおよびメニーコア・プロセッサーを最大限に利用するための創意工夫を紹介しています。

sugatech

sugatech

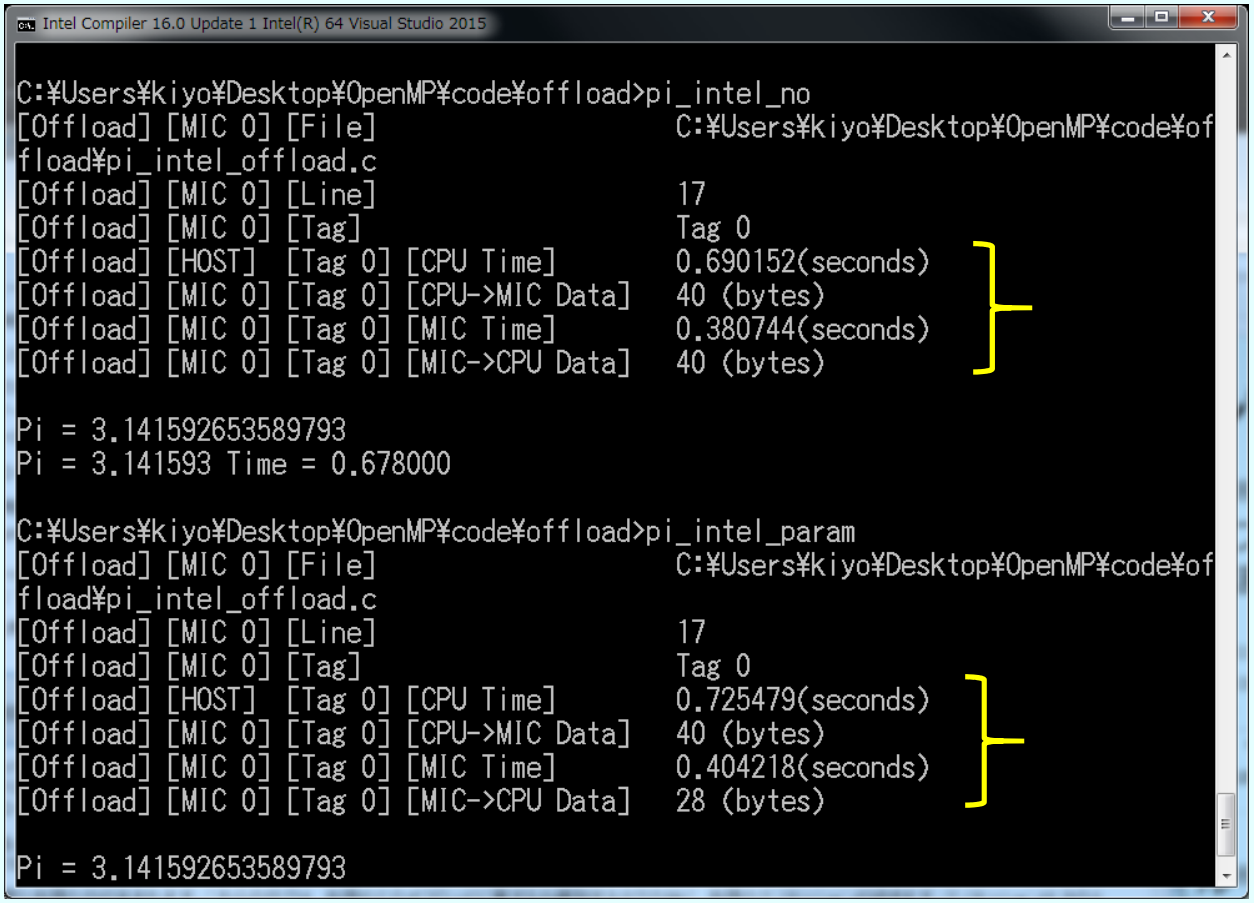
インテル® コンパイラーによるオフロード拡張の変更点 (1)

インテル® C++ および Fortran コンパイラーのバージョン 16 で OpenMP* 4.0 のオフロード拡張をサポートするにあたり、LEO のデータ属性の扱いが変更されました

```
#pragma offload target(mic) in(num_steps, step) inout(sum)
#pragma omp parallel for simd reduction(+:sum) private(x)
     for (i=0;i< num_steps; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
     }
```

インテル® コンパイラーのバージョン 15.x ではオフロード時に明示的にデータの target 属性を記述する必要がありましたが、バージョン 16 以降では省略できます

インテル® コンパイラーによるオフロード拡張の変更点 (2)



in、out、inout のデータ属性を省略するとすべての変数に inout が適用される。

inout で転送されるのは、sum のみ

戻る