CUDA 環境における高性能3次元FFT

額田 彰[†1,†2] 尾形 泰彦[†1,†2]
遠藤 敏夫[†1,†2] 松岡 聡[†1,†3,†2]

NVIDIA の最新 GPU がサポートする CUDA 環境では共有メモリを用いたスレッド間のデータ共有と, 自由度が高いメモリアクセスが可能である. 我々はこの CUDA 環境に適した高性能 3 次元 FFT アルゴリズムを提案する. GeForce 8 シリーズの GPU を用いた 3 次元 FFT において, CUFFT ライブラリ 1.1 と比較して 3.1〜3.3 倍, 最大 79.5GFLOPS の演算性能を達成した.

High Performance 3-D FFT in CUDA environment

Akira Nukada,[†1,†2] Yasuhiko Ogata,[†1,†2] Toshio Endo[†1,†2]
and Satoshi Matsuoka [†1,†3,†2]

CUDA environment, which is supported in latest NVIDIA GPUs, allows data sharing between threads using shared memory, and also provides more flexible memory accesses. We propose a high performance 3-D FFT algorithm for the CUDA environment. Using GeForce 8 series GPUs, we achieved a high performance up to 79.5 GFLOPS at 3-D FFT, which is from 3.1 to 3.3 times the performance compared with the performance of CUFFT library 1.1.

1. はじめに

Graphics Processing Unit(GPU) は 3 次元空間の画像をレンダリングする等の負荷が重い処理を高速化するアクセラレータであり, 古くからグラフィックワークステーションに搭載されていたが今日では非常に身近なものとなっている. GPU は単純な計算処理を多数繰り返すことに秀でたアーキテクチャであり, そのピーク浮動小数演算性能, メモリバンド幅ともに CPU のそれを遥かに上回る. また CPU の消費電力が高騰する傾向にある今, GPU の高い電力効率が注目されている. このため GPU を科学技術計算等に用いる事例が幾つか存在するが, GPU のアーキテクチャに適合する, グラフィック処理に似た計算に限られていた. 近年 General-Purpose computation on Graphics Processing Unit(GPGPU)[1] という, GPU を使ってより自由度の高い計算を行う技術に注目が集まっている. 開発に関しても従来は NVIDIA の Cg[2], Microsoft の High Level Shader Language(HLSL) 等のシェーダプロセッサ専用言語が用いられていたが, C 言語を拡張した BrookGPU[3] によって GPU を使用するプログラムを高級言語を用いて記述できるようになり, ソフトウェアの開発が容易にできるようになった. NVIDIA は新しい GPU アーキテクチャである CUDA (Compute Unified Device Architecture)[4] をリリースしたが, これもまた C/C++言語を拡張したプログラミング言語が提供されている.

GPU を用いた科学技術計算としては以前より N 体問題[5], 行列積[6] などメモリアクセス量に対して演算数が多いアプリケーションを対象とする研究が多く行われてきた. このようなアプリケーションは GPU による実行に適しており, GPU の高い演算性能を容易に活用できる.

高速フーリエ変換 (FFT)[7] は様々なアプリケーションで重要な役割を担う. 本研究では特に 3 次元空間のシミュレーション等で用いられる複素 3 次元 FFT の計算を CUDA 環境で高性能に行う手法を提案する. FFT の計算では $O(N \log N)$ の演算量に対して $O(N)$ のメモリアクセスが必要であり, メモリアクセスがボトルネックとなることが多い. このため CPU と比べて飛躍的に高いメモリバンド幅を持つ GPU で計算することによって高速化が可能であると考えられる. し

†1 東京工業大学
Tokyo Institute of Technology.
†2 科学技術振興機構 戦略的創造研究推進事業
Japan Science and Technology Agency, Core Research for Evolutional Science and Technology.
†3 国立情報学研究所
National Institute of Informatics.

表 1 NVIDIA GeForce 8 シリーズの諸元.

					SP		Memory			
Model	Core	Process	MP	#	clock	GFLOPS	Capacity	Interface	Clock	B/W
8800 GT	G92	65nm	14	112	1.500 GHz	336	512MB	256-bit	1800MHz	57.6 GB/s
8800 GTS	G92	65nm	16	128	1.625 GHz	416	512MB	256-bit	1940MHz	62.0 GB/s
8800 GTX	G80	90nm	16	128	1.350 GHz	345	768MB	384-bit	1800MHz	86.4 GB/s

かしながら現状 FFT では十分に GPU の性能を活用できていない[8),9)]. そこで提案手法ではメモリアクセス性能を中心に最適化を行う. 現在は 512MB のメモリを搭載する GPU 製品が主流であり, 今回はサイズ 256^3 を中心に単精度複素 3 次元 FFT の性能評価を行う.

