性能モデルに基づくCPU及びGPUを併用する効率的なFFTライブラリ

尾 形 泰 彦[†,††] 遠 藤 敏 夫[†,††]
丸 山 直 也[†,††] 松 岡 聡[†,††,†††]

General-purpose GPU (GPGPU) を HPC の分野で利用する手法が、その非常に高いピーク性能のために注目されている。しかし、ホストとの転送 I/O 帯域幅やメモリサイズの制限などのため、実効性能は大幅に低下する傾向にある。一方で、CPU のマルチコア化も近年急速に進みつつあるため、GPU と CPU 上のアプリケーションの実効性能の乖離は小さい場合が多い。そこで我々は両者を併用する 2D-FFT ライブラリを実装し、更なる性能向上を見込む。効率的な実行のためにはヘテロなプロセッサへのタスクの分割率を適切に決める必要がある。しかし、最適な分割率は問題サイズ等に依存して変化するために、自明な問題ではない。そのために我々は、2D-FFT のアルゴリズムを詳細に考慮した性能モデルを構築する。モデルのパラメータは小規模な予備実行により決定され、それを基に任意の問題サイズと分割率に対して並列実行時間を予測することができる。実験の結果、性能モデルは予備実行の 16 倍のサイズの問題について、実行時間を 15%以内の誤差で予測した。予測から得られた最適分割率は 5%の誤差に抑えられ、この誤差に起因する性能低下は 1%以内であった。また、最適分割率における並列実行により、CPU 1 コアや GPU 単体の場合と比較して 1.19 から 1.55 倍の性能向上が得られた。

An Efficient, Model-Based CPU-GPU Heterogeneous FFT Library

Yasuhiko Ogata ,[†,††] Toshio Endo ,[†,††] Naoya Maruyama [†,††]
and Satoshi Matsuoka[†,††,†††]

General Purpose computing on Graphics Processing Units (GPGPU) is becoming popular in HPC because of the high peak performance. However, in spite of the potential performance improvements, it does not necessarily perform better than the current high-performance CPUs, especially with recent trends for increases in their number of cores on a single die. This is because the GPU performance can be severely limited by such restrictions as memory size and I/O bandwidth. To overcome this problem, we propose a model-based, adaptive library for 2D-FFT that uses available heterogeneous CPU-GPU computing resources to achieve optimal performance automatically. To find optimal load distribution ratios, we construct a performance model that predicts execution time of 2D-FFT that captures the respective contributions of CPU vs. GPU. The model parameters are determined by pre-stage performance profiling; based on then, we predict the overall execution time of 2D-FFT for arbitrary problem sizes and load distribution. Preliminary evaluation shows that the performance model can predict the execution time of problem sizes that are 16 times as large as the profile runs with less than 15% error, and that the predicted optimal load distribution ratios have less than 5% error; performance overhead caused by this error is less than 1%. We show that the resulting performance improvement by parallelization can be 1.19 to 1.55 times compared to using only a CPU core or a GPU.

1. はじめに

高性能な科学技術計算のために Graphic Processing Unit(GPU) を用いる、General-purpose GPU (GPGPU)[1] と呼ばれる手法が HPC 分野でも注目されている。最新の CPU のピーク性能が 25GFLOPS 程度であるのに対して、近年のハイエンド GPU は 500GFLOPS を誇る。また価格性能比が良好であり、数値計算専用に設計された ClearSpeed[4] や GRAPE[10] 等アクセラレータと比較して非常に低コストで性能を向上させせることが可能である。

このような特性を持つ GPU を有効利用しようとする研究が数多く行われており[7)9]、その多くは多様なアルゴリズムに対して、GPU の性能を引き出すことに重点が置かれている。しかしその実効性能は、以下に述べる３つのオーバヘッドのために低下する傾向に

† 東京工業大学 (Tokyo Institute of Technology)
†† JST,CREST
††† 国立情報学研究所 (National Institute of Informatics)

ある。１つ目のオーバヘッドは、GPU 上のメモリへのデータ転送によるオーバヘッドである。GPU 内部では 100GB/s にも迫るバンド幅が存在する一方で、CPU/GPU 間のデータ転送は 1GB/s 程度のオーダーに落ち込んでしまう。２つ目は GPU メモリの搭載量の制限によるオーバヘッドである。GPU は本来グラフィック用途に設計されているため、多くのグラフィックボードでは多くても 512MB 程度のメモリしか搭載されていない。しかし、HPC アプリケーションでは、数 GB 程度のメモリを必要とする場合も多いため、データを参照する度に送り直す必要がある等、データ転送によるコストが増大する。３つ目はプログラミング API のオーバヘッドである。実行特性が画面描画向けの特性になっており、元から GPGPU を考慮している訳ではないため、GPU のポテンシャルを生かしきれない。また、従来の DirectX や OpenGL ではメモリ管理が難しいことや、アセンブリに近いプログラミング環境しか存在しないため、必要以上にプログラマに負荷が掛かる。近年、NVIDIA CUDA[11] をはじめとする効率的な GPGPU 環境が整備されてきており、この問題は大きく軽減されつつあるが、メモリサイズ制限に伴う問題の完全な解決はされていない。

