2008－HPC－117 (9)
2008／10／15

GPGPU上での流体アプリケーションの高速化手法 ～1GPUで姫野ベンチマーク60GFLOPS超～

成瀬 彰† 住元真司† 久門耕一†

本稿では，メモリバンド幅ネックの流体アプリケーションをGPGPU上で高速化する手法について述べる。GPGPUとしてはCUDA対応GPUを対象とした。各種メモリアクセス性能調査結果より、安定して高いメモリバンド幅を実現するには、スレッド進行の同期化、アクセスパターンの局所化、スレッド数の最適化が重要であることが分かった。これに基づいて姫野ベンチマークを高速化した結果、GeForce GTX280上で69.7GFLOPSの性能を達成した。これは従来実装と比べて1.69倍の性能であり、この性能達成時には115GB/sの実効メモリバンド幅が出ていたことを意味する。これは理論ピークメモリバンド幅の81%に相当する。

Acceleration Technique of Computational Fluid Dynanics on GPGPU – Over 60GFLOPS Himeno Benchmark Performance on 1 GPU –

Akira Naruse,† Shinji Sumimoto† and Kouichi Kumon†

We present the acceleration technique of CFD (Computational Fluid Dynamics) on GPGPU, that needs high memory bandwidth. The memory bandwidth and latency of GPU are measured on various conditions, and it is clarified that following techniques are required to achieve high memory bandwidth on GPU: synchronization among threads, locality of memory accesses and appropriate number of threads. We've applied these techniques to Himeno benchmark program which is the kernel routine of CFD. Our accelerated code runs 69% faster than the existing implementation and attains 69.7 GFLOPS on GeForce GTX280, that corresponds to 115 GB/s in memory bandwidth.

1. はじめに

GPU(Graphic Processing Unit)の汎用化・高速化に伴い、GPUをグラフィックス計算以外の汎用計算に用いるGPGPU(General-Purpose Computing Using Graphics Hardware)[1]が注目を集めている。汎用CPUと比べてGPUは桁違いの演算性能・メモリ転送性能を備えており、高い演算性能を必要とする行列積[2]やN体問題[3]、高いメモリ転送性能を必要とするFFT[4]や流体アプリケーション[5]など、HPCの様々な分野でGPGPU対応が進んでいる。特に、nVidiaがCUDA(Compute Unified Device Architecture)[6]をリリースして以来、GPGPU対応が加速している。これは、CUDAが提供するプログラミング環境により、CPUからGPUへのプログラム移植が従来より格段に容易になったためである。

しかし、GPU上でのプログラムの高速化はまだ歴史が浅く十分なノウハウが蓄積していない。また、GPUベンダーがGPUのアーキテクチャに関する詳細な情報を開示していないこともあり、現在はGPU上でプログラムを高速化するには試行錯誤を繰り返すしかないという状況にある。

本稿では、GPU上で安定して高いメモリバンド幅を実現する方法を明らかにし、それを流体アプリケーションのベンチマークプログラムである姫野ベンチマーク[7]に適用した結果を報告する。我々が高速化した姫野ベンチーマークは、GTX280上で69.7GFLOPSの性能を達成した。これは従来実装に比べて1.69倍の性能であり、この性能達成時には115GB/sの実効メモリバンド幅が出ていたことを意味する。これは理論ピークメモリバンド幅の81%に相当する。

以降、2章でCUDAの概要を述べ、3章でGPUのメモリアクセス特性を明らかにし、4章でGPUで高いメモリバンド幅を実現する方法を提案する。5章で提案方法に基づき姫野ベンチマークを高速化した結果について述べ、6章でまとめる。

2. CUDAの概要

CUDA対応GPUは複数のマルチプロセッサ(MP)を備えており、MPは8個のストリームプロセッサ(SP)で構成される。各MPは8K/16K本のレジスタ、16KBの共有メモリを備えている。MP間で相互アクセス可能なメモリはテクスチャメモリ、定数メモリ、グローバルメモリである。テクスチャメモリと定数

