CUDA GPU向けの自動最適化FFTライブラリ

額田 彰[†1,†2] 松岡 聡[†1,†3,†2]

NVIDIA CUDA をサポートとする GPU はその高いメモリバンド幅から，FFT などのメモリアクセスの多い計算にも有効である．CUDA 用の FFT カーネルに関しては既に幾つかの実装が存在するが，GPU のアーキテクチャに有利な 2 のべき乗などのサイズに特化したものが多い．本研究では自動最適化手法によって高性能な FFT カーネルを生成し，より多様な入力サイズに対応する．shared memory へのアクセスの最適化や網羅的な探索により生成されたカーネルは CUFFT ライブラリを遥かに超え，既存の他の実装を上回る性能を実現した．

Auto-Tuning FFT Library for CUDA GPUs

Akira Nukada[†1,†2] and Satoshi Matsuoka [†1,†3,†2]

NVIDIA CUDA capable GPUs have extremely high memory bandwidth which benefits memory intensive applications such as FFT. Already there are several implementations of FFT using CUDA but they are optimized for specific transform sizes like powers of two which are suitable for GPU architecture. In this paper, we present our auto-tuning method to generate high performance CUDA kernels for FFTs of varying transform sizes. The optimized kernels outperform not only NVIDIA CUFFT libraries but also many of existing implementations.

1. はじめに

Graphics Processing Unit (GPU) は非常に身近な出力デバイスの一つであるが，3D ビデオゲームやグラフィックスアプリケーションにおいて高画質の 3 次元空間映像を高速にレンダリングするためのアクセラレータであり，特に近年の飛躍的な演算性能の向上の結果，これまでの主要な計算資源である CPU の演算性能を上回る．このことに注目し，これまで多くの研究者・開発者が GPU による汎用計算を実現してきた[1]．従来の計算用アクセラレータと同様に N 体問題[2),3)] や行列積[4)] などの演算量の多い計算で特に性能を発揮してきたが，GPU の特徴の一つである高いメモリバンド幅を生かし，FFT[5)] 等のメモリアクセスの比率が多い計算の高速化も可能である．GPU は量産されるハードウェアであるため廉価で入手可能であることや，電力効率が高いことも魅力的な要素である．

†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.

GPU での計算は最初は DirectX や OpenGL などのグラフィック API を用い，Cg[6)]，HLSL 等のシェーダ言語で開発した独自のシェーダプログラムを実行することにより汎用計算を行っていた．その後 C 言語を拡張した BrookGPU[7)] や Microsoft の Accelerator[8)] などのより開発しやすい高級言語のプログラミング環境が整ってきた．NVIDIA の提案する CUDA[9),10)] はこれまでの GPU アーキテクチャとは異なり，GPGPU としての利用を強く意識した機能を多く備えている．CUDA の開発言語もまた C 言語を拡張したものであり，以前と比べて格段に容易に GPU を使ったプログラムを作成できるようになった．FFT の GPU 用の実装は CUDA 以前もの[11)–13)]，CUDA を利用したもの[14)–16)] と既に幾つかの FFT の実装が存在する．

GPU の場合 CPU と違い急速に進化しており，ある製品がリリースされてから次の製品がリリースされるまでの間隔が非常に短い．そこで製品を入手してから如何に迅速にコードの最適化・チューニングを完了するかが重要になる．この目的を達成するための方法の一つが ATLAS[17)] や FFTW ライブラリ[18)]，SPIRAL[19)] に代表される自動チューニング手法である．大幅なアーキテクチャの変更があると完璧に対処することは不可能であるが，多少のバランスの変化等には

追従可能である．

現在 NVIDIA GPU 用の CUDA，AMD/ATI GPU 用の Brook+や CAL[20]，Cell[21] 用の Cell SDK と様々なプログラミング環境が乱立している状態にあり，各計算デバイスに対応する開発言語を用いなければならない．これらを統合する目的で昨年 12 月に策定されたのが OpenCL 1.0[22] である．OpenCL 環境においても多種の CPU，GPU，その他のアクセラレータ等の差異を吸収し，適した実装方法を選択するために同様の自動最適化手法が必要となるであろう．

