

An Introduction to CUDA Programming

T. Aoyama

HPC-Phys workshop at Hiroshima Univ., March 18th, 2019

Table of Contents

- Notes on Hands-on
- Introduction to CUDA
- Himeno benchmark
- Programming tutorial
 - Kernels
 - Memory management
 - Launch kernel
- Compile and Run
- Advanced topics

Hands-on

- 実習用ワークステーション
 - sim01, sim02
 - NVIDIA GeForce GTX 680M (Kepler architecture)
 - CUDA Toolkit 10.0
 - PGI Compiler Community Edition 18.4
 - sim03
 - NVIDIA GeForce GTX 1080 (Pascal architecture)
 - CUDA Toolkit 10.1
 - PGI Compiler Community Edition 18.10

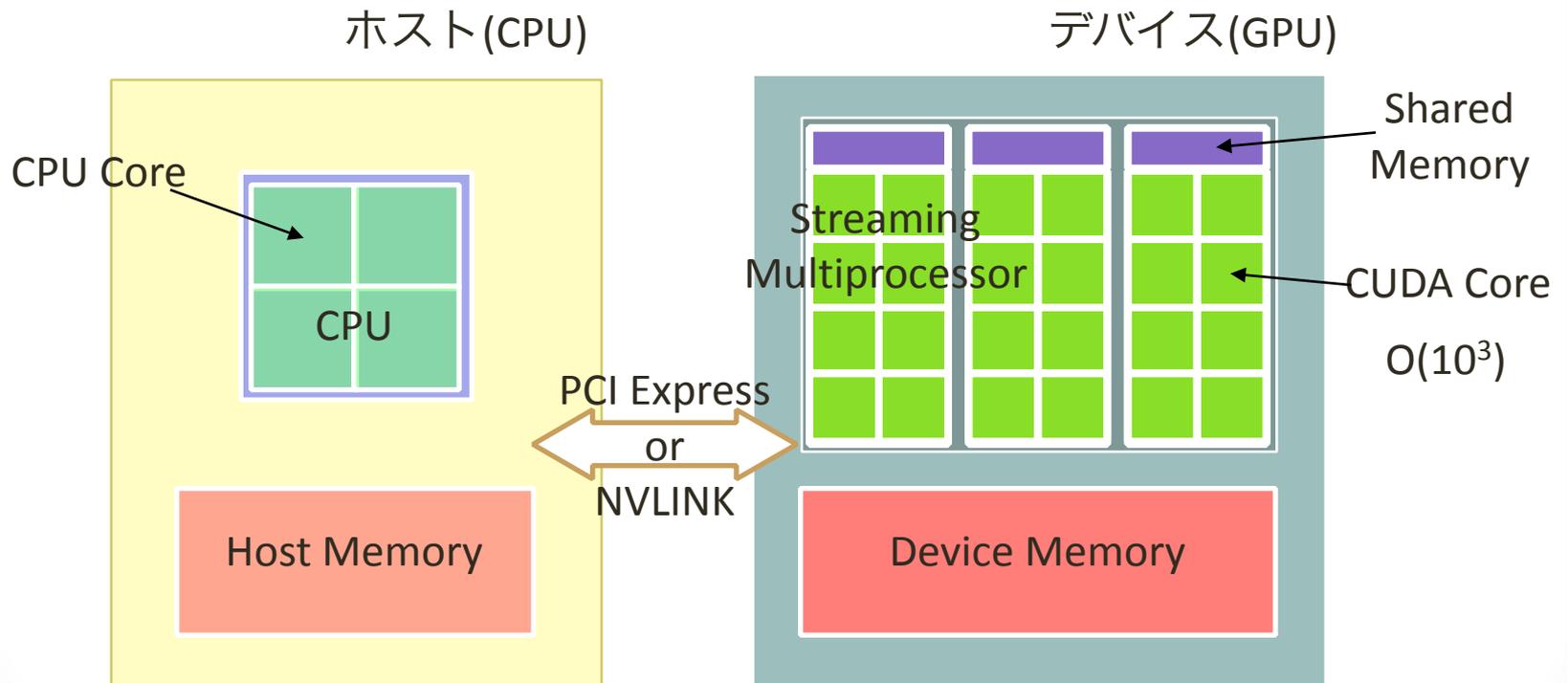
Hands-on

- 実習用プログラム: /home/aoyam/workshop/
 - himeno_C/ himenoBMTxpa_mod2.c
himenof90/ himenoBMTxp_mod.f90
 - Host用コード。これをベースに書き換え。
 - 各自ログイン後にコピー、コンパイル・実行してみる
 - オリジナルから変更:
 - offload 用に jacobi2() を用意
 - i,j,k ループ内の処理を変更: 差分計算・更新・誤差評価
 - himeno_C_CUDA/
himenof90_CUDA/ — C / Fortran90 での実装例
 - himeno_C_OpenACC/
himenof77_OpenACC/ — OpenACC での実装例
(松古氏による)

