An Introduction to CUDA Programming

T. Aoyama

HPC-Phys workshop at RIKEN, December 1st, 2018

Table of Contents

C

F

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

GPGPU Programming

- 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

CUDA

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。

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。

  a. シンプルにカーネル化
     - shared memory などの optimize は後回し
  b. Unified Memory
     - メモリの管理はdriverに任せる
  c. ライブラリの利用
     - 集約処理(reduction)に cuBLAS を使う

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

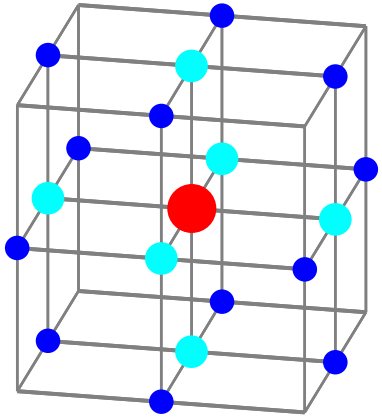
$$\frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}+\frac{\partial^2 p}{\partial z^2}+\alpha\frac{\partial^2 p}{\partial x\partial y}+\beta\frac{\partial^2 p}{\partial x\partial z}+\gamma\frac{\partial^2 p}{\partial y\partial z}=\rho$$

2次中心差分で離散化

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

- 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

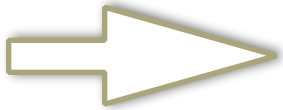
S	64x64x128
M	128x128x256
L	256x256x256
XL	512x512x1024

- 演算
  - 疎行列ベクトル積
  - 誤差 =各格子点での残差の2乗和

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 ↔ 1スレッドで実行　としてモデル化、
    多数のスレッドを 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++){
      // 格子点での演算
```



スレッドに分散

“カーネル”

- ループカウンタ(i,j,k) ↔ スレッドを識別する Index

Reorganize code flow

- 扱いやすいようにコードを変形

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

  for(i=1 ; i<imax; i++)
    for(j=1 ; j<jmax ; j++)
      for(k=1 ; k<kmax ; k++){
        s0= MR(a,0,i,j,k)*MR(p,0,i+1,j,  k)
          + MR(a,1,i,j,k)*MR(p,0,i,  j+1,k)
          + MR(a,2,i,j,k)*MR(p,0,i,  j,  k+1)
          // ...
        ss= (s0*MR(a,3,i,j,k) - MR(p,0,i,j,k))*MR(bnd,0,i,j,k);

        gosa += ss*ss;
        MR(wrk2,0,i,j,k)= MR(p,0,i,j,k) + omega*ss;
      }

  for(i=1 ; i<imax ; i++)
    for(j=1 ; j<jmax ; j++)
      for(k=1 ; k<kmax ; k++)
        MR(p,0,i,j,k)= MR(wrk2,0,i,j,k);

} /* end n loop */
```

差分をwrk2に書き出し

```
MR(wrk2,0,i,j,k) = ss;
```

gosa は wrk2 の要素の

2乗和(norm)で計算される

p を update する

```
MR(p,0,i,j,k) += MR(wrk2,0,i,j,k) * omega;
```

axpyの形になっている

CUDA-nizing code flow

- CUDA化の流れ

メモリの確保とデータ転送　cudaMemcpy(…)

```
for(n=0 ; n<nn ; n++){                                   iteration loop
  for(i=1 ; i<imax; i++)
    for(j=1 ; j<jmax ; j++)                              カーネル実行
      for(k=1 ; k<kmax ; k++){                             jacobi_step<<<grid,block>>>(args)
        s0= MR(a,0,i,j,k)*MR(p,0,i+1,j,  k)
          + MR(a,1,i,j,k)*MR(p,0,i,  j+1,k)
          + MR(a,2,i,j,k)*MR(p,0,i,  j,  k+1)
          // ...
        ss= (s0*MR(a,3,i,j,k) - MR(p,0,i,j,k))*MR(bnd,0,i,j,k);

        MR(wrk2,0,i,j,k) = ss;
      }

  for(i=1 ; i<imax ; i++)
    for(j=1 ; j<jmax ; j++)                              jacobi_update<<<grid,block>>>(args)
      for(k=1 ; k<kmax ; k++)
        MR(p,0,i,j,k) += MR(wrk2,0,i,j,k) * omega;

} /* end n loop */
```

計算結果をホストに転送

Kernel

C

- 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

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)

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。

Writing Kernels

C

```
__global__
void jacobi_step_dev(float *pa, floag *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)

Writing Kernels

```
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
```

カーネルはmodule内に配置

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)

C

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)
```

C

Memory

- 基本的な流れ
  - データ転送

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

    - kindは転送の方向

`cudaMemcpyHostToDevice`	ホスト→デバイス
`cudaMemcpyDeviceToHost`	デバイス→ホスト
`cudaMemcpyDeviceToDevice`	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

F

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
    ```

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

C

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));

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

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

Unified Memory

F

- 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)`

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殊な構文で指定
  - `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
```

```
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>`
  - 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` compute capability の指定
    - `—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 : compute capability 3.0
    - 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profiileのモードやカウンタを指定

      無指定では summary mode
  - GUI の nvvp なども利用可能
- 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

- デモ環境 GeForce GTX680M (Kepler世代)

	GTX680M	K40	P100	V100
Generation	Kepler	Kepler	Pascal	Volta
#SM	7	15	56	80
FP32 core	1344	2880	3584	5120
FP64 core	56	960	1792	2560
Peak FP32	2.03TFlops	5TFlops	10.6TFlops	15.7TFlops
Peak FP64	0.083TFlops	1.7TFlops	5.3TFlops	7.8TFlops
SP/DP	24:1	3:1	2:1	2:1
Mem B/W	**115.2GB/s**	**288GB/s**	**732GB/s**	**900GB/s**
CC	3.0	3.5	6.0	7.0

Compute Capability

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 GB/s / (4byte x 14 / 34 ops) = **70** GFlops
  - 再利用なし → 115.2 GB/s / (4byte x 32 / 34 ops) = **30** GFlops

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