アクセラレータ技術WG成果報告書

(WG活動期間:2009年10月-2012年3月)

2012年3月26日

サイエンティフィック・システム研究会

アクセラレータ技術ＷＧ

目次

1. はじめに 1
1.1 背景と目的 1
1.2 メンバー 2
1.3 活動経緯 3
2. ディレクティブ方式による GPU 利用 4
2.1 PGI Accelerator 概要 4
2.2 PGI Accelerator によるプログラム GPU 化事例 6
2.2.1 事例 1：姫野 BMT 6
2.2.2 事例 2：2D-FDTD 9
2.2.3 事例 3：分子軌道法（ERI-MO） 10
2.3 まとめ 12
3 アクセラレータ向け拡張言語による GPU 利用 14
3.1 CUDA 概要 14
3.2 CUDA によるプログラム最適化事例 15
3.2.1 事例 1:分子軌道法（ERI-MO） 15
3.2.2 事例 2:Linpack 18
3.3 OpenCL 概要 (CUDA との比較) 23
3.3.1 OpenCL と CUDA の性能比較 24
3.3.2 OpenCL のポータビリティ 26
3.4 まとめ 27
4. ハードウェア動向 29
4.1 新旧 nVidia GPU アーキテクチャ比較 29
5. おわりに 35

1. はじめに

1.1 背景と目的

スパコンシステムの計算性能は、これまで概ね 3 年で 10 倍のペースで向上してきた。近年はプロセッサの動作周波数を向上させることが技術的に難しくなったため、プロセッサコアの単体性能を向上させるのではなく、スパコンを構成するノード数を増やす、プロセッサのコア数を増やす（マルチコア化）など、数を増やすアプローチで性能向上のペースを維持してきた。その中で、近年注目されているのがアクセラレータである。

汎用プロセッサは、HPC 以外の様々な用途での使用を想定しており、様々なプログラムで簡単に安定した性能が得られる様に実装面積の半分以上をキャッシュメモリとして使用しており、演算ユニットの占める面積比率は低い。それに対して、アクセラレータは HPC 用途にフォーカスしたプロセッサで、キャッシュメモリなどの非演算ユニットのリソース量を最小限に抑え、多数の演算ユニットを搭載して、演算ユニットの面積比率を高めることで、演算性能を高めるアプローチのプロセッサである。アクセラレータは高い演算性能を持つ反面、プログラム開発が難しいなど、その性能を引き出すのは簡単でない。実際に、特定アプリでは高い性能が得られるが、それ以外のアプリでは高い性能が得られないという問題がある。しかし、エクサスケール世代では飛躍的に電力あたり演算性能を高める必要があるため、汎用プロセッサと比べて電力あたり演算性能の優れたアクセラレータ技術は、エクサスケール世代の必須技術であると期待されている。

このような背景の下、SS 研アクセラレータ技術 WG を立ち上げ、アクセラレータとして近年普及しつつある GPU を事例として、将来のアクセラレータのあるべき姿を模索・議論した。特に、GPU に代表されるアクセラレータ上でのプログラム開発の現状とその課題、また、どうすればアクセラレータで高い性能を得られるのか、具体的な事例を用いて調査した結果を報告する。本報告は 2 年にわたる WG 活動での議論・調査結果をまとめたものである。

1.2 メンバー

	氏名	所属
担当幹事	村上 和彰	九州大学
推進委員	井上 弘士 ※	九州大学
	青木 尊之	東京工業大学
	遠藤 敏夫	東京工業大学
	黒川 原佳	理化学研究所
	滝沢 寛之	東北大学
	伊野 文彦	大阪大学
	本田 宏明	九州先端科学技術研究所
	久門 耕一 ※	(株)富士通研究所
	堀田 耕一郎	富士通(株) 次世代テクニカルコンピューティング開発本部
	丸山 拓巳	富士通(株) エンタプライズサーバ事業本部
	坂口 吉生	富士通(株) TC ソリューション事業本部
	佐々木 啓	富士通(株) TC ソリューション事業本部
	成瀬 彰	(株)富士通研究所

※まとめ役

1.3 活動経緯

会合	日時	場所	活動内容
第一回	2009 年 10 月 29 日(木) 14:30～16:30	富士通本社	• WG 全体スケジュールの議論 • 今後の議題検討
第二回	2010 年 1 月 19 日(火) 14:30～17:00	蒲田 ソリューション スクエア	• 会員報告 (PGI アクセラレータコンパイラ、コーンビーム再構成の高速化、初心者による GPU プログラミング事例、複合型計算機向けソフトウェア開発環境) • 富士通報告 (GPGPU プログラミングの状況、GPUSIM から見た GPU) • 添付資料：1～7
第三回	2010 年 4 月 7 日(水) 14:30～17:10	富士通本社	• 会員報告 (Linpack on GPU 搭載スパコン Tsubame) • 富士通報告 (OpenCL の評価) • OpenCL に関する議論 • 添付資料：8～9
第四回	2010 年 7 月 14 日(水) 14:30～17:10	富士通本社	• 会員報告 (PGI コンパイラ評価) • 富士通報告 (nVidia 新 GPU 評価) • 添付資料：10～11
第五回	2010 年 12 月 3 日(金) 14:30～17:00	富士通本社	• 会員報告 (アクセラレータの大規模システム導入課題、Linpack on Tsubame2) • 添付資料：12～13
第六回	2011 年 3 月 25 日(金) 14:30～17:00	富士通本社	• 会員報告 (PGI コンパイラ評価) • 富士通報告 (分子軌道法プログラムの GPU 化検討) • 添付資料：14～15
第七回	2011 年 8 月 19 日(金) 14:30～17:00	富士通本社	• 会員報告 (CUDA Fortran 評価) • 富士通報告 (分子軌道法プログラムの GPU 化結果) • 添付資料：16～17
第八回	2012 年 1 月 19 日(木) 14:30～17:00	トラステッド クラウド スクエア	• まとめ

2. ディレクティブ方式による GPU 利用

ここでは、汎用言語で記述されたプログラムに、ディレクティブを挿入して GPU を利用する形態の評価結果を報告する。

汎用言語を用いて記述されたプログラムを GPU で動作させる一般的な方法は、既存の C や Fortran で記述されたソースコードに対して、開発者が GPU 向けのディレクティブを挿入する方法である。共有メモリ並列環境向けの並列プログラミング言語・環境である OpenMP との類似性が高いプログラミングモデルと言える。コンパイラは、挿入されたディレクティブに基づいて、ソースコード内のホスト実行部分とアクセラレータ実行部分を識別し、ホスト用オブジェクトとアクセラレータ用オブジェクトを生成する。OpenMP と同様、ディレクティブ方式の言語・開発環境では、GPU 向けにソースコードを大幅に変更する必要がないので、ソースコードが表現しているアルゴリズムの可読性を保つことができ、アルゴリズムを変更するときの開発者の手間が少ないメリットがある。その反面、後節で説明する CUDA による GPU 化と比べ、性能は低くなるといったデメリットがある。さらに、汎用言語によるオリジナルコードの記述方法によっては、ディレクティブ方式の言語面からの制約により、アクセラレータ実行用のオブジェクト生成が行われない場合もある。

2.1 PGI Accelerator 概要

ここでは、nVidia GPU 向けのディレクティブ挿入方式のプログラム開発言語・環境として、本 WG 期間に利用可能であった PGI Accelerator の評価結果を報告する。

PGI Accelerator は、PGI 社が販売している nVidia GPU に対応したディレクティブベースのコンパイラであり、下記のディレクティブを挿入して GPU 実行領域を指定することで、汎用 CPU 向けに開発されたプログラムを GPU 上で実行することができる。また、GPU コンピューティングで問題となる Host・GPU 間のデータ転送も、適切にディレクティブを挿入することで制御できる。以下に PGI Accelerator の主要ディレクティブを示す。

- **Accelerator Compute Region Directive:**

  GPU 実行部分を指定するディレクティブ。
  このディレクティブだけで既存のプログラムを GPU 上で実行させることができる。コンパイラの判定により、ある程度のデータ転送は自動で行われるが、性能を求めるのであれば、人手で細かく制御する必要がある。

```
!$acc region [clause [, clause] ..]
        structured block
!$acc end region
```

Clause:

copy(list): region の入口で CPU→GPU のコピー、
出口で GPU→CPU のコピーを行なうデータを指定
copyin(list): 入口で CPU→GPU のコピーを行なうデータを指定
copyout(list): 出口で GPU→CPU のコピーを行なうデータを指定
local(list): region 内で local なデータを指定

- **Accelerator Loop Mapping Directive:**

ループをどのように並列化するかを指定するディレクティブ。

!$acc do [clause [,clause] …]
do loop

clause: (指定しない場合はコンパイラが自動選択)

host : host 上で逐次に実行する
seq : accelarator 上で逐次に実行する
unroll : unroll を指示，accelerator 上で逐次で実行する
parallel : parallel mode で並列化する
vector : vector mode で並列化する
cache : メモリ階層の上位に移動することを試す
independent : ループ内のデータに依存関係がないことを示す
kernel : ループボディが計算カーネルであることを示す
private : デバイス上のループの 1 反復で 1 回しかコピーされない
shortloop : accelarator に十分なループかどうかを指示

- **Accelerator Data Region Directive:**

GPU でデータを保持する領域を設定するディレクティブ。

!$acc data region [clause [, clause] …]
structured block
!$acc end data region

clause:

copy : region の入口で CPU→GPU のコピー、
出口で GPU→CPU のコピーを行なうデータを指定

copyin：　入口で CPU→GPU のコピーを行なうデータを指定
copyout：　出口で GPU→CPU のコピーを行なうデータを指定
local：　region 内で local なデータを指定
mirror：　Fortran だけで有効．領域内で動的に領域の allocation, deallocation が可能
update device/host：　region，data region の外部でコピー
update device：　計算実行前か入口で CPU→GPU にコピー
update host：　計算実行後か出口で GPU→CPU にコピー

2.2 PGI Accelerator によるプログラム GPU 化事例

ここでは PGI Accelerator を用いて実際にプログラムを GPU 化した事例を報告する。事例として用いたプログラムは、姫野 BMT、2D-FDTD、分子軌道法、の 3 種類である。姫野 BMT は、開発者Ａが GPU 化を行い、2D-FDTD と分子軌道法は開発者 B が GPU 化を行った。各開発者のプログラミングスキルは以下の通りである。

開発者Ａ	OpenMP・MPI による並列プログラム開発経験はあるが、PGI Accelerator は使用したことがない。
開発者Ｂ	計算科学分野のアプリケーション開発経験は豊富。逐次プログラム開発が主体であり、プログラムの並列化はあまり詳しくない。
開発者Ｃ	並列処理のエキスパートで、CUDA・OpenCL でのプログラム開発経験がある。

2.2.1 事例 1：姫野 BMT

ここでは、PGI Accelerator を用いて姫野 BMT を GPU 化した結果を報告する。

プログラムの GPU 化	開発者Ａ
GPU 種類	Tesla C1060
PGI Accelerator	Version 10.0

PGI Accelerator 使用経験のない開発者Ａが、Fortran 版の姫野 BMT を PGI Accelerator (version 10.0)で GPU 化したところ、Tesla C1060 で 19.2GFLOPS の性能が得られた。開発者Ｃが開発した CUDA 版姫野 BMT では、同じ GPU で 50 GFLOPS 程度の性能を達成可能であり、それに比べて 40%程度の性能しか得られないという結果となった。

開発の容易さの観点では、開発に要した時間は1～2時間程度であり（PGI Accelerator の directive 学習時間を含めると数日）、また、19.2 GFLOPS を達成したソースに挿入したディレクティブは 8 行であった（ディレクティブ挿入後のコードは以下を参照）。これだけ簡単に最高性能の 40%程度の性能が得られており、開発時間あたりの効率は良いと言える。

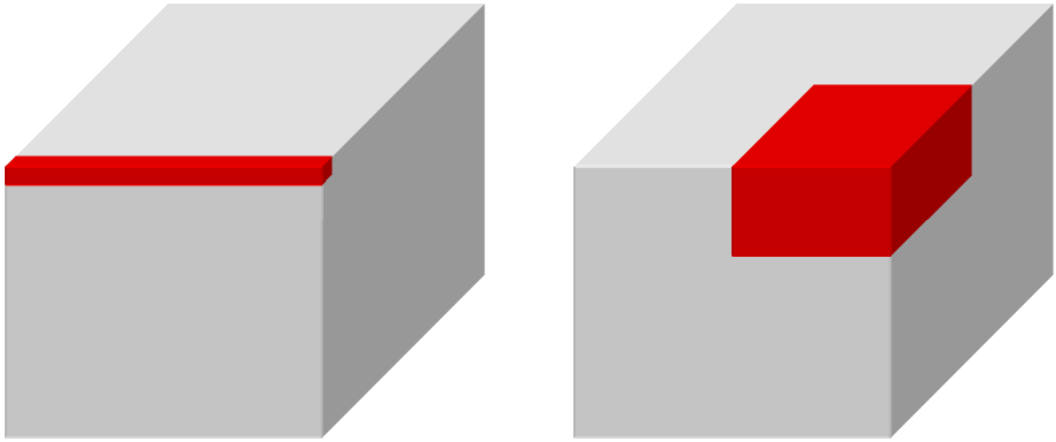
	GFLOPS	Directive 数	開発工数
PGI Accelerator 版 Compute Region Directive のみ	14.9	2 行	1 時間 (学習に数日)
PGI Accelerator 版 Loop Mapping Directive 使用	19.2	8 行	+30 分
CUDA 版 (by 開発者Ｃ)	50	…	数日
(参考：Xeon X5670 2.93GHz 1 コア)	3.9	…	
(参考：Xeon X5670 2.93GHz 4 コア)	7.8	…	

[ディレクティブ挿入後の姫野 BMT ソースコード]

```
!$acc region local(wrk2)
do loop=1,nn
        gosa= 0.0
!$acc do parallel(3)
        do k=2,kmax-1
!$acc do parallel(3)
                do j=2,jmax-1
!$acc do vector(256)
                        do i=2,imax-1
                                s0=a(I,J,K,1)*p(I+1,J,K)+…(省略)…
                                ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
                                GOSA=GOSA+SS*SS
                                wrk2(I,J,K)=p(I,J,K)+OMEGA*SS
                        enddo
                enddo
        enddo
!$acc do parallel(16)
        do k=2,kmax-1
!$acc do parallel(16)
                do j=2,jmax-1
!$acc do vector(256)
```

```
                    do i=2,imax-1
                        p(I,j,k)= wrk2(i,j,k)
                    end do
                end do
            end do
        enddo
        !$acc end region
```

開発者Ｃによる CUDA 実装版の性能に及ばない理由の一つは、並列化方法を選択するときの自由度が少ないことである。PGI Accelerator(version 10.0)では、Loop Mapping Directive により、ループ単位で並列化方法（block 並列 or thread 並列）と並列数を指示・設定することができるが、多重ループのときに（姫野 BMT は３重ループ）、複数ループを融合した並列化を指示・選択することができない。例えば、上記のコード例では、同一 block 内のスレッドは１次元上（線上）に配置されるが、姫野 BMT を GPU 化するときには、同一 block 内のスレッドを 2 次元的に、平面上に配置すると、メモリアクセスパターンがより規則的になり、高い性能が得られることが分かっている。



姫野 BMT におけるブロック形状
(左：PGI アクセラレータで選択した形状、右：高い性能が得られる形状)

このように並列化の方法（ブロック形状）を柔軟に指示・選択することができれば、ディレクティブ方式でも更に高い性能が得られると期待される。ただ、記述の柔軟性を高めると、得てして指示方法が複雑になってしまう。例えば、GUI を用いてブロック形状を直感的に指示できるようにするなど、並列化方法を柔軟かつ簡単に指示できる方法が求められる。

なお、PGI Accelerator はオプションで CUDA コードを出力することが可能であり、その CUDA コードをベースとして人手でチューニングを進めると、トータルとしての開発効率を高められるのではないかとの期待もあった。PGI Accelerator が出力する CUDA コードは行数が多く、可読性に問題があった。現状では、人手でチューニングをするのであれば、最初から人手で CUDA コードを既

述した方が良い。

2.2.2 事例 2：2D-FDTD

ここでは、PGI Accelerator を用いて 2D-FDTD プログラムを GPU 化した結果を報告する。

プログラムの GPU 化	開発者 B
GPU 種類	Tesla M2050
PGI Accelerator	Version 11.8

GPU のハードウェア特性に詳しくない開発者 B が 2D-FDTD プログラムを PGI Accelerator(version 11.8)で GPU 化したところ、Tesla M2050 で 18.5 GFLOPS の性能が得られた。CPU 1 コアの性能（Xeon X5650 2.66GHz 1 コア）は 7 GFLOPS 程度であり、CPU 1 コア性能を上回ることはできたが、CUDA 実装版の性能、45.5 GFLOPS には及ばず、CUDA 実装版の 40%程度の性能にとどまった。性能が CUDA 実装版より低い理由の一つは、姫野 BMT 同様、並列化手法を柔軟に指示できないためと考えられる。