このような GPGPU のオーバヘッドに加えて、近年 CPU のマルチコア化が進んでいるため、GPU と CPU の実効性能の乖離はそれほど大きくない場合が多い。この為、GPU に加え CPU を併用する並列処理により性能の向上が期待できる。しかし、その際に必要な、異種のプロセッサ間にどのような割合で計算を割り振るべきか決定する問題は自明ではない。GPU は上記の 3 つの問題のために CPU と異なる性能特性を持ち、最適な分割率は問題サイズ等に依存して変化しうるためである。

我々はこの問題を解決するために、性能モデルに基づき、CPU 及び GPU を併用するライブラリを提案する。モデルを用いた性能予測を行うことにより、最適な計算の分割率を決定するアプローチを採る。その第一歩として本論文では、画像解析や信号解析など広い応用分野を持つ二次元高速フーリエ変換 (2D-FFT) 計算を対象とし、ライブラリと性能モデルを構築する。

このライブラリは Row-Column 法に基づいており、計算が各軸方向への 1 次元 FFT(1D-FFT) へ分解できることを利用して、計算の割り振りを行う。この 1D-FFT 計算には、GPU 側 CPU 側共に既存のライブラリを利用した。CPU 側のライブラリとして、著名な高性能ライブラリである FFTW[6] を、GPU 側のライブラリとして、Govindaraju らによる GPUFFTW[8] と、CUDA パッケージに含まれる CUFFT を用いた。また、必要に応じて GPU 側計算を複数回に分けて実行させるため、GPU メモリサイズ以上の問題サイズの計算も可能とした。

性能モデルは 2D-FFT アルゴリズムを実行単位へ分解し、この分割された単位ごとにデータ量に対する GPU, CPU それぞれの実行時間を予測する。その際のアーキテクチャに依存するパラメータは小規模な予備実行により決定される。モデルは各単位の実行時間を総合して、任意の問題サイズと分割率に対して合計実行時間を予測する。これにより最適分割率を予測することができる。

実験の結果、予備実行の 16 倍の問題サイズの場合の実行時間を予測したところ、誤差は 15%以内であった。このときモデルが予測した最適分割率は、実際と 5%以内の誤差に抑えられ、予測に基づく分割率を用いた実行では、本来の最適実行時間に対して 1%以内の性能低下しか発生しなかった。また、CPU 1 コア、GPU 単体の場合と比較して 1.19 から 1.55 倍の性能向上が得られることが分かった。

2. 関 連 研 究

GPGPU の研究はすでに多くなされており、LU 分解[7]、FFT[9] などのアルゴリズムが提案されている。それらの多くは GPU 単体での性能に注目している。その中で大島ら[13] は、GPU と CPU を併用する行列積 (GEMM) ライブラリの提案と実装を行い、単体のみの場合よりも高性能を達成している。このライブラリでは行列積演算を GPU と CPU に分割する。GPU への計算命令をまず (OpenGL API を用いて) 発行し、その後 CPU の計算を行い、そして GPU からの結果を待つ。行列積と比べ、本研究が対象とする 2D-FFT では、フェーズ間の行列転置のために GPU と CPU 間の連携が多く必要となり、構築するモデルはそれを考慮に入れている。また、大島らの研究ではシングルコア CPU を仮定しているのに対し、本研究のライブラリはマルチコア CPU を考慮に入れている。

GPU における性能モデルの構築は今までにいくつか行われている。Buck ら[3] は、グラフィックの知識なしに GPGPU を行うことが出来る言語処理系 BrookGPU を提案し、その中で転送量と計算量に各々係数を与えるという本手法に近いモデル構築を行っている。しかし、Buck らはすべてを線形近似で行う他、実性能との比較は行っていない。

そのほかの GPU 上の性能モデルの構築としては、伊藤ら[12] の研究が挙げられる。このモデルでは、メインメモリ、GPU メモリ、GPU 演算器間の通信量に注目して性能予測を行う。このため GPU プログラム実装前から実行時間を予測が可能という特徴を持つ。一方我々は、通信量だけでは説明できない、GPU メモリ確保や解放などの処理が、実行時間の無視できない割合を占めることを確認している。またこの研究自体は、ヘテロなプロセッサの利用を目的としない。

