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

大島聡史
東京大学情報基盤センター

1 はじめに

CUDAプログラムの最適化には、既存のCPU向けの最適化とは異なる様々な要素が存在する。連載第三回である今回は、CUDAプログラムの性能に大きな影響を与えるいくつかの項目に注目し、CUDAにおける最適化プログラミングの方法を解説する。

CUDAの開発環境は不定期に更新が行われており、またハードウェアの世代によって利用可能な機能には違いがある。本稿執筆時点における開発環境(CUDA ToolkitおよびCUDA SDK)の最新バージョンは3.0であり、最新のハードウェア世代ナンバー(Compute Capability)は2.0である[*1]。CUDAプログラムの最適化においてはこれらのバージョン番号が大きな影響を持つ。最新の環境(開発環境およびハードウェア環境)ほど新たな最適化技術が利用可能になったり、以前の環境では性能的に問題があるプログラムでも高速に実行可能となっていたりすることがある。開発環境のバージョンは新たな開発環境をインストールすることで上げることが可能である一方、ハードウェア環境のバージョンはGPUに強く依存する(GPUを変更しないと上げることができない)ので注意が必要である。

本稿で用いている実験環境は表1の通りである。本稿では多くの環境で共通に利用可能な最適化方法を紹介するよう努めるが、各方法により得られる最適化効果が環境ごとに異なる可能性があることをあらかじめ断っておく。

表1 実験環境

CPU	XeonW3520 (4 コア, 2.67GHz)
GPU	GeForceGTX285 (240 コア, 1.48GHz, Compute Capability 1.3)
OS	CentOS 5.4 x86_64
開発環境	CUDA Toolkit 3.0, CUDA SDK 3.0

2 対象問題と測定範囲および性能測定方法

2.1 対象問題と測定範囲

今回は単純な行列積(行列A = 行列B × 行列C)を最適化対象問題とする。実装の内容は主にCUDA Programming Guide等を参考にして独自に実装したものである。今回実験に用いた全プログラムは筆者のwebサイト(http://www.cspp.cc.u-tokyo.ac.jp/ohshima/)にて公開する予定である。

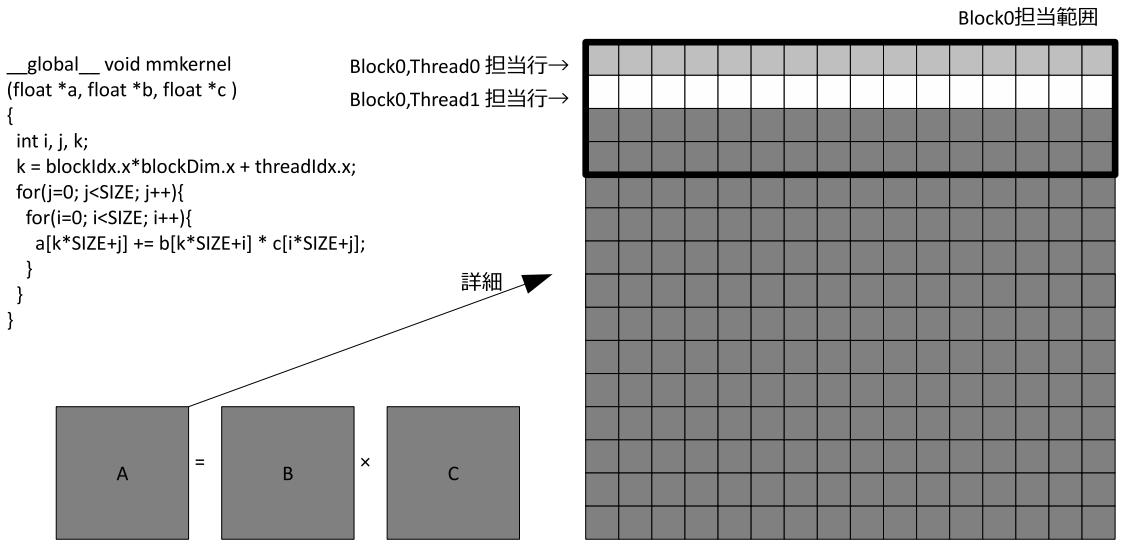
[*1] Compute Capability 2.0 の GPU は GeForceGTX480/470 および TeslaC2010 のみであり本稿執筆時点では発売直後のため用いていない

今回は単純化のため問題を以下のように定める：

- 問題サイズ(行列の一辺の長さ)は4096に固定
- データ型は単精度浮動小数点形式(float型)
- メモリの配置は一次元配列
- 行列の転置については考えない

この問題設定で単純に実装した場合、行列Bに対するメモリアクセスは連続に、行列Cに対するメモリアクセスは不連続になる。性能測定範囲(実行時間測定範囲)については、GPU上での計算のみではなくCPU-GPU間のデータ転送時間を含めて実行時間を測定し、$2 \times SIZE \times SIZE \times SIZE \div time$(ただしSIZEは行列の一辺の長さ)の式にて性能FLOPS値を算出することにする。

