CUDA による GPGPU の基礎

山尾 創輔

平成 24 年 7 月 17 日

1 目的

GPU (Graphics Processing Unit) は，計算機のグラフィックス処理を専門とするハードウェアであり，その演算性能は極めて高い．近年では，その特長をグラフィックス処理だけでなく，汎用計算に利用する GPGPU (General-Purpose computation on Graphics Processing Unit) と呼ばれる試みが注目を集めている．本実験では，統合開発環境 CUDA (Compute Unified Device Architecture) を用いた一般的なプログラムの GPU 実装と速度評価を通して，GPGPU の基礎を理解する．

2 原理

CUDA は，NVIDIA 製 GPU を用いた GPGPU を目的とする統合開発環境である．そのため，CUDA を使用するためには，NVIDIA 製 GPU のアーキテクチャや CUDA プログラムの動作原理といった知識が必要となる．そこで本章では，CUDA を用いた GPGPU の基礎として CUDA のプログラミングモデル，GPU 実装における処理手続きについて説明する．また，実験で扱うバイラテラルフィルタとテンプレートマッチングについて概要を述べる．

2.1 CUDA のプログラミングモデル

CUDA の動作環境として，ビデオカードがマザーボードへ搭載されたコンピュータを考える．このとき，マザーボードには CPU (Central Processing Unit) とメインメモリが搭載され，ビデオカードには GPU とビデオメモリ (VRAM: Video RAM) が搭載される．CUDA のプログラミングモデルでは，マザーボードおよび CPU 側をホストと呼び，GPU および VRAM 側をデバイスと呼ぶ．

CUDA のプログラムは，(a) ホスト上の CPU を動作させるホストコードと，(b) デバイス上の GPU を動作させるデバイスコードから構成される，異種混載マルチプロセッシング (heterogeneous multiprocessing) に基づく C/C++統合プログラムである．(a) ホストコードは，通常の C 言語で記述されたプログラムに加え，GPU の動作実行を命令するカーネル関数の呼び出しが含まれる．(b) デバイスコードには，カーネル関数の処理が記述される．`nvcc` コマンドは，これらのコードへプログラムを分割しコンパイルを行う．`nvcc` コマンドで生成される実行プログラムの処理の流れは以下のようになる．

step 1. ホスト側からプログラムを開始する．このとき，デバイス側へカーネル関数がロードされる

step 2. 処理に必要なデータをホスト側で用意し，ホスト側からデバイス側へデータを転送する

step 3. ホスト側からカーネル関数を起動し，デバイス側でデータを処理する

step 4. 処理結果をデバイス側からホスト側へ転送する

以降では，NVIDIA 製 GPU のアーキテクチャおよびメモリモデルの概要，CUDA におけるスレッドとその管理について説明する．

2.1.1 GPU のアーキテクチャ

NVIDIA 製ビデオカードでは，GPU と VRAM がメモリインタフェースで接続される．ハイエンド・ビデオカードでは，そのビット幅は 512bit にまで及び，一般的な CPU のビット幅 32bit, 64bit と比較して，ビデオメモリとの高速なデータ転送が可能である．

GPU チップの内部には，ストリーミング・マルチプロセッサ (Streaming Multi-processor: SM) が複数個入っている．さらに SM の内部には，最小単位の演算処理ユニットであるスカラ・プロセッサ (Scalar Processor: SP) が複数個 (標準では 8 個) 置かれる．SP は，CPU に比べると，算術演算やロジック処理等の限られた性能しか持ち合わせていないが，データ処理用途には十分な性能を持つ．この複数個の SP で同じプログラムを並列動作させることで，大量のデータを並列に処理することができる．

2.1.2 GPU のメモリモデル

GPU のメモリ構成には，複数の種類が存在する．CUDA プログラミングでは，これらのメモリの特徴をうまく利用することで，より高速な演算が可能となる．GPU のメモリ構成は，VRAM 上のメモリ (オフチップメモリ) と，GPU 上のメモリ (オンチップメモリ) に分けられる．

オフチップメモリには，中でも大きい容量をもつグローバルメモリ (Global Memory) ，スレッドの作業領域であるローカルメモリ (Local Memory) ，テクスチャ機能で利用されるテクスチャメモリ (Texture Memory) ，デバイスメモリのキャッシュ領域であるコンスタントメモリ (Constant Memory) がある．これらのメモリ構成は，容量が大きい反面，アクセス速度が遅いという特徴を共通して持つ．

一方，オンチップメモリには，SP の演算データを管理するレジスタ (Register) ，同一ブロック内の SP からアクセス可能なシェアードメモリ (Shared Memory) がある．これらのメモリ構成は，VRAM 上のメモリと比べると，アクセス速度に優れる反面，容量が小さい．レジスタは変数の格納領域として利用されるが，要求が容量を上回るとローカルメモリへレジスタのデータを退避する．そのため，無駄な変数の確保はパフォーマンスが低下する原因となる．

各デバイスメモリの特徴を表 1 へまとめる．

表 1: デバイス・メモリ

メモリの種類	場所	容量	アクセス速度	キャッシュ	アクセス	スコープ
レジスタ	オンチップ	小	速	-	R/W	当該スレッド
シェアードメモリ	オンチップ	中	速	-	R/W	ブロック内のスレッド
ローカルメモリ	オフチップ	中	遅	無	R/W	当該スレッド
グローバルメモリ	オフチップ	大	遅	無	R/W	全スレッド・ホスト
コンスタントメモリ	オフチップ	中	速	有	R	全スレッド・ホスト
テクスチャメモリ	オフチップ	大	速	有	R	全スレッド・ホスト

2.1.3 スレッド

GPU 上で並列に動作する多数のプログラムの最小単位をスレッドと呼ぶ．ホスト側からのスレッドの呼び出しには 1 スレッドあたり 1 クロックを要し，プログラム開始のタイミングがずれることでそれぞれの動作は非同期となる．また厳密には，SM 内部に搭載された SP の最低動作クロック数は 4 であるため，各 SP は一度の動作で 4 つのスレッドを呼び出す．つまり，SM あたり 8 個の SP が搭載されている場合は，各 SM 内部では，一度の動作 (4 クロック) で 32 スレッドが動作する．この 32 スレッドを 1 つのスケジューリングの単位とみなすものをウォープ (Warp) と呼ぶ．

ウォープ内のアクティブなスレッド (最大 32) は単一命令複数データ (Single-Instruction Multiple-Data: SIMD) 的に並列に実行される．条件分岐が存在する場合は，ウォープはすべての経路を逐次的に実行するため，ウォープ内にはアクティブでないスレッドが生じ，アクティブなスレッドの速度が遅くなる．このよ

うな条件分岐による性能低下を避けるために，ウォープ内のスレッドが一律にアクティブ状態のまま並列に実行されるようチューニングするのが望ましい．また GPU のアーキテクチャでは，そのような状況を単一命令複数スレッド (Single-Instruction Multiple-Thread: SIMT) と呼ぶ．SIMT では，ウォープ単位で，同一の命令を実行できるスレッドと，その間アイドル状態であるスレッドを動的に検出する．

2.1.4 スレッドの管理

CUDA では，グリッドとブロックという階層的な概念でスレッドを管理する．グリッドは，スレッド階層における最上位の階層であり，1 グリッドは 1 デバイスに相当する．グリッドの内部は，ブロックという 2 次元的な階層 (GPU の世代によっては 3 次元的に考えることも可能) で構成され，さらにブロックの内部では，スレッドが 3 次元的に配置される．ブロック内の個々のスレッドは，データを共有するために，単一の SM 内で実行される．ブロックの数が SM の総数を上回る場合は，各 SM に複数のブロックが割り当てられ，ブロックが行うメモリアクセスのレイテンシを隠蔽するなどの効率化を図るよう，各ブロックの実行がスケジューリングされる．以上のようなスレッドの階層構造は，ある問題を並列性を有する小さな問題へ分解する作業と自然に対応付けられる．

グリッドあたりのブロック数やブロックあたりのスレッド数にはハードウェアのバージョン (Compute Capability) ごとの制約がある．例えば NVIDIA Tesla C2050 は Compute Capability が 2.0 であり，グリッドの各次元における最大のブロック数は $65535 \times 65535 \times 65535$ ，ブロックの各次元における最大のスレッド数は $1024 \times 1024 \times 64$ ，ブロックあたりの最大スレッド数は 1024 である．CUDA では，3 次元的に配置されたブロックおよびスレッドの位置を示すために，以下の変数が予約されている．

`blockIdx` グリッドにおけるブロックのインデックスを示す変数．フィールド `x`, `y`, `z` がそれぞれの次元に対応する．

`blockDim` ブロック内のスレッド数を示す変数．フィールド `x`, `y`, `z` がそれぞれの次元に対応する．

`threadIdx` ブロックにおけるスレッドのインデックスを示す変数．フィールド `x`, `y`, `z` がそれぞれの次元に対応する．

2.2 GPU での処理手続き

GPU での演算は (i) デバイスメモリの確保，(ii) ホストメモリからデバイスメモリへのデータ転送，(iii) カーネル関数の呼び出し，(iv) デバイスメモリからホストメモリへのデータ転送，(v) デバイスメモリの解放の順に行う．以降では，`VEC_SIZE` 次元ベクトル `A`, `B` の加算を GPU 上で行うプログラム (表 2) を例に，上記 (i) から (v) までの処理の詳細を説明する．

(i) デバイスメモリの確保

GPU の演算処理はデバイスメモリ上のデータに対してのみ行うことができる．そのため，ホストメモリのデータをデバイスメモリへ転送する必要があるが，その前に，`cudaMalloc()` 関数を用いて，ホストメモリのデータを格納する領域をデバイスメモリ上に確保しておく必要がある．表 2 では，単精度浮動小数点型 (`float`) の `VEC_SIZE` 次元ベクトル `A`, `B`, `C` のデータを格納する領域を，`cudaMalloc()` 関数によってグローバルメモリ上へ確保している．確保された領域の先頭アドレスは，それぞれ `d_A`, `d_B`, `d_C` に割り当てられる．