速度の異なる CPU 群を持つヘテロな環境での並列計算は広く研究されており、速度に応じてデータを分割する手法も知られている[5]。一方、我々が対象とす

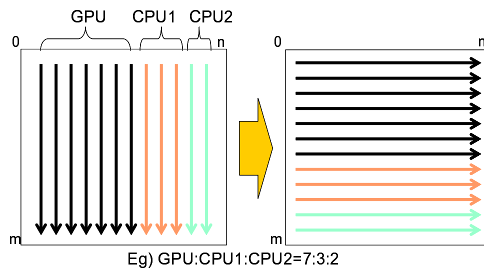

図 1　GPU/CPU への 1D-FFT の分割

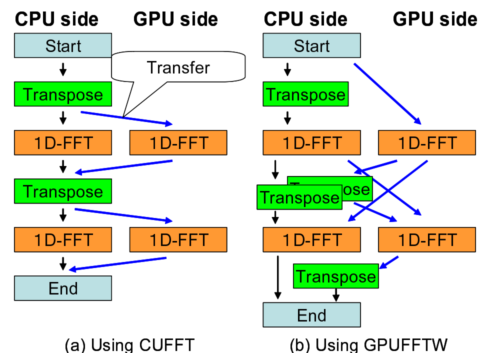

図 2　実行の詳細

る環境では、問題サイズに対する特性が大きく異なる GPU と CPU を含むため、本研究では特性を考慮したモデルを構築する。

3. GPU と CPU を併用する 2D-FFT ライブラリ

本ライブラリは、入力データとして二次元複素数配列を取り、二次元 FFT 計算を行う。我々は Row-Column 法と呼ばれる手法を採用する。この手法では、まず全ての列に対してそれぞれ 1D-FFT を行い、次に全ての行について同様に 1D-FFT を行う。我々は、この複数の 1D-FFT 処理を、CPU 用および GPU 用に設計された既存 1D-FFT ライブラリ各々に割り当ることにより、CPU と GPU の併用を行う。計算機内ではデータは 1 次元配列で表されており、2 次元の配列にアクセスする場合、特定の軸に関しては連続なアクセスとなるが、別の軸では離散したアクセスとなる。典型的な FFT ライブラリでは入力に連続データを要求する。このため、ステップ間で 2 次元配列の転置処理を行うことにより、連続なアクセスとなるようにする。

図 1 に、本ライブラリでの実行手順の例を示す。まず、列方向の 1D-FFT 処理を、後述のように与えられる GPU と CPU の分割率に応じて割り振り、実行する。次に行方向の実行も同様に行う。また、CPU 側には複数のスレッドに対して計算量を割り振ることが可能で、これにより各コアへの負荷バランスを調整する。なお、現在の GPU 上での実行が単精度しか計算出来ない制約により、本ライブラリは単精度のみの実装である。

3.1 本ライブラリが利用する FFT ライブラリ

今回、CPU 側のライブラリとして FFTW[6]、GPU 側のライブラリとして GPUFFTW[8] および CUDA[11] が提供する FFT ライブラリの 2 種類を用いた。これらは差し替えることが可能である。

FFTW[6] は MIT の Matteo Frigo および Steven G. Johnson. により開発された CPU 用 FFT ライブラリで、実行前の仮実行に基づき、実行時間の最適なアルゴリズムを自動で選択可能なライブラリである。

GPUFFTW[8] は、UNC Chapel Hill の Naga K. Govindaraju らが開発した GPU 上での FFT ライブラリである。このライブラリは、グラフィックメモリの構造を考慮した最適化を行うことが特徴である。GPUFFTW は OpenGL の NVIDIA 社による拡張である、NV_fragment_program を用いている。

CUDA (Compute Unified Device Architecture)[11] は NVIDIA が開発した GPGPU 専用環境である。NVIDIA Geforce8000 系以降の GPU に対応しており、CUDA は GPU 上のプログラムを、C 言語に近い形で簡単に書くことが可能である。このためユーザは、グラフィック API を用いずに GPGPU を行うことが可能である。また、一般的な GPGPU の実装を Geforce8000 系列上で実行した場合と比較して、より高い性能を引き出すことが可能である。今回は CUDA プログラムを記述するのではなく、CUDA パッケージ中の CUFFT ライブラリを使用した。なお、今回は用いることは出来なかったが、現在は CUDA ver1.1 がリリースされている。Ver1.1 では、後述のような、ライブラリによる CPU 時間を消費する問題が軽減されるとアナウンスされている。

3.2 実行フローの詳細

図 2 に本ライブラリにおける実行フローを示す。本ライブラリでは、1 つ以上の CPU 計算スレッドと、1 つの GPU コントロールスレッドが存在する。CPU と GPU 間の割り振り率に加え、複数 CPU 計算スレッド間の割り振り率を設定できるようにしている。

本ライブラリでは、先に列方向の 1D-FFT を行った後、行方向の 1D-FFT を行う。今回用いたライブラリのうち、FFTW および、CUFFT は Row-major のデータ並び順を要求し、GPUFFTW は Col-major のデータ順を要求する。このため、GPUFFTW を利用する図 2(b) では、CPU 側行方向の前後、GPU 側では列方向の前後と転置の順序が異なり、一方で、図 2(a) に示されている CUFFT 版では、転置は行方向の前後に行われている。転送順序に差はあるが、転送量には GPUFFTW 版/CUFFT 版で差はない。