上記の問題設定にしたがい、GPU上の単一ブロック・単一スレッドを用いて単純な三重ループ処理による行列積を行ったところ、わずか5MFLOPSという低い性能しか得られなかった。また単純な並列化の例として、64ブロック*64スレッドの4096並列を用いて各行の計算を並列実行させてみた(図1)ところ、1GFLOPS程度の性能しか得られなかった。一方、CUDAに付属の数値計算ライブラリCUBLASを用いて行列積を実行したところ、約315GFLOPSの性能が得られた[*2]。本稿はCUBLASの性能に追いつき追い越すことが趣旨ではなく、CUDAプログラミングの最適化技術を紹介することが趣旨であるが、CUBLASの性能を目指して様々な高速化手法を適用していくことにする。



右図は問題サイズ16、4ブロック*4スレッド＝16並列で実行する例。gridDim=(4,1), blockDim=(4,1) を想定している。
本文中では問題サイズ4096に対して64ブロック*64スレッド=4096並列、gridDim=(64,1), blockDim=(64,1)で実行している。

図 1　単純な並列実行の例

[*2] 残念ながら CUBLAS はソースコードが公開されていないため詳細な実装を知ることはできない

2.2 性能測定方法

CPUプログラム(C言語プログラム)の実行時間測定方法としては、時刻取得関数、特に高精度なgettimeofday関数を用いた方法が知られている。今回は性能評価対象にCPU-GPU間のデータ転送を含めているため、CPUからGPUへデータのセットを行う前とGPUからCPUへデータの書き戻しを行った後にgettimeofday関数を呼び出して差分をとることで性能FLOPS値を求めている。

ちなみに単純な時刻取得関数の利用では、CPUからGPUへのデータセット単体や、GPU上での計算単体での実行時間を測定することができない。

たとえば、CPUからGPUに対して演算開始指示を行うためには

```
funcname<<<xyz,xyz>>>(args1,args2);
```

という形式の呼び出し記述を用いる。この前後の行でgettimeofdayを行い差分を取ってもGPUに対する演算開始指示は非同期であるため、すなわちプログラムはCPUからGPUへの演算開始指示が終了した時点で呼び出し指示の次の行へ遷移してしまうため、GPU上での計算時間を取得することはできない。このように、CUDAプログラムにおいてはいくつかの処理が非同期処理であることに気をつける必要がある[*3]。GPUによる処理の時間を確実に測定するためには、`cudaThreadSynchronize` 関数を呼び出してGPUによる処理が確実に終了するのを待つ必要がある。

GPUによる演算時間を測定する別の方法としては、4章で紹介するCUDAプロファイラを利用する方法がある。CUDAプロファイラを利用すれば、プログラム実行終了後にCPU-GPU間の転送時間やGPUによるカーネル関数実行時間をそれぞれ個別に確認することができる。

3 CUDA最適化プログラミング

本章では、以下の5つの最適化手法について解説する。

1. SIMD化(GlobalMemoryへの並列連続メモリアクセス)
2. ConstantMemory・TextureMemoryの活用
3. SharedMemoryの活用
4. ループの展開(計算担当範囲の調整)
5. ホスト側のチューニング

3.1 SIMD化 (GlobalMemoryへの並列連続メモリアクセス)

本節ではGlobalMemoryの活用方法について解説する。

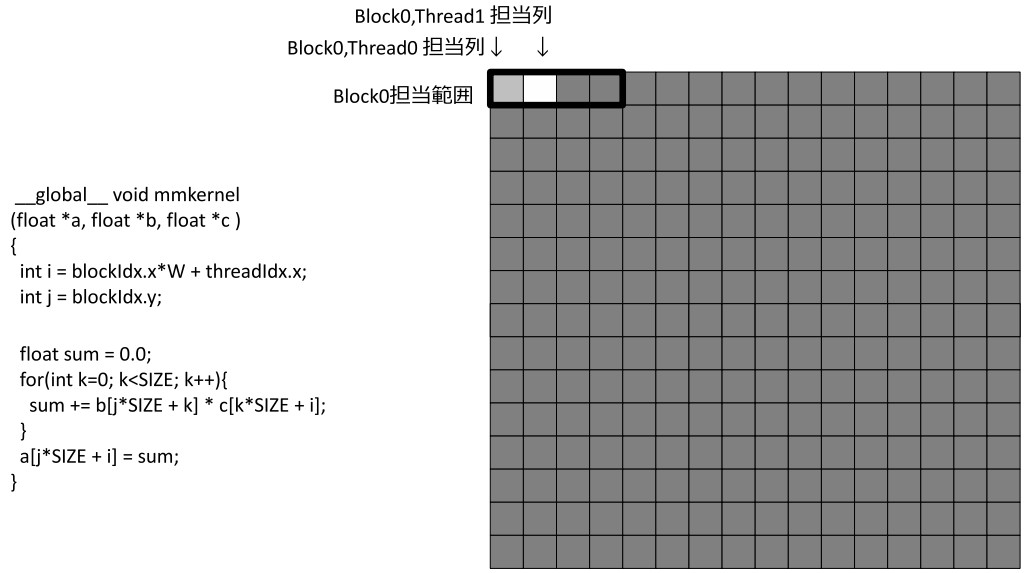
GlobalMemoryはGPU上の全てのプロセッサで共有し読み書き可能なため様々な用途に利用可能なメモリであり、連続アクセス時の転送速度は高速である一方でレイテンシが大きくランダムアクセス性能が低いという特徴を持つ。GlobalMemoryの最適化における基本的な戦略