2. NVIDIA CUDA

従来の GPU ではシェーダプロセッサ間での通信が不可能であったため各プロセッサは個別のデータに対して同じ処理を行うストリーム処理に限定されていた. CUDA 環境では共有メモリによってスレッド間の通信が可能となり, より複雑な計算を行うことができるようになった. またメモリアクセスの自由度が高く, 広範囲な計算を対象とする.

CUDA をサポートする最初の GPU は G80 コアを搭載する GeForce 8800 GTX であった. GeForce 8800 GTX は 16 個のマルチプロセッサ (MP) 群を搭載し, この各マルチプロセッサは Streaming Processor (SP) と呼ばれるプロセッサコア 8 個, 16kB の共有メモリ, 8192 個のレジスタ, 定数キャッシュメモリ, テクスチャキャッシュメモリ等から構成される. 8800 GTX は合計 128 個の SP をもつ超並列プロセッサである.

CUDA も従来の GPU と同様に各 SP で同じ命令を実行する SIMD(Single Instruction Multiple Data) 型プロセッサである. CUDA のアーキテクチャでは現在各マルチプロセッサは最大 768 個のスレッドがアクティブになる. このように多数のスレッドを実行することによってレジスタやメモリへアクセスする時の遅延を隠蔽している. 実際には 32 スレッド毎に Warp と呼ばれる単位で管理され, 最大 24Warp がアクティブな状態になる. 1 つの Warp に属する 32 個のスレッドは 8 スレッドずつ順番に 8 個の SP に投入されて実行される.

スレッドはスレッドブロックにグループ化される. 各スレッドブロックに属する全てのスレッドは一つのマルチプロセッサで実行され, 共有メモリを介してスレッド間のデータ交換が可能である. 各スレッドブロックが使用するスレッド数, 総レジスタ数, 共有メモリサイズによって各マルチプロセッサで同時に実行されるスレッドブロックの数が自動的に決定される.

表 2 ストリーム数とメモリバンド幅の関係.

	Bandwidth (GB/s)	
# of streams	8800 GT	8800 GTX
1	48.1	71.7
2	47.6	71.9
4	47.4	71.8
8	36.4	65.4
16	27.8	43.7
32	25.7	37.5
64	20.2	32.3
128	22.6	35.4
256	18.4	30.7

現時点でリリースされている CUDA 対応 GPU は基本的には同じアーキテクチャ構成を持ち, コア及びメモリの動作クロック周波数, マルチプロセッサの数, メモリ容量, メモリバンド幅, PCI-Express のバージョン等にバリエーションがある. また浮動小数演算に関しては 2008 年 1 月の時点で入手可能なハードウェアは単精度のみに対応する.

CUDA 環境に対応する GPU は GeForce 8 シリーズ以降である. その中で入手できたモデルの諸元を**表 1** に挙げる. 本論文中で 8800 GTS は G92 コアと 512MB のメモリを搭載するモデルを指し, 8800 GTS 512 と呼ばれることもある. G92 コアは 65nm プロセスで製造されており, 90nm プロセスの G80 コアよりも高い電力効率が期待できる. これらの 3 種類の GPU の中では 8800 GT の性能が最も低く, メモリバンド幅に関しては 8800 GTX が, SP の演算性能では 8800 GTS が一番高い. 表 1 中の GFLOPS 値に関しては, FFT の計算で使用する SP の積和演算器の演算性能のみを計上する. 最新の AMD Phenom 9500 プロセッサ (2.2GHz) の演算性能は 4 コアを合計して単精度で 70.4GFLOPS, 倍精度で 35.2GFLOPS 程であり, 8800 GTS はその約 6 倍の演算性能を持つ.