CUDA Tutorial

GPU Computing

- GPUシステムの構成



GPU Computing

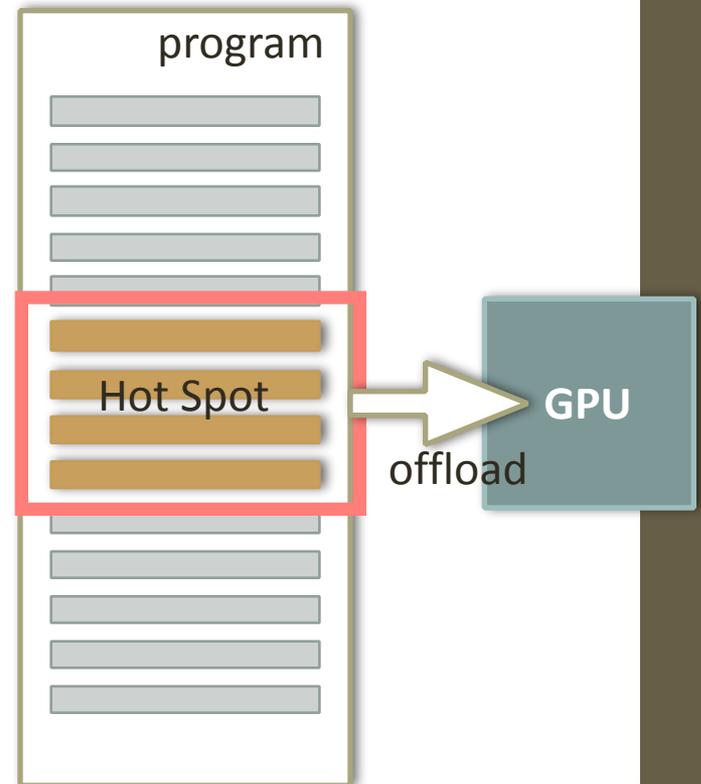
- NVIDIA GPUs

	GTX680M	GTX1080	P100	V100
Generation	Kepler	Pascal	Pascal	Volta
#SM	7	20	56	80
FP32 core	1344	2560	3584	5120
FP64 core	56	80	1792	2560
Peak FP32	2.03TFlops	8.9TFlops	10.6TFlops	15.7TFlops
Peak FP64	0.083TFlops	0.26TFlops	5.3TFlops	7.8TFlops
SP/DP	24:1	32:1	2:1	2:1
Mem B/W	115.2GB/s	320GB/s	732GB/s	900GB/s
CC	3.0	6.1	6.0	7.0

Compute Capability

GPU Computing

- GPUを効果的に使う
 - 計算時間が集中する箇所(Hot Spot)
その部分をGPUにオフロードする
 - 多数のスレッドによる並列実行
 - 独立な計算に分割可能な処理
 - データをホスト(CPU)とデバイス(GPU)の間でデータ転送する必要がある
 - オーバーヘッドを考慮



GPU Computing

- GPGPU = General-Purpose computing on Graphic Processing Unit
 - GPUを汎用の計算に利用
- Programming environments:
 - Directive-based
 - OpenACC, OpenMP (4.5-), ...
 - 既存コードからincrementalにGPU化
 - API-based
 - **CUDA**, OpenCL, ...
 - GPUでの処理を細かく制御
 - Libraries
 - GPU向けに移植されたLibraryやPackageを利用
 - cuBLAS, cuFFT, magma, ...

GPU Computing: OpenACC

GPUにoffloadする
部分をdirectiveで
指示

```
#pragma acc data copy(pp[0:size]), ¥
    copyin(size, mrows, mcols, mdeps, omega, pwrk1[0:size], ¥
        pbnd[0:size], pa[0:4*size], pb[0:3*size], pc[0:3*size]), ¥
    create(pwrk2[0:size]) // 転送する配列データは起点とサイズで指定する
{
    // 構造化ブロックを囲む
    for(n=0 ; n<nn ; n++){
        gosa = 0.0;
        #pragma acc parallel num_workers(1) vector_length(256)
        {
            // 構造化ブロックを囲む
            #pragma acc loop gang worker vector reduction(+ : gosa)
            for(int site = 0 ; site < size; site++){ // 3-level のループを1次元化済み
                int k = site % mdeps;
                if(k == 0 || k == mdeps-1) continue;
                ... (略)
                float s0 = pa[IDX(0,i,j,k)]*pp[IDX(0,i+1,j, k)]
                    + pa[IDX(1,i,j,k)]*pp[IDX(0,i, j+1,k)]
                ... (略)
                float ss = ...
                gosa += ss*ss;
                pwrk2[IDX(0,i,j,k)] = pp[IDX(0,i,j,k)] + omega * ss;
            }
        } //acc parallel // thread 間の同期

        #pragma acc parallel num_workers(2) vector_length(32)
        {
            #pragma acc loop gang worker vector
            for(int site = 0 ; site < size; site++){
                ... (略)
            }
        } // acc parallel // thread 間の同期
    } /* end n loop */
} // acc data // copy, copyout のデータがホストに送られる。メモリ領域解放
```

