各種計算ハードウェアの活用（D5）
～GPU（GPGPU）で並列演算をおこなう手法～

担当者　長谷 芳樹（神戸市立工業高等専門学校 電子工学科）

1　はじめに

複数のコンピュータをネットワークで接続して演算速度を向上させるメモリ分散型の並列化技術や複数のCPU あるいはマルチコア CPU を用いておこなうメモリ共有型の並列化技術が一般的に使用されているが，その並列化の個数には限界がある。これに対し，GPU（GPGPU: Global-Purpose computing on Graphics Processing Units）を用いた並列計算では，CPU とは比較にならない数の演算ユニットを持っているため，処理内容によっては CPU による演算と比して数倍～数 100 倍，あるいはそれ以上の演算速度を得られる場合もある。

本実験では，GPU を用いた並列計算の基礎的な手法について実習をおこなう。

2　スケジュール

- 第 1 週　5 のおわりまでと 6.1～6.5
- 第 2 週　6.6～6.8，自由課題着手
- 第 3 週　FPGA 自由課題の制作，プレゼンテーション準備
- 第 4 週　FPGA 自由課題のプレゼンテーションとディスカッション
- 第 5 週　CPU の性能を引き出す手法
- **第 6 週　GPU（GPGPU）で並列演算をおこなう手法（この資料）**

3　使用するハードウェアとソフトウェア

本実験では下記の環境を利用する。現在，GPU を用いた計算には，汎用的かつ標準化された言語である OpenCL も広く用いられているが，本実験では NVIDIA 社が C/C++ 言語を独自に拡張して開発した CUDA と呼ばれる言語を用いて開発を体験する。CUDA は NVIDIA 社の GPU でしか動作しないものの，ハードウェアの構造に直結した非常にシンプルな仕様となっているため入門は容易である。

- PC：4 コアの CPU （Intel Core i5-4460 プロセッサ：キャッシュ 6 MB, 動作クロック 3.2 GHz） およびデュアルチャンネルメモリ（DDR3 1600 MHz, 4 GB × 2）を搭載したもの
- GPU：NVIDIA GeForce GTX 1050 Ti (Pascal アーキテクチャ，CUDA コア 768 個，GDDR5 4GB)
- 開発環境： Microsoft Visual Studio Community 2019 [1]
- GPU 開発ライブラリ： CUDA 10.1 Runtime [2]

4　サンプルプログラムのダウンロードと実行

まずは最も簡単なサンプルプログラムをダウンロードし，実行してみる。

- Microsoft Visual Studio Community 2019 を起動する。
- メニューの [新しいプロジェクトの作成] を選び，一覧から [CUDA 10.1 Runtime] を選択し，[次へ] をクリックする。
- プロジェクト名の入力が求められるので，自身の名前などを入力する（例：cuda_nagatani など）。
- ウェブブラウザで https://github.com/nagataniyoshiki/kcct_d/ を開く（git がインストールされている場合は git clone https://github.com/nagataniyoshiki/kcct_d/ としても可）。[3]
- 上記ページの「CUDA」フォルダ内の hello_world_gpu_01.cu の内容を Visual Studio の入力画面にペーストする（Visual Studio に元から入力されているコードは削除する）。
- メニューの [デバッグ] - [デバッグの開始] （あるいは [F5]） を選び，実行する。
- うまく動作すると，「Hello World from GPU!」というメッセージが GPU 側から並列化されている個数だけ表示されるはずである（「コード 0 を伴って終了しました」と表示されていればエラー無く実行されたことを示す）。

◆課題： 10 行目の <<< 1, 10 >>> の 10 の部分の数値を変えて出力を確認する。

5　GPU のプログラミングの概要

5.1　ヘテロジニアスアーキテクチャ

従来のコンピュータには CPU のみが搭載されていることが一般的だったが，近年は CPU に加えて GPU や FPGA などが搭載されることが増えている。このような異種の演算リソースを搭載した計算機を「ヘテロジニアスアーキテクチャ」などと呼ぶ。

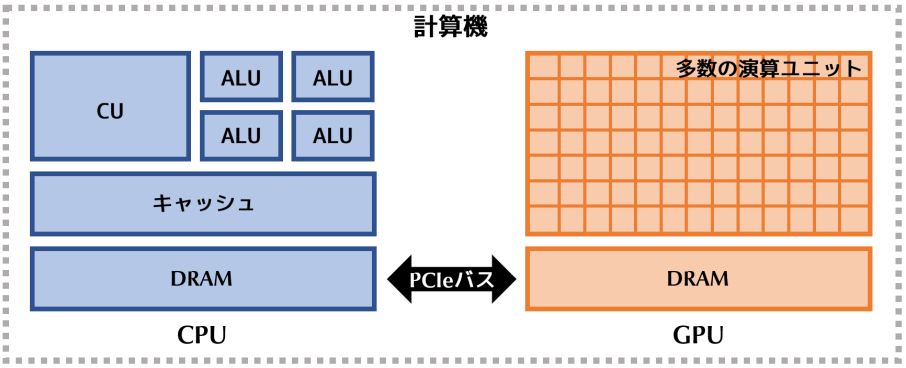

図 1: ヘテロジニアスアーキテクチャ

図 1 に，CPU と GPU を搭載したヘテロジニアスアーキテクチャの例を示す [4] 。このように，CPU と GPU それぞれが演算ユニットとメモリを個別に持っているのが特徴である。なお，CUDA においては，