GPU は高速なデバイスメモリを搭載するため, ベクトル計算機等に有効な multirow FFT[10),11)] アルゴリズムが適用できると考えられる. multirow FFT は複数の組の FFT を同時に計算する手法で, 3 次元 FFT において各次元方向の 1 次元 FFT を多数計算する場

合にも用いることができる. 各組のデータをベクトルレジスタの各要素に割り当てることによってベクトル演算化することが可能であり, GPU のように多くの SP を持つ SIMD 型のプロセッサにも適している.

multirow FFT アルゴリズムでは複数ストリームのメモリアクセスが必要となるため, ここで性能を評価しておく. **表 2** に複数のストリームでメモリコピーを行った場合のメモリバンド幅を示す. 各スレッドブロックのスレッド数は 64, スレッドブロックの数はマルチプロセッサ数の 3 倍, すなわち 8800 GT では 42, 8800 GTX では 48 とする. 各ストリームがアクセスするデータ量の合計を 128MB に固定する. 計測には以下のコードを用いた. 複素数に適したデータ型として, 2 個の float 型からなる float2 型を用いている. STREAMS は読み込むストリームの数である. 複数ストリームのコピーするタスクを全スレッドでサイクリック分割して実行している. 実際には各スレッドは SIMD 型に命令を実行するため, スレッド数分のブロック単位で各ストリームに対して順番にアクセスするような挙動になるため, ストライドアクセスとは異なる.

```
__global__ void
cuda_memcpy(float *s, float *d)
{
  int index = __mul24(blockIdx.x,blockDim.x)
              + threadIdx.x;
  int step = __mul24(gridDim.x, blockDim.x);
  int n = 256 * 256 * (256 / STREAMS);

  float2 src1 = (float2 *)s, dst1 = (float2 *)d;
  float2 src2 = src1 + n, dst2 = dst1 + n;
  float2 src3 = src2 + n, dst3 = dst2 + n;
  .....

  for (int i = index; i < n; i += step) {
    dst1[i] = src1[i];
    dst2[i] = src2[i];
    ......
    dstSTREAMS[i] = srcSTREAMS[i];
  }
}
```

同時にアクセスするストリームの数が増えるに従ってメモリバンド幅が低下する傾向にある. ストライドアクセスした場合のメモリバンド幅が 10GB/s 以下になることと比較すると複数ストリームのメモリコピーは 256 ストリームでも速度低下が小さい.

2.1 Coalesced Memory Access

各 SP はそれぞれ個別のアドレスへのメモリアクセス命令を実行することができるが, Half-Warp, すなわち 16 スレッドがスレッド番号順に連続するメモリアドレスへアクセスし, かつ先頭スレッドのアクセスするアドレスが適切にアライメントされている場合にひとつのメモリアクセス処理に融合される. 各スレッドのアクセスするデータ型のサイズは 4byte, 8byte, 16byte の何れかである必要があり, 先頭スレッドのアクセスするアドレスはそれぞれ 64byte, 128byte, 256byte 境界にアラインメントされている必要がある.

このような条件を満たさない場合には同じブロックに対する処理であっても別個にデータ転送が行われるためメモリアクセスの効率が著しく低下する. また GPU 用のメモリである GDDR メモリも連続するアドレスへのアクセスに最適化されているため Coalesced Memory Access 条件を満たすだけでは不十分であり, 高いスループットを得るためには複数の Warp やスレッドブロックから隣接するメモリアドレスへ Coalesced Memory Access を行うことによってより大きなブロック単位でメモリアクセスを行う必要がある. このため CUDA では記述可能なメモリアクセスの自由度が高いが, 効率の良いメモリアクセスパターンは事実上制限される.

3. CUDA 環境に適した 3 次元 FFT

3 次元 FFT では 3 次元の入力データの各次元方向に 1 次元 FFT を行う. キャッシュメモリを搭載する CPU で 3 次元 FFT の計算を行う場合, 低速な主記憶へのアクセスを最小限に抑えるためにキャッシュメモリを有効に活用することが重要である. 3 次元配列の各軸を X,Y,Z とし, X 軸方向のデータが連続アドレスに格納されている場合, Y 軸, Z 軸方向のデータを主記憶とキャッシュメモリの間でコピーする際にストライドアクセスが必要となる. このストライドアクセスは, 連続アドレスへのアクセスと比べると格段に転送速度が低下する. 複数ライン分のデータを一度にアクセスすることによってこの速度低下をある程度軽減することは可能であるが, キャッシュメモリの容量によって一度にキャッシュメモリに置けるライン数が制限される.

