社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－179 (15)
2008／8／6

複数GPUにおけるセルフスケジューリングによる並列数値演算

渡辺祐也[†,††] 遠藤敏夫[†,††] 松岡聡[†,††,†††]

高性能計算分野において、GPUをはじめとするコモディティアクセラレータが、その価格性能比や電力性能比のために注目されている。それらを多数用いた大規模システムの活用は重要になると期待されるが、そのようなシステムでは段階的なアップグレードにより世代の違うアクセラレータが混在しうる。特にGPUの性能向上の速さのために、性能の異なる複数GPUを効率的に利用する技術は重要になると考える。本研究ではその目的を、各GPUの性能の情報がなくとも達成するために、セルフスケジューリングを用いて動的なタスク分配を行う。計算対象として密行列積演算SGEMMをとりあげる。そして性能差のある複数のGPUを装着したマシンで性能評価と議論を行った。その結果、各GPUの性能を合計した理想的な速度と比較して94%の性能を達成した。

Parallel Numerical Computation on Multiple GPUs with Self Scheduling

Yuya Watanabe [†,††] Toshio Endo [†,††]
and Satoshi Matsuoka [†,††,†††]

In high performance computing area, commodity accelerators, especially GPUs attract considerable attention for their superior cost performance. Thus systems with a large number of those accelerators are promising. In such situations, incremental upgrade of systems will cause heterogeneity of accelerators. With the rapid advance of GPU performance, techniques to utilize heterogeneous GPUs effectively will become important. To achieve this goal without knowledge of precise performance of GPUs, we adopt the self scheduling technique for dynamic task distribution. We take the SGEMM, dense matrix multiply computation as target, and have evaluated its performance on a machine with multiple heterogeneous GPUs. The results show that self scheduling achieves 94% performance relative to the ideal speed, which is the sum of those individual speeds.

1. はじめに

グラフィック処理用のプロセッサーであるGPUを科学技術計算などグラフィックの処理以外の用途に利用するため研究が盛んに行われている。GPUの単体クロックサイクルはCPUに劣る傾向があるものの、GPUはSIMD型の計算ユニットを多数を備えておりその数はCPUよりもはるかに多く、GPUの理論ピーク性能はCPUを大きく上回っている。GPUを並列計算用のアクセラレーターとして利用することによって、いくつかのアプリケーションの実行性能の向上させる研究が報告されている[4),12),14)]。

GPUは並列アーキテクチャとみなせ、アプリケーションが複数のGPUを平行して同時に使うことでその性能を向上させることができる場合がある。実際、グラフィック用途ではGPUベンダーがGPUを複数使用して処理能力を向上させる方法を提供している(NVIDIAのSLIやAMDのCrossFire)。

また、GPUは進歩が早く、予算などの都合でGPUの購入時期が違うと、購入するGPUの種類が異なるという状況が想定される。例えば、最初の購入時点ではNVIDIA GeForce 8800 GTXが最適なGPUであると考え購入したとする。その後、新たにGPUを購入する機会にNVIDIA GeForce 9800GX2を最適なGPUであると考えて購入したとする。そうすると、これらのGPUを複数利用した場合、GPUの環境は不均質なものとなる。

以上のことを踏まえ、ヘテロな環境でも性能を向上させやすい複数GPUの並列実行方法としてセルフスケジューリングを使用する方法を提案し、計算対象として行列積を適用して性能評価をした。複数のセルフスケジューリング手法の比較を行った。

その結果、GPU単体の性能の和に対して複数GPUを同時実行したときの性能比が最大で94%を達成した。比較したセルフスケジューリング手法の中ではChunk

† 東京工業大学
†† JST
††† 国立情報学研究所

Self-Scheduling がもっともよい成績を出した。

2. GPU コンピューティング

2.1 GPU コンピューティングの方法

GPU は高い理論ピーク性能を持っており、グラフィック処理以外の用途に使うための手法は、この 1, 2 年で急速に変化している。数年前まで GPU でプログラミングするためには、DirectX[1] や OpneGL[2]、Cg[3] などのグラフィック処理のためのライブラリを使って、適応したい計算を表現しなければならなかった。そのため GPU でプログラミングするためには、グラフィック API を学習するというハードルがあった。さらに、グラフィック API ではメモリアクセスの可能な領域・タイミングの制限が厳しいなど、科学技術計算で高い性能が出すのは容易なことではなかった。