二次元配列の転置は、現在の実装では CPU 上のみで行い、CPU 上の FFT 実行を管理するスレッド、GPU

表 1 性能モデルのパラメータ

K_{trans}	CPU での転置
K_{gMa}	GPU メモリ確保
K_{c2g}	CPU から GPU へのデータ転送
K_{g2c}	GPU から CPU へのデータ転送
K_{gP}	GPU 側 FFT 前処理
K_{gDP}	GPU 側 FFT 後処理
K_{cFFT}	CPU 側 FFT 実行
K_{gFFT}	GPU 側 FFT 実行
K_{gMf}	GPU メモリ開放

コントロールスレッドの計 2 スレッドで実行する。また、各スレッドには CPU への 1D-FFT の割り当て率、GPU への 1D-FFT の割り当て率と同じ割合で転置処理を振り分ける。

本ライブラリでは下記のように、GPU のメモリサイズ制限以上のデータを扱う。GPU 側にメモリサイズ以上のデータ量の FFT を割り振る場合には、GPU メモリに収まるサイズに調整した数の 1D-FFT を複数回呼び出す。GPU のメモリは総じて CPU のメモリよりも少ない傾向にあり、市販されているグラフィックボードに搭載されているメモリは高々768MB 程度である。さらに、グラフィック API に基づく GPGPU においては、テクスチャメモリ上の入力データとフレームバッファメモリ上の出力データをそれぞれ確保する必要があり、実際にデータを格納できる容量はさらに減る。一方で、HPC 用システムではメインメモリは CPU あたり数 GB 積まれていることが多いため、GPU メモリのみで計算可能な問題サイズは CPU の場合より大幅に減りうる。本ライブラリでは GPU 単体では計算不可能な大きさの問題を解くことを可能とした。

4. 本ライブラリの性能モデルの構築

本章では、実装した 2D-FFT ライブラリにの実行時間を予測するための性能モデルを述べる。この予測実行時間を用いることにより、実際の計算より前に最適な割り当て率を探索することが可能である。現在のところ、モデルは CPU 側の計算を 1 スレッドで行う場合を対象としている。また本章での説明は、CUFFT を用いる版を対象とする。以下の説明で、問題サイズは $n \times n$ とする。

我々が構築した性能モデルは、ライブラリを細かい実行単位に切り分け、各実行単位の実行時間を問題サイズ n、割り当て率、アーキテクチャパラメータで表現する。例えば、CPU から GPU へデータを転送するための時間は、送信するデータ量に比例すると考え、$K_{c2g} * r * n^2$ (K_{c2g} は通信性能に依存する係数、r は GPU が担当する割合、n は問題サイズ) とモデル化する。アーキテクチャパラメータについては、予備実験を通して各実行単位の時間を実測し、そこから求める。

図 3 に、本ライブラリ (CUFFT 版) の実行フロー

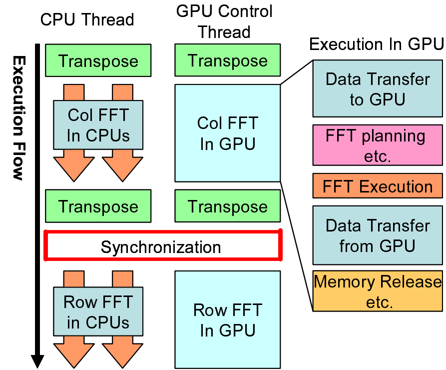

図 3 実行フローの詳細 (CUFFT 版)

を、モデル化のための実行単位ごとに示す。これらの実行単位として、ライブラリ実行のうちの主要な関数呼び出しを採用している。各実行単位に関わるアーキテクチャパラメータを、表 1 に示した。実行単位として、CPU または GPU 上の 1D-FFT 計算や通信処理以外にも、既存ライブラリを呼び出す際の前処理、後処理も考慮している。これは、既存ライブラリの利用において、実行条件を示す「プラン」の処理などに、無視できない時間がかかるためである。

なお図 3 では、GPU 内の動作が 1 セットしか書いていないが、実際には GPU メモリにおさまるサイズに列/行を分割し、複数回繰り返す。しかし、FFT の 1D-FFT 処理時間も、それに伴う通信処理も、列/行の本数に対して線形に増加するため、今回のモデル化では繰り返す回数を無視できるとした。

式 (1)(2) に CPU 計算スレッドの実行時間を表す。ここで n は問題サイズ、r は GPU 側への割り当て率である ($0 \le r \le 1$)。T_{cpu1} は列方向計算、T_{cpu2} は行方向計算についての予測実行時間を表す。二回のデータ転置の時間は、列方向に含めている。

$$\begin{aligned} T_{cpu1} &= (1-r)n^2 * (2K_{trans}) * M_{trans} \\ &\quad + (1-r)n^2 \log nK_{cFFT} && (1) \\ T_{cpu2} &= (1-r)n^2 \log nK_{cFFT} && (2) \end{aligned}$$