本研究では NVIDIA CUDA 環境において FFT ライブラリの自動最適化を実現する．特に単精度の complex-to-complex 型で多次元 FFT の一部としても頻繁に用いられる，多数の 1 次元 FFT の計算を扱う．

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 *shared memory* によってスレッド間の通信が可能となり，複数スレッドが共通して必要なデバイスメモリ上のデータへのアクセスの効率化や，細粒度並列処理，より複雑な計算等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，広範囲なアプリケーションを対象とする．

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．8800 GTX は 16 個の Streaming Multiprocessor (SM) 群を搭載し，この各 SM は Streaming Processor (SP) と呼ばれるプロセッサコア 8 個，16KByte の shared memory，8,192 個のレジスタ，constant cache memory，texture cache memory 等から構成される．GeForce 8800 GTX は合計 128 個の SP をもつ超並列プロセッサである．

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で，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．NVIDIA の最新 GPU である GeForce GTX 280 や Tesla S1070 では，SM あたりのレジスタ数が 16,384 に，SM の数が 30 に増加しているほか，命令実行機構の改良，atomic 処理，倍精度演算への対応等が追加されている．AMD Phenom 9500 Quad-Core プロセッサ (2.2GHz) の単精度演算性能は 4 コアを合計して 70.4GFLOPS であり，GeForce GTX 280 はその約 13 倍もの演算性能を持つ．

CUDA の SM は各 SP で同じ命令を実行する SIMD (Single Instruction Multiple Data) であるが，各 SM 間は同期せずに動作する SPMD (Single Program Multiple Data) 型のプロセッサである．CUDA のアーキテクチャでは現在各 SM は最大 1,024 個 (G80 では 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 32Warp (G80 では 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 SM で実行され，shared memory を介してスレッド間のデータ交換や同期が可能である．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，shared memory サイズによって各 SM 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．

3. CUDA 用 FFT カーネルの自動チューニング

CPU での FFT の計算とは異なり，CUDA GPU での FFT は実行スレッド数，shared memory のバンクコンフリクトの回避等の GPU 特有のチューニングの項目が多く存在する．

まず shared memory を用いた FFT の計算の例として 60 点 FFT を**図 1** に示す．各スレッドブロックが 1 次元 FFT を一つずつ計算していくものとする．まずはじめに入力データを global memory から読み込み，20 スレッドがそれぞれ 3 基底 FFT カーネルを計算する．shared memory を介してスレッド間でデータ交換をしながら，15 スレッドで 4 基底 FFT カーネルを，12 スレッドで 5 基底 FFT カーネルを計算し，最後に結果を global memory へ書き込む．このように複数の基底のカーネルを用いる場合には，計算に参加するスレッド数が変化する．実際には CUDA のスレッドブロックあたりのスレッド数は CUDA カーネル実行中は固定であるため，一部のスレッドが遊休状態になる．

3.1 基底の選択

CPU の計算と同様に FFT カーネルで用いる基底と順序に関して幾つかの組み合わせがある．例えば

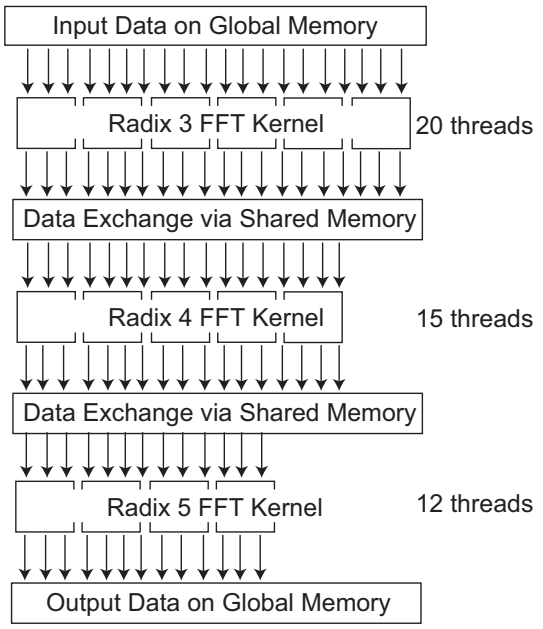

図 1 Shared Memory を用いた 60 点 FFT の計算.