	GFLOPS
PGI Accelerator 版	18.5
CUDA 版	45.5
(参考：Xeon X5650 2.66GHz 1 コア)	7.0

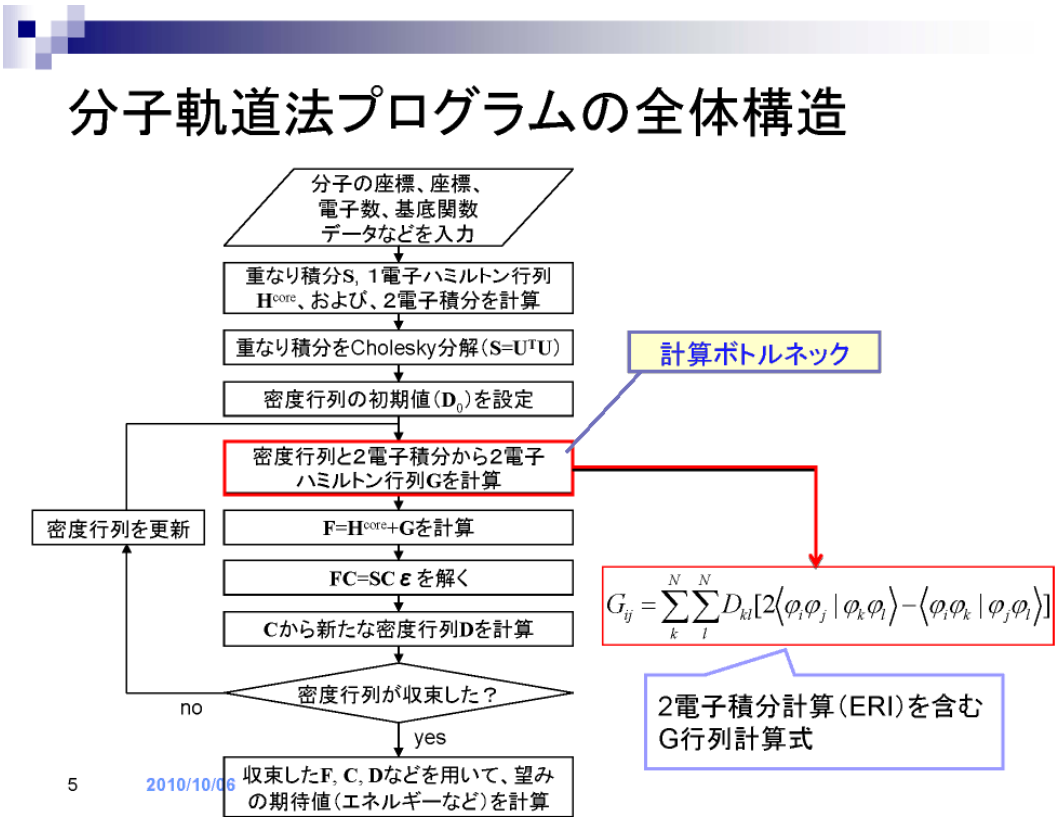
[ディレクティブ挿入後の 2D-FDTD ソースコード]

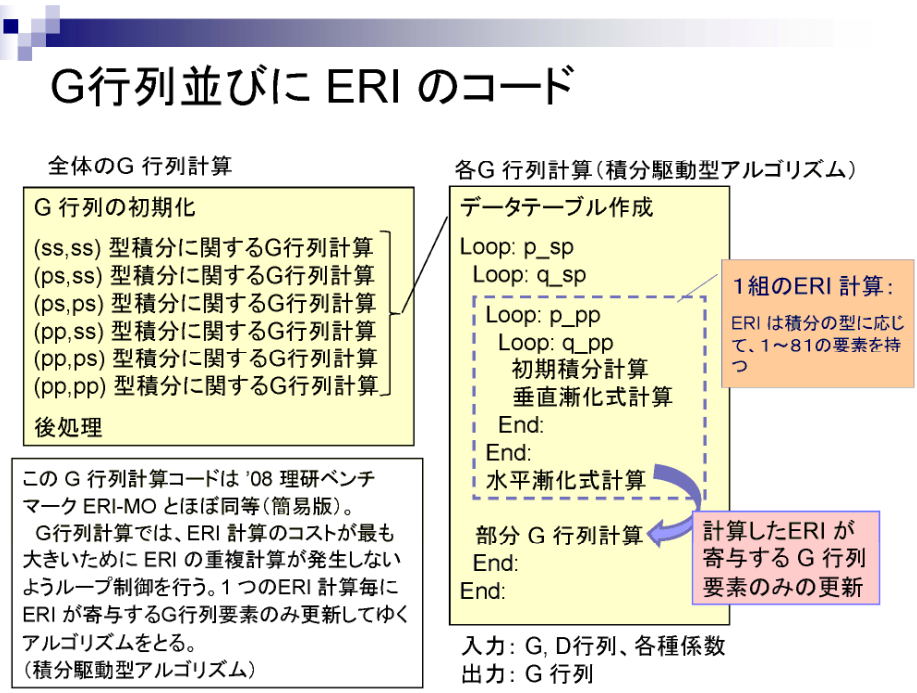
```
!$acc region                    &
!$acc copy (Ez ( 0:NMESH-1 ))   &
!$acc copy (Hx ( 0:NMESH-1 ))   &
!$acc copy (Hy ( 0:NMESH-1 ))   &
!$acc copyin (cEx, cEy, CHx, cHy)
!$acc do host
do it = 0, N -1
        do i = 1, NMESHX-1-1
!$acc do vector (256)
                do j = 0, NMESHY-1-1
                        Hx ( j, i ) = Hx ( j, i )–-cHy * ( Ez ( j + 1, i )–-Ez ( j, i ) )
```

```
                enddo
        enddo
        do i = 0, NMESHX-1-1
!$acc do vector (256)
                do j = 1, NMESHY-1-1
                        Hy ( j, i ) = Hy ( j, i ) + CHx * ( Ez ( j, i + 1)—-Ez ( j, i ) )
                enddo
        enddo
        do i = 1, NMESHX-1-1
!$acc do vector (256)
                do j = 1, NMESHY-1-1
                        Ez ( j, i ) = Ez ( j, i ) + cEx * ( Hy ( j, i )—-Hy ( j, i -1 ) ) &
                                -cEy * ( Hx ( j, i )—-Hx ( j -1, i ) )
                enddo
        enddo
enddo
!$acc end region
```

2.2.3 事例 3：分子軌道法（ERI-MO）

ここでは、PGI Accelerator による分子軌道法プログラムの GPU 化の結果を報告する。





分子軌道法プログラムは、2 電子積分計算を含む、G 行列計算と呼ばれる部分が実行時間の 99% 以上を占めており、GPU で処理時間を短縮するには、G 行列計算を GPU 化する必要がある。G 行列計算は、各種積分型に対応した 6 種類の G 行列計算から構成されている。各積分型 G 行列計算は、それぞれ 4 重ループ構成の積分駆動型アルゴリズムである。G 行列計算は ERI 計算のコストが

大きいため、ある一組の ERI を計算したら (ERI の要素数は、積分型により 1～81 と異なる)、その ERI が寄与する G 行列要素のみ更新することで、ERI 計算回数を削減している。その結果、G 行列要素の更新箇所が不連続となり、メモリアクセスパターンが不規則となる。更に、最内ループに計算値に依存する条件分岐がある。

事例１の姫野 BMT、事例２の 2D-FDTD は、いわゆるステンシルアクセス系のプログラムで GPU 実行に向いたプログラムであったが、本事例の分子軌道法プログラムは、GPU 向きとは言い難いプログラムである。

プログラムの GPU 化	開発者 B
GPU 種類	Tesla C1060
PGI Accelerator	Version 10.6

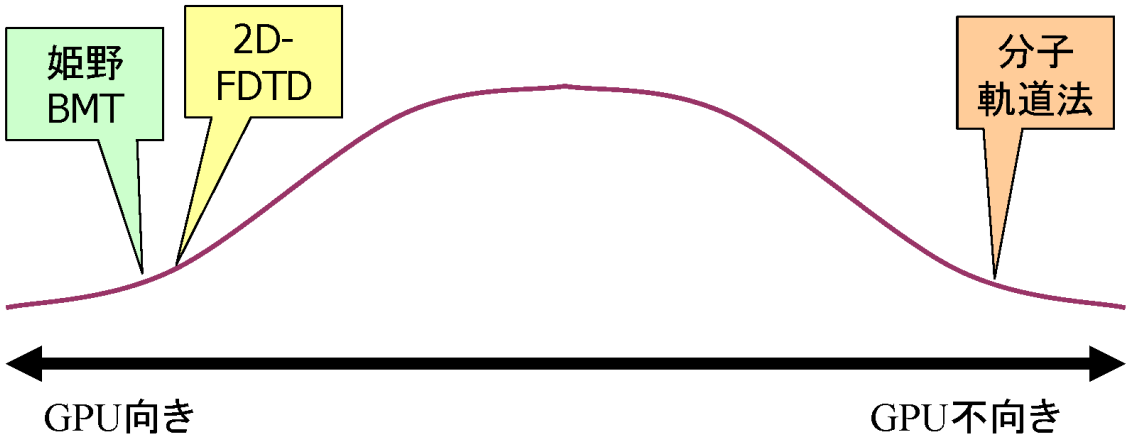
開発者 B が PGI Accelerator(version 10.6)で分子軌道法プログラムの GPU 化を試みたが、簡潔に記述できるアプローチは、図の内側２重ループの GPU 化であった。しかし、その方法で GPU 化した場合でも、ループボディ部におけるポインタ演算等により、アクセラレータ実行用オブジェクトコードが生成されない事が分かった。本プログラムを GPU 上で高速に実行するには、ソースコードを大幅に変更する必要があり、ディレクティブ挿入による GPU 化（高速化）が非常に困難であることが判明した。

本事例は、現状のディレクティブ方式によるプログラム GPU 化の限界・問題を示している。ディレクティブ方式のメリットは、ソースコードの可読性を損ねることなく、簡単にプログラムを GPU 化できることであるが、分子軌道法プログラムのような複雑なプログラムでは、現状のディレクティブ方式では、プログラムを GPU 向けに高速化することができない。このような複雑なプログラムでも、ソースコードを修正することなく、ディレクティブ挿入で並列化を指示する方法を確立しないと、ディレクティブ方式の適用可能領域は、姫野 BMT や FDTD などのメモリアクセスパターンが規則的で、条件分岐が少ないプログラムに限定されることになる。

なお、分子軌道法プログラムのソースコードを大幅に変更して CUDA 化した事例に関しては、別章にて報告する。

2.3 まとめ

本章では、ディレクティブ方式による GPU 利用に関して説明した。言語・開発環境にはワーキンググループ開催期間に使用可能であった PGI Accelerator を用い、3 種類の事例、姫野 BMT、2D-FDTD、分子軌道法を GPU 化することで、ディレクティブ挿入方式の長所・短所を明らかにした。



GPU 向きである姫野 BMT と 2D-FDTD に関しては、どちらもプログラムを短時間で GPU 化することができ、また、CPU 実行と比べて高い性能が達成可能であることを確認した。GPU 向きアプリに関しては、ディレクティブ方式は有用であると言える。一方、GPU 専用言語である CUDA で記述した場合の性能と比較すると、どちらの事例でも性能は 4 割程度に止まった。GPU に向いたアプリケーションであっても、少なくとも現時点ではディレクティブ方式で GPU 性能を最大限に生かすことは難しいことが明らかになった。

GPU に不向きである分子軌道法に関しては、現行のディレクティブ方式では、分子軌道法のような複雑な問題には容易には対処できないことが明らかになった。

今回の評価では、非常に GPU 向いている事例と、非常に GPU に不向きな事例、ある意味で両極端な事例を扱ったが、実際には、明確には GPU 向き・不向きと判別できないアプリが多いと思われる。今後は、そのような中間的な事例で、ディレクティブ方式の有効性を確認する必要がある。

3 アクセラレータ向け拡張言語による GPU 利用

ここでは、アクセラレータ向け拡張言語でプログラムを記述して GPU を利用する形態の評価結果を報告する。

アクセラレータ向け拡張言語とは、例えば CUDA や OpenCL のことであり、C/Fortran 等の既存言語を拡張することで、アクセラレータ向け処理をオフロードするプログラムを比較的容易に記述できるようになっている。ディレクティブ方式による GPU 利用の OpenMP との類似性と対比すると、CUDA や OpenCL は分散並列環境向けの並列プログラミング言語・環境である MPI との類似性の高いプログラミングモデルとも言える。

この開発環境では、既存の C や Fortran で記述されたソースコードから、開発者がアクセラレータで実行させる部分を選択し、その部分のコード(一般に kernel と呼ばれる)をアクセラレータ向けに記述する。また、ホスト実行部分には、この kernel を呼び出すコードを記述する。コンパイラは、ソースコードに記述された内容に従って、ホスト用オブジェクトとアクセラレータ用オブジェクトを生成する。CUDA や OpenCL で開発されたプログラムは、MPI と同じように、適切に記述されれば、アクセラレータ性能をフルに引き出すことができるメリットがある。その反面、逐次向けに記述されたソースコードとは記述内容・方式が大きく変わるため、汎用 CPU 向けコードとアクセラレータ向けコードの共通化が難しく、ソースのメンテナンスやポータビリティ面でのデメリットがある。

3.1 CUDA 概要

CUDA は、nVidia が自社 GPU 向けに提供しているプログラミング言語・開発環境である。CUDA は C/C++をベースとして GPU 向けに言語仕様を拡張しているが、仕様拡張は最小限に抑えられており、C/C++でのプログラム経験のある開発者であれば、それほど違和感無く CUDA を利用することができる。現在は Fortran 版の CUDA の利用可能であり、Fortran 言語をベースとした GPU 利用も進んでいる。CUDA 以前は、グラフィックス処理用の言語である OpenGL を用いることが一般的であったが、OpenGL によるプログラム開発は制約が多く、CUDA の出現により GPU を用いた汎用計算の垣根が大きく下がったことは疑いようのない事実である。

GPU は、演算ユニット単体の性能は低いが、それを大量に搭載することで高い性能を狙うアーキテクチャであり、GPU の能力を最大限引き出すにはプログラムを並列化する必要がある。しかし、CUDA 環境では、プログラムの自動並列化機能は提供されない。従って、既存のプログラムを、CUDA を使って GPU 化する時には、開発者が明示的にプログラムを並列化する必要がある。また、CUDA の並列化モデルは、block 並列と thread 並列から構成される 2 階層の並列モデルとなっている。この 2 階層の並列モデルは nVidia GPU の内部構造(Stream Multi-Processor と CUDA core)を反映したものであり、nVidia GPU 向けにプログラムを開発するモデルとしては適切なものである。

しかし、そもそもプログラムの並列化は困難な作業であるのに、1 階層でなく、2 階層で並列化を考慮する必要があることは、特に、並列処理に不慣れなアプリ開発者が、CUDA でプログラムを開発する上での障壁となっている。また、nVidia GPU の内部構造をある程度理解しないと、性能面で適切なプログラムを開発するのが難しいという問題がある。そして、頻繁に CUDA の version が上がるため、その影響で従来の CUDA コードが動かない、挙動が変わる等の問題がある。更に、CUDA は nVidia proprietary な言語・開発環境でありポータビリティがないため、プログラムを CUDA 化してしまうと他プラットフォームへのポーティングが難しくなるという問題もあり、ISV アプリの GPU 対応が進まない要因の一つとなっている。

CUDA の問題

- 開発者がプログラムを明示的に並列化（手動並列）
- 2 階層の並列化の考慮が必要（block 並列と thread 並列）
- マルチ GPU・GPU クラスタ環境では、3 階層の並列化の考慮が必要
- 頻繁な version up と version 間の非互換性
- nVidia proprietary な言語・開発環境（nVidia がなくなったらどうするのか）