(ii) ホストメモリからデバイスメモリへのデータ転送

cudaMemcpy() 関数を用いて，ホストメモリのデータを (i) で確保したデバイスメモリ上の領域へ転送する．このとき，データの転送方向を cudaMemcpyHostToDevice と設定する．表 2 では，A, B, C に格納されたデータが領域 d_A, d_B, d_C に格納される．

(iii) カーネル関数の呼び出し

オペレータ<<<>>>でグリッドの次元・ブロックの次元を指定し，カーネル関数を呼び出す．呼び出すカーネル関数には修飾子 __global__ をつけておく．表 2 では，グリッドの次元とブロックの次元を dim3 型変数 dimGrid, dimBlock として定義し，カーネル関数 VecAddKernel() を呼び出している．

(iv) デバイスメモリからホストメモリへのデータ転送

(ii) と同様に，cudaMemcpy() 関数を用いて，デバイスメモリ上のデータをホストメモリへ転送する．ただし，データの転送方向を cudaMemcpyDeviceToHost と指定する．表 2 では，デバイスメモリ上の領域 d_C のデータを C へ転送し，GPU の計算結果をホストメモリ上に格納している．

(v) デバイスメモリの解放

cudaFree() 関数を用いて，(i) で確保した領域 d_A, d_B, d_C を解放する．

表 2: ベクトルの加算を行うプログラム

```
#include<stdio.h>

#define VEC_SIZE 16
#define BLOCK_SIZE 16

//プロトタイプ宣言
__global__ void VecAddKernel(float *, float *, float *);

/**
   n 次ベクトルの和を計算
 */
int main(){
  int i;

  //ホストメモリの確保
  float *A = (float *)malloc(sizeof(float) * VEC_SIZE);
  float *B = (float *)malloc(sizeof(float) * VEC_SIZE);
  float *C = (float *)malloc(sizeof(float) * VEC_SIZE);

  //ベクトルデータ格納
  for(i = 0; i < VEC_SIZE; i++){
    A[i] = 1;
    B[i] = i;
  }

  //デバイスメモリの確保
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **)&d_A, sizeof(float) * VEC_SIZE);
  cudaMalloc((void **)&d_B, sizeof(float) * VEC_SIZE);
  cudaMalloc((void **)&d_C, sizeof(float) * VEC_SIZE);

  //ホストからデバイスへデータ転送
```

```
  cudaMemcpy(d_A, A, sizeof(float) * VEC_SIZE, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, sizeof(float) * VEC_SIZE, cudaMemcpyHostToDevice);

  //カーネル関数の実行
  dim3 dimBlock(BLOCK_SIZE, 1);
  dim3 dimGrid(VEC_SIZE/BLOCK_SIZE, 1);
  VecAddKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

  //デバイスからホストへデータ転送
  cudaMemcpy(C, d_C, sizeof(float) * VEC_SIZE, cudaMemcpyDeviceToHost);

  //計算結果の検証
  printf("A:");
  for(i = 0; i < VEC_SIZE; i++){
    printf("%5.1f", A[i]);
  }
  printf("\nB:");
  for(i = 0; i < VEC_SIZE; i++){
    printf("%5.1f", B[i]);
  }
  printf("\nC:");
  for(i = 0; i < VEC_SIZE; i++){
    printf("%5.1f", C[i]);
  }
  printf("\n");

  //メモリ解放
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  free(A);
  free(B);
  free(C);
}

//ベクトルの和を計算 (GPU)
__global__ void VecAddKernel(float *d_A, float *d_B, float *d_C){
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  d_C[tid] = d_A[tid] + d_B[tid];
}
```

2.3 バイラテラルフィルタ

バイラテラルフィルタは，(a) 対象画素からの距離による重み，(b) 対象画素との画素地の差に応じた重みを用いた平滑化フィルタである．上記 (a) と (b) の重みはガウス分布に従う．バイラテラルフィルタを適用することで，画像のエッジ成分を残したままノイズを平滑化することができる．

画像 $f(i,j)$ が与えられたとき，サイズ $2w+1$ ，分散 σ_1, σ_2 のバイラテラルフィルタによって平滑化された画像 $g(i,j)$ は次式で与えられる．

$$
\begin{aligned}
g(i,j) &= \frac{\displaystyle\sum_{n=-w}^{w}\sum_{m=-w}^{w} f(i+m,j+n)s(i,j,m,n)}{\displaystyle\sum_{n=-w}^{w}\sum_{m=-w}^{w} s(i,j,m,n)} \qquad (1)\\
s(i,j,m,n) &= \exp\left(-\frac{m^2+n^2}{2\sigma_1^2}\right)\exp\left(-\frac{(f(i,j)-f(i+m,j+n))^2}{2\sigma_2^2}\right)
\end{aligned}
$$

2.4 SSD によるテンプレートマッチング

テンプレートと画像が与えられたとき，画像中におけるテンプレートの位置を検出する処理をテンプレートマッチングと呼ぶ．テンプレートを画像全体に対してラスタスキャン順に移動し，それぞれの位置における相違度を計算する．相違度の計算には，輝度値の差の二乗値の総和を類似度とする SSD (Sum of Squared Difference), 輝度値の差の絶対値の総和を類似度とする SAD (Sum of Absolute Difference), 正規化相互相関を用いる NCC (Normalized Cross-Correlation) および ZNCC (Zero-mean Normalized Cross-Correlation) といった方式がある [6].

画像 $I(i,j)$ ，$M \times N$ 画素のテンプレート $T(i,j)$ が与えられたとき，SSD (Sum of Squared Difference) による相違度 R_{SSD} は次式で与えられる．

$$R_{\mathrm{SSD}} = \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} \{I(i,j) - T(i,j)\}^2 \tag{2}$$

相違度 R_{SSD} は，画像中でテンプレートが完全に一致した位置で 0 となり，相違が大きいほど大きな値になる．

3　実験方法

本実験は，第 2 章で述べた原理に基づく．以下では，実験環境と実験方法について述べる．

実験環境

本実験の環境を表 3 へ示す．また，実験で用いる NVIDIA Tesla C2050 のデバイス情報を表 4 へ示す．

表 3: 実験環境

OS	Ubuntu 10.04 LTS (64bit)
CPU	Intel Xeon E5450 3.00 GHz (4 cores) x2 (16 threads)
Memory	16GB
GPU	NVIDIA Tesla C2050
VRAM	3GB
CPU 実装	C 言語 / gcc version 4.3.4
GPU 実装	C 言語 / NVIDIA Cuda compiler driver 4.0

表 4: NVIDIA Tesla C2050 のデバイス情報 (一部)

CUDA ドライババージョン	4.10
CUDA ランタイムバージョン	4.0
CUDA Compute Capability	2.0
CUDA コア数	448
GPU クロック周波数	1.15 GHz
グローバルメモリ	2,687 MB
メモリバス幅	384 bit
定数メモリ	65536 bytes
ブロックあたりの共有メモリ	49152 bytes
ブロックあたりのレジスタ数	32768
ウォープサイズ	32
ブロックあたりの最大スレッド数	1024
ブロックの各次元の最大サイズ	$1024 \times 1024 \times 64$
グリッドの各次元の最大サイズ	$65535 \times 65535 \times 65535$

実験 1 ベクトルの加算

表 2 のプログラムをコンパイル・実行し，その結果を確認する．

実験 2 N 次正方行列の積算

2 つの N 次正方行列の積を計算する C 言語プログラムを作成する．また，作成したプログラムを CUDA を用いて GPU へ実装する．これらの CPU 実装・GPU 実装したプログラムの実行時間を，行列の次数を $N = 32, 64, 128, \cdots, 1024$ と変化させて計測・比較する．さらに GPU 実装したプログラムについては，ブロックあたりのスレッド数を $1^2, 2^2, 4^2, \cdots, 32^2$ と変化させて計測する．

実験 3 バイラテラルフィルタの適用

式 (1) で定義されるバイラテラルフィルタを用いて，与えられた画像を平滑化する C 言語プログラムを作成する．また，作成したプログラムを CUDA を用いて GPU へ実装する．実際に 1024×768 画素の

photo.bmp (図 1) を用いて，正しくフィルタ処理が行われることを確認する．また，CPU 実装・GPU 実装したプログラムの実行時間を，入力画像サイズを 32×24, 64×48, 128×96, 256×192, 512×384, 1024×768 と変化させて計測・比較する．さらに GPU 実装したプログラムについては，ブロックあたりのスレッド数を $1^2, 2^2, 4^2, \cdots, 32^2$ と変化させて計測する．

図 1: photo.bmp

実験 4 SSD を用いたテンプレートマッチング

式 (2) の相違度を用いて，与えられた画像とテンプレートについて SSD を用いたテンプレートマッチングを行い，相違度マップを出力する C 言語プログラムを作成する．また，作成したプログラムを CUDA を用いて GPU へ実装する．実際に，テンプレート画像として 64×64 画素の template.bmp (図 2)，対象画像として 1024×768 画素の image.bmp (図 3) を用いて，正しくテンプレートマッチングが行われることを確認する．また，CPU 実装・GPU 実装したプログラムの実行時間を，対象画像サイズを 128×96, 256×192, 512×384, 1024×768 と変化させて計測・比較する．さらに GPU 実装したプログラムについては，ブロックあたりのスレッド数を $1^2, 2^2, 4^2, \cdots, 32^2$ と変化させて計測する．

図 2: template.bmp

図 3: image.bmp

4 実験結果

第 3 章で示した方法に従って実験を行った．本章では，各実験の結果をまとめる．

結果 1 ベクトルの加算

表 2 のプログラムでは，ベクトル A, B をそれぞれ VEC_SIZE $= 16$ 次元のベクトル $(1, 1, \cdots, 1)$, $(0, 1, \cdots, 15)$ と定義し，それらの和ベクトル C $=$ A $+$ B $= (1, 2, \cdots, 16)$ を計算する．また，それぞれのベクトルの要素を確認することで GPU 計算の正当性を評価する．

表 2 で示したプログラムを vecadd.cu とし，コンパイル・実行したときの結果を表 5 へ示す．出力されたベクトルの要素は左から要素番号 $i = 0, 1, \cdots, 15$ 結果より，出力されたベクトル A, B, C の値は理論値と一致し，加算処理が GPU 上で正しく行われていることが確認できる．

