CUDA Programming

青山 龍美 (KEK)

第 3 回 HPC-Phys 勉強会 於 広島大

2019 年 3 月 18 日

1 CUDAとは

GPU 向け汎用コンピューティング (GPGPU) のプラットフォームおよびプログラミングモデルで、NVIDIA が開発・提供する。NVIDIA の GPU で利用が可能である。C/C++ の拡張として導入され、SDK にはコンパイラ・ランタイムライブラリ・デバッガ・プロファイラなどの開発環境が含まれる。CUDA Fortran は PGI compiler で利用可能な CUDA 環境で、Fortran90 の拡張として導入される。
GPU プログラムは一般に、アプリケーション内で計算量の大きい部分 (Hot Spot) をデバイスに offload して実行する。CUDA では、デバイスでの処理を記述する CUDA C と、ホストから GPU を制御するための API (Application Program Interface) および拡張構文が用意されている。

2 CUDAのプログラミングモデル

GPU システムは典型的にはホスト (CPU) とデバイス (GPU) の構成をとり、両者は PCI Express や NVLINK などで接続される。デバイスは GPU プロセッサチップおよび、ホストとは独立なメモリを備えている。デバイス内のチップとメモリ間のデータ転送は比較的高速でる。
NVIDIA GPU プロセッサチップは Streaming Multiprocessor (SM) と呼ばれる単位で構成され、世代ごとに異なるが SM 内に 32〜192 個のプロセッサコアと L1 キャッシュ/シェアードメモリ、レジスタファイルを持つ。

カーネル: デバイス上で実行されるプログラムコードをカーネルと呼ぶ。カーネルは GPU 上で多数のスレッドで並列に実行される。各スレッドは unique な id を持ち、カーネル内で組み込み変数により参照できる。GPU でのプログラム実行はホスト側から起動され、スレッドの並列数はパラメータで指定される。

スレッド: 並列スレッドはブロック・グリッドの階層にグループ化される。ブロックはスレッドの集まりであり、ブロック単位で SM に処理が割り当てられる。グリッドはブロックの集まりで、ホストから起動するカーネル実行の単位である。各ブロックは独立に実行され、同期や通信は行わない。ブロック内ではスレッド間の同期によって協調動作する。実行はワープ (Warp) と呼ばれる単位で行われ、1 命令が同時に 32 スレッドで実行される (SIMT)。

メモリ: デバイス上にはグローバルメモリと呼ぶ off-chip のメモリが搭載されている。テバイスとホストは独立なメモリ空間を持ち (Unified Memory 機構によりメモリ空間を統合して扱うこともできる)、デバイス上のメモリ確保や、ホストとデバイス間でのデータの転送は、ホストコードから API を通じて制御する。

デバイス内の SM はシェアードメモリを備え、スレッドブロック内で共有される。CUDA C にはブロック内で同期を取る組み込み関数が用意されており、適切に同期を取りながらスレッド間でデータ共有を行なうことができる。

典型的な処理の流れは次のようになる:

1. デバイスを選択 (→Sec. 3.4, 4.4)

2. デバイス上にメモリを確保　(→Sec. 3.3, 4.3)
3. データをデバイスに転送　(→Sec. 3.3, 4.3)
4. カーネルを実行して計算を行なう　(→Sec. 3.2, 4.2)
5. 計算結果をデバイスからホストに転送
6. デバイス上のメモリを解放
7. デバイスを解放

3 CUDA C

3.1 カーネルの定義

カーネルは `__global__` 修飾子のついた void 型を返す関数として定義する。

```
__global__ void kernel_name([arg,...])
```

関数の修飾子にはカーネルを示す `__global__` および、カーネルや他のデバイス関数から呼ばれる `__device__`、ホスト上で実行される関数を示す `__host__` がある。

