# OpenACCの紹介

丸山直也（東工大）

GPUコンピューティング研究会

2012年3月5日

# はじめに

- OpenACC v1.0をベースにしています
- NVIDIAによるOpenACC紹介資料から一部を用いています
- OpenACCのコンパイラがまだ入手できていないため、本資料のコードの実行等による検証はしておりません

# OpenACC

- 計算およびデータをアクセラレータにオフロードするための指示文の仕様
- C/C++、Fortranをサポート
- PGI Accelerator Compilerがベース
- NVIDIA、CAPS、CRAY、PGIによる策定
  - 次期OpenMP仕様に取り込み？
- PGI、CAPS、CRAYがコンパイラをリリース
  - PGI: 2012年夏ごろ？
  - CAPS: ？
  - CRAY:　すでに利用可能（Cray XK6に付属）
- http://www.openacc-standard.org/

# 例

```
#include <stdio.h>
#define N 1000000

int main(void) {
  double pi = 0.0f; long i;
  #pragma acc region for
  for (i=0; i<N; i++) {
     double t= (double)((i+0.5)/N);
     pi +=4.0/(1.0+t*t);
  }
  printf("pi=%16.15f\n",pi/N);
  return 0;
}
```

# 例

```
program picalc
      implicit none
      integer, parameter :: n=1000000
      integer :: i
      real(kind=8) :: t, pi
      pi = 0.0
!$acc region
      do i=0, n-1
          t = (i+0.5)/n
          pi = pi + 4.0/(1.0 + t*t)
      end do
!$acc end region
      print *, 'pi=', pi/n
end program picalc
```

# CUDA/OpenCLとの比較

| | CUDA/OpenCL | OpenACC |
|---|---|---|
| 性能 | GPUの性能を最大限引き出すことが可能 | 場合によってはCUDA/OpenCL並み，性能を出すためにはGPU／CUDAの理解要 |
| プログラミング | アプリケーションの大幅な変更が必要 | 指示文の追加のみ、もしくは多少のデータ構造の変更要 |
| 実績 | 多数 | これから |

# OpenACC利用手順例

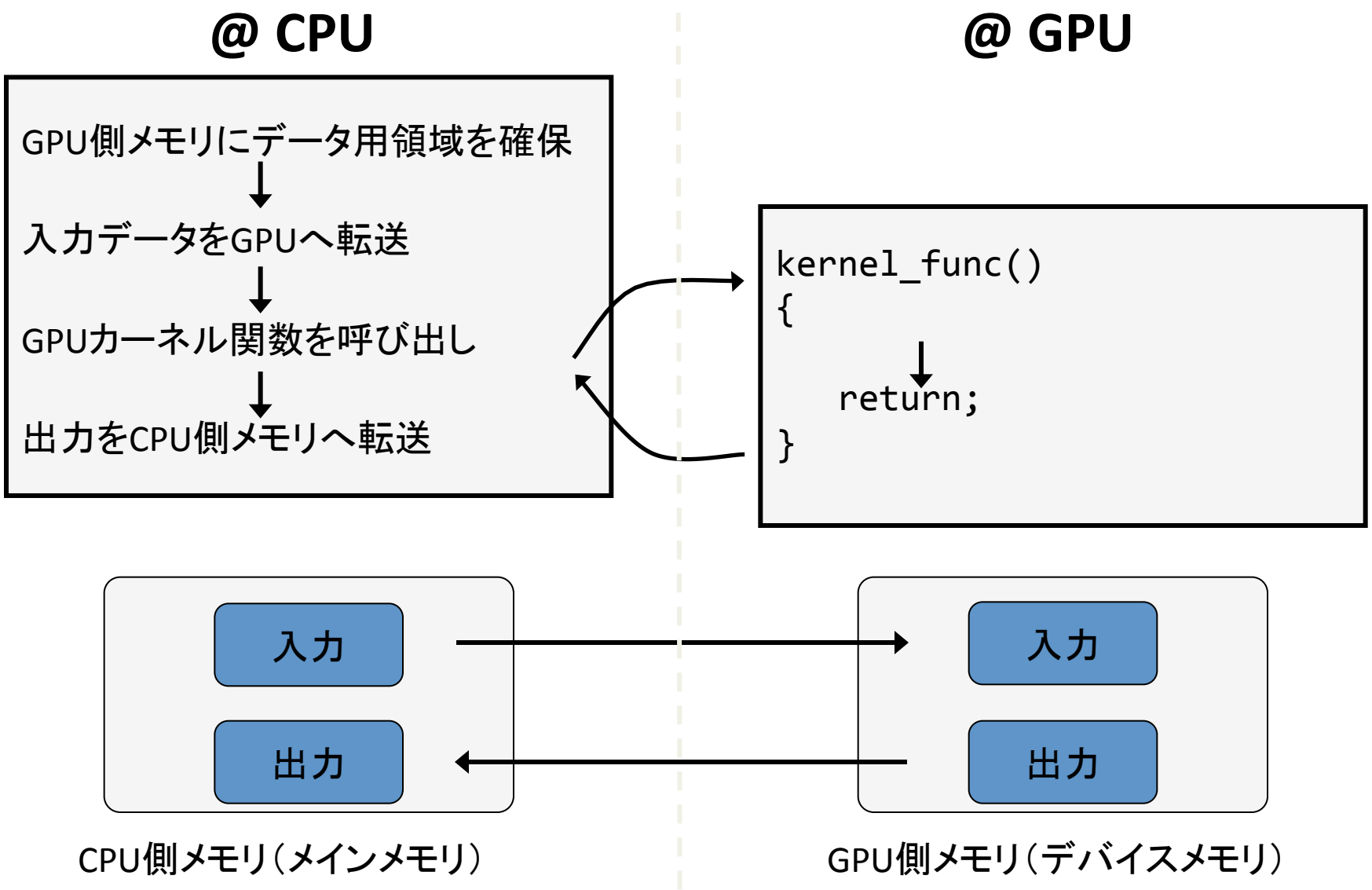
1. プログラム中のボトルネックループを特定
   - プロファイラ等ツールを利用
2. ループに最低限の指示文を追加
   - OpenACC Compute Construct
3. 実行時間の内訳を調査
   - ホスト・アクセラレータ間データ転送がボトルネック→OpenACC Data Constructを用いた最適化
   - アクセラレータ効率向上要→Loop Constructを用いた並列化効率の向上