CUDA の GPU では各マルチプロセッサ内に複数スレッドで共有可能な共有メモリがあり, これを有効活用するアルゴリズムが考えられる. しかしながら Coalesced Memory Access の条件を満たすためには Half-Warp である 16 個のスレッドが連続アドレスへアクセスする必要がある. 最小でも 32-bit の float 型の場合の $4\text{byte} \times 16 = 64\text{byte}$ のブロック単位でアクセスする. 256 ブロックにアクセスした場合には合計

16kB となり共有メモリの容量を超えてしまう．[⋆1]

3.1 提案手法

3 次元 FFT の計算においてストライドアクセスを使わずに全て連続アドレスへのアクセスを利用する手法を提案する．X 軸方向の計算には CPU での実装と同様に順番に共有メモリにコピーして共有メモリ上で 1 次元 FFT を計算する．一方 Y 軸方向と Z 軸方向の計算には multirow FFT アルゴリズムを用いるという手法である．

multirow FFT アルゴリズムを用いた場合, スレッド間のデータ共有・通信は一切必要ない．その代わりにデータを各スレッドが保持する必要がある．256 点 FFT に対して multirow FFT アルゴリズムを適用した場合, 少なくとも $512+\alpha$ 個のレジスタが必要となり, この場合端数切り上げで 1024 レジスタが確保され各マルチプロセッサ内で 8 スレッドしか同時に実行することができず, Coalescing Memory Access の条件を満たさないためメモリアクセス性能が低下する．

そこで 256 点 FFT を 2 回の 16 点 FFT に分解し, それぞれの 16 点 FFT に対して multirow FFT アルゴリズムを適用することにする．実際に実装してみたところ 51〜52 個のレジスタ数で実装可能であった．各マルチプロセッサで最大 128 スレッドを同時実行可能となり, 十分なメモリ転送速度が得られる．256 点 FFT を計算するためには 16 点 FFT を 2 回計算する必要があり, メモリアクセス回数が倍増する．8800 GT や 8800 GTX では 256 点 FFT の multirow FFT やストライドアクセスを用いた場合には転送速度が 10GB/s 以下に低下する．一方 16 点 FFT の multirow FFT では 38GB/s 以上の転送速度が得られるため, アクセス回数が増えるが結果としては短時間で実行可能である．Y,Z 軸方向の 1 次元 FFT に 16 点 FFT を用いる場合, 3 次元 FFT は次のような 5 つのステップで実現できる．

ステップ 1: 16 点 FFT(Z 軸方向 256 点 FFT の前半)
ステップ 2: 16 点 FFT(Z 軸方向 256 点 FFT の後半)
ステップ 3: ステップ 1 と同じ．(Y 軸方向)
ステップ 4: ステップ 2 と同じ．(Y 軸方向)
ステップ 5: X 軸方向の 1 次元 FFT の計算

より詳細な擬似コードを以下に示す．Y 軸,Z 軸方向のインデックスを 16×16 に分解し, 入力データを 5 次元配列 V(256,16,16,16,16) に格納するもとのし, 同様に一時的な作業領域として配列 WORK を用いる．(Fortran の多次元配列のように最初のインデックスを変化させたときにメモリアドレスが連続となるものとする．) 5 つの for 文はそれぞれ順番に各ステップに対応する．なお, 今回のように入力データの Y 軸方向と Z 軸方向のサイズが同じ場合にはステップ 1 と 3, ステップ 2 と 4 は全く同じ処理になる．

```
COMPLEX V(256,16,16,16,16),WORK(256,16,16,16,16)

for Z1,Y2,Y1,X
   WORK(X,*,Y1,Y2,Z1)=FFT256_1(V(X,Y1,Y2,Z1,*))
for Y2,Y1,Z2,X
   V(X,Z2,*,Y1,Y2)=FFT256_2(WORK(X,Z2,Y1,Y2,*))
for Y1,Z1,Z2,X
   WORK(X,*,Z2,Z1,Y1)=FFT256_1(V(X,Z2,Z1,Y1,*))
for Y2,Y1,Z2,X
   V(X,Y2,*,Z2,Z1)=FFT256_2(WORK(X,Y2,Z2,Z1,*))
for Z1,Z2,Y1,Y2
   V(*,Y2,Y1,Z2,Z1)=FFT256(V(*,Y2,Y1,Z2,Z1))
```

ここで FFT256() は 256 点 FFT の計算で, FFT256_1() と FFT256_2() は 256 点 FFT の前半と後半の 16 点 FFT を意味する． “for Z1,Y2,Y1,X” は Z1,Y2,Y1,X に対する 4 重ループを指すが, 4 重ループのオーバーヘッドは非常に大きいので実際には (Z1*Y2*Y1*X) 回反復を行う 1 重ループでインデックス変数に対する論理演算に置き換えることで実現する．メモリアクセスが連続になり, Coalesced Memory Access 条件も満たすように, ステップ 1〜4 ではループを各スレッド, 各スレッドブロックにサイクリック分割で割り当てる．一方ステップ 5 では各スレッドブロックにサイクリック分割で割り当て, スレッドブロック内の各スレッドで並列に 256 点 FFT の計算を実行する．