[*3] 逆に非同期処理を利用すれば、CPU と GPU で同時に処理を行い CPU と GPU 両方の演算能力を利用できる

は、GPU上の多数のプロセッサ(SP)による連続領域並列アクセスを行うことである。このメモリアクセスは**コアレスなメモリアクセス**と呼ばれており、CUDAの性能最適化において非常に重要である。

図1のプログラムでは、行列Bについては各スレッドが独立に行列積の各行を計算しており、また行列Cについてはメモリが不連続な方向への連続アクセスが必要である。この割り当て方の場合、CPU(C言語)プログラムであれば、行列Bについてはスレッド毎に連続メモリアクセスを行うことができる。一方CUDAでは、同一MP内の複数のSPにまたがって連続メモリアクセスを行うことが重要である。しかし今回のプログラムでは、各ブロック内のスレッドは不連続なGlobalMemoryアクセスを多く行ってしまっている。

そこで、ブロック内のスレッドが連続メモリアクセスを行うようにプログラムを変更する。図2はブロック単位で行列積の各行の計算を担当し、行内の計算においては各スレッドが各列の計算を担当するように変更した例である。各ブロック毎のスレッド数をWとしたとき、ブロック数は横に 問題サイズ/W、縦に 問題サイズ となるようにプログラムを記述している。これにより行列Bに対するメモリアクセスが連続になり性能が向上する。



右図は問題サイズ16、64ブロック*4スレッドで実行する例。W=4, gridDim=(4,64), blockDim=(4,1) を想定している。
本文中では問題サイズ4096に対して64ブロック*64スレッド=4096並列、gridDim=(4096/W, 4096),blockDim=(W,1)で実行している。

図 2　SIMD 化 (スレッドによる連続メモリアクセス) の例

各ブロック毎のスレッド数Wを変化させて性能を比較したところ、表2に示すように最大で41.1GFLOPSまで性能が向上した。このように、適切な連続領域アクセス(コアレスなメモリアクセス)を行うことがGlobalMemoryを用いた高速化の基本である。より正確には、同一MP内の複数SPが適切なGlobalMemoryアクセスを行う場合には、複数のメモリアクセスがまとめて実行されるため性能が向上する。CUDAの世代(開発環境およびハードウェアのバージョン)が進むにしたがってコアレスなメモリアクセスを行える条件が緩和されてきており、最近のGPUでは不連続でも一定領域内に収まっているGlobalMemoryアクセスはまとめて実行される

ことが多い[*4]。GlobalMemoryは容量が大きくGPU全体で共有されるため使用する機会が多いメモリである。不用意に不連続なメモリアクセスを繰り返して性能低下を招かぬように気をつけることが重要である。

表 2　SIMD 化時の性能 (GFLOPS)

W(SIMD 長)	8	16	32	64	128	256	512
性能	8.4	17.2	32.5	41.1	40.6	40.7	38.3

3.2　ConstantMemory・TextureMemory の活用

本節ではConstantMemoryとTextureMemoryの活用方法について解説する。

ConstantMemoryとTextureMemoryはGPUから見ると読み込み専用であり、またキャッシュ効果が得られるメモリである。そのため、頻繁にアクセスする読み取り専用データを配置するのに適している。ConstantMemoryは容量が64KBに制限されているため格納するデータを厳選する必要があるが、TextureMemoryは大容量のメモリを配置できるために様々な問題に適用しやすい。さらにTextureMemoryは二次元および三次元のデータを扱ったりデータの補間を行うこともできるため、利用する対象データによっては高い性能が期待できる。

行列積においては、行列Bと行列Cが読み取り専用のデータであり、またアクセス回数も多いため、ConstantMemoryやTextureMemoryの活用による性能向上が期待できる。今回は利用可能な容量が多いTextureMemoryを利用してみることにする。

図3にTextureMemoryを利用するプログラムを示す。このプログラムは前節のSIMD化した行列積プログラムに対して、行列BのみをTextureMemoryに配置するよう書き換えたものである。TextureMemoryを使うにはデータセットのための手続きがホスト側に必要なため、ホスト側プログラムの記述量が若干増加していることがわかる。またデバイス側プログラムについても、テクスチャの参照座標を計算する記述が追加されていることがわかる。

TextureMemoryを利用した場合の性能は表3の通りである。最大58.9GFLOPSの性能が得られておりGlobalMemoryのみの場合と比べて性能が向上していることがわかる。なお、行列CをTextureMemoryに配置した場合や行列Bと行列Cの両方をTextureMemoryに配置したプログラムも作成して性能を比較してみたが、残念ながら良い性能は得られなかった。TextureMemoryによる性能向上は、各MPに搭載されたTextureCacheにデータがキャッシュされることによるものであり、どの程度性能向上が得られるかはメモリアクセスパターンに依存している。行列Cのみまたは行列BとCの両方をTextureMemoryに配置したプログラムのメモリアクセスパターンよりも行列BのみをTextureMemoryに配置した方がTextureCacheをうまく活用できたものと考えられる。