3.2 CUDA によるプログラム最適化事例

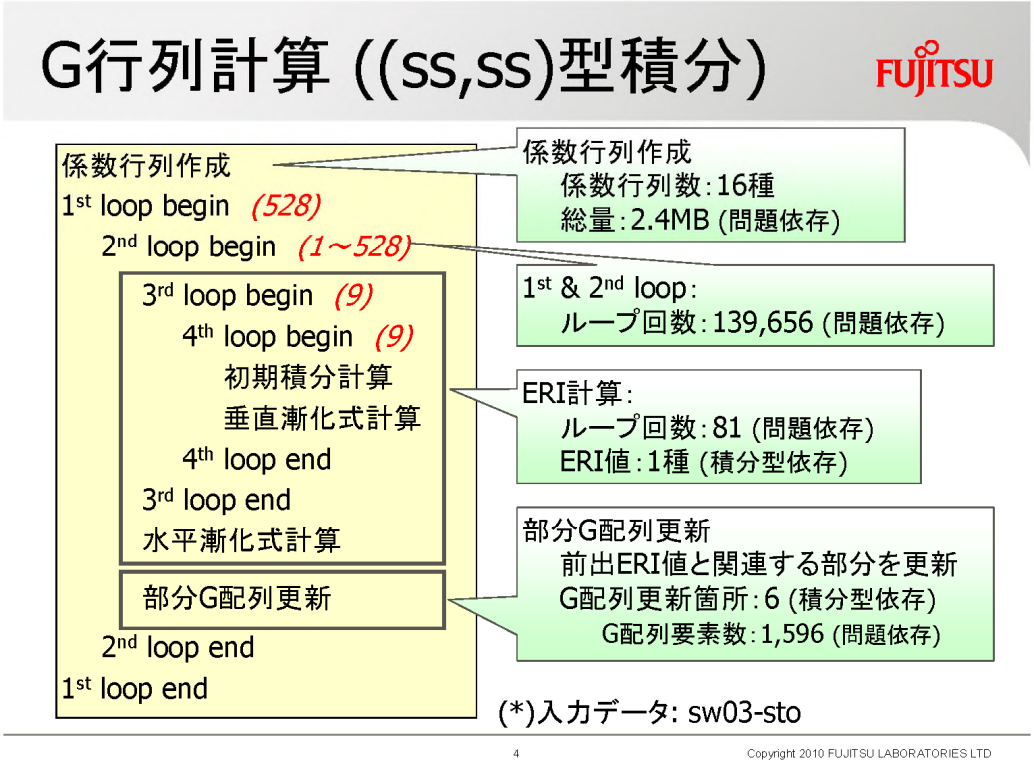
ここでは CUDA を用いてプログラムを GPU 化した事例を報告する。事例としたプログラムは、分子軌道法、Linpack の 2 種類である。なお、Linpack は Tsubame 2.0 での Linpack 性能評価事例である。

3.2.1 事例 1:分子軌道法（ERI-MO）

ここでは CUDA プログラム開発経験のある開発者 C が分子軌道法を CUDA 化した事例を説明する。分子軌道法プログラムを PGI アクセラレータで GPU 化した事例では、アクセラレータ実行用オブジェクトを生成できなかったが、GPU の高メモリバンド幅特性を生かす並列化手法を考案・採用し、G 行列計算の全てを CUDA 化することで、汎用 CPU(1 コア)に対して最大 29.2 倍の高速化を達成している。

プログラムの GPU 化	開発者 C
GPU 種類	Tesla C2050
CUDA version	4.0

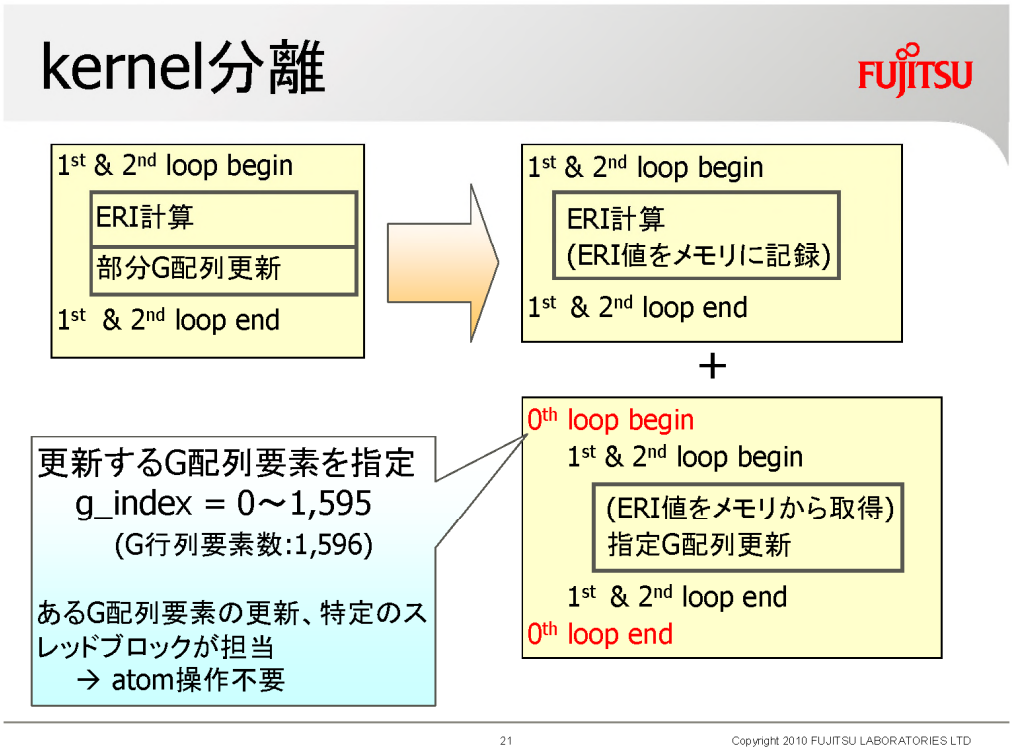
下記の図は、(ss,ss)型積分の G 行列計算の概要である。



naïve な GPU 化は、1st & 2nd ループの各イタレーションを 1 つのスレッドに割り当てる方法である。この方法では、各スレッドは、一組の ERI 計算を行い、その ERI が寄与する G 行列を更新(部分 G 行列更新)する。この方法の問題は、部分 G 行列更新の部分である。具体的には、G 行列の更新箇所が不連続でメモリアクセスパターンが不規則になること、そして、他スレッドと同時に同一箇所を更新する可能性があり、その場合でも正しい値が得られるようにするための排他制御(atomic 操作)が必要になることである。

この naïve な手法で(ss,ss)型積分 G 行列計算を GPU 化したところ、GPU 化した部分だけの性能比較で、CPU(1 コア)に対して 10 倍弱の性能向上しか得られなかった(入力データは sw03-sto)。(ss,ss)型は、6 種類の G 行列計算の中で、最もシンプルで GPU 化に向いた積分型である。他積分型ではこれより性能向上率が大幅に下がると思われるため、naïve な手法で 6 種類の積分型全てを GPU 化すると、CPU(1 コア)に対して数倍程度の性能向上しか得られないことになる。GPU で高い性能を達成するには、部分 G 行列更新の処理をなんらかの手法で高速化する必要がある。

そこで、GPU 向け並列化手法として考案したのが、以下に示す kernel 分離方式である。



通常は ERI 計算を行うと、その ERI が寄与する G 行列要素をすぐに更新するが、kernel 分離方式では ERI 計算と部分 G 行列更新を別 kernel に分離する。1 番目の kernel では、各スレッドが一組の ERI 計算を行い、その結果をメモリに格納する。2 番目の kernel では、メモリに格納された ERI を読み出して、その ERI が寄与する G 行列部分を更新する。ただ、各スレッドが一組の ERI を担当したのでは、G 行列へのメモリアクセスは不規則なままで、排他制御も必要となり性能は向上しない。そこで、2 番目の kernel では、各スレッドに対して更新する G 行列要素を割り当てることで、G 行列のメモリアクセスを規則化し、排他制御を不要にした。この kernel 分離方式では、ERI 計算結果のメモリへの読み書きが必要になるデメリットがあるが、これは GPU の高メモリバンド幅特性で相殺できるだろうという考えである。

この kernel 分離方式で(ss,ss)型積分 G 行列計算を GPU 化したところ、GPU 化した部分だけの性能比較で、naïve な方式と比較して 4 倍の性能、CPU(1 コア)と比較して 40 倍の性能向上が得られることが確認できた。

	(ss,ss)型積分 G 行列計算 1 回の実行時間 (ミリ秒)	性能向上
(CPU 1 コア)	444	(基準)
GPU naïve 方式	45	x9.9
GPU kernel 分離方式	11	x40.2

(*) 入力データは sw03-sto を使用(下記参照)

6 種類の積分型 G 行列計算を全て kernel 分離方式で GPU 化し、プログラム全体の実行時間を、様々な入力データで測定した結果を以下に示す。CPU(1 コア)と比較して、プログラム全体の実行性能が、最大で 29.2 倍向上することが確認できた(入力データとして sw04-sto を使用した場合)。

入力データ			CPU(1 コア)	GPU	
名前	原子数	基底関数	実行時間(秒)	実行時間(秒)	対 CPU 性能向上
sw01-sto	3	STO-3G	0.0310	0.165	x0.19
sw02-sto	9		0.451	0.248	x1.82
sw03-sto	24		26.3	1.19	x22.1
sw04-sto	42		254	8.69	x29.2
sw01-631	3	6-31G	0.0590	0.272	x0.22
sw02-631	9		2.72	0.638	x4.26
sw03-631	24		155	13.8	x11.2

入力データのサイズ(原子数)が小さいケースでは、CPU でも実行時間が高々数秒程度と短いため、GPU による性能向上率は僅かであった(特に原子数が少ない sw01-*では、GPU の方が遅かった)。これは、CPU・GPU 間のデータ転送時間や、GPU 起動時間がボトルネックになったためと思われ、GPU を活用するには、ある程度以上のサイズの問題を解かせる必要があると言える。基底関数に関しては、STO-3G と比べると、6-31G の場合の GPU による性能向上率はやや低めとなった。これは、STO-3G は最内ループ数がほぼ一定であるのに対して、6-31G は最内ループ数が 1～81 の間で可変であり、スレッド毎の処理量に大きなアンバランスが生じるためである。このような負荷の不均一性は、GPU に限らず、多並列で性能をかせぐアクセラレータ全般の課題である。

3.2.2 事例 2:Linpack

ここでは、東工大 TSUBAME2.0 での Linpack 最適化事例を紹介する。

TSUBAME2.0 は Tesla M2050 を 4200 枚以上備えたアクセラレータ・スパコンで、1.192pFlops の Linpack 性能を達成している。多数のアクセラレータを効率良く動作させるために Linpack 実装を大幅に改良している。アクセラレータ・スパコンでは、演算性能の向上に対して、相対的にネットワーク性能が悪くなるため、その対策として、通信コストを隠蔽する技術がきめ細かく適用されている。今後もアクセラレータ化とメニーコア化の進展で、演算性能と通信性能の乖離が進むと考えられ、本事例は、この問題に対処した先行事例の一つである。

4000GPU 超を備えるスーパーコンピュータ TSUBAME 2.0 の Linpack による性能評価

遠藤敏夫（東京工業大学学術国際情報センター

1. はじめに

GPU を中心とする数十～数百コアのメニーコアに基づくアクセラレータの，科学技術計算への応用の可能性は早くから期待されていた．しかしアクセラレータを数百以上結合してトータルで高性能が得られるか否かは，東京工業大学が TSUBAME1.0 スパコンの導入を行うまで明らかではなかった．筆者らは 2006 年に，TSUBAME1.0 の 648 計算ノードと 360 枚の ClearSpeed アクセラレータを効率的に用いるような，密行列計算ベンチマークである Linpack の新たなアルゴリズム・最適化手法を考案した[1]．それにより CPU を用いる場合の 25%増に相当する 47tFlops を達成し，アクセラレータのスパコン規模でのスケーラビリティを示した．

2010 年にはより最新の，プログラマビリティにおいても演算速度・メモリバンド幅においても優れたアクセラレータである NVIDIA Tesla M2050 GPU アクセラレータを 4200 枚以上備えた，TSUBAME2.0 スパコンが導入された．我々は上記の Linpack 実装を大幅に改良し，TSUBAME2.0 で効率的な実行を実現した．得られた性能は 1.192pFlops であり，国内で初めて 1pFlops を超え，2010 年 11 月の Top500 ランキングでは世界 4 位を達成した．さらにアクセラレータを効率的に利用することにより電力あたりの性能という指標についても大幅な利点があることを実証した．Linpack 実行時の電力性能比は 958mFlops/Watt であり，2010 年 11 月の Green500 ランキングにおいて世界 2 位を獲得した[2].

TSUBAME1.0/2.0 に渡って高効率を達成するために対処が必要であった課題は，演算性能と比較したときの相対的な通信バンド幅の悪化である．無論，インターコネクト・ノード内バス・メモリ技術の向上はあるものの，それ以上にアクセラレータの演算性能の伸びが大きく，その性能を活用するには通信コストの隠ぺいが必須であった．この問題は将来のエクサスケール・スパコンにおいてもさらに悪化する本質的な課題である．

2. TSUBAME2.0 スーパーコンピュータ

TSUBAME 2.0 は，1400 ノード以上の計算ノード(合計理論演算性能 2.4pFlops)と，合計 7.1pBytes のストレージが QDR InifiniBand により接続されている，ペタスケールのスーパーコンピュータである．以下では本稿に関連の深い事項のみ記す．

各計算ノード Hewlett-Packard Proliant SL390s G7 (図 1)は 6 コアの Intel Xeon X5670 2.93GHz プロセッサを二つ，DDR3 メモリ 54GB，および NVIDIA Tesla M2050 GPU を三つ搭載する．各 GPU はストリーミングマルチプロセッサ(SM) を 14 基持ち，各 SM は SIMD 動作する CUDA core を 32 基持つ．各 GPU の理論演算性能は 515gFlops である．GPU はホストメモリとは独立したメモリを 3GB 持ち，GPU 演算に必要なデータはそちらに置く必要がある．ホスト側からの通信経路として，三つの GPU はそれぞれ PCI-Express 2.0 x16 にて接続されている．

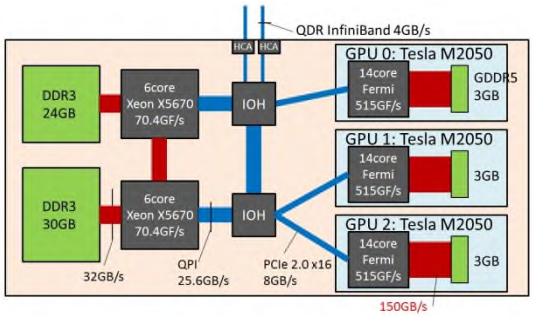


図 1: TSUBAME2.0 計算ノードの内部構成

全ノードを接続するインターコネクトは 2 段のスイッチから成るファットツリーであり，フルバイセクション構成である．Dual rail 構成であり，各 rail がファットツリーを構成する．

3. Linpack ベンチマーク

Linpack は密行列連立一次方程式をガウス消去法を用いて解くベンチマークであり，よく知られた MPI 並列実装として High-performance Linpack (HPL) [3]がある．我々の実装は HPL を大幅に改造したものである．

HPL においては，一辺の長さ N の巨大な密正方行列(TSUBAME2.0 においては N=240 万程度のものを用いた)を，二次元ブロックサイクリック分割にて，各 MPI プロセスに分散配置する．分散の単位であるブロックのサイズを B とする．計算全体での演算量は $O(N^3)$，通信量は $O(N^2)$であり，演算のほとんどは密行列積(DGEMM)カーネルとなっている．詳細は[2,3]を参照されたい．

このようなベンチマークの性質から，古典的な(CPU のみの)クラスタシステムで高性能を得るには，以下のような方策をとる．

- 高性能な DGEMM カーネルを持つ BLAS ライブラリを用いる．
- 通信量を相対的に軽減するため，行列がシステム全体のメモリに収まる範囲で，N をなるべく大きくする．

4. GPU を用いる際の課題と解決

HPL から GPU を利用するのに最も単純な手法は，DGEMM カーネルを GPU 上のものに置き換えることである．

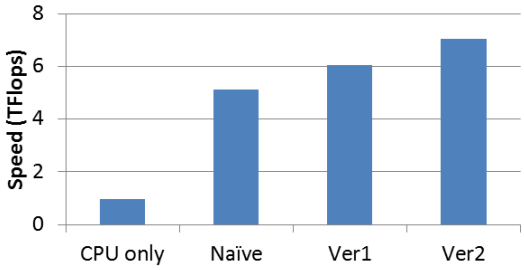


図 2: TSUBAME2.0 の 8 ノードを用いた Linpack 性能.