カーネルの組み込み変数

各スレッド・ブロックには一意的な ID が割り当てられ、デバイスコードから参照可能。dim3 型の変数。

`threadIdx`　スレッド ID。ブロック内のスレッドのインデックス
`blockIdx`　ブロック ID。グリッド内のブロックのインデックス

ブロック・グリッドのサイズと形状も定義されている。カーネル起動時のパラメータと対応。

`blockDim`　ブロック内のスレッド数
`gridDim`　グリッド内のブロック数

デバイスコードで利用できる組み込み関数

スレッドの同期　`__syncthreads()` ブロック内のスレッド間で同期をとる

算術関数、Atomic 関数 など

3.2 カーネル実行

ブロック・グリッドのサイズと形状を指定してカーネルを実行する。特殊な構文が用意されている。

```
kernel_name<<<grid_size, block_size>>>([args,...])
```

grid_size, block_size は int 型 (1 次元) または dim3 型 (2,3 次元)。ホストコードは上記呼び出し後すぐに復帰し、カーネルは非同期に実行される。カーネルの終了を待つには

```
cudaDeviceSynchronize()
```

を呼ぶ。

3.3 メモリの確保と解放

デバイス上のグローバルメモリは、APIを通じてホスト側から確保する。

3.3.1 デバイスメモリを確保

```
cudaMalloc(&devPtr, size)
```

devPtrはポインタ型の変数で、デバイス上のメモリを指す識別子。ホスト側からは直接アクセスはできない。sizeは byte数。解放は

```
cudaFree(devPtr)
```

3.3.2 データ転送

```
cudaMemcpy(dst, src, count, kind)
```

src, dstは送り元・送り先のポインタ。転送方向によってホスト側・デバイス側のポインタを指定する。kindは転送の方向。

`cudaMemcpyHostToDevice`	ホストからデバイスへ
`cudaMemcpyDeviceToHost`	デバイスからホストへ
`cudaMemcpyDevieToDevice`	デバイス上でメモリコピー

処理は同期的 (blocking)。非同期実行するには streamを作成して cudaMemcpyAsyncを用いる。

3.3.3 Unified Memory

CUDA5.0以降、ホストメモリとデバイスメモリのアドレス空間を統合して扱う Unified Memory (managed memory) が利用できるようになった。データ移動はドライバが自動的に行なう。Unified Memoryの確保は

```
cudaMallocManaged(&ptr, size)
```

ptrはホスト・デバイスの両方から参照できる。解放は `cudaFree()` を用いる。

3.3.4 シェアードメモリ

ブロック内のスレッドで共有される。デバイスコード上で `__shared__` 修飾子を付けて宣言する。

```
float __shared__ buf[1024]
```

書き込み・読み出しの競合を避けるため適切に同期をとる必要がある。

3.4 デバイスの初期化と解放

デバイスの明示的な初期化は不要。最初の API呼び出しの際に必要な初期化が行われる。複数のデバイスが接続されている場合、操作の対象となるデバイスを指定するには

```
cudaSetDevice(devid)
```

とする。devidはデバイスの ID。デバイスのプロパティを調べるには

```
cudaGetDeviceProperties(&prop, devid)
```

を実行する。各種プロパティは cudaDeviceProp 型の構造体 prop に格納される。デバイス名や compute capability、メモリサイズなどの情報を取得できる。
デバイスの解放は

```
cudaDeviceReset()
```

を呼ぶ。現在選択中のデバイスのコンテキストを破棄する。

3.5 エラー処理

ほとんどの API は cudaError_t 型のステータスコードを返す。この値が cudaSucess の場合は成功、それ以外はエラーである。エラー内容を読める形に変換するには

```
cudaGetErrorString(stat)
```

を用いる。一方、カーネル実行はステータスを返さないため、エラー状態を確認するには

```
cudaGetLastError()
```

を発行する。

Note: マクロを使って、CUDA API 呼び出しごとにエラーチェックを行なう:

```
#define CUDA_SAFE_CALL(stmt)                    \
  do {                                          \
    cudaError_t status = stmt;                  \
    if (status != cudaSuccess) {                \
      printf("ERROR: %s in %s:%d.\n",           \
             cudaGetErrorString(status),        \
             __FILE__, __LINE__);               \
    }                                           \
  } while (0)
```