† 富士通研究所
Fujitsu Laboratories

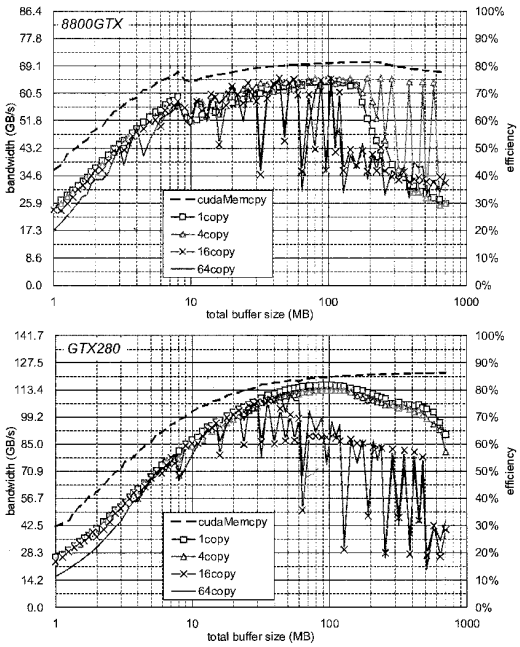

図 1 ストリーム数・バッファサイズとバンド幅 (256 スレッド/ブロック、2 ブロック/MP)
Fig. 1 # of streams, size of buffer and memory bandwidth (256 threads/block, 2 blocks/MP)

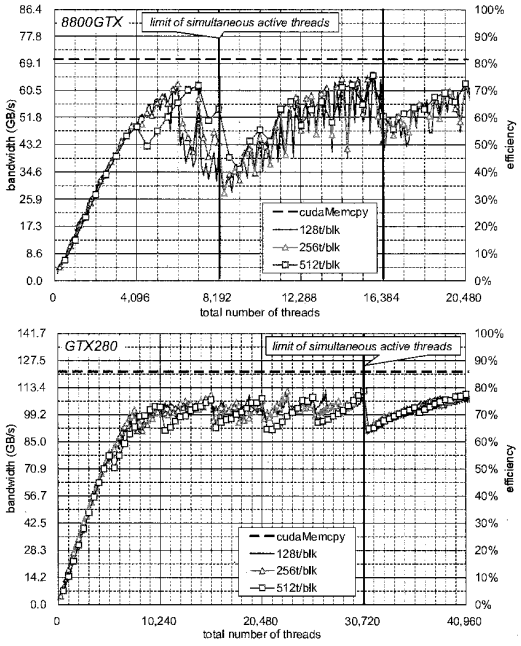

図 2 スレッド数とメモリバンド幅 (バッファサイズ:256MB)
Fig. 2 # of threads and memory bandwidth (buffer size:256MB)

メモリはキャッシュ対象となるが読み出し専用であり GPU 上では更新できない。グローバルメモリはキャッシュされないが更新可能である。主要 GPU のスペックを表 1 に示す。

CUDA プログラムにおいて、GPU で実行される処理はブロックと呼ばれる単位に分割され、各ブロックは 1 つの MP に割り当てられる。1 つの MP に対して複数のブロックを割り当てることも可能であるが、割り当て可能なブロック数はレジスタ・共有メモリ等のリソース量で制限される。また、ブロックの割り当ては CUDA runtime が行うためユーザは制御できない。

各ブロックは複数スレッドで構成されており、各スレッドが SP 上で実行される。通常、各 MP には SP 数を越える多数のスレッドが割り当てられており、MP は実行可能なスレッドを選択して実行させる。実際のスレッド選択はワープと呼ばれる 32 個のスレッドグループを単位に行われる。同一ワープ内のスレッドは命令レベルで同期しており SIMD 的に実行されるが、ワープ間のスレッドは命令レベルでは同期しない。CUDA の実行モデルは SIMD ではなく SPMD(Single Program Multiple Data) である。