GPU での汎用プログラミングを容易にするための処理系は BrookGPU[4] や Sh など多数開発されている。GPU ベンダーにおいても、まず AMD(旧 ATI) は、自社の GPU 向けに Close to Metal(CTM)[6] を提供している。これはアセンブラ言語に似た低レベルな言語で、ハードウェアに近いレベルでプログラムすることができる。さらに BrookGPU を拡張した Brook+[6] などを実装しているが、後述の CUDA[7] に比較すると今のところプログラミングの制限は大きい。

もうひとつの GPU ベンダーである NVIDIA は、CUDA(Compute Unified Device Architecture) を提供している。CUDA は C 言語を拡張した文法の言語で、プログラミングしやすく現在 GPGPU 向けの研究でよく使われている。しかし、CUDA でさえも、高性能を発揮するためには GPU のアーキテクチャなどを考慮したプログラミングが必要であるし、また CUDA 自身が複数もしくはヘテロな GPU への自動対応をするわけではない。

2.2 CUDA と CUBLAS ライブラリ

本研究では上述の CUDA 環境および、そこから利用可能な BLAS ライブラリである CUBLAS を用いる。

CUDA は NVIDIA の G80 アーキテクチャとそれ以降の世代のアーキテクチャで動作する。G80 は 1 個以上のストリーミングマルチプロセッサー (MP) から構成され、各ストリーミングマルチプロセッサーは 8 個のストリーミングプロセッサー (SP) とキャッシュメモリを持っている。SP は単精度の浮動小数点演算を行うことができる。GPU 内部にあるスケジューラーは 32 スレッド分のタスクを 1 単位として 1 個の MP に仕事を割り当てる。そのため、CUDA ではスレッドの数が 32 の倍数のときに高い性能を発揮する。

NVIDIA は CUDA と共に BLAS を実装した CUBLAS ライブラリと、FFT 演算を実装した CUFFT ライブラリを提供している。本研究では CUBLAS を行列積の計算に用い、その中の SGEMM 関数を利用する。上記の G80 アーキテクチャの性質から、与える行列サイズは 32 の倍数のときに性能が良い傾向にある。

3. セルフスケジューリング

3.1 セルフスケジューリングの概要

並列計算において重要な事は、複数のプロセッサー間で仕事を効果的に分配することである。通信のオーバーヘッドなども考慮したワークロードを分配するスケジューリングアルゴリズムが数多く提案されてきた。

そうしたアルゴリズムのひとつにセルフスケジューリングがある。セルフスケジューリングは依存関係のないループ実行などに使われる手法で、問題をプロセッサー数よりも多いタスクに分割しそれを順次実行していく。

セルフスケジューリングはマスター・スレイブモデルで表現され、基本的には次のように動作する。

(1) マスターノードは各スレイブノードにタスクを割り当てる。
(2) スレイブノードは受け取ったタスクを処理する。
(3) タスクの処理が終わったスレイブノードは、新しいタスクをマスターノードにリクエストする。
(4) マスターノードはもしまだタスクがあれば、リクエストを出したスレイブノードに新しいタスクを割り当て処理させる。
(5) もしタスクがなければ、スレイブノードがすべてのタスクを処理するのを待ってプログラムを終える。

3.2 タスク割り当て方法

スレイブノードに割り当てるタスク量について、多数の割り当て方法が提案されている[8][9]。本論文では最も単純な Chunk Self-scheduling の評価をまず行い、他の手法とも比較する。以下では I をタスクの総量、P をプロセッサーの数、T_N を N ステップ目のタスク量、R_N を N ステップ目の残存のタスク量とする。

- **Chunk Self-Scheduling:** 毎回一定のタスク量 K を割り当てる。タスク量 K はユーザーが入力できるようにしておく場合が多い。この方法は最適な K の値を見つけることが難しいが、スケジューリングのオーバーヘッドは非常に小さくなる。
- **Guided Self-Scheduling:** 計算の後半には小さいタスクを割り当てることにより、プロセッサ