- CPU 側で実行されるコードを「ホストコード」
- GPU 側で実行されるコードを「デバイスコード」

と呼んでいるので覚えておいてほしい。

CUDA では，ホストコードとデバイスコードを一つのソースコード（.cu ファイル）に記載し，NVIDIA が提供しているコンパイラ（nvcc）でコンパイルする[1] 。この際，図 2 に示す CUDA Driver API を用いても良いが，細かい調整が可能ではあるものの開発の難易度が高いため，通常は CUDA Runtime API を介して GPU の動作を指定する。

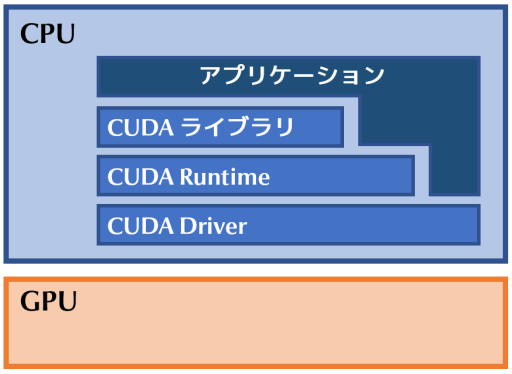

図 2: CUDA Driver API と CUDA Runtime API

5.2 ホストコード（CPU）のみの実行

CUDA（.cu）は C/C++ 言語の拡張であるので，GPU を全く使わない通常のプログラムもそのまま実行可能である。

◆ホストコードのみのサンプル

```
01: #include <stdio.h>
02:
03: int main(void){
04:     printf("Hello World from CPU!\n");
05:
06:     return 0;
07: }
```

◆課題：上記のコードを入力し，動作を確認する。

5.3 デバイスコード（GPU）を含むコードの実行

CUDA では，デバイス（GPU）で動く関数を「カーネル関数」と呼ぶ。

ここでは，5.2 節のプログラムに対し，単に文字列を出力するだけのカーネル関数を追加する。

[1]nvcc のバージョンはコマンドプロンプトで nvcc –version を実行すれば表示できる。ただし，本実験では nvcc を含む CUDA Toolkit が Visual Studio 用にあらかじめセットアップされている PC を用いるので nvcc の挙動を意識する必要は無い。

◆デバイスコードを含むサンプル

```
01: #include <stdio.h>
02:
03: __global__ void helloFromGPU() {
04:     printf("Hello World from GPU!\n");
05: }
06:
07: int main(void){
08:     printf("Hello World from CPU!\n");
09:
10:     helloFromGPU <<<1,10>>> ();
11:
12:     cudaDeviceReset();
13:     return 0;
14: }
```

◆課題：上記のコードを入力し，動作を確認する。

ここで，03 行目の「__global__」がデバイス（GPU）で実行する関数であることを示す修飾子である（表 1 参照）。そして，この関数を 10 行目の「 helloFromGPU <<< 1, 10 >>> (); 」でホスト側から呼び出して（起動して）いる。カーネルの起動については 6.3 節で詳しく説明するのでここでは詳細は割愛する。

表 1: CUDA の関数型修飾子 [4]

修飾子	実行デバイス	呼び出し（実行）	備考
__global__	デバイスで実行	ホストから呼び出し可	戻り値は void のみ
__device__	デバイスで実行	デバイスでのみ呼び出し可（ホストからは呼び出せない）	-
__host__	ホストで実行	ホストでのみ呼び出し可（デバイスからは呼び出せない）	省略可（通常の C 言語の関数と同じように記述可）

なお， __device__ と __host__ の二つの修飾子は同時に使用することが可能で，その場合は関数はホストとデバイスの両方に対してコンパイルされる。

5.4 参考：CUDA コードのコンパイルプロセス

ところで，5.3 節で示したように，CUDA のソースコード（.cu）には，通常の CPU で動作させる C 言語のソースコードと，デバイス（GPU）で動くソースコードが混在している。このため，CUDA コンパイラは，