CUDA 上でメモリバンド幅依存のプログラムを高速化する手法としては、文献 6) に記載の共有メモリ使用によるメモリアクセス量削減、Coalesced Access によるメモリバンド幅向上が良く知られており、これら手法はいろいろなプログラムにてその効果が確認されている。

表 1 CUDA 対応 GPU のスペック
Table 1 CUDA GPU specification

	8800GTX	GTX280
Core	G80	G200
# SP	128	240
# MP	16	30
# registers /MP	8,192	16,384
shared memory /MP (KB)	16	16
max active threads /MP	768	1,024
Mem. size (MB)	768	1,024
Mem. I/F (bits)	384	512
Mem. freq (GHz)	1.800	2.214
Mem. peak BW (GB/s)	86.4	141.7

3. GPU のメモリアクセス特性の調査

流体アプリケーションは基本的にメモリバンド幅依存であり、GPU で流体アプリケーションを高速化することは、GPU 上で高いメモリバンド幅を実現することとほぼ等価である。本章では、GPU のグローバルメモリのアクセス特性に関して述べる。

3.1 バンド幅

GPU のメモリバンド幅は、理論ピーク値が高いだけでなく実効値も高いことが知られており、条件が整えば、理論ピーク値の 88%に達する実効メモリバンド幅を実現できる[2)]。しかし、どのような条件でも高い

```
__global__ void mcopy(float *dst, float *src,
                      int size, int n_copy)
{
  int id = blockIdx.x*blockDim.x+threadIdx.x;
  int step = gridDim.x*blockDim.x;
  int n_total = size/sizeof(float);
  int n_each = n_total/n_copy;
  for (int i=id; i<n_each; i+=step)
      for (int j=i; j<n_total; j+=n_each)
          dst[j] = src[j];
}
```

図 3 使用したメモリコピールーチン
Fig. 3 gpu memory copy routine

メモリバンド幅が出る訳ではない。メモリバンド幅依存のプログラムを高速化するには、高いメモリバンド幅が出る条件を把握することが重要である。

以下、各種条件にてメモリコピー実施時のメモリバンド幅測定結果について述べる。

3.1.1 ストリーム数・バッファサイズとバンド幅

同時コピー数 (ストリーム数) の増減、転送量 (バッファサイズ) の増減により GPU のメモリバンド幅がどう変化するか測定した結果を図 1 に示す。図 1 は、横軸がバッファサイズ (転送元+転送先)、縦軸がメモリバンド幅である。同時コピー数は 1、4、16、64 の 4 種類で行った。測定に用いたプログラムは図 3 に示す通りで、データ型は float(4B) である。ブロックあたりのスレッド数は 256、MP あたりのブロック数は 2 とし、高バンド幅の出る条件下で測定した。

ストリーム数増によりバンド幅が低下することは文献 4) で報告されており、図 1 からもそれを確認できる。更に、図 1 よりバッファサイズ増でもバンド幅が低下していることが分かる。ストリーム数増・バッファサイズ増によるバンド幅低下は、世代の異なる 8800GTX と GTX280 の双方で観測されており、CUDA GPU に共通する特徴と言える。しかし、nVidia 提供のメモリコピールーチン (cudaMemcpy) ではバッファサイズ増によるバンド幅低下は発生していない。何か対処策があると思われる。

3.1.2 スレッド数とバンド幅

スレッド数の増減により GPU のメモリバンド幅がどう変化するか測定した結果を図 2 に示す。図 2 は、横軸が総スレッド数、縦軸がメモリバンド幅である。ブロックあたりのスレッド数は 128、256、512 の 3 種類、バッファサイズ (転送元+転送先) は 256MB とした。測定に用いたプログラムはスレッドあたりレジスタを 16 本使用するので、同時実行可能なスレッド数は☆、8800GTX が 8,192、GTX280 が 30,720 である。