図 2 の”Naïve”は，DGEMM カーネルを単純に GPU 版に置きかえた場合の性能を示す．実験条件は以下のとおりである：8 ノード 24GPU が用いられている．今回は NVIDIA 社内部の，約 350gFlops の性能を持つカーネルの提供を受けたためそれを用いた．これは CUBLAS ライ

ブラリの DGEMM(バージョン 3.2 で約 290gFlops)より高速である．前提として，N をなるべく大きくするため(本実験では N=188,416)に行列はメインメモリにおき，DGEMM のたびに PCI-Express を介し GPU と通信するものとする．

Naïve は GPU の利用により，オリジナル(CPU only)に比べ 5.3 倍の速度向上を得た．しかしながら 24GPU 分の DGEMM 性能である 8.4tFlops に比べると 6 割程度に留まっている．

問題解析のため，HPL の 1 プロセスの 1 ステップのアルゴリズムの概要を図 3 に示す．各プロセスが持つ部分行列を A'とすると，1 ステップで行うべき仕事は，U および L と呼ばれる行列の一部を MPI 通信によって獲得し，A'=A'-L*U という DGEMM 演算をすることである．なお最適化のために，L の通信については前のステップのうちに行う実装が，オリジナル HPL ですでになされている．

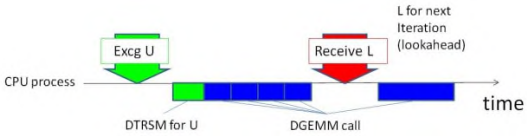


図 3: オリジナル HPL の 1 ステップのアルゴリズム

これを単純に GPU 化すると，以下の理由で問題があると分かった．

(1) U, L を MPI 通信で取得する間は計算が進まない．この点は通常のクラスタではあまり問題とされて来なかったが，GPU を用いるスパコンにおいては，演算性能に対し相対的な通信性能が低いため，問題が顕在化する．

(2) 次ステップのための L 通信を非同期に行うため，部分的な DGEMM とメッセージ到着チェックを繰り返す. GPU 版においては, 分割された DGEMM カーネルのたびに無駄な PCI 通信が繰り返される．

これらを解決するために，以下のような手法を提案・実装した．

(2)の問題の解決を以下のように行った．GPU メモリサイズがホストメモリより小さいため，行列 A'を分割して，部分的に DGEMM を行うことは必須である．ただし，Naïve と異なり，L については GPU メモリに置きっぱなしにし，毎回 L を PCI 通信することを避ける．また部分 A'および部分 U の PCI 通信と DGEMM 演算をパイプライン化させることにより，PCI 通信のコストを隠ぺいする．さらには, 次ステップのための L 通信は, GPU が計算している間に CPU により行われるため，このコストも隠ぺいされる(問題(1)も部分的に解決)．ここまでを行ったものが，図 2 の Ver1 となる．

さらに U についても問題(1)を解決するため Ver2 を実装した(図 4)．ここでは U の MPI 通信の隠ぺいのために，MPI 通信，PCI 通信，DGEMM 演算を全てパイプライン化する方式を実装した．この Ver2 の 1 ステップのアルゴリズムは図 4 のようになる．消費される時間のほとんどで GPU 上の計算を行えており，時間を有効に使う実装となっている(L の PCI 通信が隠ぺいされず残っているが，その隠ぺいも原理的には容易である)．

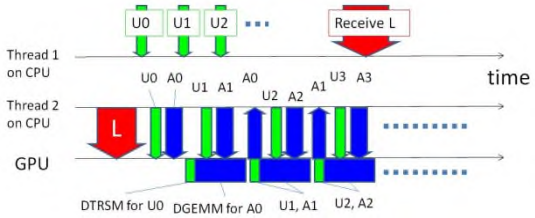


図 4: GPU 版 HPL(Ver2)の

1 ステップのアルゴリズム

結果として，図 2 のグラフに示された通り 8 ノードで 7.0tFlops の性能となり，事実上の限界(8.4tFlops)の 83%を達成した．なお，8 ノードの理論演算性能 13.5tFlops に比べると 52%程度であるが，この乖離の最大の理由は，DGEMM カーネル性能である 350gFlops がすでに理論値 515gFlops から乖離していることであることに留意されたい．

5.　TSUBAME2.0 全体を用いたペタスケールの性能評価

TSUBAME2.0 正式稼働開始の直前の 2010 年 10 月に, システムの安定性試験の一環として，TSUBAME2.0 全体で，前節の Linpack GPU 版 Ver2 を用いた評価を行った．実験に用いたシステムソフトウェアは，SUSE Linux Enterprise 11, OpenMPI 1.4.2, GCC 4.3，CUDA 3.1 である．最終的に用いたノード数は 1357 であり，GPU 数は 4071 である．

Linpack 実行時のパラメータは以下の通りとした．行列サイズ N=2,490,368 とした．この行列を 1357 ノードに分散すると，ノードあたり約 34GB となる．MPI プロセス数については，1 プロセスが 1GPU を利用することとし，ノードあたり 3 プロセス，全体で 4071 プロセスとした．行列はプロセス間で二次元分割されるが，4071=59*69 となるようにした．ブロックサイズ B=1024 とした．

この実行には約 2.4 時間かかり，結果として得られた速度性能は 1.192pFlops となった．これは国内で初めて 1pFlops を超えた計算となった．なお 1357 ノードの理論演算性能 2.288pFlops に対して 52.1%の相対性能となった．理論値との乖離については前節で議論した通りだが，8 ノードでも 1000 ノード超でも似た相対性能が得られており，スケーラビリティは非常に良好であることが分かる．

同時に実行時のシステム電力も測定しており, 消費電力は 1.24mWatt (Green500 の規定に基づく電力)，電力性能比は 958mFlops/Watt という，これも世界最高レベルの値が得られた．

6.　おわりに

国内で初めて Linpack 1pFlops 超の達成を, GPU アクセラレータを効率的に利用するソフトウェア技術により行った．2008 年に世界で初めて 1pFlops を超えた LANL RoadRunner スパコンも，PowerXCell 8i アクセラレータを用いていた．将来のスパコンが異種プロセッサ混在のものであるか否かは議論の待たれるところであるが，プロセッサのメニーコア化と，それに伴う演算性能に比べた通信性能の相対的な悪化はまず間違いない．その問題に対処する，アルゴリズム・プログラミングシステムの研究開発が必要である．

[1] T. Endo and S. Matsuoka, “Massive supercomputing coping with heterogeneity of modern accelerators”,

IEEE IPDPS08, 10pages, 2008.
[2] 遠藤 敏夫，額田 彰，松岡 聡． スーパーコンピュータ TSUBAME 2.0 における Linpack 性能 1 ペタフロップス超の達成． 情報処理学会論文誌コンピューティングシステム， Vol. 4, No.4 (ACS 35), pp.169-179, 2011 年 10 月.
[3] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL‒- a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/.

3.3 OpenCL 概要（CUDA との比較）

OpenCL は、OpenGL などの仕様策定を行っている Chronos を中心に、多数のベンダーが協力して仕様が策定されたプログラミング言語である。OpenCL の仕様策定には、GPU ベンダー以外に、汎用 CPU・モバイル・組込みベンダーも参加しており、OpenCL でプログラムを開発すれば、様々なプラットフォームでそのプログラムを実行できる、高いポータビリティが期待されている。

現在の OpenCL 言語仕様は、CUDA、特に CUDA の低レベル API との類似性が非常に高い。CUDA と同様、2 階層の並列化モデルを採用している。このモデルは GPU の内部構造にフィットしたものであり、汎用 CPU や組込み CPU にフィットするかどうかは疑問が残る。また、OpenCL のカーネル・ルーチン記述方法は CUDA との違いがほとんどないため、CUDA 使用経験のある開発者であれば、OpenCL 利用の障壁は低いことはメリットだが、カーネル・ルーチン呼び出し方法が CUDA ほど直感的ではないこと、カーネル・ルーチン呼び出し前の処理が煩雑であるなど、使い勝手の点で問題がある。CUDA は従来と比べてプログラミングを簡単にする方向性であったが、OpenCL はその流れに逆行しているとも言える。また、CUDA は version up で新しい機能が追加されるが、OpenCL ではそれら新機能を使うことができない点もデメリットである。そのため、特に nVidia GPU 向けのプログラム開発に関しては、OpenCL を採用するメリットは薄い。OpenCL は開発者が直接プログラムを記述する言語ではなく、中間言語として使用されるのではないかという意見もある。

一方で、AMD は自社 GPU 向けのプログラミング言語を、自社独自言語から OpenCL に変更したこともあり、AMD GPU の今後の状況次第では、ポータビリティがより重視され、OpenCL が使用される可能性もある。

- **CUDA と OpenCL の違い(主要点のみ)**

用語

	CUDA	OpenCL
最小実行単位	Thread	Work-item

最小実行単位のグループ	Thread-block	Work-group

カーネル・ルーチンの実行方法

CUDA: サブルーチン呼び出しに類似

```
Dim3 block(…);
Dim3 thread(…);
Gpu_kernel<<block,thread>>(…);
```

OpenCL: コマンドキューにカーネル・ルーチンを登録

```
clEnqueueNDRangeKernel( queue, gpu_kernel, … );
```

カーネル実行前に必要な処理

CUDA: 少ない

カーネル引数設定 (block と thread の作成)

OpenCL: 多い

プラットフォーム ID 取得
デバイス ID 取得
コンテキスト作成
コマンドキュー作成
メモリオブジェクト作成
カーネルオブジェクト作成
カーネル引数設定

3.3.1 OpenCL と CUDA の性能比較

ここでは nVidia GPU 上で CUDA と OpenCL の性能を比較した結果を報告する。

性能比較用プログラムには 3D-FDTD を用いた。CUDA で開発した 3D-FDTD プログラムを OpenCL に書き換えて性能を比較した。ベースとした CUDA 版プログラムは、CPU（Core-i7 2.66GHz、1 コア）に対して 20 倍以上高速な実装である（GPU は GTX-285 を使用）。CUDA 版と OpenCL 版の性能差は以下の通りであった。

CUDA 版と OpenCL 版の性能差

CUDA version 2.3	CUDA 版が 1.8 倍高速
CUDA version 3.0	CUDA 版が 1.02 倍高速 (ほぼ同性能)

CUDA version 2.3 環境では、CUDA 版と OpenCL 版で 1.8 倍程度の性能差が観測された。原理上は CUDA と OpenCL で性能差が生じる理由はない。CUDA version 2.3 の時点では OpenCL 対応の初

期段階であり、コンパイラの最適化機能のレベルが CUDA と OpenCL で差異が性能差の理由と思われる。実際、その後にリリースされた CUDA version 3.0 で、CUDA と OpenCL でほぼ同じ性能が得られることが確認できた。技術的には、CUDA と OpenCL で顕著な性能差が生じることはないが、今後どうなるかはベンダーが OpenCL をどこまで真剣にサポートするかに依存することになる。

参考までに、CUDA 版と OpenCL 版、それぞれのカーネル・ルーチンを以下に示す。OpenCL 版の赤字部分が CUDA 版と記述の異なる部分である。カーネル・ルーチンの記述方法に関しては、CUDA と OpenCL に本質的な違いのないことが、これを見ると良く分かる。

[CUDA で記述した 3D-FDTD のカーネル・ルーチン（一部）]

```
__global__ void gpu_calc_Ex( float *Ex, float *Cex, float *Dex,float *Hx,
                                        float *Hy, float *Hz )
{
        int   x = blockIdx.x * blockDim.x + threadIdx.x;
        int   y = blockIdx.y * blockDim.y + threadIdx.y;
        int   z;
        int xs = 0;
        int ys = 1;
        int zs = 1;
        int xe = NUM_X-2;
        int ye = NUM_Y-2;
        int ze = NUM_Z-2;
        int   flag = 0;
        if ( (x >= xs) && (x <= xe) && (y >= ys) && (y <= ye) ) {flag =   ;}
        for ( z = z  ; z <= z  ; z++ ) {
                if ( flag ) {
                        EX(z,y,x) = CEX(z,y,x) * EX(z,y,x)
                        +DEX(z,y,x)  *  (HZ(z,y,x)  –  HZ(z,y-1,x)  –  HY(z,y,x)  +
HY(z-1,y,x)  ;
                }
        }
}
```

[OpenCL で記述した 3D-FDTD のカーネル・ルーチン（一部）]

```
__kernel void gpu_calc_Ex( __global float *Ex, __global float *Cex, __global float *Dex,
                        __global float *Hx, __global float *Hy, __global float *Hz )
{
```

```
        int x = get_global_id(0);
        int y = get_global_id(1);
        int   z;
        int xs = 0;
        int ys =   ;
        int zs =   ;
        int xe = NUM_X-  ;
        int ye = NUM_Y-  ;
        int ze = NUM_Z-2;
        int   flag = 0;
        if ( (x >= xs) && (x <= xe) && (y >= ys) && (y <= ye) ) {flag =   ;}
        for ( z = zs; z <= ze; z++ ) {
                if ( flag ) {
                        EX(z,y,x) = CEX(z,y,x) * EX(z,y,x)
                        +DEX(z,y,x) * (HZ(z,y,x) –HZ(z,y-1,x) –HY(z,y,x) + HY(z-1,y,x)  ;
                }
        }
}
```

3.3.2 OpenCL のポータビリティ

ここでは OpenCL で記述したプログラムのポータビリティの評価結果を報告する。

ポータビリティ調査の対象としたプログラムは 3D-FDTD であり、nVidia GPU 向けに開発した OpenCL 版の 3D-FDTD プログラムが AMD GPU 上で動作するかどうか、また、そのときの性能を測定した。評価環境は以下の通りである。

	nVidia GPU 環境	AMD GPU 環境
GPU	nVidia GTX-285	AMD HD-5870
SDK	CUDA Toolkit 2.3	Stream SDK 2.0
CPU	Core-i7 2.66GHz	Core-i7 2.66GHz
OS	RHEL 5.3	OpenSUSE 11.0

動作する・しないのレベルであれば、nVidia GPU 向けに開発した OpenCL 版 3D-FDTD プログラムは、ソースコード無修正で動作することが確認できた（開発環境が異なるため、Makefile は若干変更）。ただし、ソースコード無修正で動作したのは、ベンダー提供の OpenCL サンプルコードをベースとせず、純粋に OpenCL 仕様に基づいてプログラムを開発したことが大きく寄与している。ベンダーの SDK 等に含まれるサンプルコードには、デバッグを容易にするため、また、プログラ

ム記述性を高めるためのラッパーライブラリが使われていることが多い。当然、ラッパーライブラリはベンダー独自でポータビリティがないため、SDK のサンプルコードをベースにプログラムを開発すると、ポータビリティの面で問題が生じるので注意が必要である。

一方、性能に関しては、全くポータブルでないことが分かった。nVidia GPU 向けに開発したソースコードをそのまま無修正で AMD GPU 上で実行した場合、nVidia GPU に対して AMD GPU では実行時間が 100 倍以上長くなった（CPU 1 コア実行より 10 倍以上低速）。AMD GPU で適切な性能を得るには、AMD GPU 向けにソースコードを変更する必要があり、具体的には以下の修正を実施することで、CPU 1 コア性能に対して 2 倍弱の性能が得られた。

CPU・GPU 種類	ソース種類	実行時間(秒)
Core i7 2.66GHz 1 コア	C	79.14
nVidia GTX-285	OpenCL ベース実装	6.36
AMD HD-5870	OpenCL ベース実装	843.30
	+ ストリーム化	87.92
	＋ ベクトル化	47.18

AMD GPU 向けのソースコード変更内容

1. ストリーム化（work-item の担当領域と work-group の形状変更）
2. ベクトル化（データ型を float から float4 に変更）

ソースコード変更内容の 1. ストリーム化は、プログラムの記述方式にも依存するが、基本的にはパラメータ変更で対処可能であり、ソースコードを大幅に修正する必要はない。一方、2. ベクトル化は、データ型をスカラ型からベクトル型に変更するだけでは駄目で、データ型の変更に合わせてカーネル・ルーチンを適切に書き変える必要がある。また、このベクトル化は VLIW 型の AMD GPU アーキテクチャには適切なチューニング手法だが、スカラ型の nVidia GPU アーキテクチャには不適切なものであり、ベクトル化したカーネル・ルーチンを用いると、nVidia GPU では性能が低下する。 現実的な性能を得るには、GPU 毎にカーネル・ルーチンを実装・用意する必要がある。

OpenCL 仕様に基づいてプログラムを開発すれば、アーキテクチャの異なる GPU 間でも動作レベルのポータビリティを保つことはできる。しかし、性能面のポータビリティは、少なくとも現時点では全く期待できないことが明らかになった。