- nvcc を使ったデバイス関数のコンパイル
- C/C++コンパイラ（本実験では Visual Studio）を使ったホスト関数のコンパイル

に分割してコンパイルをおこなって，図 3 に示すように最終的な一つの実行ファイルを生成している。詳しくは文献 [4] の 10.1.3 節などを参照されたい。

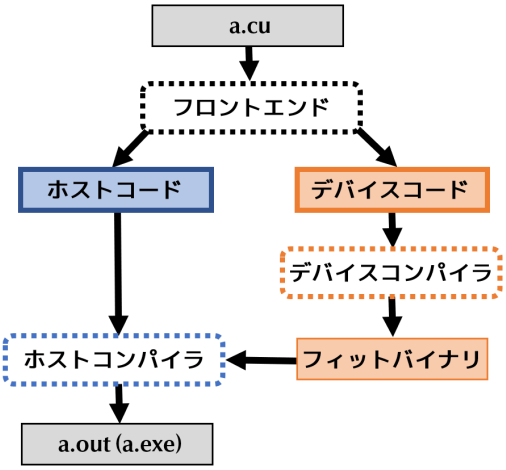

図 3: CUDA コンパイラの 2 つのコンパイルプロセス

6　スレッドの扱い

6.1　スレッドのインデックスの確認

CUDA では，デバイス（GPU）で動く「カーネル関数」が呼び出されると，たくさんの「スレッド」が生成されるが，この「スレッド」をうまく扱うことが CUDA プログラミングの肝となる。図 4 に CUDA におけるスレッドの階層を示す。複数のスレッドからなる「ブロック」と，複数のブロックからなる「グリッド」の二層構造になっていることが分かる。

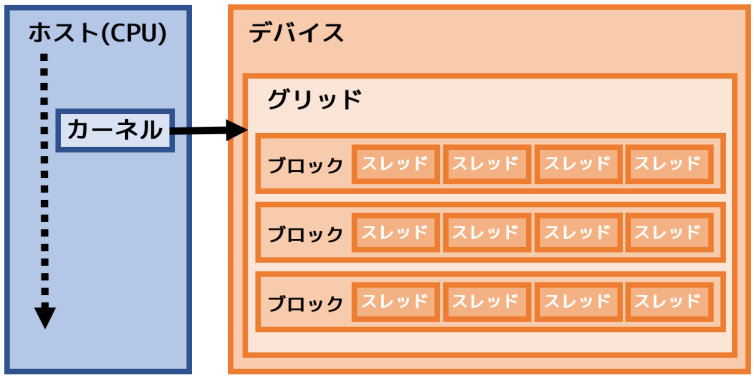

図 4: スレッド階層

さて，ここではまず，ブロック数を 1 つに固定した状態で，各スレッドの ID 番号（インデックスと呼ぶ）を表示するプログラムを作成する。

◆スレッドのインデックスを表示する
（前節のコードから変更のあった行には * を付けている）

```
 01: #include <stdio.h>
 02:
 03: __global__ void helloFromGPU() {
*04:     printf("Hello World from GPU (Thread #%d)!\n", threadIdx.x);
```

```
 05: }
 06:
 07: int main(void){
 08:     printf("Hello World from CPU!\n");
 09:
 10:     helloFromGPU <<<1,10>>> ();
*11:     cudaDeviceSynchronize();
 12:
*13:     printf("Goodbye World from CPU!\n");
 14:
 15:     cudaDeviceReset();
 16:     return 0;
 17: }
```

◆課題：上記のコードを入力し，動作を確認する。
◆課題： 10 行目の <<< 1, 10 >>> の 10 の部分の数値を変えて出力を確認する。

なお，11 行目の cudaDeviceSynchronize() は，全ての GPU スレッドの処理終了を待つ（足並みを揃える）ための命令である（「バリア」などと呼ぶこともある)。むやみに利用すると計算速度の低下を招くが，カーネル呼び出しは非同期であるためにスレッド間のデータの同期等が必要な場合などには必要に応じて利用する。

◆課題： 11 行目の cudaDeviceSynchronize(); をコメントアウトして挙動を確認する。

6.2 参考：GPU での for 文や if 文の実行

もちろん，カーネル関数（GPU）側のコードにも for 文や if 文などの制御文を記述することも可能である。例えば，前節のサンプルの 04 行目の printf の行を以下のように変更することも可能である。

◆ for 文や if 文の実行

```
 03 : __global__ void helloFromGPU() {
 04a:     if(threadIdx.x == 0){
 04b:         printf("I am zero!!\n");
 04c:     }
 04d:     else{
 04e:         for(int i=0; i<4; i++){
 04f:             printf("Hello World from GPU (Thread #%d, Loop %d)!\n", threadIdx.x, i);
 04g:         }
 04h:     }
 05 : }
```

処理速度を考えると GPU 側での分岐やループなどは極力避けるべきではあるが，確保できるスレッド数の制限やアルゴリズム上の都合（計算空間の端面の処理など）でどうしても避けられない場合はこのような処理も可能である。

6.3　カーネルの起動方法

既に利用している通り，CUDA のカーネル（デバイス側で動く関数）の呼び出し（実行）は下記のようにおこなう。

```
関数名 <<<grid, block>>> (引数リスト);
```

ここで，

- grid: グリッド内に何個のブロックを持つか（ブロック数）
- block: 各ブロックに何個ずつのスレッドを生成するか（各ブロックあたりのスレッド数）

をそれぞれ表す。従って，スレッドの総数は grid × block 個となる（それぞれの値が示すものは「個数」ではなく「サイズ」なので注意）。例えば，

```
関数名 <<<2, 4>>> (引数リスト);
```