式 (3)(4) に同様に GPU 側のモデル式を示す。GPU 側は図 3 に示したとおり、実際の FFT 計算に加え CPU-GPU 間の転送や前処理・後処理に関する項を含む。なお、転置については GPU コントロールスレッドが CPU 上で行っており、これに関する項も加わる。

$$\begin{aligned} T_{gpu1} &= rn^2(K_{gMa} + K_{c2g} \\ &\quad + K_{gP} + K_{gDP} + K_{g2c}) \\ &\quad + rn^2 * 2K_{trans} * M_{trans} \\ &\quad + rn^2 \log nK_{gFFT} && (3) \\ T_{gpu2} &= rn^2(K_{c2g} + K_{gP} \\ &\quad + K_{gDP} + K_{g2c} + K_{gMf}) \\ &\quad + rn^2 \log nK_{gFFT} && (4) \end{aligned}$$

表 2 評価環境

CPU	Core 2 Duo E6400(2.13Ghz*2)
Memory	PC6400 4GB
Chipset	intel 975X Express
HDD	250GB
GPU	Geforce 8800GTX
OS	Linux kernel2.6.18
Driver	NVIDIA Display Driver ver97.46
	CUDA ver0.81

データ転置は、単純に担当データサイズの比例というモデルでは実測と大きくずれることが分かっている。我々のライブラリでは、データの転置の際に 1 スレッドで実行せず、CPU 計算スレッドおよび GPU コントロールスレッドの両者がそれぞれの担当範囲を並列に転置する。我々が行った予備評価によると、二次元配列を半分にして転置を 2 スレッドで実行すると、実行時間は半分にはならず、約 0.85 倍となってしまう、つまり単純な予測の 1.7 倍となる。また、転置時間が並列に行われる時間帯は分割サイズ r に依存し、0.5 の場合に一番影響を受け、0 もしくは 1 に近づくほど影響は小さくなる。この性能特性を再現するため、割り当てデータ量が大きい側に合わせて各転置時間を補正する係数 M_{trans} を設けた。

$$M_{trans} = (-0.7 * |r - 0.5|/0.5 + 1.7) \qquad (5)$$

この式により、GPU への割り当て率 r が 0.5 の場合では 1.7 倍、$r = 0$ と $r = 1$ において 1 倍となる特性を表現する。r が 0 から 0.5、0.5 から 1 の区間では線形となる。この特性の原因について調査中だが、メモリバンド幅による性能の制限に拠ると考えており、今後の研究でこの原因の究明およびモデルの補正を行う予定である。

これまでに示した各式を、列方向・行方向ごとに CPU/GPU の大きいほうの実行時間を得て、合計することで全体の実行時間を式 (6) のように求める。これは、計算の方向が切り替わる際に同期を行うためである。

$$T_{total} = max(T_{gpu1}, T_{cpu1}) + max(T_{gpu2}, T_{cpu2}) \qquad (6)$$

以上に述べたモデルは、現在のところキャッシュ等の影響を考慮しておらず、また前処理・後処理の時間は GPU の担当データサイズに単純に比例するとしている。このレベルの単純さのモデルで、実行時間をどれくらいの精度で予測可能かを、5.2 章にて検証する。

5. ライブラリ性能とモデルの精度の評価

5.1 本ライブラリの実行性能

本ライブラリの評価を表 2 に示すような Dual Core CPU を持つ環境で行った。なお、今回用いた GPU である Geforce8800GTX は 1.35GHz で動作する 128 個のストリームプロセッサを搭載し、グラフィックメモリとして 768MB の GDDR3 メモリを搭載する。

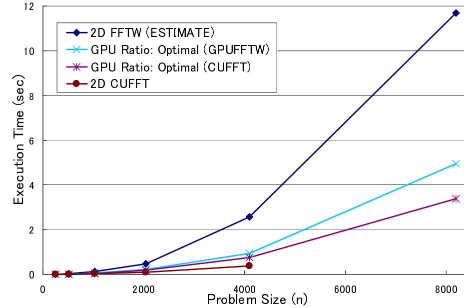

図 4 各 2D ライブラリの実行性能

図 4 に、本ライブラリの GPUFFTW 版、CUFFT 版の最適分割率での実行時間を示す。および比較のために CPU 上の FFTW と GPU 上の CUFFT の、二次元 FFT 関数 (以下、2D-FFTW および 2D-CUFFT) の実行時間を示す。横軸は問題サイズ、縦軸は実行時間である。また本ライブラリ、FFTW においては 2 スレッド利用している。

我々のライブラリは FFTW 単体よりも性能が向上していることが分かる。CUFFT 版で問題サイズが 8192 の場合、約 3.5 倍の性能向上となっている。また CPU/GPU 併用の場合、GPUFFTW 版よりも CUFFT 版の方が約 30%高速である。これは GPU 側ライブラリの性能の差によるものであり、その差は併用する場合の性能にも現れていることがわかる。