480 点 FFT の場合，3，4，5，8 基底を用いるとしても 4! = 24 通りの順序がある．それらの全てについてカーネルを生成し，実行したときの実行時間を比較して最良のものを選ぶ．単に演算数が最小になるものを選ぶ方法では不十分で，遊休状態となるスレッドや，shared memory へのアクセス，SIMD 化により増加する演算等の様々な要素が性能に影響を与える．

3.2 スレッドブロック数の選択

各 SM で同時にアクティブになるスレッドブロックの数はメモリアクセスの性能や SP を有効利用する上で重要な要素である．適切なスレッドブロック数が存在し，それより多くても少なくても性能低下を引き起こす．

表 1 に今回の実験で用いた GPU の各モデルの諸元を示す．両モデルを比較すると，SP の演算性能は Tesla S1070 の方が高く，メモリバンド幅に関しては GeForce GTX 280 の方が高いというようにバランスが異なる．**図 2** に示すようにその適切なスレッドブロック数は GPU のモデルに依存する．

図 2 に示すように最適なスレッドブロック数に至るまでの過程では性能は上昇し続ける．そこで各 SM あたりのスレッドブロック数を 1 から順に増やしていき，性能が前回より下がったところで停止するという手法により容易に最適なスレッドブロック数を得る事ができる．

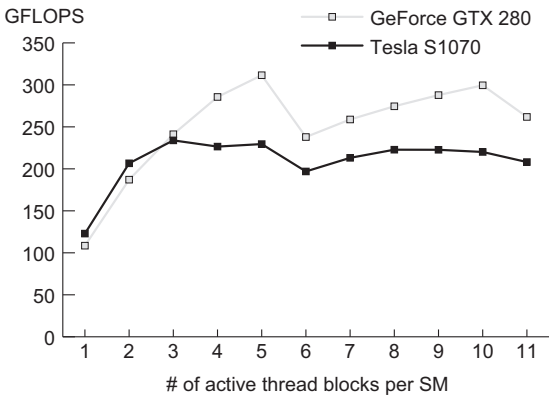

図 2 アクティブスレッドブロック数が性能に与える影響. 512 点 FFT を 32,768 組計算.

3.3 Shared Memory の最適化

NVIDIA の CUDA 対応 GPU ではスレッド間で共有される shared memory を利用することによって高速にスレッド間でデータを交換することができる．現在は全てのモデルの GPU は SM あたり 16KByte の shared memory を搭載する．一方レジスタは G80 系や G9x 系では SM あたり 32KByte(32bit× 8,192)，最新の GeForce GTX 280 等の GT200 系のコアでは 64KByte(32bit×16,384) と shared memory より容量が大きい．

このため各スレッドは基本的にレジスタにデータを保持し，スレッド間でデータを交換するときにだけ shared memory を用いる．しかしながら FFT の計算ではほぼ全てのデータを他のスレッドに送る必要があるため，複素数データの実部をまず交換し，その後で虚部を交換するといった 2 段階で行う．これによって必要な shared memory のサイズはデータ保持に使うレジスタの半分で足りることになる．

CPU のキャッシュ上で FFT を計算する際に，データの格納順を bit-reverse 順に入れ替える処理を避けるため**図 3** に示すような Stockham の自動ソートアルゴリズム[23] を使うことが多い．GPU の shared memory でデータ交換をする場合にも同様に Stockham アルゴリズムに従ったインデックスに読み書きすることでスレッド番号と保持するデータのインデックスの並び順を一致させることができる．

入力サイズ N の FFT を以下のような R 回の基底 r_k の FFT カーネルにより計算するとする．

$$N = \prod_{k=1}^{R} r_k$$

Stockham アルゴリズムを使用した場合，i 番目 ($1 \leq i \leq R-1$) のデータ交換における shared memory へ

表 1 CUDA 対応 GPU の諸元.