表 5: vecadd.cu のコンパイル・実行

```
fermi[yamao:515]$ nvcc vecadd.cu -o vecadd_cu
fermi[yamao:516]$ ./vecadd_cu
A:  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
B:  0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0 15.0
C:  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
fermi[yamao:517]$
```

結果 2 N 次正方行列の積算

作成したプログラム matprod.cu の外部仕様とソースコードをそれぞれ表 6, 7 へ示す．このプログラムでは，一様分布乱数を成分に持つ 2 つの N 次正方行列 A と B の積 C を CPU 側と GPU 側で計算する．CPU 側の計算と GPU 側の計算は，それぞれ matrixProduct() 関数と matrixProductKernel() 関数で行われる．グリッドの次元 dimGrid は (n/blockSize, n/blockSize, 1)，ブロックの次元 dimBlock は (blockSize, blockSize, 1) と定義した．これから，ブロック数は $(\text{n/blockSize})^2$, ブロックあたりのスレッド数は blockSize^2 となる．また，計算の実行時間は表 8 の util.c で定義した gettimeofday_sec() 関数を用いる．これは，過去のある時間からの経過時間を，秒数を表す倍精度実数で出力する関数である．ただし，ホスト側とデバイス側の処理は非同期であるため，カーネル関数の処理時間をホスト側で測定する際には，cudaThreadSynchronize() 関数を用いてカーネル関数の処理が終わるまで待機する必要がある．

次数 N および blockSize を，それぞれ $N = 32, 64, 128, 256, 512, 1024$，blockSize $= 1, 2, 4, 8, 16, 32$ と変化させたときの CPU 実装・GPU 実装プログラムの実行時間を表 9，10，図 4 へ示す．ただし，表中の実行時間は，試行回数 n_trial $= 3$ 回の平均値である．

結果から，CPU 実装の実行時間は，GPU 実装により 45.2% ($N = 32$, blockSize $= 1$) から 0.10% ($N = 1024$, blockSize $= 16$) まで短縮され，GPU による並列計算の効果が確認された．また CPU 実装・GPU 実装の双方で，次数 N の増加に伴い，対数グラフ上で線形に実行時間が増加した．GPU 実装に着目すると，ブロックの次元の増加に伴い，実行時間が短縮することが確認できる．ただし，ブロックの次元がある程度大きくなると，実行時間の短縮度は低下する．特に，ブロックあたりのスレッド数が 1024 (blockSize $= 32$) のときの実行時間は，ブロックあたりのスレッド数が 256 (blockSize $= 16$) のときの実行時間をわずかに上回った．

表 7: matprod.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include"util.c"
```

表 6: matplod.cu の外部仕様

プログラム	matprod.cu		
機能	N 次正方行列の積を計算し CPU 実装と GPU 実装それぞれの実行時間を測定する		
引数	n:	正方行列の次数 N	(default: 3)
	blockSize:	ブロックの次数	(default: 16)
	n_trial:	試行回数	(default: 3)
出力	CPU 実装と GPU 実装の実行時間を標準出力		

```
//プロトタイプ宣言
__global__ void matrixProductKernel(float *, float *, float *, int);
void matrixProduct(float *, float *, float *, int);

/**
   n 次正方行列の積を計算
   CPU 実装と GPU 実装それぞれの実行時間を測定

   引数 (デフォルト)
   [1]: 正方行列の次数 (32)
   [2]: ブロックの次数 (16)
   [3]: 各方式の試行回数 (3)
 */
int main(int argc, char **argv){
  int i;

  //パラメータ設定
  int n = 3;          //正方行列の次数
  int blockSize = 16; //ブロックの次数
  int n_trial = 3;    //試行回数
  switch(argc){
  case 4: sscanf(argv[3], "%d", &n_trial);
  case 3: sscanf(argv[2], "%d", &blockSize);
  case 2: sscanf(argv[1], "%d", &n);
  case 1: break;
  default:
    fprintf(stderr, "usage:\n[1]n (32)\n[2]block_size (16)\n[3]n_trial (3)\n");
    exit(1);
  }
  int size = n*n;

  //ホストメモリの確保
  float *A = (float *)malloc(sizeof(float) * size);
  float *B = (float *)malloc(sizeof(float) * size);
  float *C = (float *)malloc(sizeof(float) * size);

  //行列データ格納 (乱数)
  srand((unsigned) time(NULL));
  for(i = 0; i < n*n; i++){
    A[i] = rand();
    B[i] = rand();
  }

  //デバイスメモリの確保
  float *d_A, *d_B, *d_C;
  cudaMalloc((void **)&d_A, sizeof(float) * size);
  cudaMalloc((void **)&d_B, sizeof(float) * size);
  cudaMalloc((void **)&d_C, sizeof(float) * size);

  //ホストからデバイスへデータ転送
```

```
  cudaMemcpy(d_A, A, sizeof(float) * size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, sizeof(float) * size, cudaMemcpyHostToDevice);

  //CPU 実装の実行時間測定
  double tmp = 0.0;
  double c_tb, c_te;
  printf("(n: %d, bsize: %d, trial: %d)\n", n, blockSize, n_trial);
  for(i = 0; i < n_trial; i++){
    c_tb = gettimeofday_sec();
    matrixProduct(A, B, C, n);
    c_te = gettimeofday_sec();
    tmp += c_te - c_tb;
    printf(".");
  }
  printf("CPU_time = %.3le\n", tmp/n_trial);

  //GPU 実装の実行時間測定
  dim3 dimBlock(blockSize, blockSize);
  dim3 dimGrid(n/blockSize, n/blockSize);
  double g_tb, g_te;
  tmp = 0.0;
  for(i = 0; i < n_trial; i++){
    g_tb = gettimeofday_sec();
    matrixProductKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, n);
    cudaThreadSynchronize();
    g_te = gettimeofday_sec();
    tmp += g_te - g_tb;
    printf(".");
  }
  printf("GPU_time = %.3le\n", tmp/n_trial);

  //デバイスからホストへデータ転送
  cudaMemcpy(C, d_C, sizeof(float) * size, cudaMemcpyDeviceToHost);

  //メモリ解放
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  free(A);
  free(B);
  free(C);
}

//正方行列の積 (GPU)
__global__ void matrixProductKernel(float *d_A, float *d_B, float *d_C, int n){
  int i;

  int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  int tidy = blockIdx.y * blockDim.y + threadIdx.y;

  d_C[tidy*n+tidx] = 0.0;
  for(i = 0; i < n; i++){
      d_C[tidy*n+tidx] += d_A[tidy*n+i] * d_B[i*n+tidx];
  }

}

//正方行列の積 (CPU)
void matrixProduct(float *d_A, float *d_B, float *d_C, int n){
  int i, x, y, yn, ynx;

  for(y = 0; y < n; y++){
    yn = y*n;
```

```
    for(x = 0; x < n; x++){
      ynx = yn + x;
      d_C[ynx] = 0.0;
      for(i = 0; i < n; i++){
        d_C[ynx] += d_A[yn+i] * d_B[i*n+x];
      }
    }
  }