一方で、本ライブラリは全てを GPU 上で実行する 2D-CUFFT と比較して性能が出ていない。推測される原因として、本ライブラリが CPU 上で転置を行うことを前提に置いていることが挙げられる。このため、本ライブラリにおいては CPU-GPU 間の転送量が 2D-CUFFT の倍となっており、さらに転置処理自体の性能も、メモリバンド幅のために GPU より CPU の方が低速と考えられる。ただし、2D CUFFT が高速なのは GPU メモリサイズに収まる問題サイズに限られ、4096^2 を超えるサイズでは実行できない。本ライブラリはそれ以上のサイズでも計算可能であり、また CUFFT においても大きな問題サイズに対応しようとすると、やはり CPU-GPU 通信の量は同様に増加すると考えられる。

なお今後、GPU 側でも転置を行うライブラリを実装する予定である。また、全て GPU で行う場合をモデルに加え、問題サイズに応じて適応的に選択する手法も開発したい。

図 5 に、本ライブラリにおいて、GPU 割り当て率 r を 0%, 100%, 最適値 (Optimal) とした場合の性能を示す。CPU 側計算には 2 スレッドを用いている。GPUFFTW 版と CUFFT 版それぞれについて問題サイズに対する実行時間を示した。基本的には 0%(CPU のみ)、100%(GPU のみ) の場合よりも最適値の場合が高速である。CUFFT 版は本ライブラリを CPU の

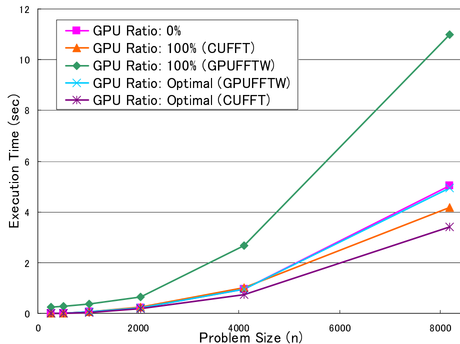

図 5 本ライブラリの各実装における実行性能

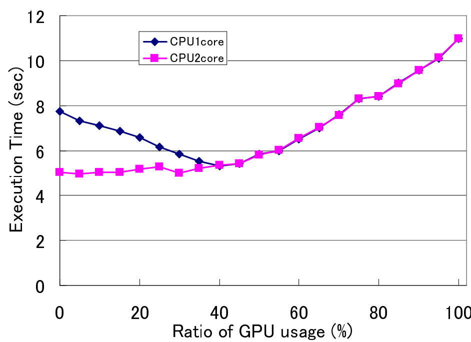

図 6 GPU 割り当て率に対する実行性能 (GPUFFTW 版)

みで使用した場合と比較して 1.5 倍性能が向上している。一方、GPUFFTW 版 (optimal) においては CPU のみと比べ性能改善が見られていないが、以下で議論する。

図 6 に GPUFFTW 版について、問題サイズ 8192 の場合の GPU 側割り当て率 r の変動と実行時間の関係のグラフを示した。CPU 側計算スレッドを 1 つ用いた場合と、2 つ用いた場合を示す。1 スレッド用いた場合には、割り当て率を 40%とした場合が最速となり、CPU/GPU を併用する利益が現れている。このとき、0%時よりも 31%, 100%時よりも 52%改善している。一方で 2 スレッド用いた場合には、CPU に加えて GPU を併用する利点表れず、さらに r が 40%以上の場合は 1 スレッド時の性能とほぼ同じとなってしまっている。この理由として、以下のように推測している。利用した GPUFFTW では、GPU 側の計算が終了するまで OpenGL の関数内で CPU 側でスピンロックを行い、CPU を消費してしまうことが分かっている。このため、2 コア CPU 上で、GPU コントロールスレッドと 2 つの CPU 計算スレッドという、3 スレッドが同時に CPU を消費し、CPU 側の計算を遅くしてしまう状況が考えられる。これにより全体性能が下がっていると考えられる。一方、CPU 計算スレッドが 1 つの場合は、この問題は起こらない。

次に CUFFT 版について、同様の評価結果を、図 7

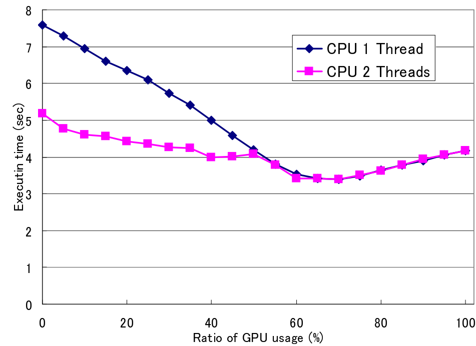

図 7 GPU 割り当て率に対する実行性能 (CUFFT 版)