	# of SP	# of SM	SP clock	GFLOPS	Mem. B/W	Mem. Capacity
GeForce GTX 280	240	30	1.30 GHz	936	141 GByte/s	1.0 GByte
Tesla S1070	240	30	1.44 GHz	1037	110 GByte/s	4.0 GByte

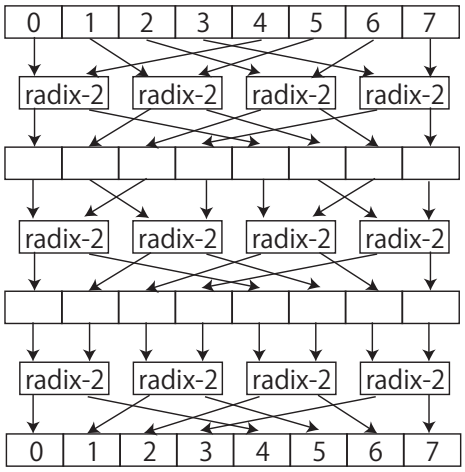

図 3 Stockham アルゴリズムを用いた 8 点 FFT の計算.

のアクセスパターンは次のようになる.

書き込み先のインデックスは,

$$\mathrm{tid} + N/r_i * k$$
$$(k = 0, 1, 2, \cdots, r_i - 1)$$

読み込み元のインデックスは,

$$\mathrm{tid}\%p_i + \mathrm{tid}/p_i * p_i * r_{i+1} + k * p_i$$
$$(k = 0, 1, 2, \cdots, r_{i+1} - 1)$$

ただし tid はスレッド番号, p_i は以下の通り.

$$p_i = \prod_{k=i+2}^{R} r_k$$

shared memory は 32bit 要素単位で 16 個のバンクに分かれており, 16 スレッドが同時にそれぞれのバンクにアクセスすることができる. 複数のスレッドが同じバンクに属する異なるアドレスにアクセスしようとするとバンクコンフリクトが発生し, より多くのサイクル数を要する. これは複数のスレッドが同時に shared memory にアクセスする CUDA GPU 特有の問題で, CPU の on-chip キャッシュ等では起こらない.

バンクコンフリクトを起こさないためには連続する 16 スレッド, tid= $\{16n, 16n+1, \cdots, 16n+15\}$ が異なるバンクにアクセスする必要がある. 書き込みに関しては各スレッドがアクセスする shared memory のインデックスは隣接しており, バンクコンフリクトは一切起こらない. 一方読み込みに関しては p_i 個連続でアクセスし $p_i(r_{i+1}-1)$ 間隔をあけるパターンを繰り返すが, ここではバンクコンフリクトが生じる可能性がある.

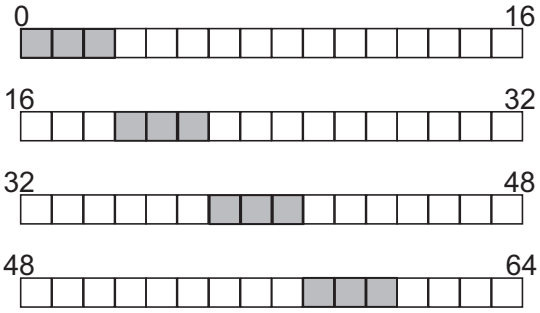

図 4 条件式 (3) を満たす場合, 連続するバンクにアクセスするためバンクコンフリクトが発生しない. ただしこの図に相当する r_{i+1} は存在せず, パディングを挿入しない限りこの図のようにはならない.

$$p_i = 2^m \times l, \qquad l : \text{odd number}$$

のようにおくと, バンクコンフリクトが起きない条件は,

$$m \geq 4 \tag{1}$$

または,

$$l = 1 \wedge r_{i+1} \text{ is odd number.} \tag{2}$$

または,

$$2^m l(r_{i+1} - 1) \equiv 0 \mod 16 \tag{3}$$

の何れかが満たされることである. 条件式 (3) は**図 4** のように順に右隣のバンクにアクセスする状況である. なお, 条件式 (1) は条件式 (3) に包含される. これらの条件を満たさない場合には, 必ずバンクコンフリクトが生じる.