上に示した提案手法の擬似コードでは 16 点 FFT の計算をした後メモリに書き戻す際に 5 次元配列に対して転置処理を行っている． 256 点 FFT を 16 点 FFT2 回で行っているため, 入力データの配列 V(X,Y1,Y2,Z1,Z2) をある種の転置によって最終的に V(X,Y2,Y1,Z2,Z1) という配列に出力する必要があるが, 提案手法では必要以上に多くの転置操作を行っている．少しずつインデックス変換をしているという点では Stockham の自動ソートアルゴリズム[11] に似ているがその意味は全く異なる．これはメモリバンド幅が最適になるようなメモリアクセスパターンを選んでいるためである．

Y 軸及び Z 軸方向の 16 点 FFT を計算するために入出力データにアクセスするパターンには**表 3** に挙げるような A から D の 4 パターンがあり得る．入力と出力のパターンの組み合わせによって, **表 4**, **表 5** のようにメモリアクセス性能が変化する．パターン C,D

⋆1 共有メモリの容量は 16kB であるが全部を利用することはできない．

表 3 メモリアクセスパターン.

A	(256,*,16,16,16)
B	(256,16,*,16,16)
C	(256,16,16,*,16)
D	(256,16,16,16,*)

表 4 入出力パターンとメモリバンド幅 (GB/s). 8800 GT で計測. スレッドブロック数は 42, 各スレッドブロックのスレッド数は 64.

	Output			
Input	A	B	C	D
A	47.4	47.9	46.8	47.1
B	48.2	48.3	46.8	47.1
C	47.3	47.1	34.4	33.3
D	45.6	45.2	32.6	27.8

表 5 入出力パターンとメモリバンド幅 (GB/s). 8800 GTX で計測. スレッドブロック数は 48, 各スレッドブロックのスレッド数は 64.

	Output			
Input	A	B	C	D
A	71.5	71.5	67.7	66.8
B	71.3	71.3	67.6	67.0
C	68.7	68.5	51.3	50.4
D	67.5	66.7	50.0	43.7

のみの場合に著しく性能低下が起こっている. 一方入力あるいは出力がパターン A,B であれば 1 ストリームのコピーに近い性能が出ている. パターン A,B ではストリーム間の距離が近く, 1 ストリームのアクセスと同じ状況になり高速なデータ転送が可能となるためである.

このような性質があるため, パターン C,D 間のメモリアクセスは避ける必要がある. また前述の 5 つのステップでは必ずステップ 2 より先にステップ 1 を, ステップ 4 より先にステップ 3 を実行する必要がある. この要件を満たす転置パターンが先に示した擬似コードのものである.

3.2 カーネルの実装

ステップ 1〜4 とステップ 5 ではカーネルに要求されるものが異なる. ステップ 1〜4 ではスレッド間でデータ交換を行う必要がなく, 共有メモリを使う必要がない. その代わり各スレッドが別々の 16 点 FFT 用のデータを保持する必要があり, 必要なレジスタ数が多い. 演算性能とメモリバンド幅のバランスからメモリアクセスがボトルネックとなる. メモリ転送速度をできるだけ上げるためにはスレッド数をある程度大きくする必要があり, その結果として各スレッドが利用可能なレジスタ数が制限される. SP の性能には余裕があるため, 定数メモリを多用することによって利用レジスタ数を削減している.

一方ステップ 5 では X 軸方向の変換を行うが, メモリアクセスが連続となるため 256 点 FFT 用のデータを高速に SP のレジスタへ転送することが可能である. しかしながら 64 スレッドがそれぞれ 256 点 FFT の計算を行うだけのレジスタ数はないため, 複数のスレッドで共有メモリを介してデータ交換をしながら一つの 256 点 FFT を計算する. SP の性能にはステップ 1〜4 ほどの余裕はなく, 共有メモリへのアクセスもできるだけ減らすべきである. 共有メモリへのアクセス回数を減らすためにはスレッド数を少なくするべきであるが, 64 スレッドより少ない場合には SP の性能を 100%利用することが出来なくなるため 64 スレッドを選ぶことにする. このとき各スレッドがデータを保持するために必要なレジスタはたったの 8 個である. レジスタ数にかなり余裕があり, SP の性能にはあまり余裕がないためステップ 5 では全ての定数データをレジスタに保持する. またスレッド間のデータ交換に用いる共有メモリは 16 バンクで構成されており, バンクコンフリクトを避けるためにパディングを挿入する等の最適化を行う.