とすれば，図 5 に示すように，grid 内にブロックが 2 つ，それぞれの block 内にスレットが 4 個ずつ，計 8 個のスレッドが生成される。

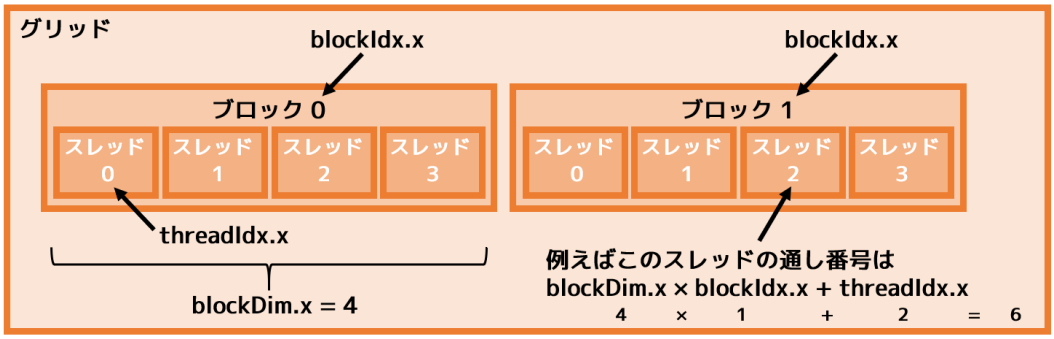

図 5: スレッドのレイアウト

なお，関数の引数については，C 言語と同等の文法で記述する。

6.4　複数ブロック＆複数スレッドの扱い

前節 6.3 で示したように，CUDA ではグリッドサイズとブロックサイズを個別に指定してカーネルの呼び出し（実行）をおこなうことができる。

ここでは，カーネル関数側でブロックサイズとブロックインデックス，そしてスレッドインデックスを得て，自身の通し番号（実際の計算では自身が担当するべき target に相当する番号）を割り出す手法を考える。

CUDA では，カーネル関数が実行されると以下の座標変数が各スレッドに自動的に割り当てられる。

- blockDim.x: ブロックのサイズ（各ブロックあたり何個ずつのスレッドが含まれるか）
- blockIdx.x: 自身のブロックのインデックス（0 から始まる通し番号）
- threadIdx.x: 自身のスレッドのインデックス（各ブロック内での通し番号）

これらの情報から，図 5 の図中にも示したように，全てのスレッドの中で自身が何番目のスレッドであるのかを「blockDim.x × blockIdx.x + threadIdx.x」という式で割り出すことが可能となる[2]。
以下にサンプルコードを示す。

◆スレッドの通し番号を計算して表示する

```
01: #include <stdio.h>
02:
03: /* DATA_SIZE = BLOCK_SIZE * GRID_SIZE で割り切れること（プログラム側ではノーチェック）*/
04: #define DATA_SIZE 16
05: #define BLOCK_SIZE 8
06: #define GRID_SIZE (DATA_SIZE/BLOCK_SIZE)
07:
08: __global__ void helloFromGPU() {
09:     int id = blockDim.x * blockIdx.x + threadIdx.x;
10:     printf("I am blockDim.x=%3d, blockIdx.x=%3d, threadIdx.x=%3d.
                My target is %3d.\n", blockDim.x, blockIdx.x, threadIdx.x, id);
11: }
12:
13: int main(void) {
14:     printf("Hello World from CPU!  DATA_SIZE(%d) =
                BLOCK_SIZE(%d) x GRID_SIZE(%d).\n", DATA_SIZE, BLOCK_SIZE, GRID_SIZE);
15:
16:     helloFromGPU <<<GRID_SIZE, BLOCK_SIZE>>> ();
17:     cudaDeviceSynchronize();
18:
19:     printf("Goodbye World from CPU!\n");
20:
21:     cudaDeviceReset();
22:     return 0;
23: }
```

◆課題：上記のコードを入力し，動作を確認する。
◆課題： 4〜5 行目の BLOCK_SIZE と GRID_SIZE の数値を変えて出力を確認する。

7 メモリのコピー（cudaMemcpy）

7.1 ホストからデバイスへのデータ転送

CUDA では，ホスト（CPU）とデバイス（GPU）との間で，互いのメモリの内容を直接参照したり直接書き換えたりすることはできない。このため，データを扱うためには事前にプログラマーが明示的にデータのコピーをおこなっておく必要がある。このために用意されている関数が cudaMemcpy である。cudaMemcpy は C 言語の memcpy 関数と似通っており，コピー先とコピー元のポインタとデータサイズ，およびコピー

[2]なお，ブロックは 2 次元（座標変数自体は 3 次元ベクトル型であるが z が 1 で固定なので実質的に 2 次元），スレッドは 3 次元の割り当てがそれぞれ可能であるが，ここでは 1 次元の事例のみを扱う。3 次元ベクトルとして指定されなかった値（本資料の例もこの事例に相当する）は 1 に初期化されるため，y=1, z=1 となるので，実質的に x のみの 1 次元として扱えることとなる。詳細については [4] などを参照されたい。

の方向を指定するだけで実行可能である。(cudaMemcpyHostToDevice だと CPU → GPU の向き，逆に cudaMemcpyDeviceToHost だと CPU ← GPU の向きのコピーとなる。)