}
```

表 8: `util.c`

```
//時間を取得
double gettimeofday_sec(){
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (double)tv.tv_usec*1e-6;
}
```

表 9: 行列積算の実行時間 (CPU)

N	実行時間 [sec.]
32	2.134×10^{-4}
64	1.733×10^{-3}
128	1.546×10^{-2}
256	1.292×10^{-1}
512	1.076
1024	4.817×10

表 10: 行列積算の実行時間 (GPU) [sec.]

N	`blockSize` $= 1$	`blockSize` $= 2$	`blockSize` $= 4$	`blockSize` $= 8$	`blockSize` $= 16$	`blockSize` $= 32$
32	9.640×10^{-5}	6.763×10^{-5}	5.666×10^{-5}	5.635×10^{-5}	5.404×10^{-5}	5.770×10^{-5}
64	4.706×10^{-4}	2.063×10^{-4}	1.187×10^{-4}	7.033×10^{-5}	6.700×10^{-5}	7.272×10^{-5}
128	6.252×10^{-3}	1.715×10^{-3}	5.660×10^{-4}	2.077×10^{-4}	1.403×10^{-4}	1.590×10^{-4}
256	5.033×10^{-2}	1.345×10^{-2}	4.023×10^{-3}	1.143×10^{-3}	6.640×10^{-4}	6.293×10^{-4}
512	4.306×10^{-1}	1.198×10^{-1}	3.832×10^{-2}	1.144×10^{-2}	5.958×10^{-3}	6.177×10^{-3}
1024	4.099	1.097	3.541×10^{-1}	9.731×10^{-2}	4.853×10^{-2}	5.127×10^{-2}

結果 3 バイラテラルフィルタの適用

作成したプログラム `bilateral.cu` の外部仕様とソースコードをそれぞれ表 11，12 へ示す．このプログラムでは，入力された画像へサイズ $w =$ `filter_w`，分散 $\sigma_1, \sigma_2 =$ `var` のバイラテラルフィルタを適用し，処理後の画像を出力する．

CPU 側の計算と GPU 側の計算は，それぞれ `bilateralFilter()` 関数と `bilateralFilterKernel()` 関数で行われる．グリッドの次元 `dimGrid` は (`image_w`/`blockSize`, `image_h`/`blockSize`, 1) ，ブロックの次元 `dimBlock` は (`blockSize`, `blockSize`, 1) で定義する．これから，ブロック数は `image_w*image_h/blockSize`^{2}，ブロックあたりのスレッド数は `blockSize`^{2} となる．

`photo.bmp` (図 1) に対し，フィルタサイズ $w = 3$，分散 $\sigma_1 = \sigma_2 = 30$ のバイラテラルフィルタを $1 \sim 3$ 回適用した時の出力画像を図 5 へ示す．図より，処理後の画像はエッジを保ったまま平滑化されており，正

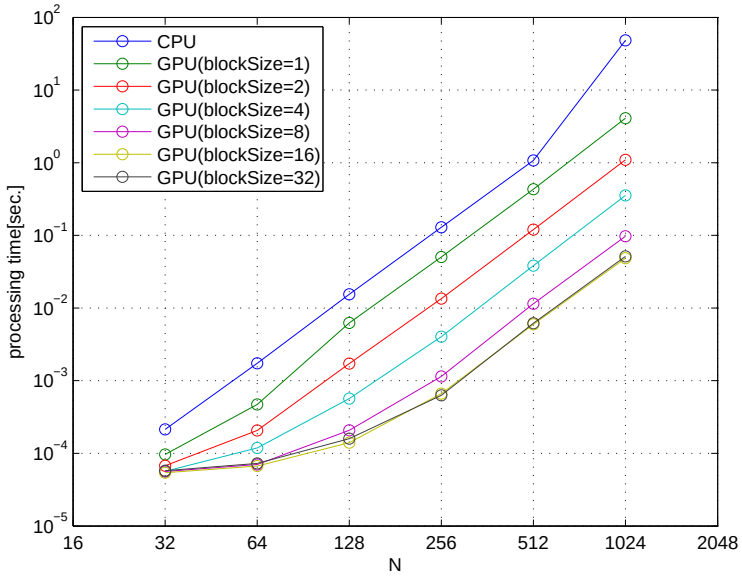


図 4: 行列積算の実行時間

しくフィルタ処理が行われていることが確認できる．

また，画像サイズおよび `blockSize` を，それぞれ 32×24, 64×48, 128×96, 256×192, 512×384, 1024×768，`blockSize` $= 1, 2, 4, 8, 16, 32$ と変化させたときの CPU 実装・GPU 実装プログラムの実行時間を表 13，14，図 6 へ示す．ただし，表中の実行時間は，試行回数 `n_trial` $= 3$ 回の平均値であり，フィルタサイズ $w = 3$，分散 $\sigma_1 = \sigma_2 = 30$ である．

結果から，CPU 実装の実行時間は，GPU 実装により 1.29% (入力画像サイズ 32×24, `blockSize` $= 1$) から 0.03% (入力画像サイズ 1024×768, `blockSize` $= 32$) まで短縮され，GPU による並列計算の効果が確認された．また CPU 実装・GPU 実装の双方で，次数 N の増加に伴い，対数グラフ上で線形に実行時間が増加した．GPU 実装に着目すると，ブロックの次元の増加に伴い，実行時間が短縮することが確認できる．ただし，ブロックの次元がある程度大きくなると，実行時間の短縮度は低下しており，ブロックあたりのスレッド数が 64 (`blockSize` $= 8$) 以上の範囲では，実行時間はほとんど等しい．特に，ブロックあたりのスレッド数が 1024 (`blockSize` $= 32$) のときの実行時間は，ブロックあたりのスレッド数が 256 (`blockSize` $= 16$) のときの実行時間をわずかに上回った．

表 11: `bilateral.cu` の外部仕様

プログラム	`bilateral.cu`		
機能	指定された画像にバイラテラルフィルタを適用し CPU 実装と GPU 実装それぞれの実行時間を測定する		
引数	`args[1]`:	入力ファイル名	(必須)
	`args[2]`:	出力ファイル名	(必須)
	`blockSize`:	ブロックの次元	(default: 16)
	`n_trial`:	試行回数	(default: 1)
	`filter_w`:	フィルタサイズ	(default: 3)
	`var`:	分散 σ_1, σ_2	(default: 30)
	`image_w`:	画像の幅	(default: 1024)
	`image_h`:	画像の高さ	(default: 768)
出力	フィルタ適用後の画像を出力 CPU 実装と GPU 実装の実行時間を標準出力		

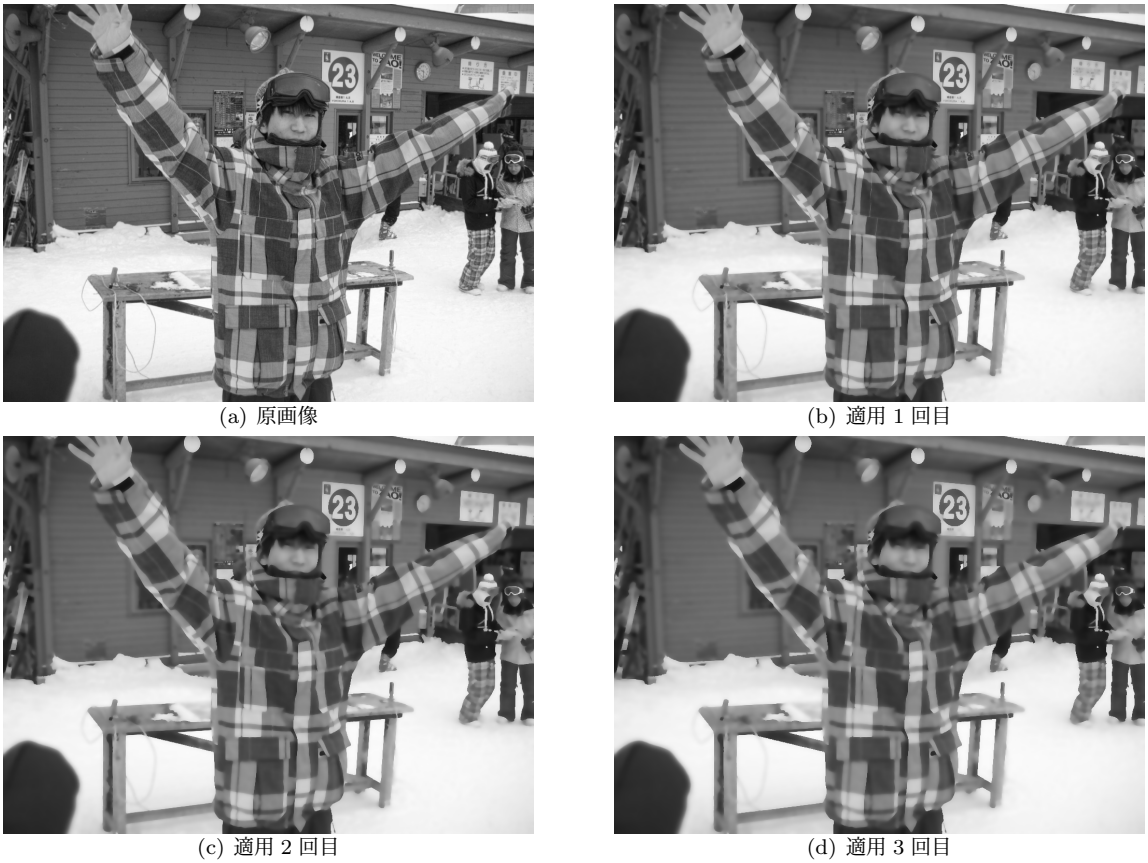
表 12: bilateral.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include"util.c"

//画像構造体
typedef struct image{
  float *d; //画素データ
  int w;    //幅
  int h;    //高さ
} image;

//プロトタイプ宣言
void bilateralFilter(image, image, int, float, float);
__global__ void bilateralFilterKernel(image, image, int, float, float);

/**
   バイラテラルフィルタの適用
   CPU 実装と GPU 実装それぞれの実行時間を測定

   引数 (デフォルト)
   [1]: 入力ファイル名 (*)
   [2]: 出力ファイル名 (*)
   [3]: ブロックの次元 (16)
   [4]: 各方式の試行回数 (1)
   [5]: フィルタサイズ (3)
   [6]: 分散 s1,s2(30)
   [7]: 画像幅 (1024)
   [8]: 画像高さ (768)
*/
int main(int argc, char **argv){
  int i;
  FILE *fp;
  image A, B;

  //パラメータ設定
  int n_trial = 1;    //試行回数
  int blockSize = 16; //ブロックの次数
  int filter_w = 3;   //フィルタサイズ
  int var = 30;       //分散
  int image_w = 1024; //画像の幅
  int image_h = 768;  //画像の高さ
  switch(argc){
  case 9: sscanf(argv[8], "%d", &image_h);
  case 8: sscanf(argv[7], "%d", &image_w);
  case 7: sscanf(argv[6], "%d", &var);
  case 6: sscanf(argv[5], "%d", &filter_w);
  case 5: sscanf(argv[4], "%d", &n_trial);
  case 4: sscanf(argv[3], "%d", &blockSize);
  case 3: break;
  default:
    fprintf(stderr, "usage:\n[1]inputfile (*)\n[2]outputfile \
(*)\n[3]blockSize (16)\n[4]n_trial (1)\n[5]filter_w (3)\n[6]var \
(30)\n[7]image_w (1024)\n[8]image_h (768)\n");
    exit(1);
  }
  int image_size = image_w * image_h;

  //ホストメモリの確保
  A.d = (float *)malloc(sizeof(float) * image_size);
```

```
  B.d = (float *)malloc(sizeof(float) * image_size);

  //画像データ格納
  if((fp = fopen(argv[1], "rb")) == NULL){
    fprintf(stderr, "can't open file: %s\n", argv[1]);
    exit(2);
  }
  for(i = 0; i < image_size; i++){
    fscanf(fp, "%f", &A.d[i]);
  }
  fclose(fp);
  A.w = image_w;  A.h = image_h;
  B.w = image_w;  B.h = image_h;

  //デバイスメモリの確保
  image d_A, d_B;
  cudaMalloc((void **)&(d_A.d), sizeof(float) * image_size);
  cudaMalloc((void **)&(d_B.d), sizeof(float) * image_size);
  d_A.w = A.w;  d_A.h = A.h;
  d_B.w = B.w;  d_B.h = B.h;

  //ホストからデバイスへデータ転送
  cudaMemcpy(d_A.d, A.d, sizeof(float) * image_size, cudaMemcpyHostToDevice);

  //CPU 実装の実行時間測定
  double tmp = 0;
  double c_tb, c_te;
  printf("(isize: %dx%d, w: %d, var: %d, bsize: %d trial: %d)\n",
A.w, A.h, filter_w, var, blockSize, n_trial);
  for(i = 0; i < n_trial; i++){
    c_tb = gettimeofday_sec();
    bilateralFilter(B, A, filter_w, var, var);
    c_te = gettimeofday_sec();
    tmp += c_te - c_tb;
    printf(".");
  }
  printf("CPU_time = %.3le\n", tmp/n_trial);

  //GPU 実装の実行時間測定
  dim3 dimBlock(blockSize, blockSize);
  dim3 dimGrid(image_w/blockSize, image_h/blockSize);
  double g_tb, g_te;
  tmp = 0.0;
  for(i = 0; i < n_trial; i++){
    g_tb = gettimeofday_sec();
    bilateralFilterKernel<<<dimGrid, dimBlock>>>(d_B, d_A, filter_w, var, var);
    cudaThreadSynchronize();
    g_te = gettimeofday_sec();
    tmp += g_te - g_tb;
    printf(".");
  }
  printf("GPU_time = %.3le\n", tmp/n_trial);

  //デバイスからホストへデータ転送
  cudaMemcpy(B.d, d_B.d, sizeof(float) * image_size, cudaMemcpyDeviceToHost);

  //画像データ出力
  if((fp = fopen(argv[2], "wb")) == NULL){
    fprintf(stderr, "can't open file: %s\n", argv[2]);
    exit(2);
  }
  for(i = 0; i < image_size; i++){
    fprintf(fp, "%f\n", B.d[i]);
```

```
  }
  fclose(fp);

  //メモリ解放
  cudaFree(d_A.d);
  cudaFree(d_B.d);
  free(A.d);
  free(B.d);
}

//バイラテラルフィルタの適用 (GPU)
__global__ void bilateralFilterKernel(image out, image in, int w, float s1, float s2){
  int m, n;
  float f_mn, dist, s;
  int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  int tidy = blockIdx.y * blockDim.y + threadIdx.y;
  float f = in.d[tidy*in.w+tidx];
  float denom = 0.0;
  float numer = 0.0;

  for(m = -w; m <= w; m++){
    for(n = -w; n <= w; n++){
      if((tidx+m) >= 0
         && (tidx+m) < in.w
         && (tidy+n) >= 0
         && (tidy+n) < in.h){
        f_mn = in.d[(tidy+n)*in.w+(tidx+m)];
      } else {
        //画像領域外なら注目画素
        f_mn = in.d[tidy*in.w+tidx];
      }
      dist = f - f_mn;
      s = exp(-((m*m + n*n)/(2*s1*s1)))
        * exp(-((dist*dist)/(2*s2*s2)));
      denom += f_mn*s;
      numer += s;
    }
  }
  out.d[tidy*out.w+tidx] = denom / numer;
}

//バイラテラルフィルタの適用 (CPU)
void bilateralFilter(image out, image in, int w, float s1, float s2){
  int x, y, m, n;
  float f, f_mn, dist, denom, numer, s;

  for(x = 0; x < in.w; x++){
    for(y = 0; y < in.h; y++){
      f = in.d[y*in.w+x];
      denom = 0.0;
      numer = 0.0;
      for(m = -w; m <= w; m++){
        for(n = -w; n <= w; n++){
          if((x+m) >= 0
             && (x+m) < in.w
             && (y+n) >= 0
             && (y+n) < in.h){
            f_mn = in.d[(y+n)*in.w+(x+m)];
          } else {
          //画像領域外なら注目画素
            f_mn = in.d[y*in.w+x];
          }
          dist = f - f_mn;
```

```
          s = exp(-((m*m + n*n)/(2*s1*s1)))
            * exp(-((dist*dist)/(2*s2*s2)));
          denom += f_mn*s;
          numer += s;
        }
      }
      out.d[y*out.w+x] = denom / numer;
    }
  }
}
```