表 6 評価に用いた環境.

CPU	AMD Phenom 9500, 2.2GHz, Quad-Core
Chipset	AMD 790FX
RAM	DDR2-800 SDRAM 1GB×4
OS	Fedora Core 8, linux 2.6.23
Driver	NVIDIA Linux driver 169.04
Software	CUDA SDK 1.1
	CUDA Toolkit 1.1
	GCC 4.1.2

表 7 提案手法と CUFFT ライブラリによる 256^3 の 3 次元 FFT の性能の比較.

	Our implementation		CUFFT3D
Model	Time(ms)	GFLOPS	GFLOPS
8800 GT	34.4	58.5	18.6
8800 GTS	31.2	64.6	20.6
8800 GTX	25.3	79.5	23.4

4. 性能評価

GeForce 8800 GT/GTS/GTX を用いて提案手法による 3 次元 FFT の性能評価を行う. 評価環境を**表 6** に示す. 790FX チップセット及び 8800 GT/GTS は PCI-Express 2.0 に対応するが, 8800 GTX だけは非対応であるため PCI-Express 1.1 で動作する.

表 8 提案手法の各ステップにおける実行時間と実効メモリバンド幅.

	Step 1&3		Step 2&4		Step 5		Total	
Model	Time(ms)	GB/s	Time(ms)	GB/s	Time(ms)	GB/s	Time(ms)	GB/s
8800 GT	6.89	38.9	6.78	39.5	7.24	37.0	34.57	38.8
8800 GTS	6.31	42.5	6.29	42.7	6.00	44.7	31.20	43.0
8800 GTX	4.52	59.3	4.84	55.3	6.92	38.7	25.64	52.3

表 9 1 次元 FFT 部分の実行時間と演算性能の比較. 65536 組の 256 点 FFT を計算する.

	Our Implementation		CUFFT1D	
Model	Time(ms)	GFLOPS	Time(ms)	GFLOPS
8800 GT	7.24	92.7	13.7	49.0
8800 GTS	6.00	111.8	11.4	58.9
8800 GTX	6.92	97.0	13.2	50.8

表 7 に各 GPU で 256^3 の 3 次元 FFT を実行した性能を示す. GFLOPS 値の算出にあたって, N^3 の 3 次元 FFT の浮動小数演算数を $15N^3 \log_2 N$ と仮定している. CUFFT3D は NVIDIA 社が提供する CUDA Toolkit 1.1 に付属する FFT ライブラリの 3 次元 FFT ルーチンである. 何れもホスト・デバイス間のデータ転送時間は含んでいない. GeForce 8 シリーズより 3 種類のモデルを使用して評価を行ったが, 何れも CUFFT と比べて 3.1〜3.3 倍高い性能が得られた.

提案手法は 5 回のカーネル実行により 3 次元 FFT の計算を実現している. **表 8** にそれぞれのカーネルの実行時間と, メモリバンド幅を示す. ステップ 1 と 3, 2 と 4 は全く同一の処理であり, 表 8 にはそれぞれの実行時間を示しており, 実行時間の合計を算出するときにはステップ 1&3 の実行時間とステップ 2&4 の実行時間を倍にする.

ステップ 1〜4 ではメモリからデータを読み込み 16 点 FFT を計算し結果をメモリに書き込む処理を行う. ステップ 1 や 3 ではひねり係数が 1.0 である部分の乗算を省略しているのでステップ 2 や 4 と比べると演算数が若干少ないが, メモリ転送速度によって制限されている.

一方ステップ 5 では 256 点 FFT の計算を行うため, 演算量は約 2 倍になる. この 256 点 FFT を複数スレッドで並列に計算するため共有メモリへのアクセスが必要である. 64 スレッド用いて 256 点 FFT を並列に計算する場合には各スレッドはそれぞれ 4 個のデータを持ち, 共有メモリを介した 3 回のデータ交換によって 256 点 FFT の計算を完了する. このためステップ 5 での実行時間はメモリバンド幅も影響するが, より SP のクロック周波数の影響が大きい. 8800 GT と 8800 GTS の場合はステップ 1〜4 に近いメモリバンド幅が得られており, 性能はメモリバンド幅によって制限されていると考えられる. 一方 8800 GTX の場合のメモリバンド幅はステップ 1〜4 より低くなっており, SP の性能によって制限されていると考えられる. SP の性能が一番高い 8800 GTS はメモリバンド幅の理論限界値では 8800 GTX に劣るがステップ 5 の性能は上回る.