間のバランスを向上させることをねらいとする。$T_1 = \lceil \frac{I}{P} \rceil, T_N = \lceil \frac{T_{N-1}}{P} \rceil$ とタスクを割り当てる。後半のタスクサイズが小さくなりすぎ、オーバヘッドが大きくなる場合がある。

- **Trapezoid Self-Scheduling:** ねらいは上記と同様だが、小さすぎるタスクサイズを避ける。最初のタスク量 F と最後のタスク量 L を既知のパラメータとし、ステップごとに一定の割合 D ずつタスクの量を減らしていく。F と L の値として Tzen と Ni が $F = \frac{I}{2 \times P}$ L=1 を提案している[11]。
- **Factroing Self-Scheduling:** タスクの実行が終わる時間が予測できない場合によい性能を発揮しやすい。$T_N = \lceil \frac{R_{N-1}}{\alpha p} \rceil, R_0 = I$ によって N ステージ目のタスク量を決定する。各ステージでは p 回同じタスク量を割り振る。パラメーター α は計算かもしくは $\alpha = 2$ と割り振る。

上で挙げたもの以外にも、タスク量を増やしていく Fixed Increase Self-Scheduling アルゴリズムや、後者の 2 つを組み合わせた Trapezoid Factoring Self-Scheduling などのヴァリエーションなどが提案されている。また、グリッド向けのアルゴリズムも提案されている[8)~10)]。

4. 提案システムと実装方法

本研究では、複数の GPU に対するロードバランシング手法としてセルフスケジューリングを用いた。計算対象として行列積を選んだ。

4.1 行列積演算の分解

3 節で述べたように、セルフスケジューリングを行うためには問題を依存関係のないタスクに分割する必要がある。行列積の計算を独立したタスクに分けるのは簡単でいくつかの方法が考えられるが、今回は次のような方法をとった。

行列 A と行列 B を掛け合わせて行列 C を求める場合を例にして説明する。行列 A のサイズを $m \times k$、行列 B のサイズを $k \times n$ とする。すると、答えの行列 C のサイズは $m \times n$ となる。

行列 A と C の行数はともに m 行であり、行列 A の第 x 行目を a_x とし行列 C の第 x 行目を c_x とする。c_x を求めるためには a_x と行列 B の積をとればよい。つまり $a_x \times B$ を計算すれば c_x を求めることができる。

このような計算方法をとれば、行列 A（もしくは行列 C) の 1 行 1 行を 1 個のタスクとみなすことができる。例えば、行列 A が 4000 行 ($m = 4000$) であれば 4000 個のタスクに分解できる。現在のマザーボードにはせいぜい 4 個程度の GPU しか備え付けることができないので、プロセッシングエレメントに対してタスクの数は十分にあり、GPU でセルフスケジューリングを行うのに適した状況であると考えられる。

4.2 実装方法

GPU での行列積の計算には CUDA 上で実装され NVIDIA から公開されている CUBLAS ライブラリを使用した。今回実験に使った GPU では倍精度の浮動小数点演算がネイティブにサポートされていないので、単精度の行列積演算を行う cublasSgemm() 関数を用いた。

CUDA 環境で複数の GPU を使う場合、1 スレッドに 1 個の GPU を結びつける必要がある。したがって、2 個以上の GPU を使う場合は 2 スレッド以上を使うマルチスレッドプログラミングを行う必要がある。

加えて、GPU 制御用スレッドとは別にマスターノードのためにもスレッドも使った。

GPU 制御用スレッドをスレイブノードとし、3 節で説明したセルフスケジューリングを実装した。

5. 評　価

5.1 実験環境

GPU 以外の実験環境についての情報を表 1 にまとめた。

表 1 実験環境

CPU	Core2 Quad Q6700(2.66GHz)
メモリ容量	4GB
オペレーティングシステム	Fedora 8
kernel	1.6.23.9-85
接続バス	PCI-Express x16(Gen 2.0)
CUDA	1.1