GPU Computing

CUDA = Compute Unified Device Architecture

- NVIDIAが開発・提供する GPGPUプラットフォームおよびプログラミングモデル
- C/C++ (NVIDIA CUDA SDK)
 - C/C++ の拡張として導入。CUDA C と API が提供される。
- Fortran (PGI)
 - PGI compiler でサポート。Fortran90 の拡張として導入。

GPU Computing

- 一般的な処理の流れ

1. デバイスを選択・初期化
2. デバイス上にメモリ領域を確保
3. データをホストからデバイスに転送
4. カーネルを起動して計算を行う
5. 計算結果をデバイスからホストへ転送
6. デバイス上のメモリ領域を解放
7. デバイスを解放

API呼び出し
malloc, memcpy に相当

カーネル(GPUスレッドでの
処理)を記述

拡張構文を用いて
カーネルを起動

CUDA Tutorial

- 今回は Himeno benchmark コードを題材に
お手軽にCUDAを導入してみる。
 - シンプルにカーネル化
 - shared memory などの optimize は後回し
 - ライブラリの利用
 - 集約処理(reduction)に cuBLAS を使う

姫野ベンチマーク

- 理研情報基盤センター・姫野龍太郎氏による
- <http://accr.riken.jp/supercom/documents/himenobmt/>

Himeno benchmark

- 非圧縮性流体などに現れるPoisson方程式をPoint-Jacobi法で解く

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial x \partial y} + \beta \frac{\partial^2 p}{\partial x \partial z} + \gamma \frac{\partial^2 p}{\partial y \partial z} = \rho$$

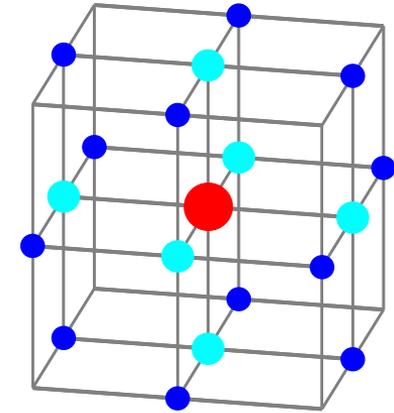
↓ 2次中心差分で離散化

$$\begin{aligned} & A_0 p_{i+1,j,k} + A_1 p_{i,j+1,k} + A_2 p_{i,j,k+1} \\ & + B_0 (p_{i+1,j+1,k} - p_{i-1,j+1,k} - p_{i+1,j-1,k} + p_{i-1,j-1,k}) \\ & + B_1 (p_{i+1,j,k+1} - p_{i+1,j,k-1} - p_{i-1,j,k+1} + p_{i-1,j,k-1}) \\ & + B_2 (p_{i,j+1,k+1} - p_{i,j+1,k-1} - p_{i,j-1,k+1} + p_{i,j-1,k-1}) \\ & + C_0 p_{i-1,j,k} + C_1 p_{i,j-1,k} + C_2 p_{i,j,k-1} - D p_{i,j,k} \\ & = W \end{aligned}$$

- p は従属変数、係数 A, B, C, D, W は定数行列(空間座標 i, j, k に依存)

Himeno benchmark

- 19点のステンシル
 - x, y, z および xy, xz, yz 方向の隣接格子点を参照
- 問題サイズ



S	64x64x128
M	128x128x256
L	256x256x256
XL	512x512x1024

- 演算
 - 疎行列ベクトル積
 - 誤差 = 各格子点での残差の2乗和

Code flow

himeno_C/himenoBMTxpa_mod2.c
jacobi2(...)

```
for(n=0 ; n<nn ; n++){  
  gosa = 0.0;
```

iteration loop

site loop

```
for(int site = 0 ; site < size; site++){  
  int k = site % mdeps;  
  int j = (site/mdeps) % mcols;  
  int i = site/(mdeps * mcols);  
  
  float s0;  
  s0 = pa[IDX(0,i,j,k)]*pp[IDX(0,i+1,j,k)]  
    + pa[IDX(1,i,j,k)]*pp[IDX(0,i,j+1,k)]  
    + pa[IDX(2,i,j,k)]*pp[IDX(0,i,j,k+1)]  
    // ...  
    + pwrk1[IDX(0,i,j,k)];  
  
  ss = (s0 * pa[IDX(3,i,j,k)] - pp[IDX(0,i,j,k)]) * pbnd[IDX(0,i,j,k)];  
  
  pwrk2[IDX(0,i,j,k)] = ss;  
}
```