(a) 原画像　(b) 適用 1 回目

(c) 適用 2 回目　(d) 適用 3 回目

図 5: バイラテラルフィルタの適用: $w = 3, \sigma_1 = \sigma_2 = 30$

表 13: バイラテラルフィルタの実行時間 (CPU)

画像サイズ	実行時間 [sec.]
32×24	2.167×10^{-2}
64×48	8.681×10^{-2}
128×96	3.464×10^{-1}
256×192	1.389
512×384	5.547
1024×768	2.246×10

結果 4 SSD を用いたテンプレートマッチング

作成したプログラム `template.cu` の外部仕様とソースコードをそれぞれ表 15, 16 へ示す．このプログラムでは，入力された画像とテンプレートを用いて SSD によるテンプレートマッチングを行い，入力画像中

表 14: バイラテラルフィルタの実行時間 (GPU) [sec.]

画像サイズ	`blockSize` $= 1$	`blockSize` $= 2$	`blockSize` $= 4$	`blockSize` $= 8$	`blockSize` $= 16$	`blockSize` $= 32$
32×24	2.800×10^{-4}	1.127×10^{-4}	8.003×10^{-5}	7.963×10^{-5}	7.963×10^{-5}	4.729×10^{-5}
64×48	9.687×10^{-4}	2.933×10^{-4}	1.203×10^{-4}	8.663×10^{-5}	9.934×10^{-5}	1.530×10^{-4}
128×96	3.724×10^{-3}	1.012×10^{-3}	3.160×10^{-4}	1.503×10^{-4}	1.507×10^{-4}	1.543×10^{-4}
256×192	1.476×10^{-2}	3.881×10^{-3}	1.115×10^{-3}	4.397×10^{-4}	4.160×10^{-4}	4.797×10^{-4}
512×384	5.907×10^{-2}	1.570×10^{-2}	4.555×10^{-3}	1.648×10^{-3}	1.555×10^{-3}	1.636×10^{-3}
1024×768	2.397×10^{-1}	6.567×10^{-2}	1.944×10^{-2}	6.386×10^{-3}	5.917×10^{-3}	5.951×10^{-3}

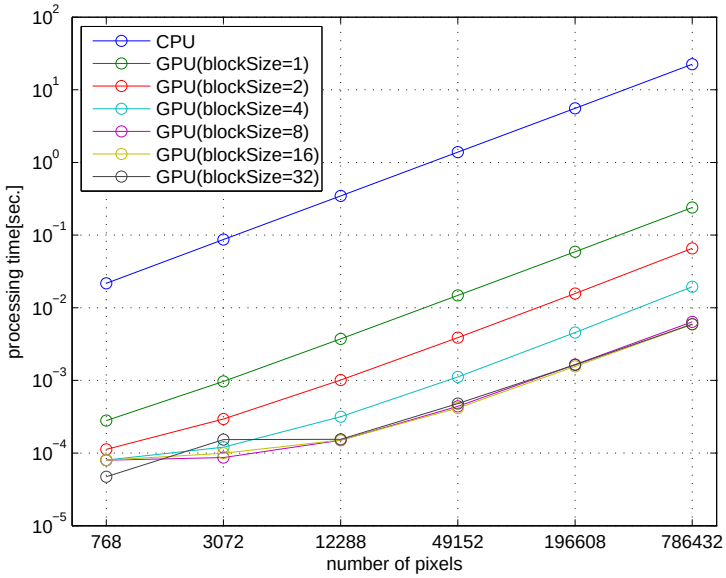


図 6: バイラテラルフィルタの実行時間

のテンプレートの位置を標準出力する．また，相違度マップを result.dat として出力する．CPU 側の計算と GPU 側の計算は，それぞれ templateMatching() 関数と templateMatchingKernel() 関数で行われる．グリッドの次元 dimGrid は $((\texttt{image\_w} - \texttt{templ\_w})/\texttt{blockSize}, (\texttt{image\_h} - \texttt{templ\_h})/\texttt{blockSize}, 1)$ ，ブロックの次元 dimBlock は $(\texttt{blockSize}, \texttt{blockSize}, 1)$ で定義する．これから，ブロックあたりのスレッド数は $\texttt{blockSize}^2$ となる．

入力画像として image.bmp (図 3) ，テンプレート画像として template.bmp (図 2) を指定した時の，GPU 実装プログラムから出力された相違度マップを図 7(a) へ示す．このときのテンプレート位置は $(582, 450)$ と出力された．この結果は，図 7(b) の赤枠の部分が検出されたことを示しうになり，正しくテンプレートマッチングが行われていることが確認できる．

また，iamge.bmp のサイズおよび blockSize を，それぞれ 32×24, 64×48, 128×96, 256×192, 512×384, 1024×768，$\texttt{blockSize} = 1, 2, 4, 8, 16, 32$ と変化させたときの CPU 実装・GPU 実装プログラムの実行時間を表 13，14，図 6 へ示す．ただし，表中の実行時間は，試行回数 $\texttt{n\_trial} = 3$ 回の平均値である．

結果から，CPU 実装の実行時間は，GPU 実装により 23.9% (対象画像サイズ 128×96, $\texttt{blockSize} = 1$) から 0.44% (対象画像サイズ 1024×768, $\texttt{blockSize} = 32$) まで短縮され，GPU による並列計算の効果が確認された．また CPU 実装・GPU 実装の双方で，次数 N の増加に伴い実行時間が増加した．GPU 実装に着目すると，ブロックの次元の増加に伴い，実行時間が短縮することが確認できる．ただし，ブロックの次元がある程度大きくなると実行時間の短縮度は低下し，ブロックあたりのスレッド数が 256 ($\texttt{blockSize} = 16$) 以上の範囲では，実行時間はほとんど等しい．ブロックあたりのスレッド数が 1024 ($\texttt{blockSize} = 32$) のときの実行時間は，ブロックあたりのスレッド数が 256 ($\texttt{blockSize} = 16$) のときの実行時間をわずかに上回った．

表 15: template.cu の外部仕様

プログラム	template.cu		
機能	指定された画像とテンプレートを用いて SSD によるテンプレートマッチングを行い CPU 実装と GPU 実装それぞれの実行時間を測定する		
引数	args[1]:	入力ファイル名	(必須)
	args[2]:	テンプレートファイル名	(必須)
	blockSize:	ブロックの次元	(default: 16)
	n_trial:	試行回数	(default: 1)
	image_w:	画像の幅	(default: 1024)
	image_h:	画像の高さ	(default: 768)
	templ_w:	テンプレートの幅	(default: 64)
	templ_h:	テンプレートの高さ	(default: 64)
出力	相違度マップを result.dat へ出力 入力画像中のテンプレートの位置を標準出力 CPU 実装と GPU 実装の実行時間を標準出力		