3.4 まとめ

本章では、アクセラレータ向け拡張言語による GPU 利用に関して説明した。

分子軌道法の事例は、GPU に不向きなプログラムを単体 GPU で利用する事例であり、GPU の長

所・短所を理解した上で、その特性を生かす並列化手法を見つけることができれば、一見するとGPU に不向きなプログラムであっても、GPU で高い性能を達成できることを示した。今回、考案・実装した並列化手法は、既存の逐次実装に対してディレクティブを挿入する GPU 利用方法では、残念ながら実現することができない（少なくとも現時点では)。しかし、全てのアプリ開発者に対して並列処理のエキスパートになることを求めるのは現実的ではない。どのようにアプリ開発者の並列化作業を支援するのかは、依然として課題である。

Linpack の事例は、今後普及すると考えられるアクセラレータ・クラスタ構成での GPU 利用事例であり、アクセラレータ・クラスタでは、演算性能に対して相対的に通信性能が悪化するため、通信処理を隠蔽する手法が重要になることが明確となった。ただ、現状では、このような通信処理隠蔽の実現方法はプログラム依存であり、また、その実装やデバッグには大きな労力がかかる。今後は、通信処理隠蔽のアルゴリズム開発、また、それを簡単に実装・デバッグできるプログラミング環境の開発が今後の課題である。

OpenCL は、異種 GPU 間(nVidia GPU と AMD GPU 間)で、ソースレベルのポータビリティがあることが確認できたが、性能面でのポータビリティは無く、使い物になる性能を達成するには、GPU 毎にカーネル・ルーチンを開発する必要があることが分かった。

4. ハードウェア動向

ここではアクセラレータのハードウェア動向に関して報告する。

アクセラレータ分野は、競争が激しいこともあり、継続的にアーキテクチャ更新が行われている。nVidia GPU は本 WG 期間中の 2010 年初に、アーキテクチャが Tesla から Fermi に変更になり、倍精度演算性能の向上、ECC メモリサポート、汎用キャッシュの導入など、HPC 用途向けの機能が強化された。AMD GPU も 2011 年末に大幅なアーキテクチャ変更が行われ、演算ユニットは VLIW 型から、nVidia GPU に近いスカラ型に変わるなど、より汎用計算に適した構造に変化した。

一方、汎用 CPU をベースにしたアクセラレータ（いわゆる Many-core）には Intel MIC がある。MIC は x86 コアをシンプルにして多数並べる構造になると予測されるが、詳細情報は依然として公開されていない。

Intel MIC 情報(Knights Corner)

倍精度性能	1TFlops 以上
コア数	50 以上
SIMD 演算	512bit 幅
メモリバンド幅	?
プログラミング言語	既存言語で OK (既存アプリがそのまま動作)

また、現在のアクセラレータは、いずれも Host マシンを必要とするタイプであるが、消費電力の観点からは Host マシンレスの stand-alone タイプになることが望ましい。アクセラレータのあるべき構造は stand-alone タイプである。nVidia は stand-alone タイプのアクセラレータ実現を目指したプロジェクトを進めており(Project Denver)、AMD は x86 CPU と GPU の統合を目指した HSA(Heterogeneous Systems Architecture)を推進している。

今後も、アクセラレータのアーキテクチャ進化は続くと思われる。

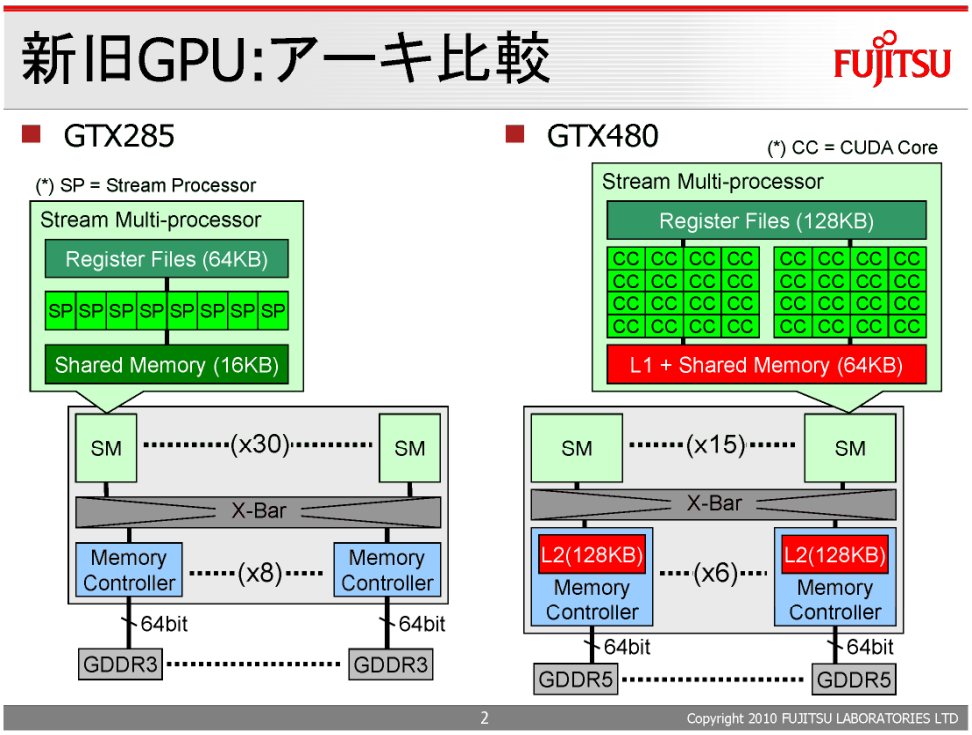
4.1 新旧 nVidia GPU アーキテクチャ比較

ここでは、1 世代前の Tesla アーキ GPU（GTX-285）と現世代の Fermi アーキ GPU（GTX-480）の比較結果を報告する。汎用計算の観点から見た Tesla から Fermi での大きな変化は、以下の 3 点である。

- 倍精度(DP)性能向上： DP/SP 比率が 1/12 から 1/2 に向上
- 信頼性向上：ECC メモリサポート
- 使い易さ向上：キャッシュ構成変更（Read-only キャッシュから Read-Write キャッシュに）

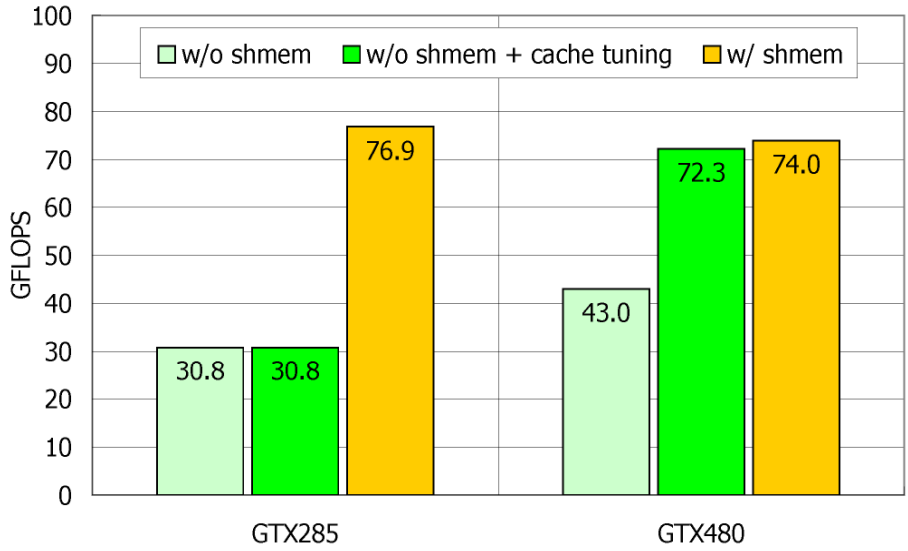
倍精度性能向上と ECC メモリサポートは、従来から HPC 分野から必須機能として要求されたいたものであるが、Read-Write キャッシュのサポートが GPU プログラムの性能チューニングの簡易化に非常に効果的であることを、姫野 BMT を事例とした評価で確認できた。

Tesla アーキ世代ではキャッシュ用途が限定されていたため、共有メモリ（いわゆるスクラッチパッド）をキャッシュの代替リソースとして利用して、グロバール・メモリへのアクセス回数を削減することが、GPU で高い性能を実現するための必須技術であった。共有メモリはプログラム側から明示的に制御するリソースであり、ソースコードを修正して利用する必要があるが、共有メモリを利用するコードは可読性が低下するため、デバッグやソースコードのメンテナンスが悪化するという問題がある。また、メモリアクセスパターンが規則的なプログラムや、どのデータが再利用されるのか予測可能なプログラムでないと、効果的に共有メモリを利用するのは困難である。Fermi 世代の GPU に導入された汎用キャッシュにより、これら問題の一部は解消される。



共有メモリ使用の有無による姫野 BMT 性能の違い

姫野 BMT を事例に、共有メモリ使用の効果を確認した結果を説明する。以下は、共有メモリを使用せずに CUDA で記述した姫野 BMT と、共有メモリを使用した場合の性能を、Tesla 世代、Fermi 世代、双方の GPU で測定した結果である。



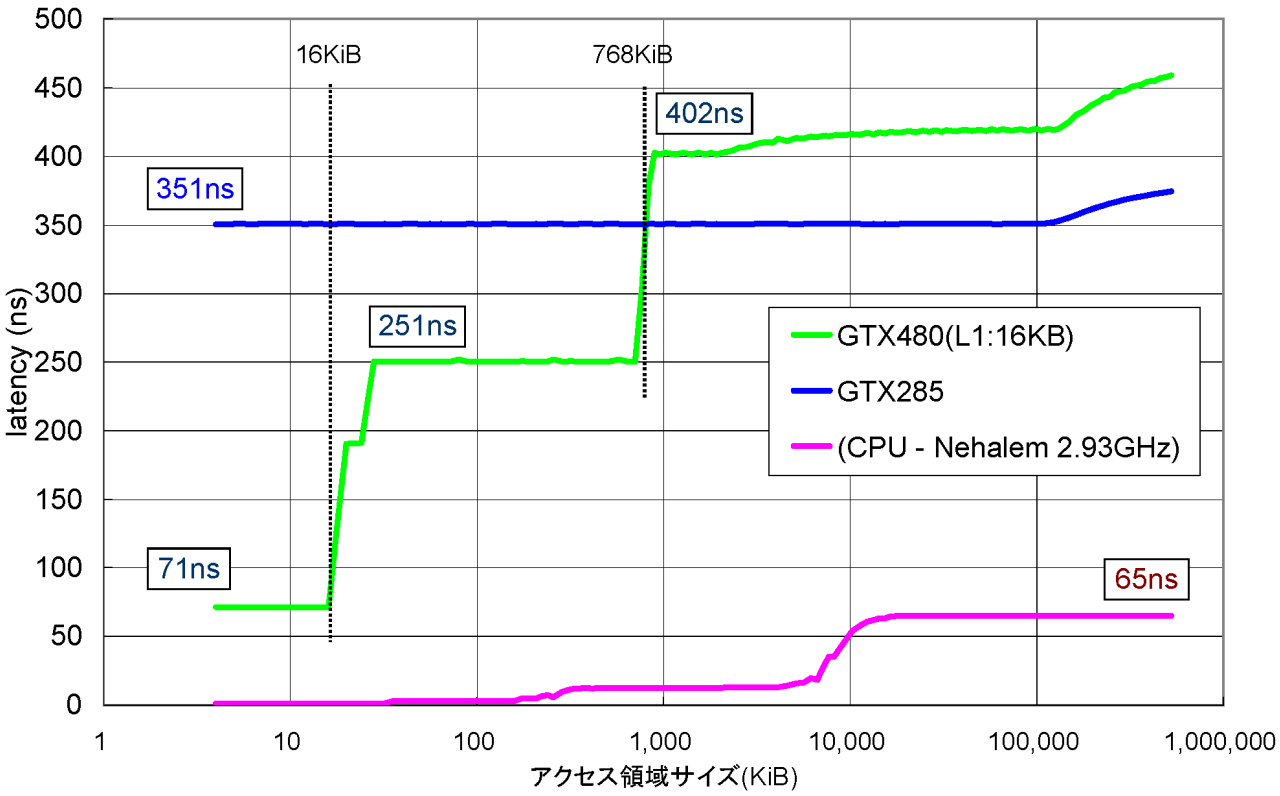
Tesla 世代の GPU である GTX-285 では、共有メモリ未使用時は 30.8 GFLOPS、使用時は 76.9 GFLOPS と、共有メモリ使用の有無で大きな性能差があった。一方、Fermi 世代 GPU である GTX-480 では、共有メモリ未使用時は 72.3 GFLOPS、使用時は 74.0 GFLOPS と、共有メモリ使用の有無は性能に大きく影響しなかった。共有メモリ使用時の方が、若干ではあるが性能は高いので、ピーク性能を追求する場合には、Fermi 世代の GPU でも、共有メモリ使用は依然として有効なチューニング手法であるが、ソースコードの可読性等を考慮すると、共有メモリ使用はあまり推奨できない。実際、姫野 BMT の場合、コアルーチン行数は、共有メモリ未使用時が 70 行程度、共有メモリ使用時が 230 行程度であった。同程度の性能を 1/3～1/4 のコード記述量で実現できていることになり、開発効率の観点でも共有メモリ使用は推奨できない。Tesla 世代の GPU では、共有メモリをキャッシュメモリの代わりとして使用することが半ば常識化していたが、Fermi 世代以降では、共有メモリはその本来の用途、例えば、近接スレッド間のデータ交換や reduction 処理などに使うのが望ましい。

キャッシュチューニング

Fermi 世代 GPU のキャッシュメモリを有効活用するには、キャッシュチューニングとしてソースコードへの__synchthreads()挿入が必要な場合がある。姫野 BMT は、このキャッシュチューニングが有効な事例であり、__synchthreads()を挿入しないと性能は 43.0 GFLOPS に留まったが、__synchthreads()の挿入で 72.3 GFLOPS まで性能が向上した。__syncthreads()は近接スレッド間の同期をとる命令であり、実行オーバーヘッドがあるため一般的には使用しないほうが性能上は望ましいが、姫野 BMT のような全スレッドが規則的に動作するプログラムでは、__syncthreads()により全スレッドの足並みが揃うことで、キャッシュヒット率が向上するという効果がある。本キャッシュチューニングが機能する仕組みに関しては、添付資料 9 を参照願いたい。

メモリアクセス時間

基本的なメモリ性能の一つである、メモリアクセス時間を、Tesla 世代 GPU（GTX-285）と Fermi 世代 GPU（GTX-480）で測定した結果を以下に示す。



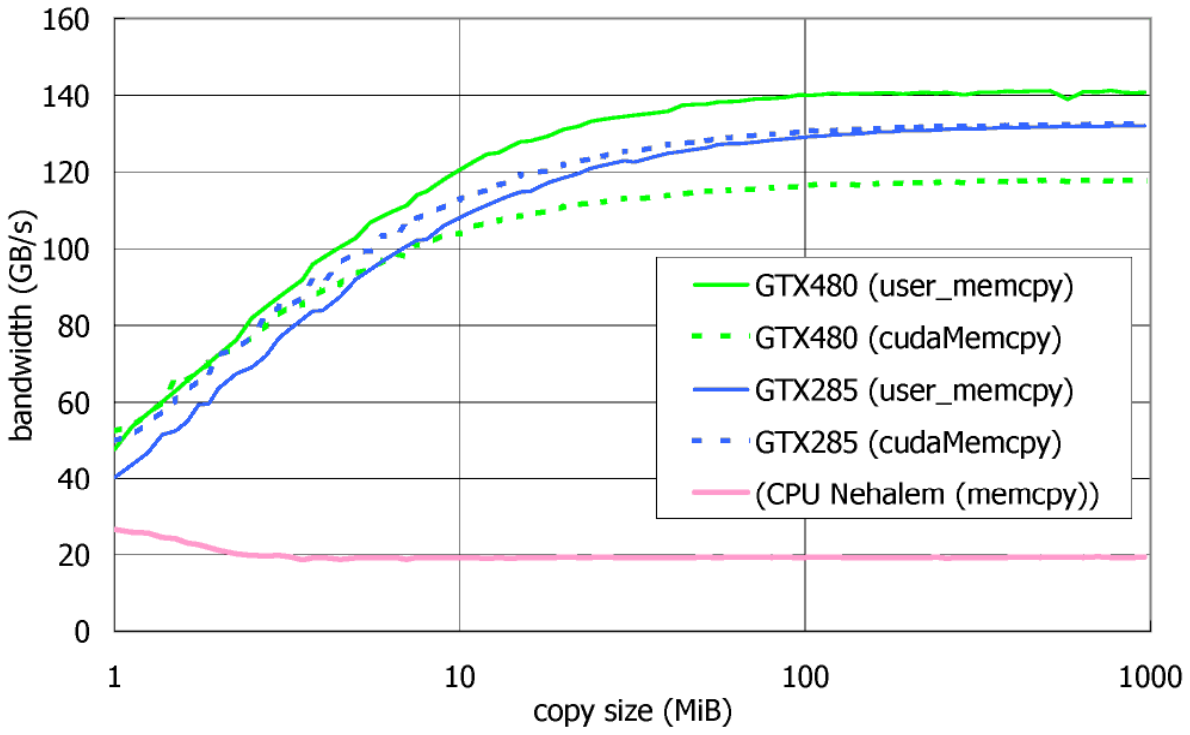
[単位:ns]	L1 / L2	共有キャッシュ	メモリ
GTX-285	---	---	351
GTX-480	71	251	402
Nehalem(2.93GHz)	1.2 / 3	13	65