図 2 より、全般的には総スレッド数が増えるほどバンド幅は上昇するが、総スレッド数によるバンド幅変動が大きく、特に 8800GTX ではバンド幅が不安定である。ブロックあたりのスレッド数に関しては、どれが適切かこの結果からは判断できない。また、どの条件でも cudaMemcpy の性能に達していない。

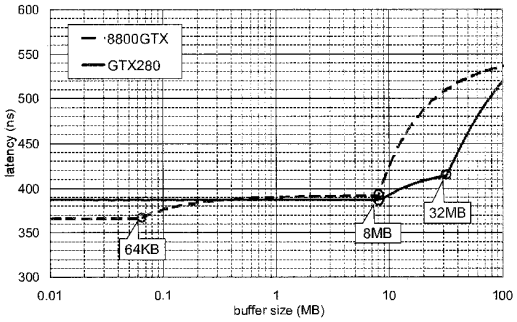

図 4 グローバルメモリ latency (ランダムアクセス, 1 ブロック x 32 スレッド)
Fig. 4 Global memory latency (random access, 1 block x 32 threads)

表 2 PTC ヒット/ミス時のメモリアクセス遅延
Table 2 memory latency in case of PTC hit and miss

(unit:ns)	8800GTX		GTX280	
L1 PTC hit	366	(64KB)	388	(8MB)
L2 PTC hit	392	(8MB)	424	(32MB)
PTC miss	550		582	

3.2 アクセス遅延

一般に、高バンド幅を得るにはアクセス遅延は短い方が良い。ランダムアクセス時の GPU のメモリアクセス遅延を測定した結果を図 4 に示す。図 4 は、横軸がバッファサイズ、縦軸がアクセス遅延である。ブロック数は 1、スレッド数は 32 とした☆☆。8800GTX は 64KB と 8MB、GTX280 は 8MB と 32MB、どちらも 2 箇所の境界で挙動が変わる。TLB 存在の可能性は文献 2) で報告されているが、図 4 より CUDA GPU には 2 レベルの TLB 相当の機能が存在する可能性が高い。仮に本稿ではこれを PTC(Page Translation Cache) と呼ぶことにする。

図 4 より推定される PTC ヒット/ミス時のメモリアクセス遅延を表 2 に示す。PTC ミスが発生すると、PTC ヒット時と比べてメモリアクセス遅延が 160〜200ns 程度増加する。一般に、アクセス遅延増はバンド幅低下を引き起こす。高いバンド幅を得るには PTC ミスを回避する必要がある。

4. GPU で高バンド幅を実現する手法の提案

4.1 低バンド幅問題の考察と対策

メモリアクセス遅延調査の結果、PTC ヒット/ミスでは 160〜200ns の遅延差があることが分かった。メモリバンド幅調査において、ストリーム数増・バッファサイズ増でバンド幅が低下する、cudaMemcpy の性能に届かない、総スレッド数によりバンド幅が変動するという問題が観測されている。これら問題の主要因

☆ MP 数 * レジスタ数 (/MP) / 使用レジスタ数 (/スレッド)
☆☆ ワープ数が 1 となるようスレッド数を設定

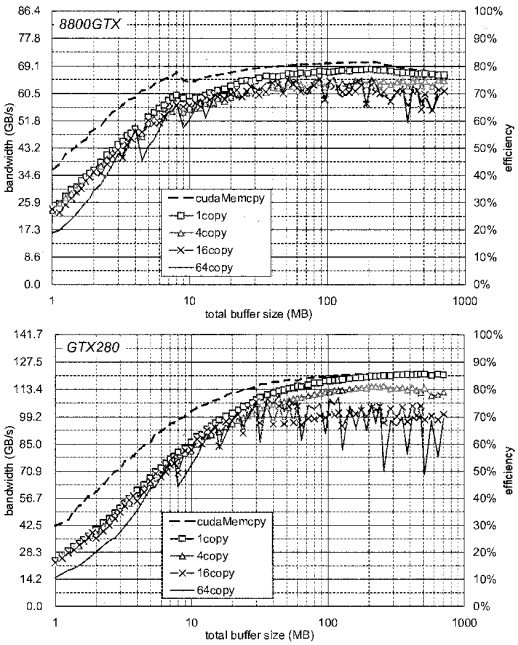

図 5 ストリーム数・バッファサイズとメモリバンド幅 (256 スレッド/ブロック、2 ブロック/MP、__syncthreads() 使用)