表 16: template.cu

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include"util.c"

//画像構造体
typedef struct image{
  float *d; //画素データ
  int w;    //幅
  int h;    //高さ
} image;

//プロトタイプ宣言
```

```
void templateMatching(image, image, image);
__global__ void templateMatchingKernel(image, image, image);

/**
   SSD によるテンプレートマッチング
   CPU 実装と GPU 実装それぞれの実行時間を測定

   引数 (デフォルト)
   [1]: 入力ファイル名 (*)
   [2]: テンプレートファイル名 (*)
   [3]: ブロックの次元 (16)
   [4]: 各方式の試行回数 (1)
   [5]: 画像幅 (1024)
   [6]: 画像高さ (768)
   [7]: テンプレート幅 (64)
   [8]: テンプレート高さ (64)
*/
int main(int argc, char **argv){
  int i;
  FILE *fp;

  //パラメータ設定
  int n_trial = 1;    //試行回数
  int blockSize = 16; //ブロックの次数
  int image_w = 1024; //画像の幅
  int image_h = 768;  //画像の高さ
  int templ_w = 64;   //テンプレートの幅
  int templ_h = 64;   //テンプレートの高さ
  switch(argc){
  case 9: sscanf(argv[8], "%d", &templ_h);
  case 8: sscanf(argv[7], "%d", &templ_w);
  case 7: sscanf(argv[6], "%d", &image_h);
  case 6: sscanf(argv[5], "%d", &image_w);
  case 5: sscanf(argv[4], "%d", &n_trial);
  case 4: sscanf(argv[3], "%d", &blockSize);
  case 3: break;
  default:
    fprintf(stderr, "usage:\n[1]inputfile (*)\n[2]templetefile \
(*)\n[3]blockSize (16)\n[4]n_trial (1)\n[5]image_w \
(1024)\n[6]image_h (768)\n[7]templete_w (64)\n[8]templete_h (64)\n");
    exit(1);
  }
  int image_size = image_w * image_h;
  int templ_size = templ_w * templ_h;
  int map_size = (image_w-templ_w) * (image_h-templ_h);

  //ホストメモリの確保
  image input, templ, ssd;
  input.d = (float *)malloc(sizeof(float) * image_size);
  templ.d = (float *)malloc(sizeof(float) * templ_size);
  ssd.d = (float *)malloc(sizeof(float) * map_size);

  //画像データ格納
  if((fp = fopen(argv[1], "rb")) == NULL){
    fprintf(stderr, "can't open file: %s\n", argv[1]);
    exit(2);
  }
  for(i = 0; i < image_size; i++){
    fscanf(fp, "%f", &input.d[i]);
  }
  fclose(fp);
  input.w = image_w;  input.h = image_h;
  ssd.w = image_w-templ_w;  ssd.h = image_h-templ_h;
```

```
  //テンプレートデータ格納
  if((fp = fopen(argv[2], "rb")) == NULL){
    fprintf(stderr, "can't open file: %s\n", argv[2]);
    exit(2);
  }
  for(i = 0; i < templ_size; i++){
    fscanf(fp, "%f", &templ.d[i]);
  }
  fclose(fp);
  templ.w = templ_w;  templ.h = templ_h;

  //デバイスメモリの確保
  image d_input, d_templ, d_ssd;
  cudaMalloc((void **)&(d_input.d), sizeof(float) * image_size);
  cudaMalloc((void **)&(d_templ.d), sizeof(float) * templ_size);
  cudaMalloc((void **)&(d_ssd.d), sizeof(float) * map_size);
  d_input.w = image_w;        d_input.h = image_h;
  d_ssd.w = image_w-templ_w;  d_ssd.h = image_h-templ_h;
  d_templ.w = templ_w;        d_templ.h = templ_h;

  //ホストからデバイスへデータ転送
  cudaMemcpy(d_input.d, input.d, sizeof(float) * image_size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_templ.d, templ.d, sizeof(float) * templ_size, cudaMemcpyHostToDevice);

  //CPU 実装の実行時間測定
  double tmp = 0;
  double c_tb, c_te;
  printf("(isize: %dx%d, bsize: %d trial: %d)\n",
 input.w, input.h, blockSize, n_trial);
  for(i = 0; i < n_trial; i++){
    c_tb = gettimeofday_sec();
    templateMatching(ssd, input, templ);
    c_te = gettimeofday_sec();
    tmp += c_te - c_tb;
    printf(".");
  }
  printf("CPU_time = %.3le\n", tmp/n_trial);

  //GPU 実装の実行時間測定
  dim3 dimBlock(blockSize, blockSize);
  dim3 dimGrid((image_w-templ.w)/blockSize, (image_h-templ.h)/blockSize);
  double g_tb, g_te;
  tmp = 0.0;
  for(i = 0; i < n_trial; i++){
    g_tb = gettimeofday_sec();
    templateMatchingKernel<<<dimGrid, dimBlock>>>(d_ssd, d_input, d_templ);
    cudaThreadSynchronize();
    g_te = gettimeofday_sec();
    tmp += g_te - g_tb;
    printf(".");
  }
  printf("GPU_time = %.3le\n", tmp/n_trial);

  //デバイスからホストへデータ転送
  cudaMemcpy(ssd.d, d_ssd.d, sizeof(float) * map_size, cudaMemcpyDeviceToHost);

  //画像データ出力
  if((fp = fopen("result.dat", "wb")) == NULL){
    fprintf(stderr, "can't open file: %s\n", "result.dat");
    exit(2);
  }
  for(i = 0; i < map_size; i++){
```

```
    fprintf(fp, "%f\n", ssd.d[i]);
  }
  fclose(fp);

  //SSD 最小値出力
  int min = 0;
  for(i = 0; i < map_size; i++){
    if(ssd.d[min] > ssd.d[i]){
      min = i;
    }
  }
  printf("min(R_SSD): (%d,%d)\n", min%ssd.w, min/ssd.w);

  //メモリ開放
  cudaFree(d_input.d);
  cudaFree(d_templ.d);
  cudaFree(d_ssd.d);
  free(input.d);
  free(templ.d);
  free(ssd.d);
}

//テンプレートマッチング (GPU)
__global__ void templateMatchingKernel(image out, image in, image templ){
  int m, n;
  float f_I, dist;

  int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  int tidy = blockIdx.y * blockDim.y + threadIdx.y;

  out.d[(tidy*out.w+tidx)] = 1.0;

  for(m = 0; m < templ.w; m++){
    for(n = 0; n < templ.h; n++){
      if((tidx+m) < in.w && (tidy+n) < in.h){
        f_I = in.d[(tidy+n)*in.w+(tidx+m)];
      }
      dist = f_I - templ.d[n*templ.w+m];
      out.d[tidy*out.w+tidx] += dist*dist;
    }
  }

}

//テンプレートマッチング (CPU)
void templateMatching(image out, image in, image templ){
  int x, y, m, n;
  float f_I, dist;

  for(y = 0; y < in.h-templ.h; y++){
    for(x = 0; x < in.w-templ.w; x++){
      out.d[y*out.w+x] = 0.0;
      for(m = 0; m < templ.w; m++){
       for(n = 0; n < templ.h; n++){
         if((x+m) < in.w && (y+n) < in.h){
           f_I = in.d[(y+n)*in.w+(x+m)];
         }
         dist = f_I - templ.d[n*templ.w+m];
         out.d[y*out.w+x] += dist*dist;
       }
      }
    }
  }
```