3.3.1 パディング挿入方法 (1)

shared memory からの読み込み時にバンクコンフリクトが起きている場合, $p_i(r_{i+1}-1)$ の間隔をあけた後にパディングを挿入することでバンクコンフリクトを解消することができる. パディングの要素数を b とすると,

$$b = 2^m \wedge l = 1 \wedge r_{i+1} \text{ is even number.} \tag{4}$$

または,

$$2^m l(r_{i+1} - 1) + b \equiv 0 \mod 16 \tag{5}$$

の何れかの条件を満たすような b を選ぶ.

このようなパディングを挿入することで shared memory から読み込む時のバンクコンフリクトは解消される. 一方 shared memory へ書き込む場合には $p_i * r_{i+1} \equiv 0 \mod 16$ でなければ新たにバンクコン

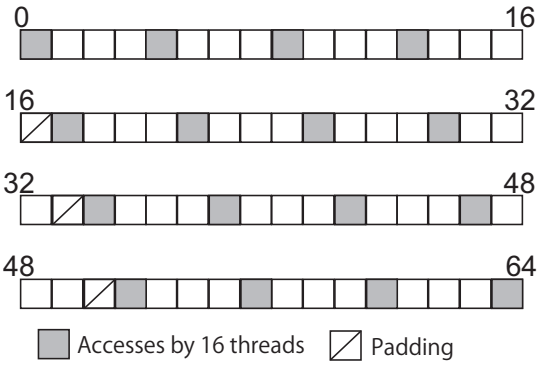

図 5 $p_i = 1$，$r_{i+1} = 4$ の場合．16 要素毎に 1 のパディングを挿入してバンクコンフリクトを回避．

フリクトが生じる．パディング挿入によって shared memory 使用量が増加することを考慮すると，$p_i * r_{i+1}$ を超えるパディングサイズを用いて shared memory 使用量を倍以上にすることはないであろう．この条件下では高々2-way のバンクコンフリクトしか起きない．書き込み時と読み込み時ではバンクコンフリクトが起きた場合の影響が同じとは限らないため単純にコンフリクトの回数を比較しても意味が無く，実際に両方のカーネルを実行して実行時間を比較する必要がある．

3.3.2 パディング挿入方法 (2)

やや限られた条件で使えるパディング挿入方法として，16 要素毎にパディングを挿入するというものがある．**図 5** は $p_i = 1$，$r_{i+1} = 4$ の場合である．この手法は p_i，r_{i+1} がともに 2 のべき乗のときのみ適用できる．$p_i = 2^m$, $r_{i+1} = 2^k$ の場合，16 要素毎に 2^m 要素のパディングを挿入することによって，shared memory 読み込み時だけでなく書き込み時もバンクコンフリクトが起きなくなる．

自動最適化ではまずパディング挿入方法 (2) の条件を満たす場合には (2) を用いる．それ以外の場合には (1) の適用を試みる．

3.4 実 装 方 法

図 6 に自動最適化の流れを示す．まず入力として 1 次元 FFT の長さ及び何組の 1 次元 FFT を計算するかの情報が与えられる．FFT の長さに対して，パディング使用の有無を含めてあらゆる基底の組み合わせを列挙し，それぞれに関して CUDA カーネルを生成し，コンパイルしてモジュールを得る．このモジュールをロードして実行することによりカーネルの実行時間を計測するが，このときカーネル呼び出し時に指定するスレッド数を調整して最適なスレッドブロック数を探索する．

GPU が実行するのは計測対象である CUDA カーネルの部分だけで，他の処理は全てホスト側の CPU が実行する．GPU でのカーネル実行時間はごく一部であり，自動最適化にかかる時間の 50%から 75%を生成したカーネルのコンパイル時間が占めている．この最適化処理にかかる時間は入力サイズに依存し，使用する基底の組み合わせが多い場合にカーネルのコンパイル回数が増加し，その結果最適化処理時間も増加する．