表 10 64^3, 128^3, 256^3 の複素 3 次元 FFT の性能 (GFLOPS).

	CUFFT3D			Our implementation		
Model	64^3	128^3	256^3	64^3	128^3	256^3
8800 GT	5.33	11.5	18.6	37.0	49.1	58.5
8800 GTS	3.80	12.3	20.6	42.2	55.6	64.6
8800 GTX	5.11	15.3	23.4	46.5	67.5	79.5

ステップ 5 では 256 点の 1 次元 FFT を 65536 組計算を行っている. この部分に関して CUFFT ライブラリの 1 次元 FFT ルーチンとの性能比較を**表 9** に示す. CUFFT ライブラリと比較すると我々の実装が遥かに上回る性能を示している.

SP の性能に制限されている 8800 GTX の場合ステップ 5 での GFLOPS 値はピーク性能の約 30%程度である. この原因の一つは共有メモリへのアクセス等演算以外の命令が多いためである. コンパイルした結果を分析したところ, 主要ループ内の 180 命令のうち 46 命令が共有メモリへのアクセスであった. 浮動小数演算の命令数は加減算が 64, 乗算が 18, 積和演算が 18 である. 加減算命令と乗算命令では SP の積和演算器の性能の半分しか使用することができない. 以上のような要因によりピーク演算性能の約 32%が上限となる.

また, 遅延が大きいメモリアクセスを行っているにもかかわらず命令数から算出されるピーク性能比からそれほど離れていないことから, 多数のスレッド, 複数のスレッドブロックを実行することによって遅延の大部分を隠蔽できていることが分かる.

提案手法はサイズに依存するものではなく, 他のサイズにも適用可能である. 64^3 及び 128^3 のサイズで

表 11 ホスト・デバイス間の転送とそれを含めた 256^3 の複素 3 次元 FFT の性能.

		Host-to-Device		3D FFT on Device		Device-to-Host		Total	
Model	PCI-Express	Time(ms)	GB/s	Time(s)	GFLOPS	Time(ms)	GB/s	Time(ms)	GFLOPS
8800 GT	2.0 x16	26.3	5.11	34.4	58.5	26.9	4.99	87.6	23.0
8800 GTS	2.0 x16	25.9	5.18	31.2	64.6	26.2	5.12	83.3	24.2
8800 GTX	1.1 x16	47.5	2.82	25.3	79.5	39.4	3.40	112	17.9

表 12 FFTW ライブラリ 3.2alpha2 の単精度版を使用した CPU による 256^3 の複素 3 次元 FFT の性能.

Processor	Clock	Socket/Core	Compiler	Time(ms)	GFLOPS
AMD Phenom 9500	2.20GHz	1/4	gcc 4.1.2	195	10.3
Intel Core 2 Quad Q6700	2.66GHz	1/4	gcc 4.1.2	188	10.7
AMD Opteron 880	2.60GHz	8/16	Intel Compiler 9.1.45	220	9.15

の実行結果を加えたものを**表 10** に示す. サイズが小さい場合にはメモリアクセス量に対する演算量の比率が低下するため GFLOPS 値としては 256^3 の場合より低下するが, やはり CUFFT ライブラリ性能を大きく上回っている.

表 11 にホスト・デバイス間の転送時間とそれを加算した実行時間から算出される性能を示す. アプリケーションが FFT の計算のみを GPU で行う場合には入出力データをホスト・デバイス間で転送する必要があり, これらを含めた性能は PCI-Express の転送速度の限界により激しく低下する. 一つ前の世代の GPU である 8800 GTX は GPU 内の演算性能は一番高いが, PCI-Express 2.0 に対応していないため転送を含めた場合の性能は一番低くなる.

全てのアプリケーションが毎回 FFT の前後でホスト・デバイス間転送を必要とするわけではない. 前後で行う処理も GPU で行うことができる場合もあれば, ほとんどの処理を GPU 側で行うことができる場合もある. また依存関係のない複数の 3 次元 FFT の計算を行うような場合には GPU で計算している間にホスト・デバイス間の転送を行うことによって転送のオーバーヘッドを軽減することができる. 従って実質的な演算性能としては, GPU での FFT の性能が上限であり, ホスト・デバイス間の転送を加味した性能が下限であると言える.