差分を計算, wrk2 にストア

```
for(int site = 0; site < size; site++) {  
  float v = pwrk2[site];  
  gosa += v * v;  
}
```

誤差は差分の2乗和(norm)
で計算される

```
for(int site = 0 ; site < size; site++){  
  int k = site % mdeps;  
  int j = (site/mdeps) % mcols;  
  int i = site/(mdeps * mcols);  
  
  pp[IDX(0,i,j,k)] += omega * pwrk2[IDX(0,i,j,k)];  
}
```

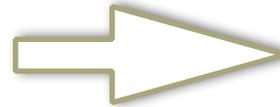
p を更新する
axpy の形になっている

```
} /* end n loop */
```

Strategy for parallelization

- 格子点をまわるループ
 - ループ内に依存性はない
 - 1格子点の演算 \leftrightarrow 1スレッドで実行 としてモデル化、格子点数のスレッドを GPU 上で並列処理

```
for(i=1 ; i<imax; i++)  
  for(j=1 ; j<jmax ; j++)  
    for(k=1 ; k<kmax ; k++){
```



スレッドに分散

// 格子点での演算



“カーネル”

- ループカウンタ(i,j,k) \leftrightarrow スレッドを識別する Index

Thread, Block, Grid

- スレッドは階層的にグループ化されて実行される
 - スレッドの集まり = ブロック
 - ブロックの集まり = グリッド

cf. OpenACC では Vector/Worker/Gang に抽象化されている
- 各スレッドは一意的な ID で識別される
 - 組み込み変数としてカーネル内から参照可
 - `threadIdx` block内のthreadのID
 - `blockIdx` grid内のblockのID
 - ブロック・グリッドの形状・サイズは `blockDim, gridDim`
 - ID は 1,2,3次元の形状をとる
 - これらの変数は dim3 型

Thread Index

- スレッドのインデックスを `threadIdx`, `blockIdx` から計算

```
idx = threadIdx.x + blockDim.x * blockIdx.x;
```

```
idx = threadIdx%x + blockDim%x * (blockIdx%x - 1)
```

- Fortranの場合、`index` は **1**からであることに注意。

CUDA-nizing code flow

メモリの確保とデータ転送

cudaMalloc(...)

cudaMemcpy(...)

```
for(n=0 ; n<nn ; n++){  
  gosa = 0.0;
```

```
  for(int site = 0 ; site < size; site++){  
    int k = site % mdeps;  
    int j = (site/mdeps) % mcols;  
    int i = site/(mdeps * mcols);  
  
    float s0;  
    s0 = pa[IDX(0,i,j,k)]*pp[IDX(0,i+1,j,k)]  
      + pa[IDX(1,i,j,k)]*pp[IDX(0,i,j+1,k)]  
      + pa[IDX(2,i,j,k)]*pp[IDX(0,i,j,k+1)]  
      // ...  
      + pwrk1[IDX(0,i,j,k)];  
  
    ss = (s0 * pa[IDX(3,i,j,k)] - pp[IDX(0,i,j,k)]) * pbnd[IDX(0,i,j,k)];  
  
    pwrk2[IDX(0,i,j,k)] = ss;  
  }
```

```
  for(int site = 0; site < size; site++) {  
    float v = pwrk2[site];  
    gosa += v * v;  
  }
```

```
  for(int site = 0 ; site < size; site++){  
    int k = site % mdeps;  
    int j = (site/mdeps) % mcols;  
    int i = site/(mdeps * mcols);  
  
    pp[IDX(0,i,j,k)] += omega * pwrk2[IDX(0,i,j,k)];  
  }
```

```
} /* end n loop */
```

カーネル実行

jacobi_step<<<grid,block>>>(...)

(総和計算 → ライブラリ利用)

カーネル実行

jacobi_update<<<grid,block>>>(...)

計算結果をホストに転送

cudaMemcpy(...)