[*4] 特に Compute Capability 1.3 では 1.1/1.2 と比べて条件が緩和されている。詳細なルールについては CUDA Programming Guide を参照していただきたい。

```
texture<float, 2>texB;  // Textureの宣言

__global__ void mmkernel
( float *a, float *c )
{
 int i = blockIdx.x*W + threadIdx.x;
 int j = blockIdx.y;
 float x1,y1, val;

 float sum = 0.0;
 for(int k=0; k<SIZE; k++){
  x1 = (float)(k) + 0.5f;  // Texture参照座標の算出
  y1 = (float)(j) + 0.5f;
  val = tex2D(texB,x1,y1);  // Textureからデータを取得
  sum += val * c[k*SIZE + i];
 }
 a[j*SIZE + i] = sum;
}

 // 以下、CPU側の記述
 // 通常のデータセット
 //cutilSafeCall(cudaMemcpy(d_B, g_B, size, cudaMemcpyHostToDevice) );
 // Textureへのデータセット
 cudaChannelFormatDesc desc = cudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
 struct cudaArray *d_texB;
 cudaMallocArray(&d_texB, &desc, SIZE, SIZE);
 cudaMemcpyToArray(d_texB, 0,0, g_B, sizeof(float)*SIZE*SIZE, cudaMemcpyHostToDevice);
 texB.filterMode = cudaFilterModeLinear;
 cudaBindTextureToArray(texB, d_texB, desc);
```

図 3　TextureMemory を用いたプログラム

表 3　TextureMemory 活用時の性能 (GFLOPS)

W(SIMD 長)	8	16	32	64	128	256	512
行列 B のみ TextureMemory に配置	6.6	13.0	24.9	46.7	58.9	57.4	54.5
行列 C のみ TextureMemory に配置	5.4	10.7	18.7	31.1	23.5	23.6	22.6
行列 B と C を TextureMemory に配置	9.0	17.1	29.1	31.0	24.5	25.5	26.7
参考:TextureMemory 未使用	8.4	16.9	31.8	39.9	39.4	39.6	37.3

3.3　SharedMemory の活用

本節ではSharedMemoryの活用方法について解説する。

SharedMemoryは、高速かつレイテンシが低い一方で1MPあたりの容量がわずか16KBしかない、各ブロック内の全スレッドでデータを共有可能な共有メモリである。SharedMemoryの位置づけとしては、既存のCPUにおけるキャッシュメモリやCell B.E.のSPEにおけるLS(Local Storage)に近いと見なされることがある。しかしSharedMemoryは、CPUのキャッシュと異なりプログラマが明示的に操作を記述する必要があり、SPEのLSと異なり共有メモリであり、そのうえキャッシュやLSと比べて容量が小さい。そのため、SharedMemoryならではの活用方法を理解する必要がある。

SharedMemoryの基本的な活用方法としては、GlobalMemoryに対するキャッシュとしての利用が挙げられる。3.1節で述べたように、GlobalMemoryは転送速度自体は高速であるもののレイテンシが大きくランダムアクセスに弱いという特徴を持っている。そのためGlobalMemory上の特定の不連続データを複数回利用する場合には、一度SharedMemoryへコピーしてから利用することで性能を改善することが可能となる。

3.1節でSIMD化したプログラムを再度見直すと、ランダムアクセスこそ無いものの、各ブロック内のスレッドが全く同じGlobalMemory(行列B)上のデータを参照していることがわかる。そこで、行列Bへのアクセスを SharedMemoryによって高速化することにする。図4にSharedMemoryを用いたプログラムを示す。本プログラムと3.1節のプログラムとの違いは、行列Aの要素に対するアクセスにSharedMemoryを用いるか否かのみである。そのためプログラムの変更点も__shared__指示子を用いたSharedMemoryの宣言およびSharedMemoryにアクセスするための記述(ksループのステップ数変更およびSharedMemory内のデータにアクセスするための内側ループの追加) のみである。小さなプログラムの修正ながらその効果は大きく、表4に示すように最大で75.4GFLOPSを達成している。

```
__global__ void mmkernel
(float *a, float *b, float *c )
{
 int tx = threadIdx.x;
 int i = blockIdx.x*W + tx;
 int j = blockIdx.y;

 float sum = 0.0;
 __shared__ float sb[W];          // SharedMemoryの宣言

 for(int ks=0; ks<SIZE; ks+=W){
  sb[tx] = b[j*SIZE + ks + tx];    // SharedMemoryへのコピー
  __syncthreads();                 // スレッド間の同期（コピー終了待ち）
  for(int k=0; k<W; k++){          // SharedMemoryのデータを用いて計算
   sum += sb[k] * c[(ks+k)*SIZE+i];
  }
  __syncthreads();
 }
 a[j*SIZE + i] = sum;
}
```

図 4　SharedMemory を用いたプログラム

表 4　SharedMemory 活用時の性能 (GFLOPS)