また提案手法は 2 次元 FFT にも応用することが可能である. X 軸方向のデータが共有メモリに収まるデータサイズであれば, そのまま適用可能である. それより大きい場合にはさらに転置をする必要があり, やや複雑な実装になる.

表 12 に 256^3 の複素 3 次元 FFT を CPU で計算した場合の性能を示す. CPU では FFTW ライブラリ[12] のバージョン 3.2alpha2 の単精度ルーチンを用いた. OpenMP 並列化及び SSE 拡張命令を有効にしており, 全ての CPU コアを使用している. コンパイラには各環境で利用可能な OpenMP 対応コンパイラの中で性能が最大になるものを選択した. これらの CPU での性能と比較した場合, 転送時間を加味した GPU による性能 (表 11) が遥かに上回る.

現時点では CUDA 対応 GPU は単精度演算しかサポートしていないため高い精度を必要とするアプリケーションには適用できないが, 将来倍精度演算をサポートした製品がリリースされればより実用範囲が広まるであろう. それらを入手でき次第, 提案手法を用いて性能評価を行う予定である.

5. ま と め

GPU を用いた科学技術計算では汎用 CPU と比べて非常に高い演算性能とメモリアクセス性能を用いることができる. CUDA 環境では従来の GPU と比べてメモリアクセスの自由度が増加し, また共有メモリの利用により複雑なアルゴリズムを適用することが可能となった. 我々は CUDA 環境に適した 3 次元 FFT アルゴリズムを提案した. X 軸方向の計算には共有メモリを活用し, Y 軸, Z 軸方向の計算には高いメモリアクセス性能を活かしてベクトル計算機向けの multirow FFT アルゴリズムを用い, さらに転置処理を組み合わせることによってメモリアクセスパターンを最適化した. GeForce 8 シリーズの GPU における性能評価では 256^3 の 3 次元 FFT の計算において NVIDIA の CUFFT ライブラリと比較して 3.1〜3.3 倍となる最大 79.5GFLOPS の演算性能を達成した.

謝辞 本研究の一部は科学技術振興機構戦略的創造研究推進事業『ULP-HPC: 次世代テクノロジのモデル化・最適化による超低消費電力ハイパフォーマンスコンピューティング』, 及び Microsoft Technical Computing Initiative “HPC-GPGPU: Large-Scale Commodity Accelerated Clusters and its Applica-

tion to Advanced Structural Proteomics" によるものである.

参 考 文 献

1) General-Purpose Computation Using Graphics Hardware: http://www.gpgpu.org/.
2) Mark, W.R., Glanville, R.S., Akeley, K. and Kilgard, M. J.: Cg: A System for Programming Graphics Hardware in a C-like Language, *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, Vol. 22, No. 3, pp. 896–907 (2003).
3) Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware, *SIGGRAPH '04: ACM Transactions on Graphics*, Vol. 23, No. 3, pp. 777–786 (2004).
4) NVIDIA CUDA – Compute Unified Device Architecture: http://developer.nvidia.com/object/cuda.html.
5) Stock, M.J. and Gharakhani, A.: Toward efficient GPU-accelerated N-body simulations, *46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608* (2008).
6) Larsen, E. S. and McAllister, D.: Fast Matrix Multiplies using Graphics Hardware, *the 2001 ACM/IEEE conference on supercomputing (CDROM)*, ACM Press (2001).
7) Cooley, J. W. and Tukey, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. Comput.*, Vol.Vol. 19, pp. 297–301 (1965).
8) Moreland, K. and Angel, E.: The FFT on a GPU, *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 112–119 (2003).
9) Govindaraju, N.K., Larsen, S., Gray, J. and Manocha, D.: A Memory Model for Scientific Algorithms on Graphics Processors, *the 2006 ACM/IEEE conference on supercomputing (CDROM)*, IEEE (2006).
10) Swarztrauber, P.N.: FFT algorithms for vector computers, *Parallel Computing*, Vol.1, pp. 45–63 (1984).
11) Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).
12) Frigo, M. and Johnson, S.G.: The Design and Implementation of FFTW3, *Proceedings of the IEEE*, Vol.93, No.2, pp.216–231 (2005). special issue on "Program Generation, Optimization, and Platform Adaptation".