Kernel

- スレッドで実行する処理を関数の形式で定義

```
__global__ void kernel (int n, float *dA)
{
    // kernel contents
}
```

__global__	カーネルの記述
__device__	デバイスコード用の関数 カーネルや他のデバイスコードから呼ばれる
__host__	ホストコード向けの関数

- 引数
 - デバイス上の配列を指すポインタ
 - スカラー値

Writing Kernels

```

__global__
void jacobi_step_dev(float *pa, float *pb, float *pc, float *pp,
                    float *pbnd, float *pwrk1, float *pwrk2, int mrows, int mcols, int mdeps)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;

    int k = index % mdeps;
    int j = (index / mdeps) % mcols;
    int i = (index / (mdeps * mcols)) % mrows;

    if (i == 0 || i == mrows-1 || j == 0 || j == mcols-1 || k == 0 || k == mdeps-1)
    {
        pwrk2[IDX(0,i,j,k)] = 0.0f;
    } else {
        float s0 = 0.0f;
        s0 = pa[IDX(0,i,j,k)]*pp[IDX(0,i+1,j,k)]
            + pa[IDX(1,i,j,k)]*pp[IDX(0,i,j+1,k)]
            + pa[IDX(2,i,j,k)]*pp[IDX(0,i,j,k+1)]
            + pb[IDX(0,i,j,k)]
            *( pp[IDX(0,i+1,j+1,k)] - pp[IDX(0,i+1,j-1,k)]
              - pp[IDX(0,i-1,j+1,k)] + pp[IDX(0,i-1,j-1,k)] )
            + pb[IDX(1,i,j,k)]
            *( pp[IDX(0,i,j+1,k+1)] - pp[IDX(0,i,j-1,k+1)]
              - pp[IDX(0,i,j+1,k-1)] + pp[IDX(0,i,j-1,k-1)] )
            + pb[IDX(2,i,j,k)]
            *( pp[IDX(0,i+1,j,k+1)] - pp[IDX(0,i-1,j,k+1)]
              - pp[IDX(0,i+1,j,k-1)] + pp[IDX(0,i-1,j,k-1)] )
            + pc[IDX(0,i,j,k)] * pp[IDX(0,i-1,j,k)]
            + pc[IDX(1,i,j,k)] * pp[IDX(0,i,j-1,k)]
            + pc[IDX(2,i,j,k)] * pp[IDX(0,i,j,k-1)]
            + pwrk1[IDX(0,i,j,k)];
        float ss = (s0 * pa[IDX(3,i,j,k)] - pp[IDX(0,i,j,k)]) * pbnd[IDX(0,i,j,k)];

        pwrk2[IDX(0,i,j,k)] = ss;
    }
}

```

インデックス計算

loop内の処理

(Cプログラムから切り出す)

Kernel

- global属性付きサブルーチンの形式で定義
 - カーネルはモジュールの中に置く必要がある

```
module kernel_module
contains
  attributes(global) subroutine kernel (n,dA)
    integer,value :: n
    real,dimension(n),device :: dA
    !...
end module kernel_module
```

- 引数
 - デバイス配列は device を付ける (e.g. dA)
 - スカラー値の受け渡しは value を付ける (e.g. n)

Writing Kernels

カーネルはmodule内に配置

```

module cuda_kernel
  use cudafor
  contains
    attributes(global) subroutine jacobi_step (p, a, b, c, bnd, wrk1, wrk2, &
      imax, jmax, kmax)
      implicit none
      real, dimension(::, ::, ::, ::), device :: a, b, c
      real, dimension(::, ::, ::), device :: p
      real, dimension(::, ::, ::), device :: bnd, wrk1, wrk2
      integer, value :: imax, jmax, kmax
      integer :: index
      integer :: i, j, k
      real :: s0, ss

      i = threadIdx%x + blockDim%x * (blockIdx%x-1)
      j = threadIdx%y + blockDim%y * (blockIdx%y-1)
      k = threadIdx%z + blockDim%z * (blockIdx%z-1)

      if ( (i.lt.1).or.(i.gt.imax).or.&
        (j.lt.1).or.(j.gt.jmax).or.&
        (k.lt.1).or.(k.gt.kmax) ) then
        ! index out of range.
        return
      else if ( (i.eq.1).or.(i.eq.imax).or.&
        (j.eq.1).or.(j.eq.jmax).or.&
        (k.eq.1).or.(k.eq.kmax) ) then
        ! edge points
        ss=0.0
      else
        s0 = a(I, J, K, 1)*p(I+1, J, K) &
          +a(I, J, K, 2)*p(I, J+1, K) &
          +a(I, J, K, 3)*p(I, J, K+1) &
          +b(I, J, K, 1) * (p(I+1, J+1, K)-p(I+1, J-1, K) &
            -p(I-1, J+1, K)+p(I-1, J-1, K)) &
          +b(I, J, K, 2) * (p(I, J+1, K+1)-p(I, J-1, K+1) &
            -p(I, J+1, K-1)+p(I, J-1, K-1)) &
          +b(I, J, K, 3) * (p(I+1, J, K+1)-p(I-1, J, K+1) &
            -p(I+1, J, K-1)+p(I-1, J, K-1)) &
          +c(I, J, K, 1)*p(I-1, J, K) &
          +c(I, J, K, 2)*p(I, J-1, K) &
          +c(I, J, K, 3)*p(I, J, K-1)+wrk1(I, J, K)
        ss=(s0*a(I, J, K, 4)-p(I, J, K))*bnd(I, J, K)
      end if
      wrk2(i, j, k) = ss
    end subroutine jacobi_step
  end module cuda_kernel