に示す。全体の性能は、GPU 側への割り当てが 70%付近で最高となっている。CPU 計算スレッドが 1 つの場合、r=0%の場合に比べ 55%, r=100%の場合に比べ 19%高速となっている。また、50%以上の場合に、1 スレッドの場合と 2 スレッドの場合の性能が同じとなってしまっている。この原因は GPUFFTW 版と同様と考えている。CUDA においても、今回使ったバージョンにおいては、GPU の計算終了を待つ間 CPU を消費してしまう☆ためと推測している。なお、前の結果と異なり、2 スレッドの場合でも、r=0%よりも併用時が向上しているが、これは CUFFT 単体の性能が GPUFFTW より優れており、その差が現れていると考えられる。

5.2 性能モデルの検証

性能モデルの評価を、予測時間と実測時間の比較、および最適な GPU と CPU の分割率を推定出来るか否かについて行う。性能モデルの各パラメータについては、特定の問題サイズについて予備実験を行い、各実行単位の実行時間を測定することにより決定する。前述のように、現在のモデルは CPU 計算が 1 スレッドのときに対応しているので、その場合を調べる。

表 3 に今回用いたパラメータを示した。予備実験時に問題サイズ 512 と用いた場合と、8192 を用いた場合を示す。モデルが挙動を完全に予測可能であれば、両者のパラメータは一致するはずだが、そうはなっておらず、この点による誤差は生じると考えられる。この点は後で議論する。

図 8 に、CUFFT 版、問題サイズ 8192 の実行時間の実測値と予測値を示す。このときの性能モデルのパラメータは、同じ問題サイズ 8192 による予備実行から決定した。図から、我々のモデルは非常に性格に併用の場合の実行時間を予測していることが分かる。予測実行時間の誤差は最大で 6.9%、多くの場合は 5%未満である。また、最適な GPU への割り当て率である

☆ NVIDIA Forums[2] トピック 28524 の nVidia 開発者のコメントより。また、我々もその現象を、実行中プロセスの CPU 利用率を監視することにより観測している。

表 3 予備実験から得られたパラメータ

予備実験サイズ	8192	512
K_{trans}	1.47×10^{-8}	1.11×10^{-8}
K_{gMa}	6.93×10^{-11}	5.95×10^{-10}
K_{c2g}	5.92×10^{-9}	6.16×10^{-9}
K_{g2c}	5.09×10^{-9}	5.68×10^{-9}
K_{gP}	2.83×10^{-10}	2.73×10^{-11}
K_{gDP}	2.46×10^{-9}	7.28×10^{-12}
K_{cFFT}	3.21×10^{-9}	2.61×10^{-9}
K_{gFFT}	1.88×10^{-10}	1.65×10^{-10}
K_{gMf}	6.08×10^{-10}	3.01×10^{-9}

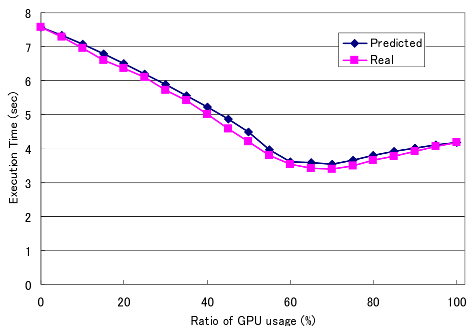

図 8 モデルと実性能の比較 (CUFFT 版,8192to8192)

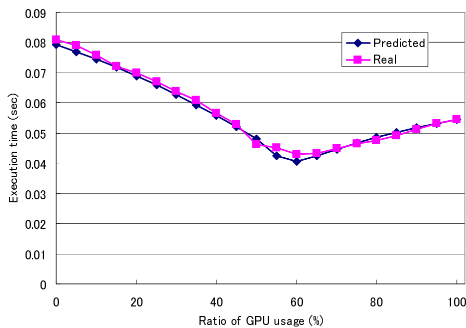

図 9 モデルと実性能の比較 (CUFFT 版,512to1024)

70%を予測できた。なお、割り当て率は 5%刻みで実測と予測を比較したので、少なくとも 5%以内の精度で、最適割り当て率を予測したと言える。

次に、小さい予備実験から、大きい問題サイズの性能を予測する場合を図 9 に示す。図 9, 図 10 では、問題サイズ 512 で予備実験を行い、問題サイズ 1024 および 8192 の場合を予測した結果を示す。

図 9 の場合は、実行時間を平均で 2%程度の誤差、最大誤差で 6%程度で予測している。また、最適分割率である 60%の予測に成功している。また図 10 によると、実行時間を少なく見積もってしまう傾向にあり、最大 15%程度の誤差が出ている。

この誤差の原因は、実行時間の内訳をとることにより、主に CPU 上で行う処理の実行時間を低く予測す

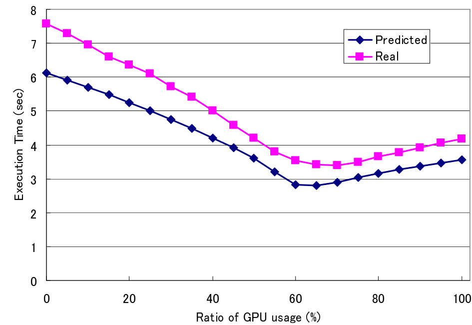

図 10 モデルと実性能の比較 (CUFFT 版,512to8192)

