2010 年 5 月 17 日 情報システム基盤学専攻合同輪講発表資料

Auto-Tuning 3-D FFT Library for CUDA GPUs

著者： Akira Nukada, Satoshi Matsuoka

出典： *SuperComputing 2009 PP.1-10*

発表者： 高性能コンピューティング学講座　1053018 長塚　郁

1　はじめに

Graphics Processing Unit(以後 GPU) は膨大な計算領域と広いメモリバンド幅を持つため，特に N-body 問題や FFT などの計算量やメモリがボトルネックとなるアプリケーションにおいて，マルチコア CPU と比較して性能が良いと言われている．一方，GPU は CPU に比べて，メモリの取り扱いによって性能が大きく異なるため，不安定となる．

これらの問題点より，今まで多数提案された GPU を用いた FFT の手法は問題サイズ毎に人間の手によってチューニングが行われてきた．さらに，GPU の開発サイクルは CPU と比べて非常に短く，GPU チップ毎に最適化を行う必要がある．ただし，人間の手による最適化を迅速に行うことは難しい．この問題の解決案として，機械的に自動の最適化を行う Auto-Tuning が注目されている．しかし，GPU は特殊なマルチスレッド構造など，アーキテクチャが特殊であるため，従来の方法を用いることができない．そこで本研究では GPU の特殊なアーキテクチャを対象とした，3-D FFTs における Auto-Tuning の手法を提案する．

2　Auto-Tunnig

GPU 上での 3-D FFT の Auto-Tuning を実装は，次元 X と次元 Y,Z で仕様が異なる．次元 X の場合を考える．次元 X の 1-D FFT を実行する際，各 Thread を連動させる．そのため，Thread 間のデータのやり取りは高速アクセスが可能な Shared Memory を用いて行う．

このとき，最適化の際には Thread 数の選択や Shared Memory の使用による Bank Conflict の発生の回避が必要となる．

2.1　Thread-Block 数 [1] の選択

Thread-Block の数は問題サイズだけではなく，GPU チップに依存する．同じハードウェアチップであっても，GTX280 ではメモリバンド幅が大きく，Tesla10p では Streaming Processor(以後 SP) 当たりの性能が大きい．このように特徴が異なるため，Thread-Block の数の最適な数も変化する．そこで，Streaming Multi Processor(以後 SM) 当たりの Thread-Block の数を 1 からカーネル実行し，最適値を評価する．

2.2　Shared Memory を用いた最適化

Shared Memory は NVIDIA CUDA GPU が SM 一個当たり 16[Kb] 用意した，Video Memory と比較してアクセスが高速なメモリである．次元 X の 1-D FFT ではこの Shared Memory を使い，データのやり取りを行う．

CUDA GPU の Shared Memory は 16 個の 32-bit バンクで成立し，16 個の Thread は同時にアクセスすることができる．しかし，複数の Thread が同時に同じバンクにアクセスしようとすると，Bank Conflict が生じる．この Bank Conflict は性能低下の大きな原因となる．

この Bank Conflict を避けるために，連続した 16Thread が異なるバンクにアクセスする必要がある．次元 X の FFT を実行する際，Thread は一定間隔毎に，連続したアドレスの Shared Memory からデータを読み込む．この間隔の開け方や連続したアドレス数（以下連続数）によって，Bank Conflict が生じる．以下の条件のいずれかが満たされる場合に Bank Conflict を避けることができる．

(1) 連続数を 16 の倍数にする．

(2) 読込の間隔を 16 の倍数にする．

(3) 連続数を 2 のべき乗にし，かつ 1Thread が読み込むデータ総数を奇数にする．

2.3　Padding

Bank Conflict が Shared Memory からデータを読み込むときに生じる場合，Padding を一定間隔後に挿入することで解決できる．Padding 処理は無意味なデータを挿入することで，アクセスするメモリの位置をずらす．Padding 処理を行うことで，(1),(2),(3) の条件のうち (2) と (3) のどちらかの条件が満たされるようにする．

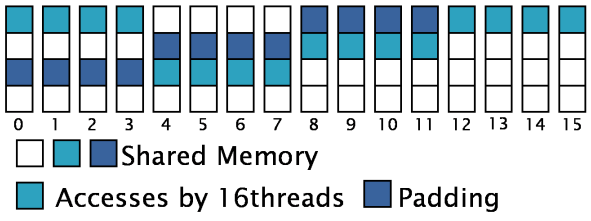

Fig. 1: Padding の挿入例

Fig. 1 は (3) の条件が Padding 処理を行った結果満足した場合の Shaderd Memory へのアクセスを示す．

ただし，Padding 処理によっては Shared Memory への書き込みで Bank Conflict が発生してしまう．さらに大量の Padding を挿入することで，Shared Memory の容量を浪費してしまう．