GTX-285 のメモリアクセス時間は、キャッシュメモリがないため、351ns で一定である。一方、汎用キャッシュを搭載している GTX-480 は、L1 キャッシュ、共有キャッシュ、グロバール・メモリでアクセス時間が異なり、それぞれ、71ns、251ns、402ns であった。L1 キャッシュのアクセス時間 71ns は、汎用 CPU のキャッシュメモリ・アクセス時間と比べて非常に遅く、汎用 CPU のメモリアクセス相当の時間がかかっている。実行スレッドを頻繁に切替えてメモリアクセス時間を隠蔽する GPU なら、これで十分なのかもしれないが、GPU と汎用 CPU では設計コンセプトが違うことがよく分かる結果である。

なお、グローバル・モリへのアクセス時間は、GTX-285 から GTX-480 で 50ns 以上遅くなっているが、原因は不明である。メモリ構造が階層化されたこと（汎用キャッシュが追加されたこと）が影響しているのかもしれない。

メモリバンド幅

基本的なメモリ性能の一つであるメモリバンド幅を測定した結果を、Tesla 世代 GPU（GTX-285）と Fermi 世代 GPU（GTX-480）で測定した結果を以下に示す。メモリバンド幅はメモリコピー時の実測値である。



	GTX-285	GTX-480	Nehalem
メモリタイプ	GDDR3	GDDR5	DDR3
データ転送レート	2.484 GHz	3.696 GHz	1.333 GHz
ビット幅	512 bit	384 bit	192 bit
ピークバンド幅	159.0 GB/s	177.4 GB/s	32.0 GB/s
CUDA version	3.0	3.0	---

GTX-285 では、標準 API の cudaMemcpy を使用した場合と、自作メモリコピールーチンの user_memcpy を使用した場合で、ほぼ同じ実効メモリバンド幅となったが、GTX-480 では、標準 API と自作メモリコピールーチンで実効メモリバンド幅が異なり、標準 API のときは性能が低い結果が得られた。これは、cudaMemcpy が、まだ Fermi 世代 GPU 向けに最適化されていなかったためと思われる。

また、ピークメモリバンド幅は GTX-480 の方が上であるにも関わらず、GTX-285 の方が実効メ

モリバンド幅は良いという結果となった。メモリバンド幅は多くの HPC アプリケーションにとって重要なポイントであり、Fermi 世代の GPU でメモリバンド幅が強化されなかったのは、HPC 観点では残念な点の一つである。

5. おわりに

アクセラレータは、エクサスケール・スパコンに求められる、現在よりも桁違いに良い電力あたり演算性能を実現する一つの手法として有望視されているが、エクサスケール世代におけるアクセラレータのあるべき姿は、まだ明確にはなっていない。本アクセラレータ技術 WG では、アクセラレータの長所・短所、そして課題を浮き彫りにするため、近年普及しつつあるアクセラレータの一形態とも言える GPU を題材に、具体的な事例を用いて調査・議論を重ねた。

第２章では、ディレクティブ方式によるアクセラレータ利用方式に関する調査をまとめた。ディレクティブ方式は GPU 利用の壁を引き下げる効果が大きく、一般的なアプリ開発者が容易に GPU を使うことを可能とし、ループ構成が規則的であり GPU に向いたアプリケーションであれば、汎用 CPU を上回る性能を実際に達成できることを確認した。しかし、性能的にはアクセラレータ向け拡張言語で開発したものには及ばす、また、ループ構造が複雑なアプリケーションでは、そもそもどのようにディレクティブを挿入してよいのか分からないという課題が浮き彫りになった。ディレクティブの種類を豊富にして様々な並列化手法を記述可能にすることも大切であるが、アプリ開発者の並列化作業を支援する仕組みが今後は重要になるだろう。

第３章では、アクセラレータ向け拡張言語によるアクセラレータ利用方式に関する調査をまとめた。CUDA や OpenCL 等の拡張言語を用いることで、ディレクティブ方式では達成できない性能を実現可能なこと、本来は GPU に不向きなアプリケーションであっても、実装アルゴリズムを GPU 向きに変更することで汎用 CPU を上回る性能を達成できることを事例ベースで確認した。しかし、拡張言語で高い性能を達成するには、GPU の動作原理・特性の理解が不可欠であり、一般的なアプリ開発者にこれを求めるのは酷である。

現時点では、アクセラレータ環境において、プログラミングの容易性と高性能を両立する解は残念ながら存在しない。これを解決するアプローチとしては、アクセラレータ HW 強化、ディレクティブ方式の記述性向上、並列化支援機能、自動性能チューニング、DSL(Domain specific language) の開発、アクセラレータ向け拡張言語教育など、いろいろな方向性が考えられる。現在はアクセラレータを使いこなす障壁は高いが、エクサスケール世代に向けて、今後はこの障壁を下げることがますます重要になる。今後も継続的に、アクセラレータの技術動向・方向性に関する議論を行う必要があるだろう。

最後に、SS 研事務局の WG 運営に感謝の意を表すると供に、各種調査や議論に参加して頂いた WG メンバの皆様に深く感謝致します。また、本報告書が、アクセラレータ技術に関する議論の一助となれば幸いである。

[添付資料 1]

コーンビーム再構成の高速化

伊野　文彦

大阪大学大学院情報科学研究科

Cone Beam Reconstruction

- A technique for producing volume data
  - Input data: a series of 2-D projections taken by CT scanner
  - Output data: 3-D volume
- Feldkamp, Davis and Kress (FDK) algorithm
  - Standard but memory-intensive reconstruction method
  - 135 s on a single-core CPU using SSE instructions
- Acceleration is needed for real-time imaging
  - GPU (8.9 s), Cell B.E. (19 s), and FPGA (40 s)

2010/1/19　SS研究会　2

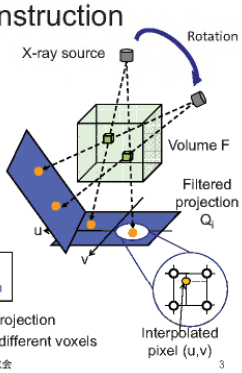
FDK Reconstruction

- Input/output data
  - Input: a series of projections
  - Output: Volume
- Filtering stage
  - Convolution between projections and Shepp-Logan filter
- Backprojection stage
  - accumulates pixels on filtered projections to voxels

$$F(x,y,z) = \sum_i W_i(x,y)\, Q_i(u,v)$$

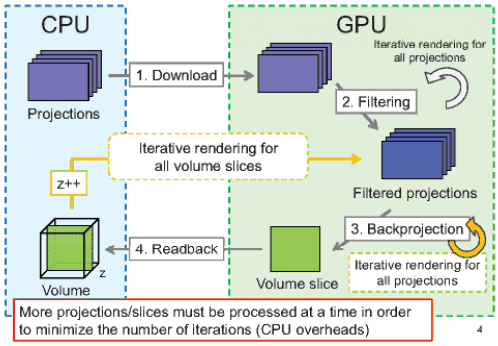
Volume　Weight　Projection

- iterates accumulation for every projection
- has perfect parallelism between different voxels



2010/1/19　SS研究会　3

GPU-Accelerated Reconstruction



4

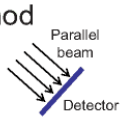
OpenGL版

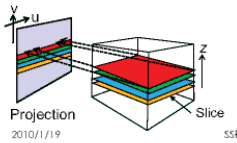
2010/1/19　SS研究会　5

Previous Packing Method

- Assumption
  - Parallel beam geometry
  - Projections have the same resolution as the volume
- 1/4 resolution in v and z axes
  - Succeeding 4 volume slices and 4 rows in a projections are packed into an RGBA pixel

# of iterations is reduced to 1/4






```
Backprojection for (x,y,z)-(x,y,z+3)
// determine (u,v) from (x,y,z)
float u = …
float v = …
// weight computation
float W = …
// fetch pixels on projections
float4 value = Q(u,v);
// Accumulation
F(x,y) += W * value;
```

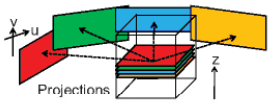
fetches 4 pixels at a time

2010/1/19　SS研究会　6

Our Packing Method

- No assumption
- 1/4 resolution in z axis
  - z-independent computation should be placed outside the loop to perform data reuse
- 4 scalar projections are packed into an RGBA projection
  - with the same resolution
  - to compute coordinates/weights of 4 projections at a time

# of iterations is reduced to 1/16




```
Backpropjection for (x,y,z)-(x,y,z+3)
// determine u from (x,y,z)
float4 u = …
// Weight computation
float4 W = …
// Loop for 4 slices
for (j = 0 ; j < 4 ; j ++ ) {
    // determine v from (x,y,z+j)
    float4 v = …
    // fetch pixels on projections
    float value = Q1(u1,v1) + Q2(u2,v2)
                  + Q3(u3,v3) + Q4(u4,v4);
// Accumulation
    F(x,y,z+j) += W * value;
}
```

Vectorization for 4 projections

2010/1/19　SS研究会　7

Experiments

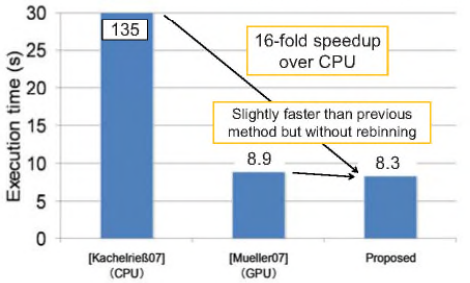
- Performance and quality evaluation
- Machine
  - CPU: Intel Core 2 Extreme 2.93 GHz
  - GPU: nVIDIA GeForce 8800 GTX
- Implementation
  - C++, OpenGL and Cg Toolkit
- Dataset
  - Shepp-Logan phantom

Shepp-Logan phantom

2010/1/19　SS研究会　8

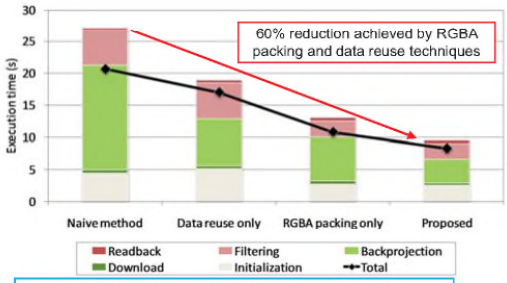
Execution time

512^3-voxel volume from 360 512^2-pixel projections



2010/1/19　SS研究会　9

Breakdown Analysis



RGBA packing is more effective than data reuse because it reduces filtering time as well as backprojection time

10

Efficiency of Backprojection

Effective performance		Measured	Peak	Measured/Peak
VRAM bandwidth (GB/s)		70.8	86.4	82%
Floating point (GFLOPS)	w/ interpolation	231.6	518.4	45%
	w/o interpolation	99.3	345.6	28%

- Our method achieves 82% of peak bandwidth between the GPU and video memory
- VRAM bandwidth limits the entire performance
  - Computational efficiency results in 45%
  - Processing elements in the GPU waits for data to be fetched

2010/1/19　SS研究会　11

CUDA版

2010/1/19　SS研究会　12

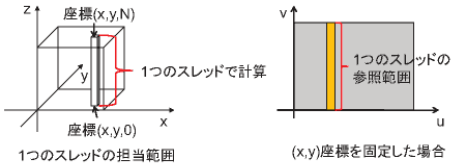
提案手法の実装概要

- アクセス回数の削減
  - グローバルメモリへのアクセス回数を削減
  - ローカルメモリへのアクセス回数の削減
- アクセス遅延の隠蔽
  - Coalesced参照の実現
  - 専攻計算による隠蔽
  - ローカルメモリへのアクセスの除去

2010/1/19　SS研究会　13

スレッド割当

- 1つのスレッドで複数のボクセルを処理
  - (x,y) 座標を固定し全てのz座標について計算
    - 逆投影時に参照する投影像のu座標を固定できる
    - v座標の計算結果を再利用可能



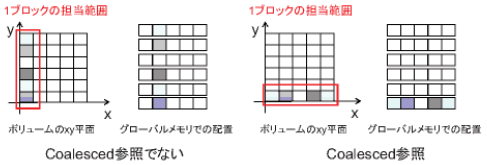
2010/1/19　SS研究会　14

Coalesced参照の実現

- 1つのブロックで複数のx座標を持つボクセルを計算
  - ある時点でスレッドが処理するボクセルの
    グローバルメモリ上での配置が連続となる

複数のy座標を持つ
ボクセルを計算する場合は
メモリ上のアドレスが**連続しない**

複数のx座標を持つ
ボクセルを計算する場合は
メモリ上のアドレスが**連続する**



Coalesced参照でない　　Coalesced参照

2010/1/19　SS研究会　15

グローバルメモリへのアクセス回数の削減

- グローバルメモリに格納しているボクセルVへのアクセスを削減
- 複数の投影像をまとめて処理

1枚毎に投影像を処理する場合

```
CPU  for ( n = 1;n <= N; ) {
       GPU (u,v)を導出;
           V += W_n P_n(u,v);
     }
```

Vのアクセス回数は2N回

4枚の投影像をまとめて処理する場合

```
CPU  for ( n = 1;n <= N;) {
       for(i = 0;i < 4;i++,n++){
       GPU (u,v)を導出;
           tmp += W_n P_n(u,v);
       }
       V += tmp;
     }
```

ローカルメモリへのアクセスが発生

Vのアクセス回数はN/2回

- 投影像をまとめると1つのブロックで使用するレジスタ数が増加
  - 予備実験より5枚以上の投影像をまとめるとレジスタ数の上限を超える

2010/1/19　SS研究会　16

ローカルメモリへのアクセス回数の削減

- ローカルメモリに格納している変数tmpへのアクセス回数を削減
- 複数の投影像に対する逆投影値をまとめてtmpに加算

逆投影値をまとめて加算しない場合

```
CPU  for ( n = 1;n <= N;) {
       for(i = 0;i < 4;i++,n++){
       GPU (u,v)を導出;
           tmp += W_n P_n(u,v);
       }
       V += tmp;
     }
```

tmpのアクセス回数は2N回

逆投影値をまとめて加算する場合

```
CPU  for ( n = 1 ; n <= N ; n += 4 ) {
       GPU (u_1,v_1) を導出;
           (u_2,v_2) を導出;
           (u_3,v_3) を導出;
           (u_4,v_4) を導出;
           tmp += W_1P_1(u_1,v_1) + W_2P_2(u_2,v_2)
                  + W_3P_3(u_3,v_3) + W_4P_4(u_4,v_4);
           V += tmp;
     }
```

tmpのアクセス回数はN/2回

2010/1/19　SS研究会　17

ローカルメモリへのアクセス回数の削減

- ローカルメモリに格納している変数tmpへのアクセス回数を削減
- 複数の投影像に対する逆投影値をまとめてtmpに加算

逆投影値をまとめて加算しない場合

```
CPU  for ( n = 1;n <= N;) {
       for(i = 0;i < 4;i++,n++){
       GPU (u,v)を導出;
           tmp += W_n P_n(u,v);
       }
       V += tmp;
     }
```

tmpのアクセス回数は2N回

逆投影値をまとめて加算する場合

```
CPU  for ( n = 1 ; n <= N ; n += 4 ) {
       GPU (u_1,v_1) を導出;
           (u_2,v_2) を導出;
           (u_3,v_3) を導出;
           (u_4,v_4) を導出;
           V   += W_1P_1(u_1,v_1) + W_2P_2(u_2,v_2)
                  + W_3P_3(u_3,v_3) + W_4P_4(u_4,v_4);
     }
```

ローカルメモリへのアクセスを除去

tmpのアクセス回数はN/2回 → 0回

2010/1/19　SS研究会　18

計算によるアクセス遅延の隠蔽

- アクセス遅延をスレッド内の計算で隠蔽
  - スレッド内で依存関係のない式を先に計算
  - 4枚の投影像に対する処理をスレッド内で繰り返す
    - 予備実験の結果より3回繰り返す