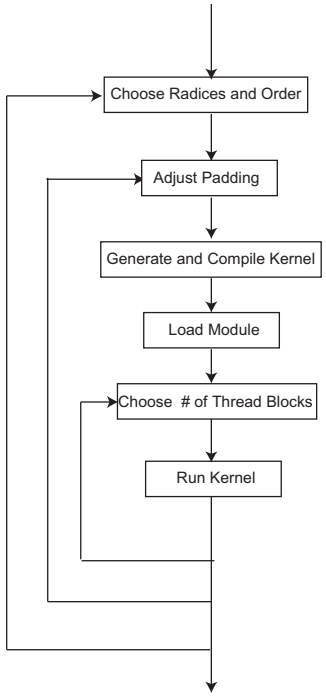

図 6 CUDA 環境における FFT カーネルの自動最適化の流れ．入力として 1 次元 FFT のデータの長さ，計算する 1 次元 FFT の数が与えられ，最も高性能な CUDA カーネルが出力される．

4. 性 能 評 価

前節で述べた自動最適化の性能評価を**表 2** に示すシステムを用いて行う．性能値 (GFLOPS) は実行時間 t，入力サイズ N，計算する入力の数 B より，以下の式で計算するものとする．

$$5N \log_2 N \times B/t \times 10^{-9}$$

図 7 にパディング挿入が性能に与える効果を示す．入力サイズが 2 のべき乗の場合に特にパディングの効果が大きいが，これはパディングを使わない場合に 4-way や 8-way のバンクコンフリクトが発生しているためである．他のサイズに関してはあまり効果大きくなく，最大でも 3%程の性能向上にとどまる．2 のべき乗以外では演算数が格段に多い 3 基底や 5 基底カーネルを使用するため，バンクコンフリクトの影響

表 2 性能評価環境.

CPU	AMD Phenom 9500 Quad-Core Processor (2.2GHz, L2=512KByte× 4, L3=2MByte.)
Memory	DDR2-1066 SDRAM 2GB× 4
Chipset	AMD 790FX
Interface	PCI-Express Gen 2.0 mode, 16 lanes.
OS	OpenSUSE 11.0 (X86-64), linux 2.6.25
Driver	NVIDIA ForceWare 180.22.
Compiler	gcc 4.3.1 (host code)
	NVIDIA CUDA Toolkit 2.1 (device code)
GPU	NVIDIA GeForce GTX 280

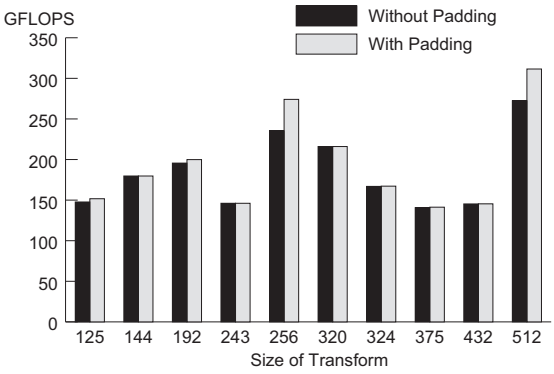

図 7 パディング挿入によるバンクコンフリクト調整の効果．1 次元 FFT を 32,768 組計算．GeForce GTX 280 を使用.

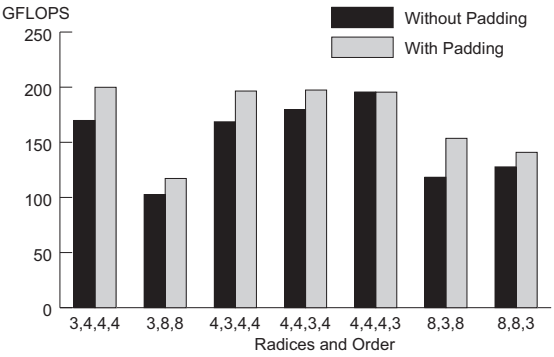

図 8 様々な基底の選択に対するパディング挿入の効果．192 点 FFT を 32,768 組計算．GeForce GTX 280 を使用.