Fig. 5 # of streams, size of buffer and memory bandwidth (256 threads/block, 2blocks/MP, using __syncthreads())

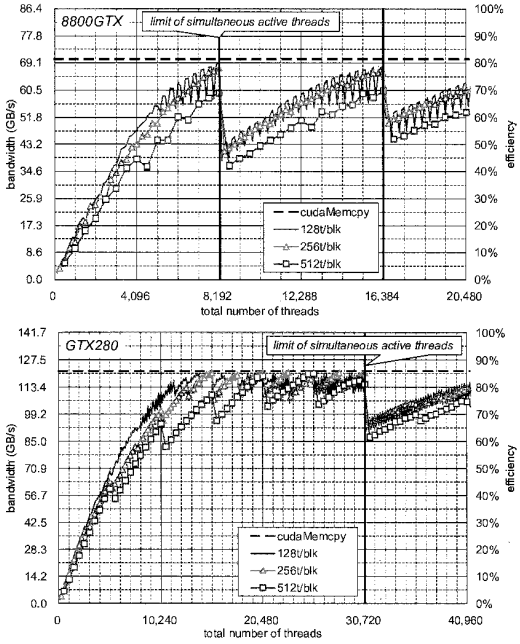

図 6 スレッド数とメモリバンド幅 (バッファサイズ:256MB、__syncthreads() 使用)

Fig. 6 # of threads and memory bandwidth (buffer size:256MB, using __syncthreads())

は PTC ミスであると我々は考える。

我々の仮説 (バンド幅低下が発生するシナリオ) は次の通りである。CUDA の実行モデルは SPMD なのでスレッドは同期して動いておらず、進行の速いスレッドと遅いスレッドが存在する。時間が経過するほどスレッド間の進行差が広がる。本来は局所的なメモリアクセスを行うプログラムでも、スレッド間の進行差が広がるとメモリアクセス箇所が広がって単位時間あたりにアクセスするページ数が増加する。アクセスページ数が PTC エントリ数を越えると PTC ミスが発生し、バンド幅が低下する。

このシナリオが正しいとすると、スレッドの進行を同期させることでバンド幅低下を解消できる。CUDA には異ブロック間のスレッドを同期させる機能は無いが、ブロック内のスレッドは __syncthreads() で同期可能である。これを頻繁に呼び出すことでスレッド進行を同期状態に近づけることができる。各スレッドがバッファの要素を 1 つコピーする度に __syncthreads() を呼び出した場合のメモリバンド幅測定結果を図 5 と図 6 に示す。

図 5 より、__syncthreads() 使用により、ストリーム数増・バッファサイズ増によるバンド幅低下は大幅に軽減されることが分かる。ストリーム数増によるバンド幅低下は若干残っているが、__syncthreads() 未使用時 (図 1) と比べるとバンド幅低下は小さい。また、バッファサイズが大きいときは cudaMemcpy 相当の性能が出ている。

図 6 より、__syncthreads() 使用により、総スレッド数によるバンド幅変動が小さくなることが分かる。また、ブロックあたりのスレッド数は 128・256 が適切である。これは文献 6) の説明と一致する結果である。

更に、図 6 より総スレッド数とバンド幅の関係が良く分かる。CUDA 環境では、総スレッド数が同時実行可能スレッド数を越える場合、最初に MP に割り当てられるスレッドは同時実行可能スレッド数で制限され、先行スレッドの実行が完了すると、待機スレッドが MP に割り当てられ実行される。従って、総スレッド数次第でメモリバンド幅が変わる可能性がある。例えば、総スレッド数が同時実行可能スレッド数を若干上回る場合、一巡目は実行スレッド数が多く高バンド幅を得られるが、二巡目は実行スレッド数が少なく高バンド幅を得られない。二巡目がネックとなり全体のバンド幅が低下する。この傾向は図 6 に明確に表れている。