4枚の投影像の処理を3回繰り返す場合

CPU

for (n = 1 ; n < N ; n += 12) {

GPU

$(u_1,v_1) \sim (u_4,v_4)$ を導出;

$V \mathrel{+}= W_1P_1(u_1,v_1) + W_2P_2(u_2,v_2) + W_3P_3(u_3,v_3) + W_4P_4(u_4,v_4);$

アクセス遅延の発生

$(u_5,v_5) \sim (u_8,v_8)$ を導出;

$V \mathrel{+}= W_5P_5(u_5,v_5) + W_6P_6(u_6,v_6) + W_7P_7(u_7,v_7) + W_8P_8(u_8,v_8);$

$(u_9,v_9) \sim (u_{12},v_{12})$ を導出;

$V \mathrel{+}= W_9P_9(u_9,v_9)+W_{10}P_{10}(u_{10},v_{10}) +W_{11}P_{11}(u_{11},v_{11})+W_{12}P_{12}(u_{12},v_{12});$

}

アクセス遅延発生中に計算可能

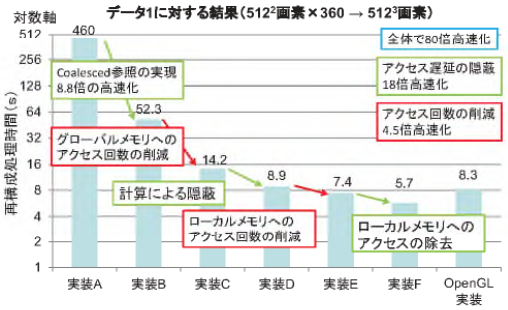
2010/1/19　SS研究会　19

各実装の概要

高速化手法	実装 A	B	C	D	E	F
Coalesced参照	×	○	○	○	○	○
グローバルメモリへのアクセス回数の削減	×	×	○	○	○	○
計算によるアクセス遅延の隠蔽	×	×	×	○	○	○
ローカルメモリへのアクセス回数の削減	×	×	×	×	○	○
ローカルメモリへのアクセスの除去	×	×	×	×	×	○

2010/1/19　SS研究会　20

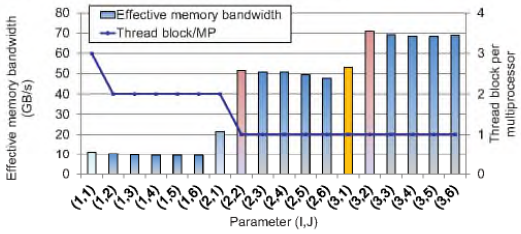
実験結果



2010/1/19　SS研究会　21

Parameter optimization

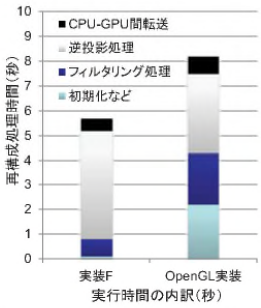
- First, determine I such that each multiprocessor executes a single thread block（I枚の投影像の処理をJ回繰り返し）
- Then, determine J according to available resources



2010/1/19　SS研究会　22

OpenGL実装との比較

- OpenGL実装と比較して実装Fの逆投影処理が低速
  - 1ブロックが使用するレジスタ数が多い
- OpenGL実装と比較して実装Fのフィルタリング処理は高速
  - 共有メモリの使用によりグローバルメモリへのアクセス回数を削減
- OpenGL実装と比較して実装Fの初期化時間が短い
  - OpenGL実装はグラフィクス処理用のCPU側の処理が必要



2010/1/19　SS研究会　23

謝辞

実験用データを提供していただいた
株式会社島津製作所様に感謝いたします

2010/1/19　SS研究会　24

[添付資料 2]

SS研アクセラレータWG
第2回会合資料
～初心者によるGPUプログラミング事例～
2010/01/19

九州大学
山口光章，平山平太，井上弘士，村上和彰

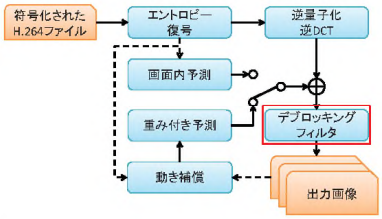
目次

- 対象アプリケーション
  - H.264/AVC　デブロッキングフィルタ
- CUDAによる実装方針
  - マクロブロック内の並列化
  - フレーム内の並列化
- 最適化と効果測定

2010/1/19　2

対象アプリケーション：H.264/AVC　デブロッキング・フィルタ

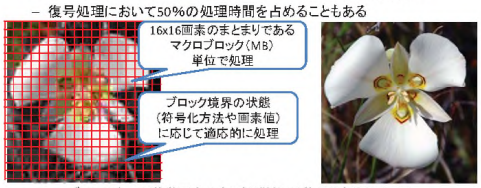
- H.264/AVC　復号器の全体構成



2010/1/19　3

デブロッキング・フィルタ

- 画像の符号化・復号時に生じるブロック歪を減少させるためのフィルタ
  - 動き補償などのフレーム参照において、ブロック歪の伝播を防止
  - 復号処理において50%の処理時間を占めることもある

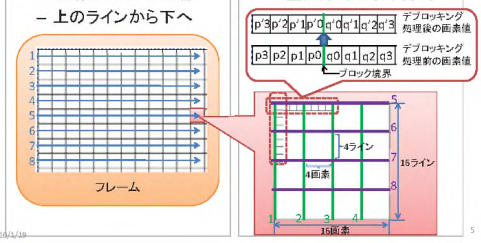


ブロックベース（複数画素のまとまり単位の）符号化処理によってブロックの境界に歪が生じた例

2010/1/19　4

デブロッキング・フィルタの処理順序

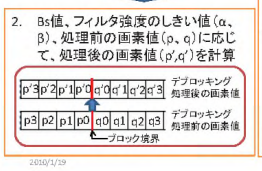
- MBの処理順
  - 左端のMBから右端へ
  - 上のラインから下へ
- MB内の境界の処理順
  - 垂直方向→水平方向

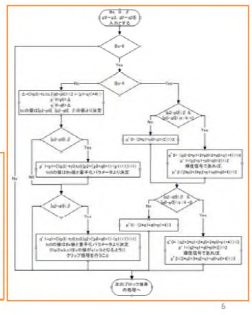


2010/1/19　5

適応的なフィルタ処理の流れ

1. 符号化の状態により、ブロック境界の強度 Bs値（Boundary Strength）を計算
   - 画面内予測
   - マクロブロック境界
   - 動きベクトルの差異
   - 直交変換係数の有無
2. Bs値、フィルタ強度のしきい値（α、β）、処理前の画素値（p、q）に応じて、処理後の画素値（p′,q′）を計算





2010/1/19　6

目次

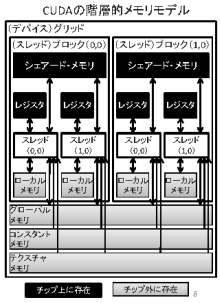
- 対象アプリケーション
  - H.264/AVC　デブロッキングフィルタ
- CUDAによる実装方針
  - マクロブロック内の並列化
  - フレーム内の並列化
- 最適化と効果測定

2010/1/19　7

CUDAによる実装方針（1/2）

- マクロブロックの処理
  - 並列性
    - 垂直、水平方向がそれぞれ16並列に計算可能
  - 境界の処理のデータ依存関係
    - 垂直方向→水平方向
- スレッド・ブロックに割り当て、16スレッドで実行
  - スレッド間で共有可能なオンチップ・メモリ（シェアード・メモリ）を利用
    - 画素値、Bs値のデータ
  - ブロック内のスレッドは同期可能
    - __syncthreads()

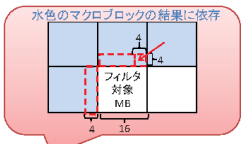
CUDAの階層的メモリモデル



2010/1/19　8

CUDAによる実装方針(2/2)

- フレーム内の並列化
  - データ依存関係が解消されたマクロブロック(MB)から実行
    - ブロック間の同期が必要
      - 現在は、複数回カーネル関数を実行
  - それぞれのMBの処理をCUDAのスレッド・ブロックに割り当て
    - 1920x1088画素の場合、最大で60ブロックが同時実行可



1	2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10	11
5	6	7	8	9	10	11		
7	8	9	10	11				
9	10	11						
11	12							

番号：MBが実行される順番
(同じ番号は同時に実行可)

2010/1/19　9

目次

- 対象アプリケーション
  - H.264/AVC　デブロッキングフィルタ
- CUDAによる実装方針
  - マクロブロック内の並列化
  - フレーム内の並列化
- 最適化と効果測定
  - 実験環境
  - 各種最適化

2010/1/19　10

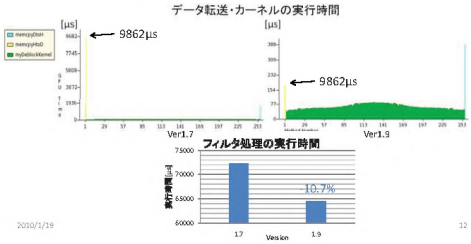
実験環境

- CPU
  - Intel(R) Xeon(R) CPU　E5520
    - コア数：(使用されるのは)1
    - 周波数：2.27GHz
- GPU
  - Tesla C1060
    - デバイスメモリ：4GB
    - SM(ストリーミング・マルチプロセッサ)数：30
    - SP(ストリーミング・プロセッサ)数：240
    - 周波数：1.3GHz
    - シェアード・メモリ：16KB
- CUDA　toolkit, sdk
  - version:2.3
- 入力データ
  - 画像サイズ：1920x1088
  - フレーム数：1
  - ピクチャの符号化方式
    - Pピクチャ
    - メイン・プロファイル

2010/1/19　11

最適化1：ホスト-デバイス間のデータ転送量削減

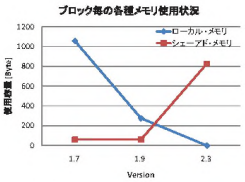
- 画素値とBs値のデータ型を変更
  - int型(32bit)　→　unsigned char型(8bit)

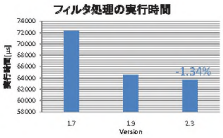


2010/1/19　12

最適化2：ローカルメモリの使用量削減

- 画素値、Bs値のデータ　→　シェアード・メモリ
  - 画素値：スレッド・ブロック毎に　32x20画素 = 640[byte]
- その他のローカル・メモリのデータ　→　コンスタント・メモリ

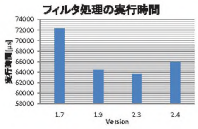




2010/1/19　13

最適化3：カーネル呼び出しのホストプログラムの高速化

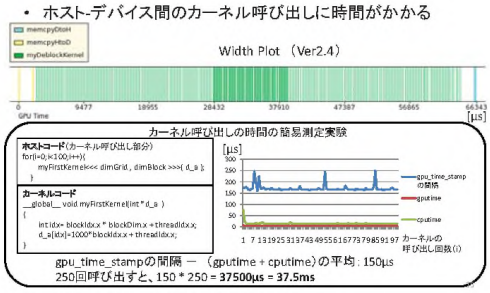
- ホスト関数のカーネル関数呼び出し部分の除算、mod演算をシフト演算やAND演算に変更
  - 除算：5箇所、mod演算：1箇所



2010/1/19　14

復号処理の実行時間の内訳

- ホスト-デバイス間のカーネル呼び出しに時間がかかる



gpu_time_stampの間隔 ー (gputime + cputime)の平均：150μs
250回呼び出すと、150 * 250 = 37500μs = 37.5ms

最適化4(案)：カーネル呼び出し回数削減

- ブロック間の同期方法
  - 現状
    - カーネルを複数回呼び出す
  - 考えられるアプローチ
    - 1スレッド・ブロックあたりの仕事を増やす
      - 現状　1スレッド・ブロック：1マクロブロック
        → 1スレッド・ブロック：2,3マクロブロック
    - デバイス上でブロック間の同期を取る
      - グローバルメモリに各マクロブロックの状態を保持する
      - __threadfence()の利用

2010/1/19　16

デバイス上で同期

- 各マクロブロックの状態（フィルタの完・未完）をグローバルメモリ上で共有
- __threadfence()を用いて、状態確認のグローバルメモリ・アクセスの同期をとる

グローバルメモリ上にテーブルを保持

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

フィルタが完了後、状態を更新

1	2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	0	0
7	8	9	10	11	0	0	0	0
9	10	11	0	0	0	0	0	0
11	12	0	0	0	0	0	0	0

2010/1/19　17

実装の経過

Version	内容	日付
1.7	フレーム単位での正常なデブロッキング・フィルタ処理が可能。	1/9
1.8	ホスト、デバイスの画素値データの型ををint型からunsigned char型に変更。	1/10
1.9	Bs値およびフィルタ処理用の定数配列データclip_Tabをint型からunsigned char型に変更。	1/30
2.0	shared memoryにスレッド・ブロック毎に20x20画素値分のデータを格納するように変更。変更前はローカルメモリに存在。	1/13
2.1	shared memoryにBs値のデータにも利用。	1/13
2.2	ローカルメモリを使用していたその他のconstなデータ（CLIP_TAB、edgeなど）を__constant__領域（コンスタント・メモリ）を利用するように変更。これによりローカルメモリの使用量が0に。	1/14
2.3	画素値のデータを32x20画素をshared memoryに格納（グローバルメモリからのコピー作業において、すべてのスレッドで同じ挙動に変更。）	1/14
2.4	ホスト側のカーネル関数を呼び出す部分の計算方法を変更。 除算→算術シフト演算　5箇所 MOD演算→AND演算　1箇所	1/14

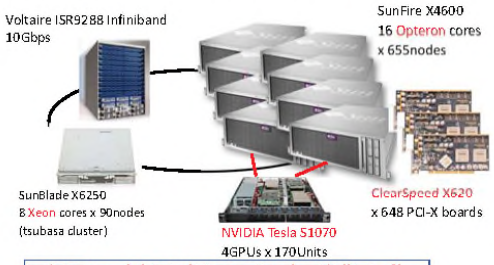
2010/1/19　18

[添付資料 3]

GPUを搭載したスパコンTSUBAMEにおけるLinpackベンチマーク

東京工業大学
情報理工学研究科
遠藤敏夫

東工大TSUBAME 1.2 (2008)



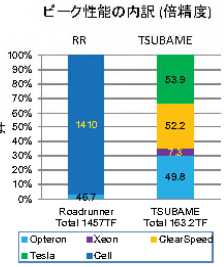
アクセラレータを用いたシステムの中では世界三位
ピーク性能 163TFlops, Linpack 性能 87TFlops
Nov 09のTop500では56位

LinpackとHPL

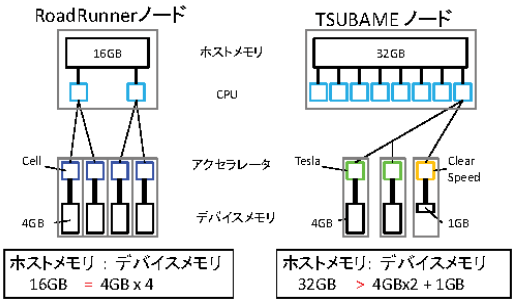
- Linpack: 密行列連立一次方程式を直接法で解くベンチマーク
  - 行列サイズN, 計算量O(N^3)
  - 倍精度
- HPL (High-performance Linpack) by A. Petitet
  - 著名なMPI並列実装
  - ブロック化されたLU分解+部分ピボッティングに基づく
  - Top500ではNの選択は自由
    - メモリにおさまる範囲でNが大きいのが有利
    - TSUBAMEでは Nは約1,000,000
  - DGEMMカーネルの占める時間がほとんど

誰が計算する?