```

デバイス配列は device
スカラー引数は value で修飾

3次元のindexを使用
i,j,k に対応させる

loop内の処理
(Fortranコードから切り出し)

Memory

- メモリ管理
 - APIを通じてホスト側から制御
- 基本的な流れ
 - デバイス上にメモリを確保 (global memory)
`cudaMalloc(&devPtr, size)`
 - devPtr はデバイス上のメモリへのポインタ
ホスト側からは単なる識別子(直接読み書きはできない)
 - データ転送 (次ページ)
 - 確保したメモリを解放

`cudaFree(devPtr)`

Memory

- 基本的な流れ
 - データ転送

`cudaMemcpy(dstPtr, srcPtr, size, kind)`

- kindは転送の方向

<code>cudaMemcpyHostToDevice</code>	ホスト→デバイス
<code>cudaMemcpyDeviceToHost</code>	デバイス→ホスト
<code>cudaMemcpyDeviceToDevice</code>	デバイス上のメモリコピー

- `dstPtr`, `srcPtr` は方向に応じてデバイス上またはホスト上のポインタを指定
間違えると実行時エラー

Memory

- デバイス配列は device 属性をつけて宣言する
allocate文で動的に確保できる
 - `real, device, allocatable, dimension(:, :)` :: dA
`allocate(dA(M, N))`
MxN の2次元配列をデバイス上に確保
- 配列の代入文でデータ転送
 - `dA = hA`
 - ホスト上の配列 hA からデバイス配列 dA へ転送
 - 部分配列も可
- deallocate文で解放

Memory

- APIを利用することも可
 - cudafor モジュールを use する
- 例
 - ```
real, device, allocatable :: dB(:)
status = cudaMalloc(dB, 1024)
```
- 多次元配列は扱いにくい

# Kernel Launch

- 関数呼び出しと同じ形式
- ブロック・グリッドのサイズと形状を特別な構文で指定
  - `kernel<<<grid,block>>>(args,...)`
  - `call kernel<<<grid,block>>>(args,...)`
    - grid, block は int型(1次元) または dim3型(2,3次元)

```
for(i=1 ; i<imax; i++)
 for(j=1 ; j<jmax ; j++)
 for(k=1 ; k<kmax ; k++){
```

// 格子点での演算

並列度の情報

`kernel<<<grid,block>>>(args)`

# Kernel Launch

- カーネル実行は非同期
  - カーネル起動後、すぐにホスト側に処理が復帰する
  - カーネルの終了を待つには
    - `cudaDeviceSynchronize()`

# Initialize Devices

- デバイスの明示的な初期化は不要
  - 最初のAPI呼び出しで自動的に初期化が行われる
  - (GPUが複数あるとき) 使うデバイスを id で指定
    - `cudaSetDevice(devid)`
- デバイスの機能や設定を調べるには
  - `cudaGetDeviceProperties(&prop, devid)`
  - prop は `cudaDeviceProp`型の構造体
  - compute capability やデバイスメモリのサイズなど取得できる
- デバイスの解放は
  - `cudaDeviceReset()`
  - 現在activeなデバイスのcontextを破棄する

# cuBLAS Library

- BLAS (Basic Linear Algebra System)
  - 線形代数演算のセット
- 使い方
  - `#include <cublas_v2.h>` (C)
  - `use cublas` (Fortran)
  - `cublasHandle_t`型のハンドルを作成
    - `cublasCreate(&handle)`
- BLASルーチンの呼び出し例
  - 2乗ノルムの平方根 (単精度)
    - `cublasSnrm2(handle, size, x, xstride, &val)`
  - `axpy`
    - `cublasSaxpy(handle, size, a, x, xstride, y, ystride)`

# Compile and Run

- `nvcc [options] source.cu`
  - コンパイラ `nvcc` (NVIDIA CUDA SDK)
  - ソースファイル名の convention は `～.cu`
  - オプション (詳しくは `nvcc --help`)
    - `-arch sm_30` (Kepler) compute capability の指定
    - `-arch sm_60` (Pascal)
    - `-ptxas-options=-v`
      - kernel の使用リソースなど詳細情報を出力
- カーネルコードとホストコードを分離してそれぞれコンパイルし、まとめて一つの実行形式にして出力する