総スレッド数は高バンド幅を実現するためには重要であり、常に十分な数のスレッドが実行されるよう、総スレッド数を設定するのが望ましい。

4.2 提案手法

CUDA 上で高バンド幅を実現する方法を提案する。

4.2.1 スレッド進行の同期化

CUDA の実行モデルは SPMD であり、スレッド間に進行差が生じることにより PTC ミスを誘発する特性を持つ。CUDA には全スレッドを同期させる機能はないが、__syncthreads() により同ブロック内のスレッドは同期可能である。これを頻繁に呼び出すと、スレッド進行を同期状態に近づけることができ、PTC ミスが減り高バンド幅を得ることができる。

4.2.2 アクセスパターンの局所化

メモリアクセスパターンが複雑だと、単位時間あたりにアクセスするページ数が多く、PTC ミス発生の可能性が高い。メモリアクセスパターンが逐次的、もしくは局所的なアルゴリズム・データ構造を選択することで、PTC ミスが減り高バンド幅を得ることができる。

4.2.3 スレッド数の最適化

CUDA ではスレッド数が少ないと高バンド幅を得ることができない。特にピークバンド幅の高い GTX280 ではその傾向が顕著で、例えば GTX280 で 100GB/s のバンド幅を得るには 10,000 スレッド必要である (図 6)。常に十分な数のスレッドが実行されるアルゴリズムを選択することで、高バンド幅を得ることができる。

5. 提案手法の姫野ベンチマークへの適用

姫野ベンチマーク[7] (以下、姫野ベンチ) は、流体アプリケーションで良く用いられる処理、Poisson 方程式解法時の性能を測定するベンチマークである。本章では、GPU 上で高バンド幅を実現する方法を姫野ベンチに適用した結果について述べる。

5.1 従来実装

GPU 上の姫野ベンチ高速化事例としては文献 5) が知られており、これに基づく実行ファイル (M モデル) は Web 上に公開されている[8]。以下、姫野ベンチの特徴、文献 5) の実装方法を簡単にまとめる。

- 姫野ベンチには同サイズの 3 次元配列が 14 個存在。12 個は係数用の配列で参照用。2 個は圧力変数用の配列で参照/更新用。
- 圧力変数には更新前と更新後があり、サイクル毎に入れ替わる。更新前圧力変数は参照用で、これは隣接参照される (共有メモリによるメモリアクセス量削減の対象)。更新後圧力変数は更新用。
- 各 3 次元配列はブロック形状 (16, 16, 8) で分割される。姫野ベンチ (M モデル) の配列サイズは (256, 128, 128) なので、総ブロック数は 2,048。
- 各ブロックは 256 スレッドで処理される。各スレッドが担当する格子点は 8 個。
- ブロック内のスレッドは、格子点計算を開始する前に、スレッド間で共用するサイズ (16+2, 16+2, 8+2) の圧力変数を共有メモリにロードする。ブロックあたりの共有メモリ使用量は 12,960B。

5.2 高速化

文献 5) の実装方法に対して、4 章で提案した GPU 上で高メモリバンド幅を実現する方法を適用する。以下、実際に適用した高速化手法を説明する。

5.2.1 スレッド進行の同期化 (同期処理の多用)

各スレッドが 1 格子点の計算を終える度に、__syncthreads() を呼び出す。スレッド進行が同期状態に近くなり、PTC ミスが減る。

5.2.2 アクセスパターンの局所化 (配列の次元入替え)

姫野ベンチには 3 次元の係数配列が 12 個存在する。これらは同じ形状なので、4 次元配列が 1 つ存在すると考えても良い。この 4 次元配列の次元を次のように入れ替える。