# プログラム構成

ホストプログラム + GPUカーネル関数

- ホストプログラム
  - CPU上で実行されるプログラム
  - ほぼ通常のC言語として実装
  - GPUに対してデータ転送、プログラム呼び出しを実行
- （GPU）カーネル関数
  - GPU上で実行されるプログラム
  - ホストプログラムから呼び出されて実行
  - 再帰、関数ポインタは非サポート

# 典型的な制御とデータの流れ



# OpenACC指示文

```
C       #pragma acc directive-name [clause…]

Fortran  !$acc directive-name [clause…]
```

- 主な指示文
  - Compute Construct （計算のオフロード）
  - Data Construct （データのオフロード）
  - Loop Construct（ループ並列化の指示）

# Compute Construct

- `parallel`
  - 領域をアクセラレータで並列に計算
- kernels
  - 複数の領域をまとめてオフロード

# parallel 指示文

```
#pragma acc parallel [clause…]
  structured block

!$acc parallel [clause…]
  structured block
!$acc end parallel
```

- 主なオプション
  - if: 真のときのみオフロード
  - async: 非同期処理を指定
  - reduction: 縮約操作を定義

# OpenACCにおける並列化

- ベクトル並列（SIMD並列）
  - SIMD並列性を有効活用するため
- スレッド並列階層その1
  - OpenACC用語では “**Worker**” (ワーカー)
- スレッド並列階層その2
  - OpenACC用語では“**Gang**”（ギャング）
  - gangの中にworkerが含まれる
  - CUDAでいうスレッドブロック（？）

# parallel指示文オプション（一部）

- num_gangs(n)
  - n個のギャングを生成
- num_workers(n)
  - n個のワーカーを生成
- vector_length(n)
  - ベクトル長を指定

# リダクション

```
#pragma acc parallel reduce(+:sum)
for (x = 0; x < N; ++x) {
   sum += a[x];
}
```

- 2項演算子による縮約操作を定義
  - C: +, *, max, min, &, |, ^, &&, ||
  - Fortran: +, *, max, min, iand, ior, ieor, .and., .or., .eqv., .neqv.