Shared Memory へ格納できるデータの数も減るので，一度に動かせられる Thread 数も限られる．

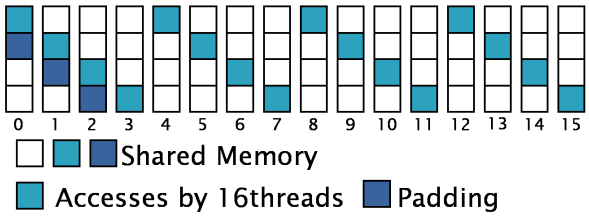

Fig. 2: 毎 16 要素後に Padding を挿入する場合

そこで Padding を毎 16 要素の後ろに挿入する方法をとる．Fig. 2 にその一例を示す．これは連続数と間隔の両方が 2 のべき乗である場合に限り有効である．その時連続数と同数の Padding を毎 16 要素後に挿入することによって，書き込みと読み込みの Bank Conflict を解決することができる．本論文では，条件が合う場合にはこの方法を用い，そうでない場合では，一定間隔毎に挿入する方法の適用を試みる．

2.4 Auto-Tuning の仕様

次元 X の 1-D FFTs を Auto-Tuning する際，まずデータサイズとバッチ数が入力される．入力されたデータサイズから考えれる基底の組み合わせや配列をコンピュータ側が決め，Padding の挿入を適用する．そして CUDA カーネルを生成し，コンパイルを行う．さらにモジュールを読み込み，Thread-Block の数を決めてカーネルを実行し，カーネルの実行時間を測定する．そしてまた Thread-Block の数を決め，カーネルを実行することを繰り返していく．

3 結果

Auto-Tuning FFT ライブラリによる最適化を行った次元 X に

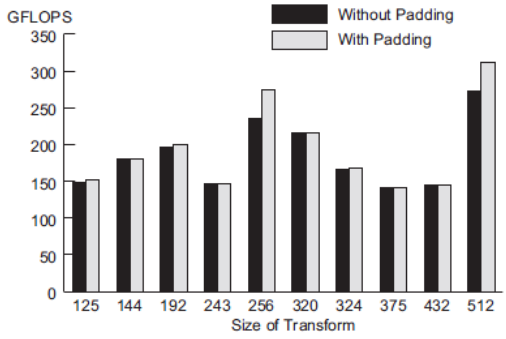

Fig. 3: Padding の有無における各要素数での性能比較

Fig. 3 に Padding による性能改善の結果を示す．このとき，GPU は GeforceGTX280 を使用した．データサイズが 2 のべき乗の場合，特に高速化が得られた．これは 2 のべき乗サイズで，Padding がない場合に 4-Way もしくは 8-Way Bank Conflict が生じるためである．これに対し，データサイズが 2 のべき乗ではない場合では，Padding を施した場合の性能は，そうでない場合に対して 3% 程度の上昇になった．また更に，2 のべき乗ではないサイズの FFT は 3 基底や 5 基底カーネルなどの負荷が大きい浮動小数演算を必要とするため，Bank Conflict の負荷が相対的に低くなり，性能の差がないと考えられる．

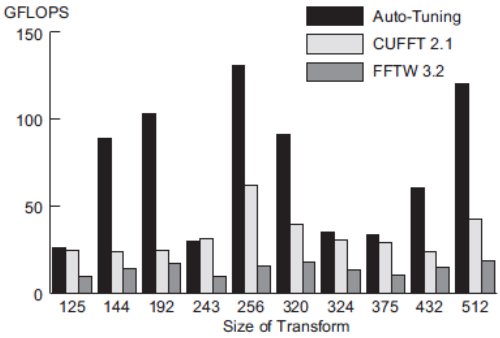

Fig. 4: 各システムとの性能比較

Fig. 4 は GTX280 上で動かした Auto-Tuning およびハンドチューニング，CUFFT の結果と FFTW を Phenom9500 4 cores で実行した場合での性能比較である．FFTW ではたかだか 16GFLOPS の性能であり，どの GPU の結果よりも低い．Auto-Tuning に比べ，NVIDIA CUDA CUFFT ライブラリは 2.1〜4.7 倍性能が低い結果が得られた．またハンドチューニングに対しては，同じようなアルゴリズムを使っているのにもかかわらず，512 の場合を除いて性能が勝る結果を得た．なぜ 512 のサイズでハンドチューニングの性能が Auto-Tuning に勝るかという理由は，CUDA 言語レベルでははっきりとはわからないため，他の Auto-Tuning への適用のためにも更なる調査が必要となる．

4 結論

本研究では CUDA GPU を用いた 3-D FFT の Auto-Tuning algolism を提案した．CPU で指定するパラメータを含め，Thread の数の調整と Padding を用いた Shared Memory を使うにあたっての Bank Conflict の回避の最適化を行った．性能比較を行った結果，Auto-Tuning で最適化した FFT のカーネルは CUFFT に比べ，最大 2.1-4.7 倍性能が向上する．一方で，2 のべき乗ではないカーネルの場合，浮動小数点演算を必要な計算が存在するため，性能の差は小さい．

参考文献

[1] nVIDIA, NVIDIA CUDA Programing Guide Version 2.3, nVIDIA, 2009.