次元入替前: `coef[12][MIMAX][MJMAX][MKMAX]`

次元入替後: `coef[MIMAX][12][MJMAX][MKMAX]`

メモリアクセスが局所化され、PTC ミスが減る。

(ブロック形状変更)

ブロック内のメモリ連続方向が長くなるよう、ブロック形状を (16, 16, 8) から (64, 4, 8) に変更する。メモリアクセスがより逐次的になる。

5.2.3 スレッド数の最適化 (同時実行スレッド数の増加)

ブロック形状が (64, 4, 8) の場合、1 ブロックあたりの共有メモリ使用量は 15,840B である。MP あたりの共有メモリ量は 16KB なので、共有メモリ量がネックとなり、各 MP に同時に割り当て可能なブロック数は 1 に制限される。ブロックあたりのスレッド数は 256、GTX280 の MP 数は 30 なので、この場合の同時実行スレッド数は 7,680 である。図 6 より、これでは同時実行スレッド数が不十分で、高バンド幅が得られない。

同時実行スレッド数を増やすため、ブロックあたりの共有メモリ使用量を減らす。格子点計算を開始する前に、ブロック内で参照される全ての圧力変数を共有メモリにロードしているが、これを改め、格子点計算の途中に必要な圧力変数だけ共有メモリにロードする。この場合、共有メモリはサイズ (64+2, 4+2, 3) の圧力変数を格納できる領域があれば十分であり、ブロックあたりの共有メモリ使用量は 4,752B で済む。各 MP に同時に割り当て可能なブロック数は 1 から 3 に増え、同時実行スレッド数は 23,040 となる (GTX280)。図 6 より、これなら高バンド幅を期待できる。

なお、z 軸方向は分割する必要が無くなるので、ブロック形状は (64, 4, 128) とする。

(総スレッド数調整)

ブロック形状 (64, 4, 128) の場合、総ブロック数は 128、総スレッド数は 32,768 である。同時実行スレッド数は 23,040 なので (GTX280)、一巡目の実行スレッド数は 23,040、二巡目は 9,728 となる。スレッド数 9,728 では最高バンド幅は得られない (図 6)。

実行スレッド数を多くするため、ブロック形状を (64, 4, 64) とする。これで総ブロック数は 256、総スレッド数は 65,536 に増える。この場合、一巡目・二巡目の実行スレッド数は 23,040 スレッド、三巡目は 19,456 スレッドとなり、常に十分なスレッド数が存在するので、高バンド幅を期待できる。

5.2.4 その他 (パディング量調整)

汎用 CPU では、配列間のパディング量を調整することによりバンド幅が変化することが知られている。GPU でも同様の効果が期待できる。係数配列 coef[MIMAX][12][MJMAX][MKMAX] を、サイズ MKMAX*MJMAX の要素が 12*MIMAX 個存在すると捉え、この要素間にパディングを挿入する。パディング量は 0〜16KB の範囲で調査したところ、8800GTX では 15,360B、GTX280 では 12,800B のときに最高性能が得られた。

5.3 性能測定と評価

従来実装[8]、並びに高速化手法を順次適用した場合の姫野ベンチ (M モデル) 性能測定結果を表 3 に示す。テスト環境は表 4 の通りである。全ての高速化を適用した場合、8800GTX では 36.1GFLOPS、GTX280 では 69.7GFLOPS の性能を達成した。これは従来実装と比べて 1.69 倍の性能である。なお、従来実装はベース実装より高い性能が出ており、本稿で述べた高速化手法のどれかが適用されている可能性が高い。

姫野ベンチは 1 格子点あたり 14 変数のメモリをアクセス、データ型は float(4B) なので、メモリアクセス量は 1 格子点あたり 56B である。演算量は 1 格子点あたり 34FLOP なので、1 演算あたりのメモリアクセス量は 1.647B/FLOP である。従って、GTX280 で 69.7GFLOPS を達成したときには 115GB/s のバンド幅が出たことになる。これは理論ピークメモリバンド幅 (141.7GB/s) の 81%に相当する。また、図 5 より、バッファサイズ 224MB☆の GTX280 の最大実効メモリバンド幅は 120GB/s であり、ほぼ最大実効メモリバンド幅を引き出せている。