```
}
```

(a) 相違度マップ

(b) テンプレートマッチング結果

図 7: テンプレートマッチング結果: テンプレート `template.bmp`, 対象画像 `image.bmp`

表 17: テンプレートマッチングの実行時間 (CPU)

画像サイズ	実行時間 [sec.]
128×96	1.085×10^{-1}
256×192	1.369
512×384	8.199
1024×768	3.861×10

表 18: テンプレートマッチングの実行時間 (GPU) [sec.]

画像サイズ	`blockSize` $= 1$	`blockSize` $= 2$	`blockSize` $= 4$	`blockSize` $= 8$	`blockSize` $= 16$	`blockSize` $= 32$
128×96	2.592×10^{-2}	8.320×10^{-3}	4.664×10^{-3}	2.565×10^{-3}	1.728×10^{-3}	2.677×10^{-3}
256×192	3.635×10^{-1}	1.028×10^{-1}	3.403×10^{-2}	1.041×10^{-2}	6.216×10^{-3}	5.337×10^{-3}
512×384	2.166	6.050×10^{-1}	1.962×10^{-1}	5.339×10^{-2}	2.694×10^{-2}	2.676×10^{-2}
1024×768	1.055×10	2.901	9.316×10^{-1}	2.544×10^{-1}	1.352×10^{-1}	1.261×10^{-1}

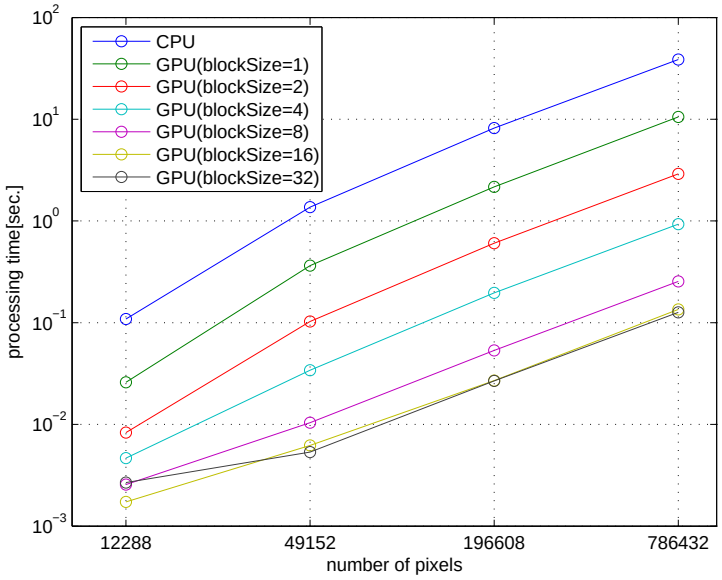


図 8: テンプレートマッチングの実行時間

5　考察

本章では，第 4 章の実験結果をもとに各項目の考察を行う．

5.1　Amdahl の法則

Amdahl の法則とは，“ある改善を行った結果の性能向上の度合いは，改善した機能がどの程度使用されるかによって制約される” という法則である．改善の影響を受ける実行時間を t_p ，改善度を N とすると，アムダールの法則より，改善後の実行時間 t' は次式で与えられる．

$$t' = \frac{t_p}{N} + (1 - t_p) \tag{3}$$

また，元の実行時間に対する，改善の影響を受ける実行時間の割合 P を考えると，式 (3) から，速度向上比 $S(N)$ を次のように表せる．

$$S(N) = \frac{1}{(1 - P) + \dfrac{P}{N}} \tag{4}$$

この議論は，プログラムの並列化による性能向上性の考察に適用することができる．つまり，プログラム全体における並列化可能部分の割合を P ，プロセッサの個数を N と見なすと，並列化による速度向上比 $S(N)$ を推定することができる．式 (4) からわかるように，プロセッサの個数 N あるいは並列化可能部分 P を大きくすることで速度向上比を高めることができる．ただし，P 一定の場合を考えると，プロセッサの個数 N を限りなく多くしても，速度向上比が $1/(1 - P)$ を超えることはなく，速度向上比 $S(N)$ に上限があることがわかる，並列化可能部分の割合が $P = 0.5, 0.75, 0.9, 0.95$ の場合の速度向上比を図 9 へ示す．例えば，プログラムの並列化可能部分の割合が 0.95 であった場合，速度向上比が 20 を超えることはない．

以上より，プロセッサの個数 N を増やすよりも，並列可能部分 P を大きくしたほうが速度向上比の改善を期待できる．見方を変えると，プログラムの並列化が有効であるのは，プロセッサ数が少ない場合か，並列化可能部分が大きい場合であると言える．

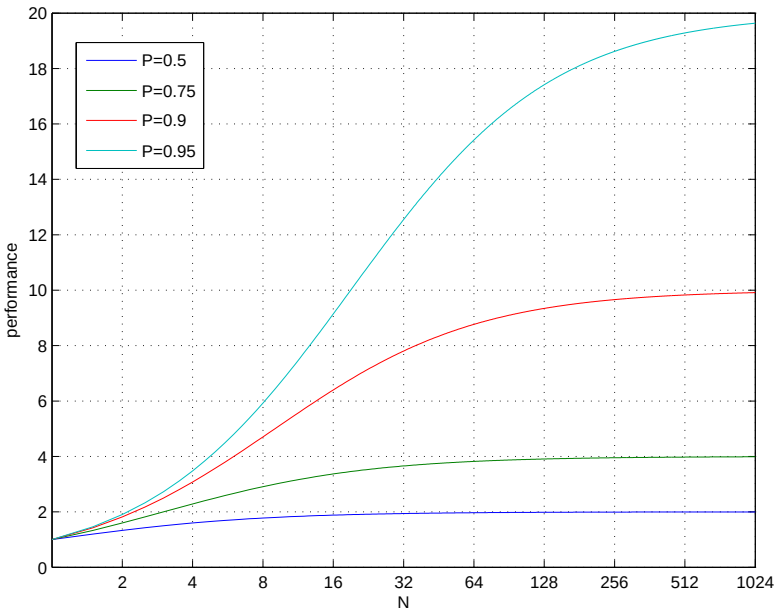
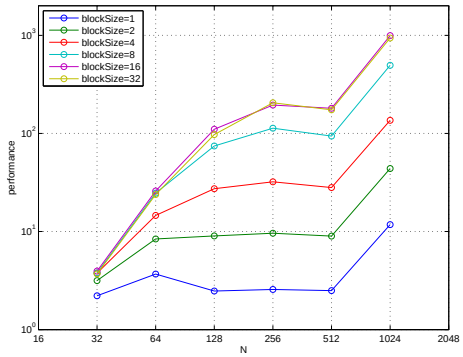


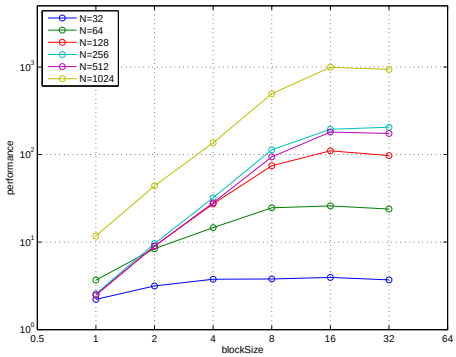
図 9: Amdahl の法則

5.2 GPU を用いた並列化による速度向上比

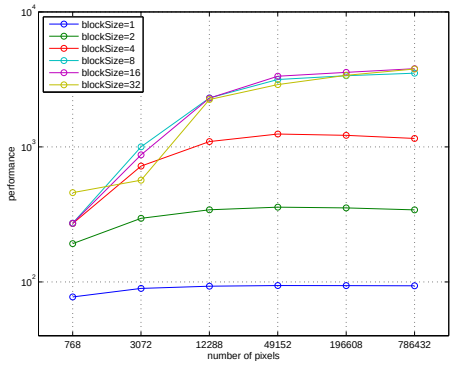
各実験で得られた実行時間から，GPU を用いた並列化による速度向上比を考察する．図 10 へ (a), (b) N 次正方行列の積算プログラム `matprod.cu`，(c), (d) バイラテラルフィルタ適用プログラム `bilateral.cu`，(e), (f) テンプレートマッチングプログラム `template.cu` における，GPU 実装による速度向上比を示す．ただし，(d) および (f) の凡例はそれぞれ入力画像のサイズ，対象画像のサイズである．グラフを観察するとわかるように，プログラムを並列化することで性能向上を図ることが可能であるが，その程度は，扱うデータのサイズと割り当てるブロックの次元に依存する．以降では，速度向上比とデータサイズ・ブロックの次元の関係について考える．



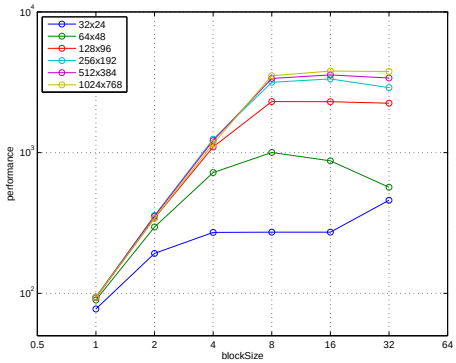
(a) `matprod.cu` (対データサイズ)



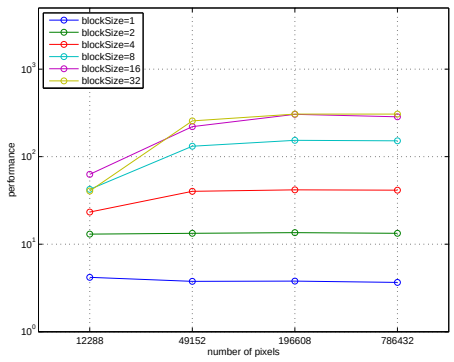
(b) `matprod.cu` (対 `blockSize`)



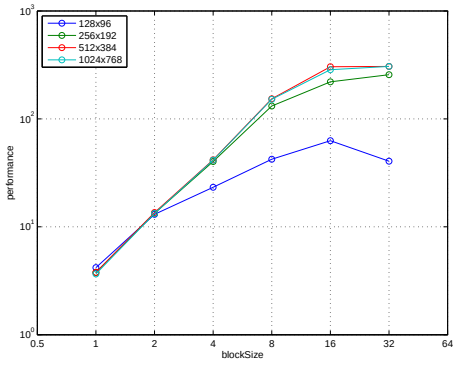
(c) `bilateral.cu` (対データサイズ)



(d) `bilateral.cu` (対 `blockSize`，凡例は入力画像サイズ)



(e) `template.cu` (対データサイズ)



(f) `template.cu` (対 `blockSize`，凡例は対象画像サイズ)

図 10: 各実験における速度向上比

データサイズと速度向上比

扱うデータのサイズと速度向上比の関係を考察する．それぞれの実験において，CPU による逐次的な計算量と GPU による並列化後の計算量から理論的な速度向上比を求めると表 19 のようになる．ただし，バイラテラルフィルタプログラムについては，サイズ $W \times H$ の画像に対するサイズ $w \times w$ のフィルタを考え，テンプレートマッチングについては，サイズ $W_i \times H_i$ の画像とサイズ $W_t \times H_t$ のテンプレートを用いたマッチングを考える．また，GPU 実装の計算量は，二次元データの走査に要するループを取り払うことによる計算量の減少のみを考えており，DRAM へのアクセスコストについては計算量のオーダに影響を与えない程度であると仮定している．

表 19 より，理論上ではデータサイズの増大に伴い速度向上比が高くなることがわかる．この予想を実際の速度向上比 (図 10(a), (c), (e)) と比較すると，データサイズが小さいうちは速度向上比の上昇のオーダは予想とほぼ一致するが，データサイズが増大するにつれて速度向上比は頭打ちになり，理論値と外れることがわかる．これは，データサイズの増加により，GPU のメモリアクセスにかかるコストが増大するためである．そのため，データサイズが大きいようなケースでは，グローバルメモリの他に，より高速なアクセスが可能な共有メモリやテクスチャメモリを利用することで，実行時間を改善できる見込みがある．例えばテンプレートマッチングプログラム `template.cu` (表 16) の場合，テンプレート画像へのアクセス範囲は全スレッドで共通であるため，テンプレート画像を共有メモリへ格納することで速度の向上を期待できる．なお，N 次正方行列の積算プログラム `matprod.cu` では，$N = 1024$ で大きく速度向上比が上昇しているが，これは GPU 実装による速度向上に起因するのではなく，図 4 を見てもわかるように，CPU 実装の実行時間が大きく増大したためであると考えられる．

表 19: 各実験における速度向上比 (理論値)

プログラム	CPU 実装	GPU 実装	速度向上比
N 次正方行列の積算	$O(N^3)$	$O(N)$	$O(N^2)$
バイラテラルフィルタの適用	$O(WH \cdot w^2)$	$O(w^2)$	$O(WH)$
テンプレートマッチング	$O(W_iH_i \cdot W_tH_t)$	$O(W_tH_t)$	$O(W_iH_i)$

ブロックの次元と速度向上比

ブロックの次元と速度向上比の関係を見てみると，どの実験においても，ある程度の `blockSize` までは対数グラフ上で比例の関係にあるが，それ以降では速度向上比は改善されないことがわかる．

`blockSize` が小さいと速度向上比も低い理由は，ウォープに生じる空きスレッド領域が GPU の性能をうまく引き出せていないためである．例えば `blockSize` $= 1$ のとき，ブロックあたりのスレッド数は 1 となるため，ウォープが 32 スレッドでアクティブに動作する場合に比べるとパフォーマンスが大きく低下する．

`blockSize` に対する速度向上比の非線形な上昇は，理論的な計算が難しい．そこで，ブロックの次元と GPU のパフォーマンスの関係をより詳しく調べるために，CUDA SDK より提供されている CUDA Occupancy Calculator を用いて SM あたりの占有状況を計算する．まず，各プログラムのカーネル関数で使用されるレジスタ数およびメモリを，`-Xptxas=-v` オプションを付加した `nvcc` コマンドより調べる (表 20)[4]．この情報をもとに CUDA Occupancy Calculator を用いると，`matprod.cu`, `bilateral.cu`, `template.cu` の SM の占有状況はそれぞれ表 21, 22, 23 のように計算される．なお，今回の場合，3 つの表すべてが同じ結果となったが，一般にカーネル関数あたりのレジスタ数・メモリは異なり，SM の占有状況はプログラムに依存することに注意する．

