CUDA技術を利用したGPU(Graphics Processing Unit)コンピューティングの実際

後編

関連データ

FFTを利用した光波の伝播(フレネル回折)をGPUで高速計算

下馬場 朋禄，伊藤 智義

ここではGPU(Graphics Processing Unit)コンピューティングを利用したプログラミングの事例を紹介する．行列の乗算を例に，プログラミングの流れを示す．また，処理を高速化するためのカーネルの実装方法についても解説する．最後に，FFT(高速フーリエ変換)を利用するフレネル回折の計算をGPUで行った例を紹介する．
(編集部)

1．CUDAを用いたプログラミング

ここでは行列の乗算を例として，CUDA(クーダ)プログラミングの流れを把握していきます．このプロジェクト(CQ_CUDA_matrix)は，Interface誌のWebサイト(http://www.cqpub.co.jp/interface/)からダウンロードできます．

行列の乗算はCUDAのプログラミング・ガイド[1]にも例として登場します．行列の乗算は誰もが知っている計算方法で理解しやすいため，本稿でもこの計算を例として取り上げたいと思います．また，行列の乗算には計算量を減らすアルゴリズムが提案されていますが，ここではそのようなアルゴリズムを採用せず，行列の乗算を筆算の要領で行う方法を実装することにしました．

本稿では3種類のカーネルを用意し，プログラミングの仕方によってどの程度，計算速度に差が出るのかを見ていきます．

まず，ホスト側のソース・コードを**リスト1**に示します．**リスト1**を見ると，普通のC言語とほとんど変わらないことが分かります．

CUDAを用いたホスト側のプログラミングの流れは，一般に以下のようになります．

- GPUの初期化
- GPU上のデバイス・メモリ領域の確保
- ホストがGPUに計算させるデータを用意
- CUDAのAPI(Application Programming Interface)を使用して，ホストからデバイス・メモリへデータを転送
- スレッド数，ブロック数を指定してカーネルを実行
- ホストがCUDAのAPIを使用してデバイス・メモリ上の計算結果を回収

ここでは，この流れに沿って**リスト1**を見ていきます．

```
cudaMalloc(void** devPtr, size_t count)
引き数
  devPtr：GPU上に確保したメモリ領域へのポインタ
  count：確保するメモリ容量(バイト単位)
```

図1　cudaMallocの書式
GPU上のグローバル・メモリを確保する際に利用する．

● ホスト側の処理の流れを見る

まず，**リスト1**の①はGPUの初期化を行う部分です．初期化には，ホストに接続されているCUDA対応のGPUの個数を調べたり，認識できたGPUの特性(ブロック当たりのレジスタ数や搭載メモリ容量など)を取得したりするステップを踏む必要があります．この手順を一まとめにしたマクロCUT_DEVICE_INITがヘッダ・ファイルcutil.hに用意されています．このマクロを使用することで，手順を簡略化できます(ただし，実際にはこのマクロを省いても動作する)．

次に②の部分で，GPU上のグローバル・メモリに行列データを格納する領域を確保します．GPUはこの確保されたメモリ中のデータに対して演算を行うことになります．GPU上のグローバル・メモリを確保するにはCUDAのAPIであるcudaMallocを使用します(**図1**)．このAPIを使用することで1次元から多次元までのメモリ領域を確保できます．ほかにもメモリ確保用のAPIが用意されていますが，本稿ではこのAPIのみを使用します．

このプログラムでは，WIDTH × WIDTHサイズの行列Aと行列BをGPU上で乗算し，その結果をGPU上のメモリCという領域に格納するため，float型でWIDTH × WIDTHサイズの配列をそれぞれd_a，d_b，d_cに確保します．