# Compile and Run

- `pgf90 [options] source.cuf`
  - コンパイラは `pgf90` (PGI Fortranコンパイラ)
  - ソースファイル名の convention は `～.cuf`
- オプション (詳しくは `man pgf90`)
  - `-Mcuda=cc30,ptxinfo`
    - `cc30` : (Kepler) compute capability
    - `cc60` : (Pascal)
    - `ptxinfo` : kernel 関数の情報を表示
  - `-Mfixed` 固定形式のソースファイル

# Profiling and Debug

- 実行時性能を評価
  - 各関数の経過時間・実行回数をカウント
  - ロード・ストアの効率などをモニタ
  - `nvprof [type] ./a.out [args]`
    - `type` に `profile` のモードやカウンタを指定  
無指定では `summary mode`
  - GUI の `nvvp` など利用可能
- デバッグ
  - `cuda-gdb`
    - コンパイル時に `-G -lineinfo` をつけてコンパイル
    - 使い方は `gdb` とほぼ同じ。特定のスレッドを調査できる
  - `printf` デバッグ
    - カーネル内で `printf` が使える

# Performance

- 演算量
  - 格子点あたり 34演算
- データ量
  - 12個の係数・1個の一時変数・1個の変数 = 14変数
  - 隣接格子点を毎回ロード → + 18変数
    - **Byte/Flop = 4byte x 14 or 32 / 34 ops**
- メモリバンド幅が律速とすると、期待される性能は
  - 再利用あり →  $115.2 \text{ GB/s} / (4\text{byte} \times 14 / 34 \text{ ops}) = \mathbf{70 \text{ GFlops}}$
  - 再利用なし →  $115.2 \text{ GB/s} / (4\text{byte} \times 32 / 34 \text{ ops}) = \mathbf{30 \text{ GFlops}}$

# Performance

- 実習環境 GeForce GTX680M (Kepler) / GTX 1080 (Pascal)

|            | GTX680M          | GTX1080        | P100           | V100           |
|------------|------------------|----------------|----------------|----------------|
| Generation | Kepler           | Pascal         | Pascal         | Volta          |
| #SM        | 7                | 20             | 56             | 80             |
| FP32 core  | 1344             | 2560           | 3584           | 5120           |
| FP64 core  | 56               | 80             | 1792           | 2560           |
| Peak FP32  | 2.03TFlops       | 8.9TFlops      | 10.6TFlops     | 15.7TFlops     |
| Peak FP64  | 0.083TFlops      | 0.26TFlops     | 5.3TFlops      | 7.8TFlops      |
| SP/DP      | 24:1             | 32:1           | 2:1            | 2:1            |
| Mem B/W    | <b>115.2GB/s</b> | <b>320GB/s</b> | <b>732GB/s</b> | <b>900GB/s</b> |
| CC         | 3.0              | 6.1            | 6.0            | 7.0            |

Compute Capability

# Performance

- Himeno Benchmark: 実習環境で測定 (単位は GFlops)

|                    | GTX 680M | GTX 1080 |
|--------------------|----------|----------|
| C                  | 0.56     | 0.98     |
| C with OpenACC     | 20.22    | 103.35   |
| CUDA C             | 33.25    | 103.25   |
| CUDA C (optimized) | 43.46    | 192.72   |
|                    |          |          |
| Fortran90          | 1.07     | 3.39     |
| F90 with OpenACC   | 23.71    | 88.75    |
| CUDA Fortran       | 20.35    | 94.69    |

- Mサイズ(128x128x256)

# 実習

(1)C/F90版をそのまま compile ・ 実行してみる

(2)CUDA化する

- a) kernel 作成 (差分計算 ・ 更新)、kernel 呼び出しで置き換え
- b) デバイスメモリの確保と解放処理を追加
- c) データ転送を追加
- d) 総和計算をcuBLASルーチン呼び出しに置き換え

(3)パラメータを変えてみる、など

- ブロックサイズ ・ グリッドサイズを変更

# Advanced Topics

## 1. Shared memory の利用

- block内のthreadで共有 → thread間のcommunication
- Global memory より高速 → Programmable な cache

## 2. Block size/Grid size の最適化

- block, grid のサイズを調整し、SMの占有率を上げる
  - なるべく多数のthreadをSMに割り当て
  - SMのリソース(レジスタ・共有メモリ)は有限

# Advanced Topics

## 3. Memory layout の検討

- threadが協調的にglobal memoryにアクセスすると効率がよい  
→ coalesced access
- 今の場合、係数行列の読み出しがメイン (再利用されない)  
→ データストリームを最適化