使い方

```
CUDA_SAFE_CALL( cudaMalloc(&devptr, sizeof(float) * N) );
```

3.6 コンパイルと実行

コンパイルには NVIDIA CUDA SDK に含まれる nvcc コンパイラを使う。ソースコードのファイル名には ～.cu が使われる。

```
nvcc [options] source.cu
```

代表的なコンパイルオプション

`-arch sm_30`	compute capability の指定
`--ptxas-options=-v`	レジスタ使用量などカーネルの詳細情報を表示
`-G -lineinfo`	デバッグ情報を追加

作成された実行形式を実行する

```
./a.out
```

3.7 プロファイルとデバッグ

実行時性能を評価。各カーネルや API の実行時間や実行回数をカウント。ロード・ストアの効率などをモニタする。

```
nvprof [type] ./a.out [args]
```

type にはプロファイルのモードやカウンタの種類を指定する。無指定時は summary mode でカーネル等の実行時間の一覧が表示される。この他に GUI の nvvp なども利用可。
CUDA SDK には cuda-gdb が用意されている。コンパイル時に -G -lineinfo をつけてコンパイルする。使い方は gdb に類似で、特定のスレッドにアタッチして調査できる。
カーネル内で printf を使うことができる。出力はバッファを通してホスト側に送られる。全てのスレッドから出力すると出力量が膨大になるので注意。

3.8 ライブラリの利用

CUDA SDK には cuBLAS(線形代数) や cuFFT(高速フーリエ変換)、cuRAND(擬似乱数) などのライブラリが提供されている。ここでは cuBLAS の使い方を簡単に紹介する。

ヘッダの読み込み:

```
#include <cublas_v2.h>
```

初期化: cublasHandle_t 型のハンドルを作成する。

```
cublasCreate(&handle)
```

BLAS ルーチンの例

- 2 乗ノルムの平方根

  ```
  cublas[T]nrm2(handle, size, x, xstride, &val)
  ```

- axpy

  ```
  cublas[T]axpy(handle, size, a, x, xstride, y, ystride)
  ```

  BLAS ライブラリの命名規約と同じく、[T] はデータ型によって次の記号が入る:

S:	単精度実数	float
D:	倍精度実数	double
C:	単精度複素数	cuComplex 型
Z:	倍精度複素数	cuDoubleComplex 型

4 CUDA Fortran

CUDA API を利用するには cudafor モジュールを use する。

```
use cudafor
```

4.1 カーネルの定義

カーネルはモジュール内に配置する必要がある。attributes(global) を付けたサブルーチンとして定義する。

```
module cuda_kernel
contains
  attributes(global) subroutine kernel(args,...)
    ! kernel body
    !
  end subroutine
end module
```

note: サブプログラム (サブルーチン・ 関数) の属性

global	カーネル。通常はサブルーチンに付ける
device	デバイス上で実行されるサブプログラム。カーネルや他のデバイスサブプログラムから呼ばれる
host	ホスト上で実行されるサブプログラム

カーネルの組み込み変数

各スレッド・ブロックには一意的な ID が割り当てられ、デバイスサブプログラムから参照可能。type(dim3) 型の変数。ID は 1 からであることに注意。

`threadIdx`	スレッド ID。ブロック内のスレッドのインデックス
`blockIdx`	ブロック ID。グリッド内のブロックのインデックス

ブロック・ グリッドのサイズと形状も定義されている。カーネル起動時のパラメータと対応。

`blockDim`	ブロック内のスレッド数
`gridDim`	グリッド内のブロック数

dim3 型の要素は `threadIdx%x`, `threadIdx%y`, `threadIdx%z` のように参照する。

デバイスコードで利用できる組み込み関数等

スレッドの同期　ブロック内のスレッド間で同期をとる

```
call syncthread()
```

算術関数、Atomic 関数など

4.2 カーネル実行

ブロック・グリッドのサイズと形状を指定してカーネルを実行する。特殊な構文が用意されている。

```
call kernel_name<<<grid_size,block_size>>>([args,...])
```

grid_size, block_size は integer 型 (1 次元) または type(dim3) 型 (2,3 次元)。ホストコードは上記呼び出し後すぐに復帰し、カーネルは非同期に実行される。カーネルの終了を待つには

```
stat = cudaDeviceSynchronize()
```