③の部分で，行列Aと行列Bのデータをホスト上に用意します．このデータはホスト側の配列h_a，h_bに格納し

リスト1　ホスト側のソース・コード（main.cuファイルからの抜粋）

```
#include <stdio.h>
#include <conio.h>
#include <cutil.h> //CUDAのユーティリティ関連ヘッダ・ファイル

//マクロ宣言
#define BLOCK 16
#define WIDTH 512

//プロトタイプ宣言
void Host(float *a, float *b, float *c);
__global__ void Kernel1(float *a, float *b, float *c);
__global__ void Kernel2(float *a, float *b, float *c);
__global__ void Kernel3(float *a, float *b, float *c);
void SetTimer(unsigned int *t);
float EndTimer(unsigned int *t);

//行列データと計算結果格納用配列
float h_a[WIDTH*WIDTH];
float h_b[WIDTH*WIDTH];
float h_c[WIDTH*WIDTH];

//メイン関数
int main()
{
  int i;
  unsigned int timer;

  //GPUの初期化 ←①
  CUT_DEVICE_INIT();

  //GPU上にメモリを確保 ←②
  float *d_a, *d_b, *d_c;
  cudaMalloc((void**) &d_a, sizeof(float)*WIDTH*WIDTH);
  cudaMalloc((void**) &d_b, sizeof(float)*WIDTH*WIDTH);
  cudaMalloc((void**) &d_c, sizeof(float)*WIDTH*WIDTH);
  cudaMemset(d_c, 0, sizeof(float)*WIDTH*WIDTH);
                                          //d_cを0で初期化する

  //行列データ作成 ←③
  for(i=0; i<WIDTH*WIDTH; i++)
  {
    h_a[i]=i;
    h_b[i]=i;
  }

  SetTimer(&timer);

  //変数をGPU上のメモリへコピー ←④
  cudaMemcpy(d_a, h_a, sizeof(float)*WIDTH*WIDTH,
      cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, h_b, sizeof(float)*WIDTH*WIDTH,
      cudaMemcpyHostToDevice);

  //ブロックとグリッドの定義とカーネルの起動 ←⑤
  dim3 grid(WIDTH/BLOCK, WIDTH/BLOCK, 1);
  dim3 threads(BLOCK, BLOCK, 1);
  Kernel1<<< grid, threads >>>(d_a, d_b, d_c);
                                          ←カーネル関数名

  //計算結果を取得 ←⑥
  cudaMemcpy(h_c, d_c,
      sizeof(float)*WIDTH*WIDTH,cudaMemcpyDeviceToHost);

  printf("計算時間=%f(ms)",EndTimer(&timer));
  printf(" 計算結果=%f\n",h_c[WIDTH*WIDTH-1]);

  //GPU上のメモリを開放 ←⑦
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  //ホスト側での計算（比較用）
  SetTimer(&timer);
  Host(h_a, h_b, h_c);
  printf("ホスト計算時間=%f(ms)",EndTimer(&timer));
  printf("計算結果=%f\n",h_c[WIDTH*WIDTH-1]);

  getch();
}
```

ます．なお，h_cはGPUが計算した乗算結果をホストが回収するための配列になっています．

④の部分で，ホスト側で用意したデータを，先ほどGPU上に確保したメモリ領域（グローバル・メモリ）へコピーします．コピーの作業にはCUDAのAPIであるcudaMemcpy（**図2**）を使用します．ここでは，行列Aのデータh_a（ホスト側）をd_a（GPU側）へ，行列Bのデータh_b（ホスト側）をd_b（GPU側）へ，float型でWIDTH×WIDTH個コピーします．

⑤の部分で，実際にGPUで行列の乗算を行わせるカーネルを起動します．カーネルを起動する際に，そのカーネルをどの程度のブロック数とスレッド数で実行するかを指定しなければなりません．カーネル起動の書式を**図3**に示します．

C言語にはない書式ですが，<<< 〜 >>>の部分がブロック数とスレッド数を指定する部分です．そのあとに，通常

```
cudaMemcpy(void* dst, const void* src, size_t count,
enum cudaMemcpyKind kind)
引き数
  dst：転送先のポインタ
  src：転送元のポインタ
  count：転送データ量（バイト単位）
  kind：転送方向．下記のパラメータを指定できる
```

パラメータ	データ転送の方向
cudaMemcpyHostToHost	ホスト→ホスト
cudaMemcpyHostToDevice	ホスト→GPU
cudaMemcpyDeviceToHost	GPU →ホスト
cudaMemcpyDeviceToDevice	GPU →GPU

図2　cudaMemcpyの書式

ホスト側で用意したデータをGPU上に確保したメモリ領域（グローバル・メモリ）へコピーする際に利用する．

```
関数名<<<Dg, Db, Ns >>>(関数の引き数)
カーネル起動パラメータ
  Dg：dim3型でグリッド（カーネル）当たりのブロック数を2次元
      まで指定可能
  Db：dim3型でブロック当たりのスレッド数を3次元まで指定可能
  Ns：カーネル内で使用するシェアード・メモリを動的に割り当て
      る場合，その使用量をバイト・サイズで指定．動的に割り当
      てを行わない場合，省略することができる
```

図3　カーネル起動の書式

GPUで行列の乗算を行わせるカーネルを起動する際に利用する．カーネルをどの程度のブロック数とスレッド数で実行するかを指定する．