ることに拠ると分かった。実際に、表 3 によると、問題サイズ 512 から得たパラメータと 8192 から得たパラメータでは、特に K_{trans} の差が大きい。転置については、係数 M_{trans} の導入により並列性能の特性を考慮したが、それでもなお、より正確なモデルが必要と考えられる。例えば問題サイズがキャッシュに収まる場合と収まらない場合を分けるなどが必要と考えており、それは将来の課題の一つである。また、実行時間の内訳においては K_{cFFT} の差も同様に大きいと分かった。これは、CPU 側の前処理 (プラン作成など) を FFT 処理と同一実行単位に含めてしまうと推測され、GPU 側同様に分離することにより改善を見込んでいる。

以上のような問題はあるが、それによる最適分割率の誤差は比較的小さく、実際の 70%の代わりに 65%が得られている。この (誤差を含む) 分割率を採用して実行を行った場合、本来の最適実行時間に比べた性能低下は 1%以内である。

最後に、最適分割率を決定するためにかかった時間を述べる。図 10 の場合、実際の実行が 3.5 秒程度の計算を行うための予備実行は、計 0.032 秒であった。この予備実行を行うことで、CPU2 スレッドで実行する場合と比較して、1.78 秒の実行時間短縮が可能であり、パラメータ決定を行うコストを払っても、合計実行時間は得になる。

6. まとめと今後の課題

本研究では、GPU と CPU を併用する FFT ライブラリを提案し実装した。また、そのライブラリについて、性能のモデル化を行い、構築した性能モデルを用いて、CPU と GPU への最適な割り振り率を予測する方法の提案を行った。この性能モデルは、最適な CPU と GPU への割り振り率を最大で 5%程度の誤差で実行時間の予測が可能である。それを用いて実行を行った場合、本来の最適割り振り率の場合の実行時間と比較して 1%以内の性能低下率であった。また、このパラメータを求めるための予備実験の時間は実際の計算時間の 100 分の 1 で済むことを示した。

今後の課題は以下のとおりである。まず挙げられる課題は、CPU 側の性能モデルの精度の向上である。現状の性能モデルは、主にデータ転置時間の予測精度に改善の必要があるため、キャッシュミスの影響等を取り入れる予定である。また、現在のところモデルは CPU 側計算を 1 スレッドと仮定し、また GPU は 1 つとしている。より多数のスレッドによる計算や、複数の GPU、さらには異種の GPU からなる環境にも対応できるよう性能モデルを改良していく。なお本研究の実験では 2 コア計算機で 2 スレッドを計算に用いた場合の性能が想定より低くなった。これはグラフィック API が CPU を占有しているためと推測しているが、その問題が軽減されている新しいバージョンの CUDA での性能評価および、モデルの検証評価を行っていく。

また、一般的な問題サイズにおいて更なる高速化を行うために、CUDA の特徴を活かしたライブラリの改良を行う。たとえば GPU 側でのデータ転置を取り入れることにより、転置自体の高速化と通信削減が可能となると考えられる。今後、その手法を取り入れた場合の、GPU/CPU 併用の有用性をモデルと実験から調査していく予定である。

謝辞　本研究の一部は JST-CREST「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」および Microsoft Technical Computing Initiative の援助による。

参 考 文 献

1) GPGPU.org. `http://gpgpu.org/`.
2) NVIDIA Forums. `http://forums.nvidia.com/`.
3) Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, Vol. 23, No. 3, pp. 777–786, 2004.
4) ClearSpeed Technology plc. ClearSpeed white paper: Csx processor architecture. `http://www.clearspeed.com/`.
5) P. E. Crandall and M. J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 42–49, 1993.
6) Frigo, et al. 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".
7) Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 3, Washington, DC, USA, 2005. IEEE Computer Society.
8) Naga K. Govindaraju and Dinesh Manocha. GPUFFTW: High performance power-of-two fft library using graphics processors. `http://gamma.cs.unc.edu/GPUFFTW/`.
9) Moreland K and Angel E. The FFT on a GPU. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings*, pp. 112–119, July 2003.
10) Junichiro Makino, Eiichiro Kokubo, and Toshiyuki Fukushige. Performance evaluation and tuning of grape-6 - towards 40 "real" tflops. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 2, Phoenix, AZ, 2003.
11) NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. `http://developer.nvidia.com/cuda`.
12) 伊藤信悟, 伊野文彦, 萩原兼一. GPGPU アプリケーションの開発を支援するための性能モデル. 先進的計算基盤システムシンポジウム SACSIS2007 論文集, pp.27-34, May, 2007.
13) 大島聡史, 吉瀬謙二, 片桐孝洋, 弓場敏嗣. CPU と GPU を用いた並列 GEMM 演算の提案と実装. 情報処理学会論文誌 コンピューティングシステム, Vol. 47, No. SIG12(ACS 15), pp. 317–328, 2006.