以下に，ホストからデバイスへのデータ転送のプログラム例を示す。この例では，ホスト側で用意した値 (h_data) をデバイス側に転送 (d_data) し，その後デバイス側で値を 2 倍にして表示をおこなっている。少々長いコードになっているが，どの部分でどのような処理がおこなわれているのかを考えながらぜひ自身で入力して欲しい。(なお，この例では GPU での計算結果は単に画面に表示しているだけで，CPU にはデータとしては転送していない。CPU への転送方法は次節 7.2 で紹介する。)

◆ CPU → GPU へのデータコピー

```
01: #include <stdio.h>
02:
03: /* DATA_SIZE = BLOCK_SIZE * GRID_SIZE で割り切れること (プログラム側ではノーチェック) */
04: #define DATA_SIZE 8
05: #define BLOCK_SIZE 4
06: #define GRID_SIZE (DATA_SIZE/BLOCK_SIZE)
07:
08: /*-------------------------------------------------------------*/
09: /* GPU 側でデータ内容を 2 倍して表示する関数 */
10: __global__ void DoubleOnGPU(float* d_data) {
11:     int id = blockDim.x * blockIdx.x + threadIdx.x;
12:
13:     /* GPU では for 文ではなく，自分の担当のデータ (id) だけ計算すれば OK */
14:     printf("My target is d_data[%d] : %f*2.0=%f.\n", id, d_data[id], d_data[id]*2.0);
15: }
16:
17: /*-------------------------------------------------------------*/
18: int main(void) {
19:
20:     float* h_data;    /* Host(CPU) 側メモリ */
21:     float* d_data;    /* Device(GPU) 側メモリ */
22:
23:     /* ホスト (CPU) 側メモリ領域の確保 (可読性重視のためエラーチェック無しなので注意) */
24:     h_data = (float*)malloc(DATA_SIZE * sizeof(float));
25:
26:     /* デバイス (GPU) 側メモリ領域の確保 (可読性重視のためエラーチェック無しなので注意) */
27:     cudaMalloc((void**)&d_data, DATA_SIZE * sizeof(float));
28:
29:     /* 初期値の代入 (CPU 側で生成：各自で好みの数値を代入すれば良い) */
30:     printf("Data before processing: ");
31:     for (int i = 0; i < DATA_SIZE; i++) {
32:         h_data[i] = (float)(i) * 10.0;
33:         printf("%f, ", h_data[i]);
34:     }
35:     printf("\n");
36:
```

```
37:     /* デバイスにメモリ内容をコピー (CPU → GPU) */
38:     cudaMemcpy(d_data, h_data, DATA_SIZE * sizeof(float), cudaMemcpyHostToDevice);
39:
40:     /* デバイス (GPU) で 2 倍処理を実行 */
41:     DoubleOnGPU <<<GRID_SIZE, BLOCK_SIZE>>> (d_data);
42:
43:     cudaDeviceSynchronize();
44:     cudaDeviceReset();
45:
46:     return 0;
47: }
```

◆課題：上記のコードを入力し，動作を確認する。

ここで特筆すべきは，GPU 側（DoubleOnGPU 関数）では for 文を用いることなく 8 点全てのデータに対する演算を実現している点である。この演算は，for 文で 8 回のループをおこなうかわりに 8 個のスレッドを生成することによって実現している。このように，GPU では各スレッドでのループ処理をおこなわずに，単に自分の担当のデータ (11 行目で計算している id) だけを一個だけ計算すれば済むように工夫してコーディングをおこなうことで，一括して多数の演算を実施することが可能となる。これにより，CPU よりもシンプルでクロック数でも劣る GPU であっても格段に高速な処理が期待できるわけである[3] 。

なお，CUDA にはこのサンプルプログラムで使用しているもの以外にも標準 C 関数と似通った関数が用意されているので，必要に応じて使用する。表 2 に標準 C 関数と CUDA C 関数の対応を示しておく。

表 2: ホストとデバイスのメモリ関数

標準 C 関数	CUDA C 関数	動作
malloc	cudaMalloc	メモリの確保
memcpy	cudaMemcpy	メモリのコピー
memset	cudaMemset	メモリの値をセット
free	cudaFree	メモリの解放

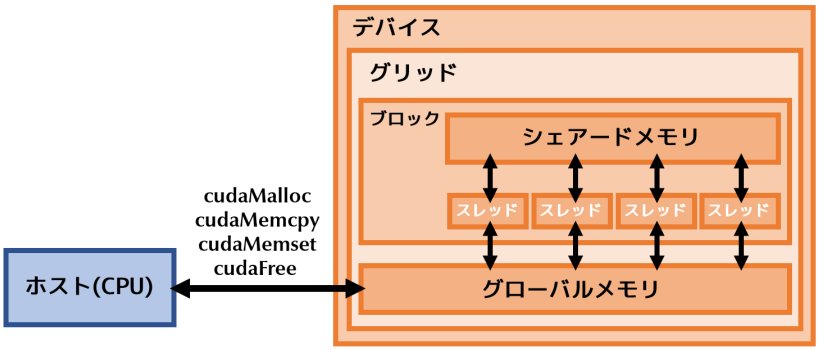

図 6: GPU のメモリ構造

[3]もちろん，GPU で演算をおこなうためには CPU ←→ GPU 間のメモリ転送が必要となる。このメモリ転送のためのオーバーヘッドよりも演算時間の短縮効果が大きい場合にのみ，トータルの演算時間が短縮される。なお，アプリケーションのどの部分にどれだけ時間が掛かっているのかを調べるためには nvprof というツールが役に立つので，必要に応じて各自で試してみて欲しい。