姫野ベンチは実アプリケーションに近いメモリアクセスを行うプログラムである。上記結果は、適切な高速化を施せば、実アプリケーションでも最大実効メモリバンド幅に近い性能を実現できることを示している。

表 3 姫野ベンチ (M モデル) 性能測定結果
Table 3 results of himeno benchmark (model:M)

	姫野ベンチ性能 (GFLOPS)	
	8800GTX	GTX280
従来実装[8]	21.4	41.2
ベース実装	14.4	24.7
[スレッド進行の同期化]		
同期処理の多用	20.3	29.1
[アクセスパターンの局所化]		
配列次元入替	20.6	38.9
ブロック形状変更	24.2	44.9
[スレッド数の最適化]		
同時実行スレッド数の増加	35.3	62.3
総スレッド数調整	35.3	67.3
[その他]		
パディング量調整	36.1	69.7

表 4 テスト環境
Table 4 testing environment

CPU	AMD Phenom 9600 (2.3GHz)
M/B	Gigabyte GA-MA790FX-DQ6 (AMD 790FX)
Mem	4GB (2 x 2GB DDR2-800 DIMM)
GPU	Leadtek WinFast PX8800 GTX (8800GTX)
	Leadtek WinFast GTX280 (GTX280)
OS	CentOS 5.1 (x86_64, kernel:2.6.18-53.el5)
SW	nVidia Linux driver 177.67
	nVidia CUDA 2.0 (x86_64, for RHEL5.1)
	gcc 4.1.2

6. まとめ

これまで、GPU のメモリアクセス特性は不透明な部分が多く、高いメモリバンド幅を実現するには試行錯誤を繰り返す必要があった。本稿では各種メモリアクセス性能調査結果より、GPU 上で安定して高いメモリバンド幅を実現するには、スレッド進行の同期化、アクセスパターンの局所化、スレッド数の最適化が重要であることを明らかにした。

この方法に基づいて、流体アプリケーションのベンチマークプログラムである姫野ベンチマークを高速化した結果、GTX280 上で 69.7GFLOPS の性能を達成した。これは従来実装と比べて 1.69 倍の性能であり、この性能達成時には 115GB/s の実効メモリバンド幅が出ていたことを意味する。これは理論ピークメモリバンド幅 (141.7GB/s) の 81%に相当する。適切な高速化を施すことで、実アプリケーションでも高いメモリバンド幅実現が可能であることを示した。

参考文献

1) General-Purpose Computation Using Graphics Hardware: http://www.gpgpu.org/
2) V. Volkov, J. Demmel: LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs, LAPACK Working Note 202 (2008).
3) R. Belleman, J. Bedorf, S. Zwart: High Performance Direct Gravitational N-body Simulations on Graphics Processing Units - II: An implementation in CUDA, ArXiv:0707.0438v2 [astro-ph] (2007)
4) 額田 彰, 尾形 泰彦, 遠藤 敏夫, 松岡 聡: CUDA 環境における高性能 3 次元 FFT, 先進的基盤システムシンポジウム (SACSIS2008), pp. 81-88 (2008).
5) 小川 慧, 青木 尊之: CUDA による定常反復 Poisson ベンチマークの高速化, 情報処理学会研究報告会, 2008-HPC-115, pp. 19-23 (2008).
6) nVidia: CUDA Programming Guide 2.0, http://www.nvidia.com/object/cuda_develop.html
7) 姫野ベンチマーク: http://accc.riken.jp/HPC/HimenoBMT/
8) 2007 年度 理研ベンチマークコンテスト優勝 実行ファイル: www.nr.titech.ac.jp/~taoki/Whatsnew/bench-riken.html

☆ 姫野ベンチ (モデル M) の総配列サイズ