GPU についてはに表 2 に重要な性能とともにまとめた。今回の実験もちいた GPU は NVIDIA の GeForce 8800 (G92)、GeForce 8800 GTX、GeForce 8800 GT、GeForce 8600 GTS である。表ではスペースの都合で略記した。

表 2 GPU 環境

	88gts	88gtx	88gt	86gts
MP	16	16	14	4
SP	128	128	112	32
メモリ (MB)	512	768	504	256
Clock rate(MHz)	1650	1350	1500	675
ピーク性能 (gflops)	634	512	504	138

5.2 評価方法

今回の実験では行列のサイズは全て正方行列とした。また、行列のサイズは32の倍数とし、セルフスケジューリングのタスクサイズも32の倍数とした。前にも触れたようにNVIDIAのGPUはスレッドの実行単位が32スレッドとなっており、CUBLASのsgemm関数の実装でも今回扱う行列はほとんどが内部で32×32の単位で計算を実行している。そのため、行列のサイズとセルフスケジューリングのタスクサイズも32の倍数とした。

5.3 Chunk Self-Schedulingの性能と単体性能の合計の比較

8800 GTSと8800 GTX2個GPUを個別に測定した合計の合計値と、同じGPU 2個を使ってチャンクセルフスケジューリングを使用して同時に実行したときの性能を比較した。結果を図1に示す。Chunk Self-Schedulingの値は、入力値を32から32ずつ増加させ704まで変化させたときの最良値を取った。セルフスケジューリングで2個のGPUを同時に使って性能向上していることが確認できる。図の中のSumは8800 GTSと8800 GTXの性能の単純な和である。単独では実行できないサイズの行列の計算もできた。さらに、行列サイズが8000を超えたあたりの性能はSumに対して90%を超える性能を発揮した。最大で94%の性能で実行することができた。

8800 GTXに対して8800 GTS、8800 GT、8600 GTSを組み合わせたときの単体性能の和に対するチャンクセルフスケジューリングの性能比を図2に示す。どの組み合わせでも行列サイズが大きくなると性能比が向上し、単体のGPUが実行できる限界近くでは90%を超える性能を発揮した。上に書いたように、単体で実行できない行列サイズの領域もChunk Self-Schedulingは実行可能で、この領域ではさらに性能が向上した。

5.4 静的割付とチャンクセルフスケジューリングの比較

8800GTSと8800GTXを用いて、(1)理論的な性能比(45%)でタスクを分割 (2)静的割り当てによる各行列のサイズ毎の最良値 (3)セルフスケジューリングを行った場合を比較した。行列サイズによらずセルフスケジューリングが静的割り当ての場合を上回った。性能差は最大で5.4%であった。結果を図3に示す。静的割り当てのサイズ毎の最適値は、理論性能比の周辺を1%ずつずらして探し出した。

5.5 セルフスケジューリング手法の比較

3節で述べたセルフスケジューリング手法を比較した実験の結果を図4に示す。この実験で使ったGPUはGeForce 8800 GTとGe Force 8600 GTSの2種である。この2個のGPUは性能差が大きく理論性能値や実行時間は約3倍違う。