また，図 6 に GPU のメモリ構造を示す。このように，表 2 に示す関数を用いてホストからデバイス側のメモリーを操作して所望の演算を実現するのである。

7.2 デバイスからホストへのデータ転送

次に，デバイスからホストへのデータ転送方法を示す。単に，cudaMemcpy の引数の順番を変え（1 番目が転送先，2 番目が転送元），また 4 番目の引数を cudaMemcpyDeviceToHost に変更するだけである。

以下にプログラム例を示す。この例では，正しく転送されたかを確認するため，最後に CPU 側で再度データ内容を表示している。

◆ CPU → GPU へのデータコピー ＆ CPU ← GPU へのデータコピー

（前節のコードから変更のあった行には * を付けている）

```
 01: #include <stdio.h>
 02:
 03: /* DATA_SIZE = BLOCK_SIZE * GRID_SIZE で割り切れること (プログラム側ではノーチェック) */
 04: #define DATA_SIZE 8
 05: #define BLOCK_SIZE 4
 06: #define GRID_SIZE (DATA_SIZE/BLOCK_SIZE)
 07:
 08: /*-------------------------------------------------------------*/
 09: /* GPU 側でデータ内容を 2 倍して表示する関数 */
*10: __global__ void DoubleOnGPU(float* d_data, float* d_data2) {
 11:     int id = blockDim.x * blockIdx.x + threadIdx.x;
 12:
 13:     /* GPU では for 文ではなく，自分の担当のデータ (id) だけ計算すれば OK */
*14:     d_data2[id] = d_data[id] * 2.0;
*15:     printf("My target is d_data[%d] : %f*2.0=%f.\n", id, d_data[id], d_data2[id]);
 16: }
 17:
 18: /*-------------------------------------------------------------*/
 19: int main(void) {
 20:
 21:     float* h_data;    /* Host(CPU) 側メモリ */
*22:     float* h_data2;   /* Host(CPU) 側メモリ */
 23:
 24:     float* d_data;    /* Device(GPU) 側メモリ */
*25:     float* d_data2;   /* Device(GPU) 側メモリ */
 26:
 27:     /* ホスト (CPU) 側メモリ領域の確保（可読性重視のためエラーチェック無しなので注意） */
 28:     h_data = (float*)malloc(DATA_SIZE * sizeof(float));
*29:     h_data2 = (float*)malloc(DATA_SIZE * sizeof(float));
 30:
 31:     /* デバイス (GPU) 側メモリ領域の確保（可読性重視のためエラーチェック無しなので注意） */
 32:     cudaMalloc((void**)&d_data, DATA_SIZE * sizeof(float));
*33:     cudaMalloc((void**)&d_data2, DATA_SIZE * sizeof(float));
```

```
 34:
 35:     /* 初期値の代入（CPU 側で生成） */
 36:     printf("Data before processing: ");
 37:     for (int i = 0; i < DATA_SIZE; i++) {
 38:         h_data[i] = (float)(i) * 10.0;
 39:         printf("%f, ", h_data[i]);
 40:     }
 41:     printf("\n");
 42:
 43:     /* デバイスにメモリ内容をコピー（CPU → GPU） */
 44:     cudaMemcpy(d_data, h_data, DATA_SIZE * sizeof(float), cudaMemcpyHostToDevice);
 45:
 46:     /* デバイス（GPU）で 2 倍処理を実行 */
*47:     DoubleOnGPU <<<GRID_SIZE, BLOCK_SIZE>>> (d_data, d_data2);
 48:
 49:     /* デバイスからメモリ内容をコピー（CPU ← GPU） */
*50:     cudaMemcpy(h_data2, d_data2, DATA_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
 51:
 52:     cudaDeviceSynchronize();
 53:     cudaDeviceReset();
 54:
 55:     /* 結果の表示（CPU 側で） */
*56:     printf("Data after processing: ");
*57:     for (int i = 0; i < DATA_SIZE; i++) {
*58:         printf("%f, ", h_data2[i]);
*59:     }
 60:
 61:     return 0;
 62: }
```

◆課題：上記のコードを入力し，動作を確認する。

このプログラムのフローを図 7 に示す。このように，CPU 側でデータの準備や転送の指示をおこない，GPU 側で計算をおこなって終了したらまたそれを CPU 側に戻してくる，というような流れが標準的である。