## 4. 複数GPUの利用

- single node/multi-node上の複数のGPUを並列に使う
- 通信と演算のoverlapによりデータ転送遅延を隠蔽

# References

- CUDA C プロフェッショナル プログラミング,  
John Cheng, Max Grossman, Ty McKercher 著, 株式会社クイープ 訳
- CUDA C PROGRAMMING GUIDE, v10.0, NVIDIA
- PGI Compilers & Tools CUDA FORTRAN PROGRAMMING GUIDE AND REFERENCE, Version 2018, NVIDIA

# Backup

# Optimization

## (a) shared memory の利用

- 圧力変数  $p(i,j,k)$  は複数の隣接サイトから参照される  
→ shared memory を介して隣接サイト間で再利用

## (b) stream optimization

- 係数行列  $A, B, C, \text{bnd}, \text{wrk1}$  は1回のみ load される  
→ 別々の配列をまとめることで load を効率化  
データレイアウトを最適化

# Optimization

## (a) shared memory の利用

- 格子を tile に分割、1 tile を 1 thread block で計算する
- thread block 内で  $p(i,j,k)$  を共有し、差分計算で再利用する  
(shared memory へのアクセスは global memory より高速)
- パラメータ
  - tile のサイズ・形状 — e.g.  $1 \times 8 \times 128$
  - thread block のサイズ — e.g. 256
    - Note: 1 thread は  $1 \times 8 \times 128 / 256 = 4$  site の演算を担当

# Optimization

- 計算手順
  - shared memory 上に袖領域を含めて array を確保
  - tile 部分 (+袖領域) の  $p(i,j,k)$  を shared memory にコピー
  - shared memory を参照して差分を計算
- あまり大きな効果はない
  - L1 cache が有効

# Optimization

## (b) stream optimization

- 係数行列 A, B, C, bnd, wrk1 を一つの配列に pack する:
  - $A[0..3][Nx][Ny][Nz] \rightarrow \text{Coeff}[0..3][Nx][Ny][Nz]$
  - $B[0..2][Nx][Ny][Nz] \rightarrow \text{Coeff}[4..6][Nx][Ny][Nz]$
  - $C[0..2][Nx][Ny][Nz] \rightarrow \text{Coeff}[7..9][Nx][Ny][Nz]$
  - $\text{bnd}[Nx][Ny][Nz] \rightarrow \text{Coeff}[10][Nx][Ny][Nz]$
  - $\text{wrk1}[Nx][Ny][Nz] \rightarrow \text{Coeff}[11][Nx][Ny][Nz]$
- 配列の index を入れ替え:
  - $\text{Coeff}[12][Nx][Ny][Nz] \rightarrow \text{Coeff}[Nx][12][Ny][Nz]$ 
    - tile と格子のサイズを考慮
- かなり効果がある
  - memory access の効率化が重要

# References

- CUDAによる定常反復Poissonベンチマークの高速化  
小川慧・青木尊之 情報処理学会研究報告 2008-HPC-115
- GPGPU上での流体アプリケーションの高速化手法  
成瀬彰・住元真司・久門耕一  
情報処理学会研究報告 2008-HPC-117
- 姫野ベンチマークのマルチノード実行における通信と演算のオーバーラップによる高速化  
加藤季広・青木尊之・額田彰・遠藤敏夫・松岡聡・長谷川篤史  
情報処理学会研究報告 2009-HPC-120

# Unified Memory

- ホストとデバイスのメモリ空間を統合
  - メモリ管理を単純化
    - 別々のメモリ空間によるポインタの重複を排除
  - 明示的なデータ転送が不要
    - 必要に応じてドライバが裏で転送を行う
    - (Pascal以降) on-demand で paging する.  
GPUメモリ容量を超える oversubscribe 可能.
- 使い方
  - `cudaMallocManaged(&ptr, size)`
  - ホスト側の malloc 呼び出しを置き換え
  - データを書き込み、そのままカーネルの引数に渡せる

# Unified Memory

- Himeno benchmark では
  - Matrix 型の初期化 `newMat()` の中で行列のデータを確保している箇所を置き換える

```
int
newMat(Matrix* Mat, int mnums, int mrows, int mcols, int mdeps)
{
 Mat->mnums= mnums;
 Mat->mrows= mrows;
 Mat->mcols= mcols;
 Mat->mdeps= mdeps;
 Mat->m= NULL;
 Mat->m= (float*)
 malloc(mnums * mrows * mcols * mdeps * sizeof(float));

 int vol = mnums * mrows * mcols * mdeps;
 cudaMallocManaged((void**)&(Mat->m), sizeof(float) * vol);

 return(Mat->m != NULL) ? 1:0;
}
```

# Unified Memory

- Fortran では managed 属性を付ける
  - `real,managed,allocatable :: A(:)`  
`allocate(A(N))`
- API を通じて確保もできる
  - `stat =`  
`cudaMallocManaged(A,N,cudaMemAttachHost)`