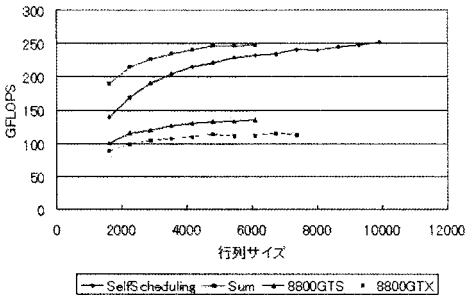

図1 セルフスケジューリングと合計性能の比較

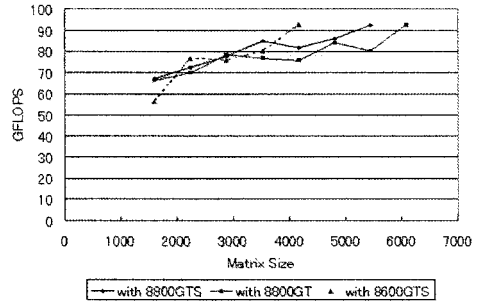

図2 セルフスケジューリングの性能

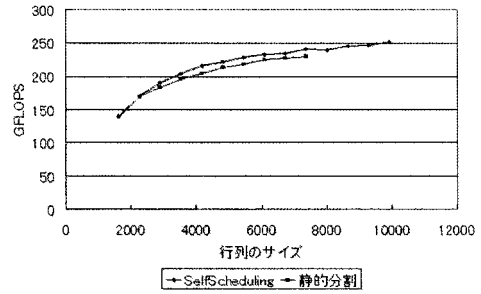

図3 静的割付とセルフスケジューリングの比較

各Self-Schedulingアルゴリズムの入力値とプロットに使った代表値は次のように選んだ

- **Chunk Self-Scheduling** 行列サイズ1つにつき入力Kの値を32から704まで32区切りで実行し、最良値を代表値に使った。
- **Trapezoid Self-Scheduling** 入力値はFと

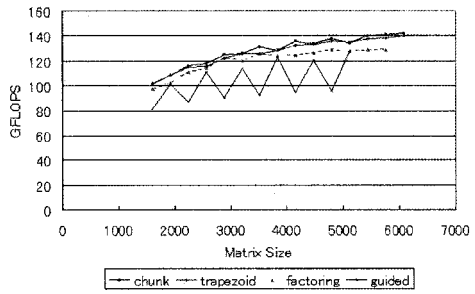

図 4 セルフスケジューリング手法の比較

L である。L の値を 32 と固定した。F の値は 32 から 704 まで 32 区切りで変化させて実行し、最良値を代表値に選んだ。

- **Factoring Self-Scheduling** 入力値は α。今回は文献で推薦されていた $\alpha = 2$ だけを試した。
- **Guided Self-Scheduling** 入力値はない。複数回実行した中で最良値をとった。

今回の実験した行列サイズの種類は 15 サイズである。サイズごとに性能が最高であったものを調べると、Chunk が 12 サイズで最高の性能を出し, 残りの 3 サイズは Trapezoid が最高だった。今回の実験では、Chunk と Trapezoid は行列サイズ 1 つにつき 15 パラメータで実験し、その最良値をプロットしている。スペースの都合でデータは省略するが、行列サイズごとの平均値は Trapezoid の方が良い例が多かった。行列 15 サイズ中、14 サイズで Trapezoid が Chunk の平均に勝っていた。

Factoring は最良値をとるサイズはなかった。しかし、行列サイズが小さいときは比較的良い性能を示した。

Guided は実行ごとに実行時間の変化が大きい。どの行列サイズでも他のセルフスケジューリングよりも性能が悪かった。繰り返し実験したが、Factoring にも勝るケースはなかった。

できるだけ高い性能で実行したい場合は Chuk と Trapezoid からパラメータサーチでよい実行例を探すのがいいと思われる。また、パラメーターサーチをわざわざする必要がない場合には、平均的に性能の高い Trapezoid もしくは小さな行列サイズでは Factoring を使うべきだと考えられる。

6. 関連研究

本章では紙面の都合のため、GPU などのアクセラレータを用いたヘテロな環境での並列計算についての関連研究を述べる。大島らは CPU と GPU を使った行列積の並列計算を行っている[12]。行列を適切な比率で分割し、CPU 上と GPU 上の BLAS ライブラリを並列に呼び出すことにより両者を併用する。また著者らはすでにアクセラレータを用いた大規模システムの性能を評価している[13]。10000 個以上の CPU コアと 600 個以上の ClearSpeed アクセラレータを併用した Linpack ベンチマークの実行において、50TFlops 以上の性能を実現した。CPU とアクセラレータ間のタスク分割率については手動でチューニングを行っている。

一方尾形らは CPU と GPU 間のタスク分割率を半自動的に見つける研究を行っている[14]。多次元高速フーリエ変換 (FFT) 演算を対象とし、適切なタスク分割率を半自動的に見つけるために、詳細な性能モデルを構築している。性能モデルは、CPU・GPU 単体の FFT 速度や PCIe 通信速度などのアーキテクチャパラメータや問題サイズを考慮する。アーキテクチャパラメータについては予備実行により求めておく。これを用い任意の分割率による並列実行時間を求めることができるので、最適な分割率を得ることができる。