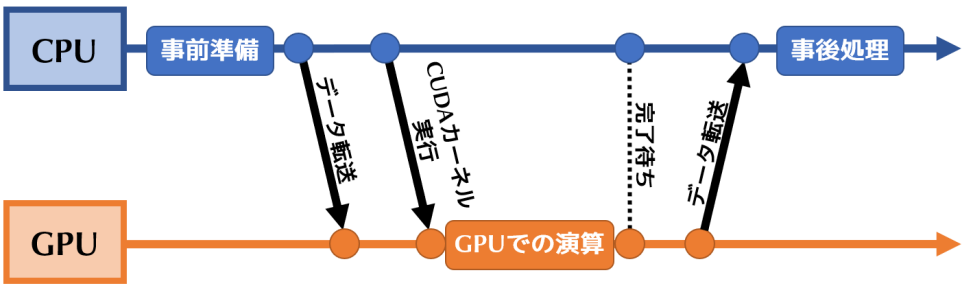

図 7: CUDA プログラムの一般的なフロー

8 GPUによる演算速度の向上

さて，GPU の概要をある程度理解したところで，いよいよ次は GPU で演算することによってどの程度処理速度が向上するのかを体験する。

8.1 CPUによる演算

まずは GPU は使わずに CPU のみで演算をおこなうプログラムを試す。筆者の GitHub リポジトリ [3] より multiply_cpu_01.cu をダウンロードして実行する。これは DATA_SIZE のサイズの配列を 2 つ用意し（値はランダム），それらの値の積を CPU のみで求めるプログラムである。

以下に，配列同士の積を求める関数（CPU で実行される関数）を示す。

◆ DATA_SIZE 個の浮動小数点の積演算を CPU でおこなう（抜粋）

```
13: /* 積演算 R=A*B をおこなう関数 (単一コア) */
14: void MultiplyOnCPU(float* h_data_A, float* h_data_B, float* h_data_R) {
15:     long i;
16:
17:     /* CPU ではデータの数だけ for 文をまわす */
18:     for (i = 0; i < DATA_SIZE; i++) {
19:         h_data_R[i] = h_data_A[i] * h_data_B[i];
20:     }
21: }
```

このように，これまでにも何度も述べたとおり，CPU で計算するためにはデータの数だけ for 文をまわす必要がある。

8.2 GPUによる演算

次に， GPU で演算をおこなうプログラム multiply_gpu_01.cu を試す。これは，前節 8.1 と同じく，DATA_SIZE のサイズの配列を 2 つ用意し（値はランダム），それらの値の積を求めるプログラムであるが，積の演算を GPU 側でおこなっているという違いがある。

以下に，GPU で配列同士の積を求めている部分のみを示す。

◆ DATA_SIZE 個の浮動小数点の積演算を GPU でおこなう（抜粋）

```
27: /* GPU 側で積演算をおこなう関数 */
28: __global__ void MultiplyOnGPU(float* d_data_A, float* d_data_B, float* d_data_R) {
29:     int id = blockDim.x * blockIdx.x + threadIdx.x;
30:
31:     /* GPU では for 文ではなく，自分の担当のデータ (id) だけ計算すれば OK */
32:     d_data_R[id] = d_data_A[id] * d_data_B[id];
33: }
```

```
36: int main(void) {
        /* デバイス (GPU) で積演算を実行 */
84:     MultiplyOnGPU <<<GRID_SIZE, BLOCK_SIZE >>> (d_data_A, d_data_B, d_data_R);
99: }
```

この演算は，図 8 に示すように，for 文で DATA_SIZE 回（図 8 では 8 回）のループをおこなうかわりに DATA_SIZE 個（図 8 では 8 個）のスレッドを生成することによって実現しているため，MultiplyOnGPU 関数は非常にシンプルなものとなっていることがわかる。

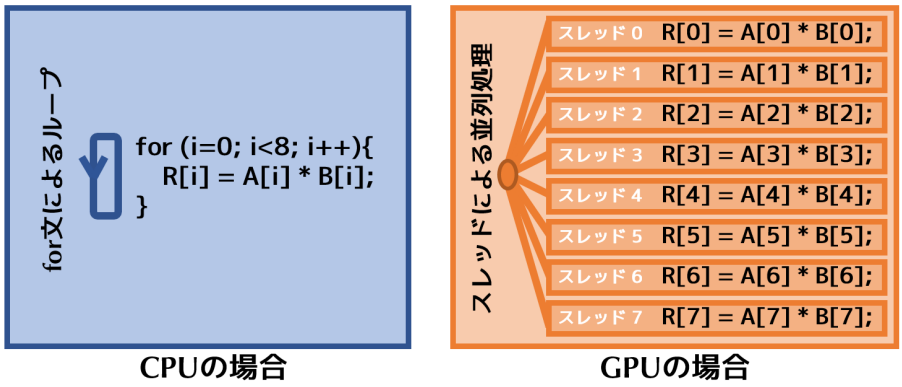

図 8: for ループによる逐次処理とスレッドによる並列処理

◆課題：上記のコードには結果の一部を表示する機能も含めてある（93〜96 行目）ので，CPU での実行結果と GPU での実行結果が一致していることを確認する。

8.3 CPU と GPU の演算速度の比較

最後に，CPU と GPU の演算速度の比較をおこなう。multiply_gpu_02.cu [3] には演算速度の表示機能を実装してある。（演算内容は前節 8.1 と同じである。）