W(SIMD 長)	8	16	32	64	128	256	512
性能	22.1	49.8	38.6	68.8	75.1	75.4	75.3
参考:SharedMemory 未使用	8.4	16.9	31.8	39.9	39.4	39.6	37.3

今回のプログラムは各スレッドからSharedMemoryへのアクセスが単純で偏りのないものであったが、SharedMemoryの特定の範囲に対して偏ったアクセスがある場合にはあまり高い性能が得られないことがある。これはSharedMemoryにおけるバンクコンフリクトの問題として知られている。SharedMemoryは全体が16のメモリバンクに分割されており、特定のバンクにアクセスが集中するとメモリアクセスが待たされてしまい性能が低下するという問題である。

そのためSharedMemoryへのアクセスパターンによっては参照・格納順序の変更やパディングなどを行うことで初めて性能が向上することがある。また16KBという限られた空間では適切にデータを配置しきれずSharedMemoryを活かせないこともあるので注意が必要である。

3.4 ループの展開 (計算担当範囲の調整)

CPUプログラムの高速化において有用な最適化手法の1つにループの展開(アンローリング)がある。特にGPUは分岐処理に弱いという特徴があるため、アンローリングにより分岐処理(ループの終了判定)が削減されることによる性能向上が期待できる。

表5に、SIMD化されたプログラム、TextureMemoryを用いたプログラム、そしてSharedMemoryを用いたプログラムのそれぞれをアンローリングした際の性能および性能比(それぞれアンローリングを行わない場合を1とした相対性能) を示す。プログラムの差分は単純なアンローリングの有無のみであるため、プログラムの詳細な説明は省略する。実験の結果、残念ながら単純なアンローリングによる性能向上は得られなかった。

表 5　アンローリング後の性能 (GFLOPS、括弧内はアンローリング前に対する性能比)

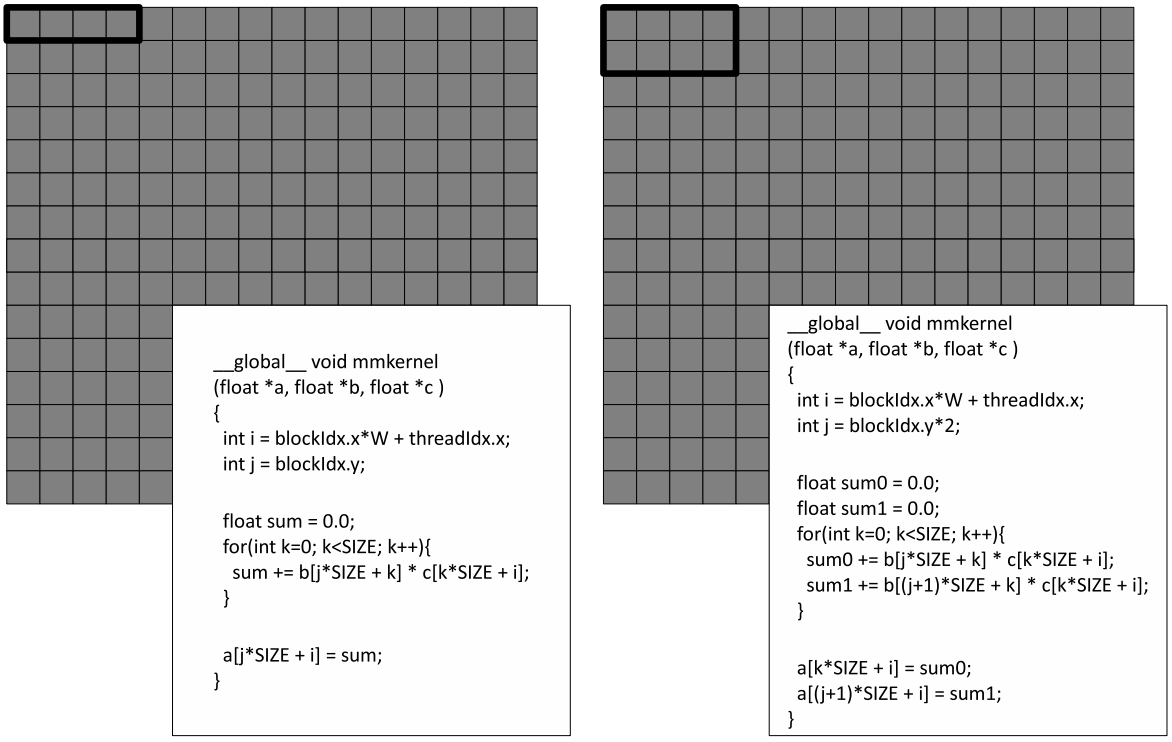
アンローリング段数 (SIMD 長は全て 256)	1	2	3	4
SIMD 化 + アンローリング	40.3 (0.99)	40.6 (1.00)	40.3 (0.99)	40.0 (0.98)
TextureMemory(行列 B のみ)+ アンローリング	55.3 (0.92)	55.1 (0.92)	55.2 (0.92)	55.3 (0.92)
SharedMemory+ アンローリング	75.5 (1.00)	58.9 (0.78)	75.4 (1.00)	75.4 (1.00)