以上の研究はヘテロなプロセッサ間のタスク分割率を前もって (手動で、もしくは半自動的に) 求めておくアプローチである。一方本研究ではセルフスケジューリングを採用することにより、そもそも分割率を求めなくとも効率的にヘテロな GPU を利用可能であることを示した。そしてその性能は (原因は究明中であるが) 静的な分割の場合よりも良好であった。

7. まとめ

本研究ではセルフスケジューリングという手法を用いて複数の GPU に対する仕事の割り当てを行う実験をした。その結果、実験した GPU の組み合わせではどの組み合わせでも単体の GPU の性能合計に対して 90%を超える性能を発揮することと、静的にタスクを割り当てたときより最大 5.4%セルフスケジューリングの方が性能が高いことを示した。さらに、セルフスケジューリングアルゴリズムの比較も行った。最も高い性能を出したのは Chunk Self-Scheduling だった。

今後の検討課題として、第一に FFT など他の計算に対してセルフスケジューリングを適用することが挙げられる。行列積の計算量はデータ量 n に対して $O(n^3)$ であり、工夫をしなくてもデータ転送に対する実行時間の割合が高く複数の GPU の性能を発揮しやすかった。例に挙げた FFT はデータ転送に対して計算量が $O(nlog(n))$ であり、複数の GPU を利用したときに

データの転送時間が大きな問題になる。このような問題でセルフスケジューリングがどの程度効果があるのか確かめる必要があると思われる。

第二に、データ転送と計算の実行をオーバーラップさせるアルゴリズムをセルフスケジューリングに組み込む必要があると考えられる。現在提供されているCUBLASライブラリのインターフェースではこの機能を利用できないが、公開されているソースコードに手を加えることなどによって、この機能を追加できないか検討中である。

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，科学研究費補助金特定領域研究 (18049028) の補助による．

参 考 文 献

1) C. Peeper and J. L. Mitchell, "Introduction to the DirectX 9 high level shading language" in Shader X2, W. R. Engel, Ed. Plano, Wordware, 2003.
2) R. J. Rost, "OpenGL Shading Language, 2nd ed. "Addison-Wesley, 2006.
3) W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language" ACM Trans. Graph., vol. 22, no. 3, pp. 896.907, 2003.
4) Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. "Brook for GPUs: stream computing on graphics hardware" ACM Trans. Graph., Vol. 23, No. 3. (August 2004), pp. 777-786.
5) M. McCool and S. Du Toit"Metaprogramming GPUs With Sh. "Wellesley AK Peters, 2004.
6) AMD Streaming Computing http://ati.amd.com/technology/streamcomputing/index.html
7) NVIDIA CUDA Homepage http://www.nvidia.com/object/cuda_home.htm
8) A. T. Chronopoulos, M. Benche, D. Grosu, and R. Andonie. A class of loop self-scheduling for heterogeneous clusters. Proceeding of the 2001 IEEE International Conference on Cluster Computing, pages 282.294, Newport Beach, USA, October 2001.
9) Anthony T. Chronopoulos, Satish Penmatsa, Ning Yu, Du Yu. "Scalable loop self-scheduling schemes for heterogeneous clusters" International Journal of Computational Science and Engineering (IJCSE), Vol. 1, No. 2/3/4, 2005
10) Javier Díaz ,Sebastián Reyes ,Alfonso Niño and Camelia Muñoz-Caro. "A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems", Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona,Spain
11) T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: Apractical scheduling scheme for parallel compilers. IEEETransactions on Parallel and Distributed Systems, 4(1):87.98, January 1993.
12) 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 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)
13) Toshio Endo and Satoshi Matsuoka. "Massive Supercomputing Coping with Heterogeneity of Modern Accelerators". Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2008), April 2008.
14) 尾形 泰彦, 遠藤 敏夫, 丸山 直也, 松岡 聡. "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" 第 8 回ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集（HPCS 2008），p. 107-114, (2008-01).