◆課題：CPU での実行結果と GPU での演算時間を確認する。ただし，このサンプルでは秒単位でしか表示がされないので，必要であれば REPEAT 回数を変更して比較する。
◆課題：このサンプルでは cudaMemcpy に掛かる時間は測定していないので，時間計測対象（116〜120 行目）に cudaMemcpy（「CPU → GPU」のコピーと「CPU ← GPU」のコピーの両方）が繰り返しをおこなっている for 文の中に含まれるようにプログラムを書き換えて，再度時間を計測する。

8.4 OpenMP の利用（CPU）

ところで，前節 8.3 のプログラムは，GPU はマルチスレッディングを用いているのに対して CPU はシングルコアでの演算であり，少々不公平な比較である。

前節のプログラム multiply_gpu_02.cu には OpenMP によるスレッド並列化のためのディレクティブを記載した関数も用意してある（40 行目）のでこれを用いる。

◆ **DATA_SIZE 個の浮動小数点の積演算を OpenMP (CPU) でおこなう（抜粋）**

```
35: /* CPU 側で積演算 R=A*B をおこなう関数 (OpenMP) */
36: void MultiplyOnCPU_OpenMP(float* h_data_A, float* h_data_B, float* h_data_R) {
37:     long i;
38:
39:     /* CPU ではデータの数だけ for 文をまわす（ここを OpenMP で並列化する） */
40:     #pragma omp parallel for
41:     for (i = 0; i < DATA_SIZE; i++) {
42:         h_data_R[i] = h_data_A[i] * h_data_B[i];
43:     }
44: }
```

ただし，OpenMP を有効化するにはホスト（CPU）コードのコンパイルオプションを指定してコンパイルをおこなう必要がある。Visual Studio では [プロジェクト] - [(自身が指定したプロジェクト名) のプロパティ] - [CUDA C/C++] - [Host] - [Additional Compiler Options] に「 -Xcompiler "/openmp" 」を追加すれば OpenMP が有効化される。（なお，gcc の場合は「 /openmp 」のかわりに「 -fopenmp 」とする[4]。）

◆課題：CPU での実行結果（シングルコア，OpenMP によるスレッド並列），GPU での演算時間を確認する。また，その際の CPU 利用率の変動をタスクマネージャーで観察する。
◆課題：メモリコピーの回数は変えずに演算量を増やして，CPU と GPU での計算時間の増え方を比較する。例えば R=A*A*B*B*B のように積演算を増やしてみる，あるいは log や sqrt の追加などを試すと比較がしやすいかもしれない。

9 簡易レポート

第 6 週の課題はレポートは不要である。この替わりに，実験終了時に簡易レポートを提出する（A4 用紙 1 枚程度)。PC 等で作成した場合はファイルを nagatani 宛に送信しても良い。

- 各条件で実行したプログラムの実行時間（あるいは速度の向上率）を列挙する。
- 簡単なコメントを記す（簡易考察)。
- その他，アイデアや感想等があれば記す。

10 発展課題

ここまでで紹介した手法は CUDA を用いた演算手法のごく一部である。各自，興味があれば深めて欲しい。

- 課題で実施した計算以外の計算（第 5 週に実施した FDTD 法の計算や自身の卒業研究のテーマなど）

[4]「-Xcompiler」で指定したオプションがホストコンパイラに渡される。Visual Studio 等を使わずに直接 nvcc を使う場合でも同様に「nvcc ファイル名.cu -Xcompiler "-fopenmp"」のようにオプションを追加するだけでよい。

- cuFFT ライブラリ（CUDA に最適化された高速フーリエ変換ライブラリ）や cuRAND ライブラリ（準乱数および疑似乱数を発生させるライブラリ）の活用

- さらに効率的なコードの追求（計算量の低減，アムダールの法則 etc.）
- シェアードメモリ（容量は小さいがより高速なメモリ）の活用
- CUDA 以外の GPU 演算手法との比較，FPGA との比較

11 注意事項

- ソフトウェア工学実験室の PC は随時ファイル削除等をおこなうので，作成・更新したプロジェクトやスライドはこまめに各自の USB メモリやオンラインストレージ等に保存すること。
- ソフトウェア工学実験室の PC，プリンタ，ネットワーク機器などはていねいに扱うこと（**これは担当者からの強いお願いです！**）。

参考文献

[1] “ダウンロード | IDE、Code、Team Foundation Server | Visual Studio,” https://visualstudio.microsoft.com/ja/downloads/ .

[2] “CUDA Toolkit 10.1 Update 1 Download | NVIDIA Developer,” https://developer.nvidia.com/cuda-downloads .

[3] “Documents for the Experiments in the Department of Electronics, Kobe City College of Technology,” https://github.com/nagataniyoshiki/kcct_d/ .

[4] John Cheng 他著, 株式会社クイープ訳，森野慎也監訳，“CUDA C プロフェッショナル プログラミング,” インプレス (東京, 2015).

(URL は 2019 年 5 月 22 日閲覧)
(rev. 201907)