この表から，どのプログラムも `blockSize` $= 16$ のときに SM 占有率が 100%，アクティブブロック数が 6 となることがわかる．この結果は，SM が処理できる最大限のスレッド数を使うことが可能であり，かつブロックのメモリアクセスにおけるレイテンシを隠蔽できることを示しており，GPU の性能を高く引き出せていると言える．図 10(b), (d), (f) をみると，この傾向がグラフにも現れていることが確認できる．

一方，どのプログラムも blockSize = 32 となると，SM 占有率が 67%，アクティブブロック数が 1 となる．この結果は，SM が処理可能なスレッド数に余裕があるうえ，ブロックのメモリアクセスにおけるレイテンシを隠蔽する能力が不足していることを示しており，blockSize = 16 の場合に比べると，GPU の性能をうまく引き出せていないと言える．

以上の考察からわかるように，ブロックあたりのスレッド数を増やしても，期待するような性能が得られない場合がある．GPU の処理性能をうまく引き出すには，各プログラムにおいて適切なグリッド・ブロックの設定を行う必要がある．

表 20: 各プログラムのカーネル関数あたりのレジスタ数・メモリ

プログラム	matprod.cu	bilateral.cu	template.cu
レジスタ数	9	18	14
共有メモリ [bytes]	44	60	64
定数メモリ [bytes]	4	32	8

表 21: matprod.cu の SM 占有率: レジスタ数 9, 共有メモリ 2092 bytes

blockSize	1	2	4	8	16	32
SM あたりのアクティブスレッド数	256	256	256	512	1536	1024
SM あたりのアクティブウォープ数	8	8	8	16	48	32
SM あたりのアクティブブロック数	8	8	8	8	6	1
SM 占有率 [%]	17	17	17	33	100	67

表 22: bilateral.cu の SM 占有率: レジスタ数 18, 共有メモリ 2108 bytes

blockSize	1	2	4	8	16	32
SM あたりのアクティブスレッド数	256	256	256	512	1536	1024
SM あたりのアクティブウォープ数	8	8	8	16	48	32
SM あたりのアクティブブロック数	8	8	8	8	6	1
SM 占有率 [%]	17	17	17	33	100	67

表 23: template.cu の SM 占有率: レジスタ数 14, 共有メモリ 2112 bytes

blockSize	1	2	4	8	16	32
SM あたりのアクティブスレッド数	512	256	256	512	1536	1024
SM あたりのアクティブウォープ数	8	8	8	16	48	32
SM あたりのアクティブブロック数	8	8	8	8	6	1
SM 占有率 [%]	17	17	17	33	100	67

5.3 GPU と並列ハードウェア・並列ソフトウェア

並列計算を実現するためのハードウェア (並列ハードウェア) およびソフトウェア (並列ソフトウェア) について調査する．また，それらの特徴について GPU と比較する．

並列ハードウェア

並列ハードウェアは (i) すべてのプロセッサが単一の物理アドレス空間を共有する方式と，(ii) 各プロセッサが独自の物理アドレス空間を持つ方式に分類できる [3].

(i) は，共有記憶形マルチプロセッサ (Shered Memory Multiprocessor: SMP) と呼ばれる．この方式では，各プロセッサは主記憶内の任意の場所へアクセス可能で，プロセッサ間では共有変数を通じて情報が交換される．SMP には，主記憶へのアクセス時間の観点から 2 通りのタイプが存在する．1 つは，どのプロセッサがどの語を要求しても，主記憶へのアクセス時間がほぼ同じであるタイプで，均等メモリ・アクセス (Uniform Memory Access: UMA) 型マルチプロセッサと呼ぶ．もう 1 つは，プロセッサが要求する語によって主記憶へのアクセス時間が異なるタイプで，非均等メモリ・アクセス (Nonuniform Memory Access: NUMA) 型マルチプロセッサと呼ぶ．

一方 (ii) の場合，マルチプロセッサ同士は明示的なメッセージ交換 (message passing) を通じて情報を交換する．メモリアクセスは NUMA である．この方式の一般的な形態としては，クラスタ (cluster) やグリッド・コンピューティング (grid computing) がある．低コスト・高アベイラビリティなどの理由で， World Wide Web の主要なサービスはこの技術に立脚している．

また，もう 1 つの並列ハードウェアの分類法として，命令流の数とデータ流の数に基づくものがある．この分類では，ユニプロセッサは単一命令流・単一データ流 (Single-Instruction stream, Single-Data stream: SISD) の区分に，またマルチプロセッサは複数命令流・複数データ流 (Multiple-Instruction stream, Multiple-Data stream: MIMD) の区分に該当する．一般的な MIMD プログラミングモデルは，単一のプログラムがすべてのプロセッサにまたがって実行される単一プログラム複数データ (Single-Program Multiple-Data: SPMD) の方式に基づく．また，ベクトル・プロセッサやアレイ・プロセッサのような単一命令流・複数データ流 (Single-Instruction stream, Multiple-Data stream: SIMD) 型コンピュータでは，同じ命令が多数のデータ流に適用される．なお，複数命令流・単一データ流 (MISD) に分類されるコンピュータは実例がない．

GPU は，VRAM を主記憶とする共有記憶型マルチプロセッサである．ストリーミング・マルチプロセッサの集合として見れば MIMD であるが，各ウォープのアクティブスレッドは SIMD で動作する．また，CUDA のプログラミングモデルにおいて，各スレッドは同一のカーネル関数を実行するため SPMD でもある．

並列ソフトウェア

共有記憶型マルチプロセッサ上で動作する並列ソフトウェアは，POSIX スレッド (Pthreads) や OpenMP といった標準規格に準拠する．より簡単に並列プログラミングを可能にするという理念のもとに，様々な並列プログラミングモデルがこの規格から派生している．

一般に並列ソフトウェアは次の 4 ステップを含む [5].

step 1. ループの反復などの並行性 (concurrency) のある処理を特定する

step 2. 並行性のあるコード領域を構成し直し，並行領域を明確にする

step 3. 並列プログラミング言語を用いてプログラムを並列化する

step 4. 作成したプログラムのチューニングを行い，最適な性能を引き出す

CUDA による並列プログラミングも上記の手続きに従う．特に CUDA では step 4. に関して，ホスト側のチューニングだけではなく，各種メモリの利用やスレッド・ブロックの割り当てなど．デバイス側のチューニングに対する配慮が求められる．

6 所感

擬似並列でない並列プログラミングは初めての経験でした．GPU のアーキテクチャ，CUDA のプログラミングモデルをしっかりと理解できれば，CUDA による GPGPU はとても強力であると実感しました．CUDA によるプログラミングはそれほど厳しい作法もなかったので，実験そのものはすぐに終わりましたが，アーキテクチャの調査と速度向上に関する考察が大変でした．いろいろと文献をみてみると，GPU を用いた並列プログラミングは，デバイスのチューニングが肝心であると感じました．機会があれば，共有メモリなどを用いた高速化を実際に検証してみたいです．また，考察に関しては，やや思いつきで進めた部分があります．おかしな点があればアドバイスをいただけると嬉しいです．以上で無事に研修 A と実験 D を終わらせることができました．お忙しい中のご指導ありがとうございました．

参考文献

[1] 青木尊之, 額田彰, “初めての CUDA プログラミング,” 2009, 工学社

[2] 岡田賢治, “CUDA 高速 GPU プログラミング入門,” 2010, 秀和システム

[3] David A. Patterson, John L. Hennessy, 成田光彰, “コンピュータの構成と設計 第 4 版 [下],” 2011, 日経 BP 社

[4] 田原哲雄, “高速演算器 第 3 回 「チューニング技法その 1 CUDA プログラミングガイドからピックアップ」,” http://www.gdep.jp/column/view/3, 2012 年 7 月 11 日アクセス

[5] Tim Mattson, “並列プログラミングのエキスパートのようになるには - パート 4: 並列ソフトウェアの作成,” http://www.isus.jp/article/howtobeexpert/part4/, 2012 年 7 月 12 日アクセス

[6] “パターンマッチング（正規化相互相関など）,” http://imagingsolution.blog107.fc2.com/blog-entry-186.html, 画像処理ソリューション, 2012 年 7 月 12 日アクセス

[7] 大島聡史, “これからの並列計算のための GPGPU 連載講座 (III) GPGPU プログラミング環境 CUDA 最適化編,” 東京大学情報基盤センター, スーパーコンピューティングニュース Vol.12 No.3