を実行する。

4.3 メモリの確保と解放

デバイス配列は device 属性を付けて宣言する。動的配列を宣言できる。

```
real,device,allocatable,dimension(:,:) :: A
```

メモリ割り当ては allocate 文で動的に行える。

```
allocate(A(M,N))   ! MxN の 2 次元配列を確保
```

メモリの解放は deallocate 文を用いる。API を使ってデバイスメモリを割り当てることもできる。

```
stat = cudaMalloc(devArray, count)
```

count は要素数。

4.3.1 データ転送

配列の代入文でデータ転送を記述できる。

```
devA = hostB
```

devA はデバイス配列、hostB はホスト上の配列とする。上記で hostB から devA へのデータ転送を行なう。部分配列の指定も可能。

4.3.2 Unified Memory

ホストメモリとデバイスメモリのアドレス空間を統合して扱う Unified Memory (managed memory) が利用できる。データの移動はドライバが自動的に行なう。Unified Memory の指定は managed 属性を付ける。

```
real,managed,allocatable :: A(:)
allocate(A(N))
```

配列 A はホスト・デバイスの両方から参照できる。

4.3.3 シェアードメモリ

ブロック内のスレッドで共有される変数。デバイスサブプログラム中で、shared 属性を付けて宣言する。

```
real,shared :: buf[1024]
```

書き込み・読み出しの競合を避けるため適切に同期をとる必要がある。

4.4 デバイスの初期化と解放

デバイスの明示的な初期化は不要。最初の API 呼び出しの際に必要な初期化が行われる。複数のデバイスが接続されている場合、操作の対象となるデバイスを指定するには

```
stat = cudaSetDevice(devid)
```

とする。devid はデバイスの ID。デバイスの解放は

```
stat = cudaDeviceReset()
```

を呼ぶ。現在選択中のデバイスのコンテキストを破棄する。

4.5 エラー処理

API は関数として定義され、integer 型のステータスコードを返す。この値が cudaSuccess の場合は成功、それ以外はエラーである。エラー内容を読める形に変換するには

```
cudaGetErrorString(stat)
```

を用いる。カーネル実行はサブルーチン呼び出しであるので、ステータスコードを返さない。エラー状態を確認するには

```
stat = cudaGetLastError()
```

を発行する。

4.6 コンパイルと実行

コンパイルには PGI Fortran コンパイラを使う。ソースコードのファイル名には ～.cuf が使われる。

```
pgf90 [options] source.cuf
```

代表的なコンパイルオプション

-Mcuda=[params]	params は (カンマで区切って複数指定可)	
	cc30	compute capability の指定
	cuda8.0	CUDA version の指定
	ptxinfo	レジスタ使用量などカーネルの詳細情報を表示
-Mfixed	固定形式のソースファイル	

4.7 ライブラリの利用

cuBLAS の使い方

モジュールの読み込み

```
use cublas
```

初期化

type(cublasHandle) 型のハンドルを作成する

```
stat = cublasCreate(handle)
```

BLAS ルーチンの例

- 2 乗ノルムの平方根

```
stat = cublas[T]nrm2_v2(handle, size, x, xstride, val)
```

- axpy

```
stat = cublas[T]axpy_v2(handle, size, a, x, xstride, y, ystride)
```

BLAS ライブラリの命名規約と同じく、[T] はデータ型によって次の記号が入る:

S: 単精度実数
D: 倍精度実数
C: 単精度複素数
Z: 倍精度複素数

参考文献

[1] 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

[2] CUDA C PROGRAMMING GUIDE, v10.0, NVIDIA

[3] PGI Compilers & Tools CUDA FORTRAN PROGRAMMING GUIDE AND REFERENCE, Version 2018, NVIDIA