が相対的に小さい.

図 8 に 192 点 FFT において様々な基底選択時のパディングの効果を示す．”4,4,4,3” 以外の全ての基底の選択時にはパディングの効果が大きく現れている．しかし最高性能となるのはパディングなしの場合に”4,4,4,3” であるのに対して，パディングありの場合は”3,4,4,4” と異なる．パディングなしの場合，バンクコンフリクトが少ない基底の組み合わせ・順序が選択されている．一方パディングありの場合はもともと生じていた 4-way バンクコンフリクトを全て解消

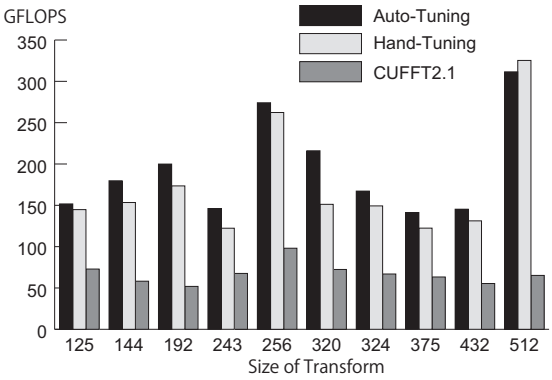

図 9 自動最適化と手動最適化，及び CUFFT ライブラリの性能比較．1 次元 FFT を 32,768 組計算．GeForce GTX 280 を使用.

している.

パディングなしの場合 “4,4,4,3” では 2-way バンクコンフリクトが，その他では 3-way 以上のバンクコンフリクトが shared memory からの読み込み時に生じる．“4,4,4,3” で生じる 2-way バンクコンフリクトは我々の考案した方法で対処すると shared memory への書き込み時に 2-way バンクコンフリクトを生じてしまうため性能改善に繋がらない.

この計測結果より，少なくとも使用した GPU においては，パディングの効果が現れるのは 3-way 以上のバンクコンフリクトがある場合，またはパディングを挿入することによって書き込み部分で新たにバンクコンフリクトが生じない場合と考えられる.

図 9 に自動最適化により生成されたカーネルと他のライブラリの性能比較を示す．”Hand-Tuning” はこの自動最適化手法を確立する前に我々が手作業による最適化で作成したカーネルであるが，512 点 FFT を除く全てにおいて自動最適化によるカーネルの方が高い性能を示している．512 点 FFT に関しても自動最適化により生成されたカーネルと手作業による最適化されたカーネルでは shared memory へのアクセス方法は完全に同じであり，この性能差は記述の違い等から来ているものと考えられる.

自動最適化されたカーネルを用いた GeForce GTX 280 と Tesla S1070 の性能比較を**図 10** に示す．2 のべき乗などの演算量の少ないサイズに関しては高いメモリバンド幅を持つ GeForce GTX 280 の方が高性能であるが，演算量の多いサイズに関してはシェーダクロックの高い Tesla S1070 の方が高性能という結果になった．入力サイズのうち下線が付いているものは，自動最適化の結果選択された基底が異なるものであるが該当するものが多く，自動最適化が有効に機能して

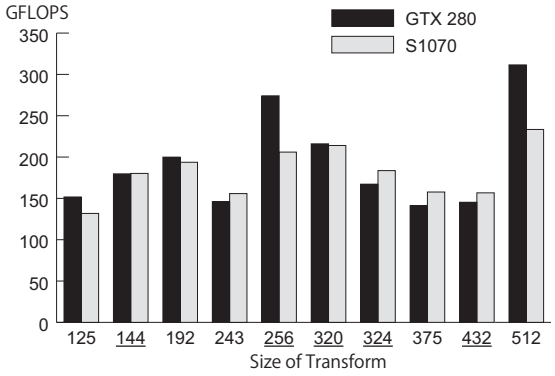

図 10 自動最適化されたカーネルを用いた GTX 280 と S1070 の性能比較．1 次元 FFT を 32,768 組計算.