ところで、これまでに挙げたプログラムはいずれもブロック形状が縦に1のもの(gridDim.yが1のもの)のみを扱ってきた。一方でブロック形状を変更して各スレッドが担当する行数を増やし、ブロック数を減らすと、プログラム全体で見たループの終了判定回数(全スレッドが行う終了判定の累計回数)を減少させることができる。これは、**三重ループによる行列積において外側のループをアンローリングすること**に相当する。

SIMD化されたプログラムの計算担当範囲を調整したプログラムのソースコードと動作イメージを図5に、SIMD化されたプログラム、TextureMemoryを用いたプログラム、そしてSharedMemoryを用いたプログラムのそれぞれについて計算担当範囲を調整したプログラムの性能と性能比を表6に示す。今回のアンローリングはいずれも高い性能向上が得られており、特にSharedMemoryを用いたプログラムについては3倍を超える速度向上比を達成している。最大性能が得られたのはSharedMemory+範囲拡大4倍にした場合で、245.8GFLOPSを達成した。

3.5 ホスト側のチューニング

ここまではデバイス(GPU)側のチューニングについて紹介してきた。本節ではホスト(CPU, メインメモリ)側のチューニング手法を紹介する。



左図は変更前、右図は変更後（ブロック数を半分にしてmmkernelを呼び出す）。太枠内が１ブロックの担当する範囲。

図 5　計算担当範囲の調整

表 6　問題割り当てを変更した際の性能 (GLOPS、括弧内はアンローリング前に対する性能比)

割り当て範囲拡大率 (SIMD 長は全て 256)	2	4	8	16
SIMD 化 + 範囲拡大	55.8 (1.37)	60.3 (1.48)	78.7 (1.93)	65.6 (1.61)
TextureMemory+ 範囲拡大	104.2 (1.73)	104.4 (1.74)	104.4 (1.74)	104.5 (1.74)
SharedMemory+ 範囲拡大	146.2 (1.94)	245.8 (3.26)	137.0 (1.82)	- -

(SharedMemory 16 倍は SharedMemory の容量が 16KB を超えてしまったため測定不可能)

特に簡単で効果が高い手法としては、pinned memory(Page-Locked memory)の使用が挙げられる。pinned memoryを使用するとCPU-GPU間のデータ転送性能を向上させることができる。使い方はCPU側でメモリを確保する際に

```
cudaHostAlloc((void**)&g_data, sizeof(data)*length, cudaHostAllocWriteCombined);
```

のようにcudaHostAlloc関数を使えば良いだけであり、非常に簡単である。ただし、pinned memoryを使いすぎるとCPU側の性能が低下することがあるので注意せねばならない。

例として、3.4節のSharedMemory+範囲拡大(4倍)プログラムに対してpinned memoryを適用して性能を測定した。その結果、245.8GFLOPSから247.0GFLOPSへとわずかながら性能を向

上させることができた。CPU側のメモリ確保方法を変更するだけで若干の性能向上が得られていることから、データ送受信のサイズが大きかったり通信回数が多くプログラム実行時間に対するCPU-GPU間のデータ転送時間の占める割合が大きな場合には、pinned memoryを有効に活用すべきである。

他のホスト側のチューニング手法としては、計算と通信のオーバーラップが挙げられる。CUDAではCPU-GPU間のデータ通信とGPUによる計算をオーバーラップさせることができるため、複数回のカーネル実行を行う場合には適切にオーバーラップ実行することで性能向上が可能となる。しかし、今回は行列積を一度だけ実行して性能を測定しているため、通信と計算をオーバーラップ実行する機会がない。本手法による性能向上については機会があれば改めて紹介したい。

別のオーバーラップの考え方として、CPUとGPUで問題を分割し同時に計算することが考えられる。行列積はCPUでも高速に計算できる問題であり、また問題分割がとても容易である。そのため、これまでに解説してきたプログラムはいずれもGPUが計算をしている間にCPUは何もせずに待っていたが、対象問題をデータ分割してCPUでも計算すればより高い性能が得られることが期待できる。なおこれはCUDA特有の手法ではなく、グラフィックスAPIを用いたGPGPUでも可能だった手法である。

3.6 CUDA 最適化プログラミング:まとめ

以上、本章では5つの最適化手法について解説した。今回解説した最適化手法のいくつかは複数種類を組み合わせて利用することでより高い性能を得ることができる。(実際、いくつかの解説中では特に説明せずに組み合わせて利用していた。) 本章にて解説した最適化手法の中で高い性能が得られた組み合わせとしては、SIMD化・SharedMemory・カーネル内最内ループのアンローリング・計算担当範囲の調整・pinned memoryの活用を組み合わせて実行したケースで最大277.7GFLOPSを達成している。

今回は主に行列Bのメモリアクセスを中心とした最適化を行ってきたが、さらに高い性能を得るためには、行列Cのメモリアクセスについても最適化する必要がある。CUDA Programming Guideに掲載されている行列積の例では二次元ブロック化によるメモリアクセスを行っているので参考になるだろう。