# 例： ヤコビ法

```
while ( error > tol && iter < iter_max ) {
  error=0.f;
  for( int j = 1; j <= m; j++) {
    for(int i = 1; i <= n; i++) {
       Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
            + A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  tmp = Anew;
  Anew = A;
  A = tmp;
  iter++;
}
```

# 例： ヤコビ法

```
while ( error > tol && iter < iter_max ) {
  error=0.f;
  #pragma acc parallel reduce(max:error)
  for( int j = 1; j <= m; j++) {
    for(int i = 1; i <= n; i++) {
       Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
            + A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  tmp = Anew;
  Anew = A;
  A = tmp;
  iter++;
}
```

# kernels 指示文

```
#pragma acc kernels [clause…]
  structured block

!$acc kernels [clause…]
  structured block
!$acc end kernels
```

- 主なオプション
  - if: 真のときのみオフロード
  - async: 非同期処理を指定

# kernels 例

```
!$acc kernels
   do i=1,n
     a(i) = 0.0
     b(i) = 1.0
     c(i) = 2.0
   end do
   do i=1,n
     a(i) = b(i) + c(i)
   end do
!$acc end kernels
```

2つの別個のGPU
カーネルを生成

# Data Construct

```
#pragma acc data [clause…]
  structured block

!$acc data [clause…]
  structured block
!$acc end data
```

- データのアクセラレータへの転送を定義
- 指示文で指定した領域の前後でデータを転送
- 使わなくても良いが、性能チューニングのために重要
  - 指定しなかった場合はコンパイラが自動的に判断

# データ転送オプション

- copy
  - 領域の前後でホスト・アクセラレータ間で転送
- copyin
  - 領域の実行前にホストからアクセラレータへ転送
- copyout
  - 領域の実行後にアクセラレータからホストへ転送
- create
  - アクセラレータ上に領域を確保。ホストとの転送無し
- present
  - アクセラレータ上にすでに領域確保済み

# 例： ヤコビ法

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max ) {
  error=0.f;
  #pragma acc parallel reduce(max:error)
  for( int j = 1; j <= m; j++) {
    for(int i = 1; i <= n; i++) {
       Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
            + A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  tmp = Anew;
  Anew = A;
  A = tmp;
  iter++;
}
```

# 例： ヤコビ法

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.f;
  #pragma acc parallel reduce(max:error)
  for( int j = 1; j <= m; j++) {
    for(int i = 1; i <= n; i++) {
       Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
            + A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
  tmp = Anew;
  Anew = A;
  A = tmp;
  iter++;
}
```

# Loop Construct

```
#pragma acc loop [clause…]
  for loop

!$acc loop [clause…]
  do loop
!$acc end loop
```

- ループ並列化の詳細な指示
- 記述しなくてもコンパイラが自動的に判断
- 詳細に指示することで最適化可能

# ループ並列化オプション

- collapse
  - ループ並列化指示を多重ループへ適用
- gang
  - ループをギャングによって並列実行
- worker
  - ループをギャング内のワーカーによって並列実行
- seq
  - ループを逐次実行
- vector
  - ループをSIMDモードで実行
- independent
  - ループが独立に並列化可能であることを指示

# その他の指示文

- キャッシュ指示文
  - ループ内にてキャッシュするべき変数を指示
  - コンパイラも自動的に判断→明示的な指示による効率向上

```
#pragma acc cache(list)

!$acc cache (list)
```

# OpenACCライブラリAPI

- アクセラレータ利用に関する情報の取得や設定のためのライブラリAPI
  - OpenMPにおける omp_*関数のようなもの
- Cでは openacc.h ヘッダーファイルに定義
- Fortranでは openacc_lib.h ヘッダーファイルおよび openacc モジュールに定義

# 主なAPI

- acc_get_num_devices
- acc_set_device_type
- acc_init
- acc_shutdown
- acc_async_test
- acc_async_wait

# まとめ

- OpenACC
  - 指示文によるGPU化のための仕様
  - OpenMPと同様にループを並列実行
  - データ転送やループマッピングの指示が可能
  - CUDA/OpenCLより簡便なプログラミング
  - 性能？
- OpenACCの今後
  - 今年夏ごろまでにはコンパイラが利用可能に
  - 今後はCUDA/OpenCLより主流に？