いるといえる．

図 10 の入力サイズの中で 432 が最も自動最適化の時間が長く，Phenom 9500 + GeForce GTX 280 の構成で約 54 秒を要した．その中で約 28 秒がコンパイル時間，約 11 秒が CPU で行う計算結果のチェックにかかった時間である．この最適化時間は十分許容範囲にあると考えられるが，GPU のハードウェアに対して不変なので毎回実行時に最適化を行う必要はなく，オフラインであらかじめ最適化を行うことが可能である．

5. まとめと今後の課題

多次元 FFT の一部としても頻繁に用いられる，複数の 1 次元 FFT の計算を行う処理を CUDA 対応 GPU で高速に実行するための自動最適化手法を提案した．CPU の場合と同様の基底の選択などに加えて，CUDA GPU 独自の要素として最適なスレッドブロック数の選択や shared memory のバンクコンフリクトを避ける手法を用いている．手作業による最適化では見落としがちな可能性まで網羅的に探索することにより，512 点 FFT 以外の全てのサイズにおいて手作業で最適化したコードを超える性能を達成できた．

現在は 1 次元 FFT の自動最適化にとどまっているが，今後は我々が以前提案した 3 次元 FFT[14] や 2 次元 FFT に拡張する予定である．また本手法の倍精度演算への対応も容易であると考えられる．

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 Application to Advanced Structural Proteomics” によるものである．また NVIDIA 社の方々には様々な技術的サポートをいただき感謝いたします．

参 考 文 献

1) General-Purpose Computation Using Graphics Hardware: http://www.gpgpu.org/.
2) Stock, M.J. and Gharakhani, A.: Toward efficient GPU-accelerated N-body simulations, *46th AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2008-608* (2008).
3) Nyland, L., Harris, M. and Prins, J.: Fast N-Body Simulation with CUDA, *GPU Gems 3* (Nguyen, H., ed.), Addison-Wesley, chapter31, pp.677–695 (2007).
4) Larsen, E. S. and McAllister, D.: Fast Matrix Multiplies using Graphics Hardware, *the 2001 ACM/IEEE conference on supercomputing (CDROM)*, ACM Press (2001).
5) 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).
6) 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).
7) 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).
8) Tarditi, D., Puri, S. and Oglesby, J.: Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses, *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems 2006* (2006).
9) NVIDIA CUDA: Compute Unified Device Architecture. http://developer.nvidia.com/object/cuda.html.
10) Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol.28, No.2, pp.39–55 (2008).
11) Moreland, K. and Angel, E.: The FFT on a GPU, *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 112–119 (2003).
12) Spitzer, J.: Implementing a GPU-efficient FFT, *SIGGRAPH Course on Interactive Ge-*

ometric and Scientific Computations with Graphics Hardware (2003).
13) 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*, IEEE (2006).
14) Nukada, A., Ogata, Y., Endo, T. and Matsuoka, S.: Bandwidth Intensive 3-D FFT Kernel for GPUs using CUDA, *the 2008 ACM/IEEE conference on supercomputing* (2008).
15) Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B. and Manferdelli, J.: High Performance Discrete Fourier Transforms on Graphics Processors, *the 2008 ACM/IEEE conference on supercomputing* (2008).
16) Volkov, V. and Kazian, B.: Fitting FFT onto the G80 architecture (2008). http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf.
17) Whaley, R.C., Petitet, A. and Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project, *Parallel Computing*, Vol.27, No.1–2, pp.3–35 (2001).
18) 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".
19) Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W. and Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms, *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, Vol.93, No.2, pp.232–275 (2005).
20) Advanced Micro Devices, Inc.: ATI Stream Computing (2009). http://ati.amd.com/technology/streamcomputing/sdkdwnld.html.
21) Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y. and Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture, *IEEE Micro*, Vol. 26, No. 2, pp. 10–24 (2006).
22) Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.
23) Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, PA (1992).