4 CUDAプロファイラを用いた性能の解析

4.1 CUDA プロファイラ

GPUはCPUと同等以上にメモリアクセスパターンやプロセッサへの計算割り当てによって性能が左右されるハードウェアである。しかしCUDA最適化プログラミングにおいては、様々な最適化手法を組み合わせた場合にどの手法がどの程度影響を与えているのか、またプログラム記述の変更がハードウェアの動作にどのように影響しているのかを知ることは容易ではない。

CUDA最適化プログラミングの手助けになりうるツールとしては、CUDA Toolkitに付属のNVIDIA社製CUDAプロファイラが挙げられる。本章ではCUDAプロファイラの簡単な使い方を紹介し、また前章のいくつかの実行結果に対するプロファイル出力結果を確認してみることにする。

CUDAプロファイラはCUDA toolkitインストール時に自動的にインストールされる。CUDAプロファイラを使うために最低限必要な準備は、環境変数CUDA_PROFILEに1を設定することである。CUDA_PROFILEを設定した状態でCUDAプログラムを実行すると、実行後にカレントディレクトリにcudab_profile.logというプロファイル結果が書き込まれたファイルが作られる。出力ファイル名は環境変数CUDA_PROFILE_LOGにて変更することも可能である。

図6に行列積プログラムの1つを実行した際のプロファイル結果を示す。デフォルトでプロファイル出力されるデータはCPU時間、GPU時間()、occupancy(GPU占有率)のみであるが、環境変数CUDA_PROFILE_CONFIGによって設定ファイルを指定することで出力される項目を変更することもできる。CUDAプロファイラの詳細な使用方法は付属のドキュメント[*5] を参照されたい。

```
> cat cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 1.6
# CUDA_DEVICE 0 GeForce GTX 285
# TIMESTAMPFACTOR fffff72f1365e070
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 11760.608 ] cputime=[ 11928.000 ]
method=[ memcpyHtoD ] gputime=[ 11759.840 ] cputime=[ 11926.000 ]
method=[ memcpyHtoD ] gputime=[ 11767.712 ] cputime=[ 11936.000 ]
method=[ mmkernel ] gputime=[ 4120903.500 ] cputime=[ 10.000 ] occupancy=[ 0.250 ]
method=[ memcpyDtoH ] gputime=[ 11828.768 ] cputime=[ 12003.000 ]
```

図 6　プロファイル出力結果例

この他、CUDAプロファイラの出力を可視化するVisual Profilerも提供されており、OSを問わず(Windows,Linux,MacOSXのいずれでも)使うことができる。さらにVisualStudioと統合された開発環境nsightも開発が進んでいる。今後これらについても機会があれば紹介する。

4.2　CUDA プロファイラによるプログラム解析の例

本節ではこれまでに取り上げたプログラムのいくつかについてプロファイル出力結果を確認する。なお、同時にプロファイル可能なパラメタ数が限られているため、複数回実行して取得し比較を行っている。

まずはじめに、2.1節で作成した単純な並列実行のプログラム(1.0GFLOPS)と3.1節で作成したSIMD長8(8.4GFLOPS)のプログラムのプロファイル出力結果を比較してみることにする。以下に示すのはそれぞれのプロファイル出力結果である。

(単純な並列実行)

```
method=[ mmkernel ] gputime=[ 140879904.000 ] cputime=[ 140881600.000 ] occupancy=[ 0.500 ]
local_load=[ 0 ] local_store=[ 0 ] gst_request=[ 100663296 ] divergent_branch=[ 4294967295 ]
branch=[ 100687899 ] sm_cta_launched=[ 3 ] gld_incoherent=[ 0 ] gld_coherent=[ 4294967295 ]
gld_32b=[ 4294967295 ] gld_64b=[ 0 ] gld_128b=[ 0 ] gst_incoherent=[ 0 ]
gst_coherent=[ 4294967295 ] gst_32b=[ 4294967295 ] gst_64b=[ 0 ] gst_128b=[ 0 ]
instructions=[ 906313776 ] warp_serialize=[ 0 ] cta_launched=[ 7 ]
```

[*5] toolkit をインストールしたディレクトリ (デフォルトでは/usr/local/cuda/) 直下の doc ディレクトリにある CUDA_Profiler_3.0.txt

(SIMD 長 8)
```
method=[ mmkernel ] gputime=[ 16227636.000 ] cputime=[ 16227876.000 ] occupancy=[ 0.250 ]
local_load=[ 0 ] local_store=[ 0 ] gst_request=[ 69906 ] divergent_branch=[ 4294967295 ]
branch=[ 286824318 ] sm_cta_launched=[ 69906 ] gld_incoherent=[ 0 ] gld_coherent=[ 4294967295 ]
gld_32b=[ 1717993472 ] gld_64b=[ 0 ] gld_128b=[ 0 ] gst_incoherent=[ 0 ]
gst_coherent=[ 419432 ] gst_32b=[ 209716 ] gst_64b=[ 0 ] gst_128b=[ 0 ]
instructions=[ 2291798310 ] warp_serialize=[ 0 ] cta_launched=[ 209716 ]
```

これらのプログラムを比較すると、単純な並列実行はSIMD化したものに比べてgld_32b(GlobalMemoryに対するloadの回数)やgst_32b(GlobalMemoryに対するstoreの回数)が非常に多いことがわかる。これはSIMD化することによってGlobalMemoryへのアクセスがまとめて行われるようになったことを反映しているものであると考えられる。

つづいて、3.1節で作成したSIMD長8(8.4GFLOPS)のプログラムとSIMD長256(40.7GFLOPS)のプログラムのプロファイル出力結果を比較してみる。以下に示すのは各SIMD長256におけるプロファイル出力結果である。

(SIMD 長 256)
```
method=[ mmkernel ] gputime=[ 3337158.750 ] cputime=[ 3337238.000 ] occupancy=[ 1.000 ]
local_load=[ 0 ] local_store=[ 0 ] gst_request=[ 17480 ] divergent_branch=[ 2165478906 ]
branch=[ 71705145 ] sm_cta_launched=[ 2185 ] gld_incoherent=[ 0 ] gld_coherent=[ 1708726971 ]
gld_32b=[ 429522944 ] gld_64b=[ 429522944 ] gld_128b=[ 0 ] gst_incoherent=[ 0 ]
gst_coherent=[ 419456 ] gst_32b=[ 0 ] gst_64b=[ 104864 ] gst_128b=[ 0 ]
instructions=[ 573064352 ] warp_serialize=[ 0 ] cta_launched=[ 6554 ]
```

SIMD長8とSIMD長256で異なる値として、occupancy(GPU上の演算器がどの程度使われたか) が挙げられる。本プログラムではSIMD長=ブロックあたりのスレッド数であることから、SIMD長8ではGlobalMemoryへのアクセスレイテンシをアクティブスレッドの切り替えによって隠蔽し切れておらず、GPU上の演算器を使い切れていなかったためにこのような差が生じたと考えられる。また、SIMD長8では_32bという32bit単位のメモリ読み書きばかりなのに対して、SIMD長256では_64bという64bit単位のメモリ読み書きが多くなっており、SIMD長を大きくすることでメモリの読み書きがまとめて行われるようになったことが確認できる。

つづいて、SIMD長256のプログラム(40.7GFLOPS)と、これに対して3.4節で計算割り当て範囲の拡大(8倍)を行ったプログラム(78.7GFLOPS)のプロファイル出力結果を比較してみる。

(割り当て範囲 8 倍)
```
method=[ mmkernel ] gputime=[ 1810732.000 ] cputime=[ 1810776.000 ] occupancy=[ 0.500 ]
local_load=[ 0 ] local_store=[ 0 ] gst_request=[ 17536 ] divergent_branch=[ 1100197911 ]
branch=[ 8991858 ] sm_cta_launched=[ 274 ] gld_incoherent=[ 0 ] gld_coherent=[ 1122244519 ]
gld_32b=[ 429916160 ] gld_64b=[ 53739520 ] gld_128b=[ 0 ] gst_incoherent=[ 0 ]
gst_coherent=[ 419840 ] gst_32b=[ 0 ] gst_64b=[ 104960 ] gst_128b=[ 0 ]
instructions=[ 269445042 ] warp_serialize=[ 0 ] cta_launched=[ 820 ]
```

割り当て範囲を拡大した結果、occupancyは低下してしまったが、branch(プログラム全体における分岐回数)やinstructions(プログラム全体における実行命令数)などの大幅な減少が確認できており、複数行で同様に行われていたループ処理(ループ終了判定に関わる分岐命令)がまとめられたことにより性能向上したことが読み取れる。

さらにSharedMemoryを用いた場合のプロファイル出力結果と比較を行いたいところであるが、残念ながらCompute Capability 2.0以上のGPUでないとSharedMemoryの読み書き回数は

取得できないため、比較することができなかった。

以上のように、CUDAプロファイラを使うことで詳細な実行情報取得することが可能となり、最適化の効果を確認することができる。CUDAプロファイラを用いると様々な情報を確認することができる一方で、現在のところプロファイラ出力のどの値がどの程度性能に影響するかについてはあまり明らかになっていない。そのため単純に最適化の指標として使うことは容易ではないが、意図したとおりに最適化が行えているかを確認する際などに役に立つだろう。今後は様々な最適化手法や各手法におけるパラメタの選択にプロファイラが有効活用できるようになることが期待される。

以上、第三回の今回は行列積を対象問題としてとりあげてCUDA向けの最適化プログラミングについて紹介した。

今回は対象問題が単純なこともあり多くの最適化手法が少なからず性能向上を得られたが、実アプリケーションでは常に性能向上が得られるとは限らない。また、行列積のように一度GPUに計算命令を出して終わりではなく、CPUとGPUそれぞれが様々な計算を繰り返し行うアプリケーションや複数台のGPUを使うようなアプリケーションでは、CPU-GPU間の通信がボトルネックにならないようにアプリケーション全体の最適化を行わなくては高い性能を得ることはできない。アプリケーションの特徴とGPUの特徴を見極めて適切な最適化を行う必要がある。

(次回に続く)