- カーネル計算
  - RoadRunnerでは, Cellが性能の96%を占める
    - CPUsは4%
  - ⇒ Cellのみ利用
  - TSUBAMEでは, CPUが35%
    - いずれかのプロセッサ種類を省くのは性能低下が大きい
  - ⇒ CPU, GPU, ClearSpeedの全てを利用
- カーネル以外の処理
  - MPI通信、ピボッティングにはCPU利用



行列データをどこにおく? (1)

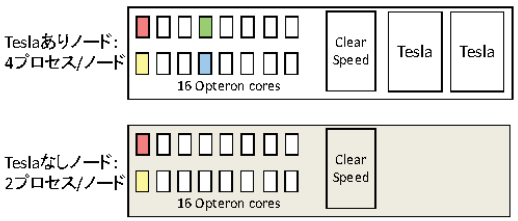


行列データをどこにおく?(2)

- Linpackの効率のためには行列サイズNが大きいのが望ましい
  - ⇒ ホストメモリサイズに近付けたい
- RRでは,
  - (1) デバイスメモリ容量 = ホストメモリ容量
  - (2) カーネル計算はCell上
  - ⇒ 行列データはCellデバイスメモリに
- TSUBAMEでは,
  - デバイスメモリ容量 < ホストメモリ容量
  - ⇒ 行列データは通常、ホストメモリに

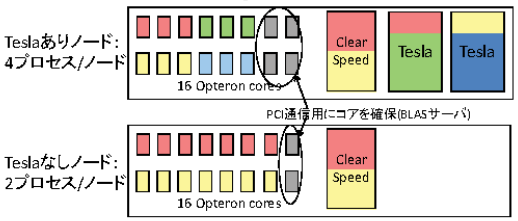
TSUBAME上Linpackの挙動(1)

- カーネル計算以外



TSUBAME上Linpackの挙動(2)

- カーネル計算中
  - 他CPUコア・アクセラレータも利用
  - 性能に応じた比率で、A_Lの更新(DGEMM)を依頼



実験条件

改造HPLから、以下のライブラリを利用

- Voltaire MPI
- BLASライブラリ
  - CPU: Goto BLAS 1.26
  - ClearSpeed: CSXL 3.1, on-board kernel
  - Tesla: 本グループ(額田)作成のDGEMM/DTRSM、CUDA 2.0利用
- ブロックサイズB=1152
  - RoadRunnerのB=128や、TSUBAME CPUのみ時のB=240より大幅に大きい
- システム全体実験:
  - TSUBAME648ノード(うち312にTesla) + Xeon 80ノード
  - 行列サイズN=1059839 → 3時間強

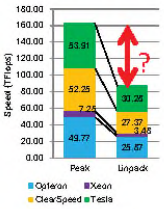
TSUBAMEのLinpack速度

	Jun06	Nov06	Jun07	Nov07	Jun08	Nov08	Jun09
Linpack 性能(TF)	38.18	47.38	48.88	56.43	67.70	77.48	87.01
順位	7	9	14	16	24	29	41

HW/SWの改良により、2006年以来6回連続の速度向上(Top500史上初)
Linpack性能/Peak = 53%

ピーク性能比についての考察

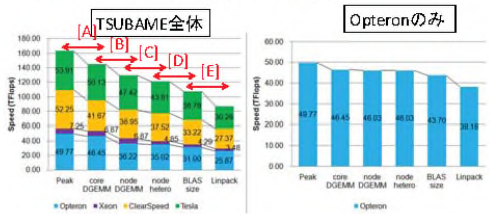
	ピーク (TFlops)	Linpack (TFlops)	性能比
TSUBAME	163.2	87.01	53%
TSUBAME Opteronのみ	49.87	38.18	77%
RoadRunner	1457	1105	76%



- ピークとLinpackの乖離の原因は？
  - PCI通信オーバヘッド? 異種ノード混在?
  - 以下、主にカーネル性能に注目して考察

オーバヘッドの解析

- [A] コア単体のDGEMMのオーバヘッド
- [B] PCIe通信＋バス干渉のオーバヘッド
- [C] ノードの不均一性のオーバヘッド
- [D] カーネルの行列サイズのオーバヘッド
- [E] その他のオーバヘッド(ノード間通信など)



[添付資料 4]

PGIアクセラレータコンパイラについて

黒川 原佳
情報基盤センター
独立行政法人 理化学研究所

内容

- PGIアクセラレータコンパイラの概要
- Programming Model(指示行)について
- SAXPYの例
- HimenoBMTの例
  - Step by Step
- HimenoBMT チューニング

PGIアクセラレータコンパイラの概要

- Portland Group Inc.が開発したNvidia GPU(CUDA)向けに開発した指示行モデルのコンパイラ
  - 評価版として9.0が2009年6月にリリース
  - 正式版としてPGI2010(PGI 10.0)が11月にリリース
  - x86-64 + NVIDIA CUDAが利用可能なGPUで利用可能
  - CUDAコードへ変換して，CUDAコンパイラを利用
- OpenMPに近い指示行
  - GPUで実行する領域を指定し，その部分をGPUで実行
  - DOループをオフロードするような指定
- 元コードをほとんど変更せずにとりあえずGPUを利用可能
  - コードにもよるが，GPU上での実行領域を設定
    - Fortranなら2行入れるだけで，領域内で利用している配列をコンパイラが自動判定．とりあえずGPUが利用できる

コンパイル・オプション

- PGI2010でサポートされているオプション
  - -ta: !$acc指示行を有効にする
    - サブオプション
      - Nvidia：NVIDIA GPUを対象とする
      - cc1[123]：Compute capability 1.1,1.2または1.3向けのコードを生成する
      - Time：簡易性能プロファイリングを出力する ← バグあり？
      - keepgpu：生成したCUDAコードを出力する
  - -Minfo=accel:
    - Regionに対応するGPUカーネル生成の際のメッセージを出力する

Programming Model(指示行)について(1)

- GPUで実行する領域を設定
  - Accelerator Compute Region Directive
    - 構文

```
!$acc  region [clause [, clause] ..]
      structured block
!$acc  end region
```

    - Clause
      - copy(list): regionの入口でCPU→GPUのコピー、出口でGPU→CPUのコピーを行なうデータを指定
      - copyin(list): 入口でCPU→GPUのコピーを行なうデータを指定
      - copyout(list): 出口でGPU→CPUのコピーを行なうデータを指定
      - local(list): region内でlocalなデータを指定
      - Listには基本的に配列指定．配列はインデックスも記載可能
  - 基本的にこの設定だけでCUDA化が可能
    - データの移動はコンパイラが自動判定(結構賢い)
    - ただし，やはり細かく指定するには人手が必要

Programming Model(指示行)について(2)

- GPUでデータを保持する領域を設定
  - Accelerator Data Region Directive
    - 構文

```
!$acc  data region [clause [, clause] ..]
      structured block
!$acc  end data region
```

    - Clause
      - copy(list)：regionの入口でCPU→GPUのコピー、出口でGPU→CPUのコピーを行なうデータを指定
      - copyin(list)：入口でCPU→GPUのコピーを行なうデータを指定
      - copyout(list)：出口でGPU→CPUのコピーを行なうデータを指定
      - local(list)：region内でlocalなデータを指定
      - mirror(list)：Fortranだけで有効．領域内で動的に領域のallocation, deallocationが可能
      - update device/host(list)：region，data regionの外部でコピー
      - update device(list)：計算実行前か入口でCPU→GPUにコピー
      - update host(list)：計算実行後か出口でGPU→CPUにコピー

Programming Model(指示行)について(3．1)

- Accelerator Loop Mapping指示行
  - 構文
    - !$acc do [clause [, clause]…]
      do loop
  - Clause
    - host[(width)]：host上で逐次に実行する
    - seq[(width)]：accelarator上で逐次に実行する
    - unroll [(width)]：UNROLLを指示，accelerator上で逐次で実行する
    - parallel[(width)]：parallel modeで並列化する
    - vector[(width)]：vector modeで並列化する
    - cahe[(list)]：メモリ階層の上位に移動することを試す
    - independent：ループ内のデータに依存関係がないことを示す
    - kernel：ループボディーが計算カーネルであることを示す
    - private(list)：デバイス上のループの1反復で1回しかコピーされない
    - shortloop：accelaratorに十分なループかどうかを指示

Programming Model(指示行)について(3．2)

- 指定したループについてどのような並列化を適用するかを指定する
  - Parallel modeがblock並列に、vector modeがthread並列に対応している
- (width)で並列数を指定
  - 指定がない場合、自動的に決定される
  - 様々な制限がある
    - Parallel指定ループは2ネストまで(CUDAにおけるGridが2次元までであることから)
    - Vectorのwidthは256まで

Programming Model(指示行)について(4)

- GPUにデータ宣言する設定
  - Accelerator Declarative Data Directive
    - Accelerator region 内で利用可能
    - 構文
      !$acc declclause [, declclause] …
    - Declclause
      - copy(list), copyin(list), copyout(list), local(list), mirror(list)はdata regionと同一
      - reflected(list)：Fortranで利用可能．実配列をlist中の仮配列に結びつける．Fortranのインターフェースブロックが必要．呼び出された時点で指定データをコピー

Programming Model(指示行)について(5)

- GPUにアップデート宣言する設定
  - Accelerator Update Directive
    - Accelerator region 内で利用可能
    - 構文
      !$acc update updateclause [, declclause] …
    - updateclause
      - device(list), host(list)
    - ただし，計算領域かデータ領域外でコピーされる

実行環境

- ホストノード
  - NEC Express 5800/56Xg
    - CPU: Xeon 5570 (2.93GHz) X 2
      Memory: 24GB
    - I/O: PCI-Express Gen2 ×16
    - Accelerator: NVIDIA Tesla C1060

```
% pgaccelinfo
Device Number:                  0
Device Name:                    Tesla C1060
Device Revision Number:         1.3
Global Memory Size:             4294705152
Number of Multiprocessors:      30
Number of Cores:                240
Concurrent Copy and Execution:  Yes
Total Constant Memory:          65536
Total Shared Memory per Block:  16384
Registers per Block:            16384
Warp Size:                      32
Maximum Threads per Block:      512
Maximum Block Dimensions:       512 x 512 x 64
Maximum Grid Dimensions:        65535 x 65535 x 1
Maximum Memory Pitch:           2621440
Texture Alignment               256B
Clock Rate:                     1296 MHz
Initialization time:            1003094 microseconds
Current free memory             4240998240
Upload time (4MB)               1013 microseconds
( 718 ms pinned)
Download time                   978 microseconds
( 731 ms pinned)
Upload bandwidth                4140 MB/sec (3641
MB/sec pinned)
Download bandwidth              4288 MB/sec (3737
MB/sec pinned)
```


SAXPYの例

```
program main
  integer :: i
  real(4), dimension(500) :: x, y
  real(4) :: b

  b = 3.0

! initialize the two arrays x and y
  do i = 1, 500
    y(i) = i
    x(i) = i + 3.0
  end do

! add a scalar multiple of x to y
!$acc region
  do i = 1, 500
    y(i) = y(i) + b * x(i)
  end do
!$acc end region
end program
```

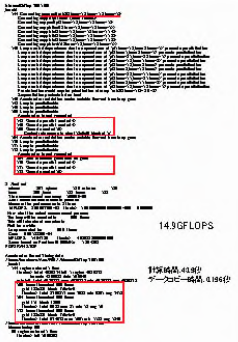
```
% pgfortran -fast -ta=nvidia,time -Mreporocess -
Minfo=accel saxpy.f90 /usr/local/pgi/linux86-
64/10.0/lib/kvsetta.o
saxpy.f90:
main:
    18, Generating copyin(x(1:500))
        Generating copy(y(1:500))
    19, Loop is parallelizable
        Accelerator kernel generated
        19, !$acc do parallel, vector(256)
$ ./a.out

Accelerator Kernel Timing data
/home/kurokawa/Acc/ /saxpy.f90
main
    18: region entered 1 time
        time(us): total=1049917 init=1049766
region=151
                  kernels=24 data=127
        w/o init: total=151 max=151 min=151
avg=151
    19: kernel launched 1 times
        grid: [2]  block: [256]
        time(us): total=24 max=24 min=24
avg=24
```

HimenoBMT

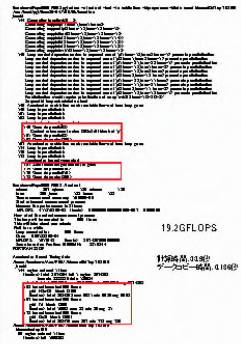
- 条件
  - Fortran90
  - 反復回数は500回固定
- 私の習熟度
  - 第1段階に至るまで：数日
  - 第1段階：1時間
  - 第2段階：10分
  - 第3段階：30分
  - 第4段階：数時間

HimenoBMTの例（1段階）

HimenoBMTの例（2段階）



HimenoBMTの例（3段階）



まとめ＆HimenoBMTの結果分析

- PGIアクセラレータコンパイラについて
  - 指示行とその内容
  - HimenoBMTへの適用とその性能について
- ベンチマークコードとして，チューニングすると21.8GFLOPS程度
  - 計算内容の変更はなし．（GOSA計算あり）
    - ちなみにGOSA計算しない場合，22.4GFLOPS
  - 内容を確認中
- CUDAコードとPGIコンパイラの性能比較
  - C1060でCUDAのチューニングコードを実行した場合をGTX280とのメモリ帯域比で推定すると約51.8GFLOPS
  - 今回の結果と比較すると，第3段階で37%，第4段階で42%
- コーディングの楽さで言えば，PGIは指示行を8行入れるだけ
- MPI版を考えた場合，data宣言指示行やupdate指示行を入れると，袖領域のみをアップデート出来そうなので，ある程度の性能はだせるかもしれない

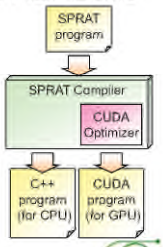
[添付資料 5]

複合型計算機向け ソフトウェア開発環境 SPRATのご紹介と性能評価

東北大学大学院情報科学研究科
滝沢寛之

SPRATの自動最適化機構

- 一般的に効果が見込める最適化手法を自動化
  - プロセッサに合わせた言語で出力
  - GPU用をCUDA言語で出力
    - メモリアクセスの調整が大きな影響
    - 実行時パラメータの調整が必須
  1. 効果的な最適化手法の自動適用
     - 共有メモリを活用したメモリバンド幅の改善
  2. 実行パラメータの自動決定
     - 最適スレッドブロックサイズの自動探索



スレッドブロック形状の自動調整

- SPRATのプロファイリング機構を利用
  - 処理データサイズと実行時間の関係を保持
  - 線形性能モデルを最小二乗近似法によって自動生成
- 最適なスレッドブロック形状の自動抽出
  1. 異なるスレッドブロック形状でプログラムをビルド
  2. サンプルデータに対するプロファイリングを行う
  3. 集まったデータから最適スレッドブロック形状を推定

性能評価

- ベンチマーク
  - 姫野ベンチマーク（512×256×256）
  - LU分解（4096×4096）
- 評価環境
  - CPU：Intel Core i7 920 2.66GHz
  - GPU：NVIDIA Tesla C1060
  - OS：Linux 2.6.18-164.el5 (CentOS 5.4)
  - Video Driver：Forceware 190.29
- 評価対象
  - Baseline ・・・ SPRATによる基本実装
  - Optimized ・・・ 最適化1を自動適用
  - Auto-tuned ・・・ 最適化1＋最適化2を自動適用

姫野ベンチのSPRATコード

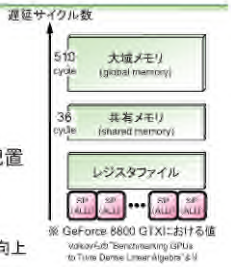
```
/ the main kernel of HONMA benchmark /
kernel map jacobi( gather stream<float> p,
    in stream<float> a0, in stream<float> a1,
    in stream<float> a2, in stream<float> a3,
    in stream<float> b0, in stream<float> b1,
    in stream<float> b2, in stream<float> c0,
    in stream<float> c1, in stream<float> c2,
    in stream<float> bnd, in stream<float> wrk1,
    out stream<float> wrk2, out stream<float> gosa)
{

    float ss, s0;

    s0 = + a0*p[+1][ 0][ 0] + a1*p[ 0][+1][ 0] + a2*p[ 0][ 0][+1]
    + b0*( p[+1][+1][ 0] - p[+1][-1][ 0] - p[-1][+1][ 0] + p[-1][-1][ 0] )
    + b1*( p[ 0][+1][+1] - p[ 0][-1][+1] - p[ 0][+1][-1] + p[ 0][-1][-1] )
    + b2*( p[+1][ 0][+1] - p[-1][ 0][+1] - p[+1][ 0][-1] + p[-1][ 0][-1] )
    + c0*p[-1][ 0][ 0] + c1*p[ 0][-1][ 0] + c2*p[ 0][ 0][-1] + wrk1;

    ss = ( s0*a3 - p[0][0][0] )* bnd;
    gosa = ss*ss;
    wrk2 = p[0][0][0] + 0.8f *ss;
}
```


最適化2：実行時パラメータの自動調整

- CUDAにおけるカーネル起動パラメータ
  - グリッドサイズ ・・・ スレッドブロック数の決定
  - スレッドブロックサイズ ・・・ 同時に処理されるスレッド数の決定
  - 動的共有メモリ割当量 ・・・ SPRATでは不使用
- スレッドブロックサイズが性能におおきな影響
  - 大きいスレッドブロック ＝ グローバルメモリアクセス遅延を隠蔽
  - 小さいスレッドブロック ＝ 同時に動作するスレッドブロック数が増加

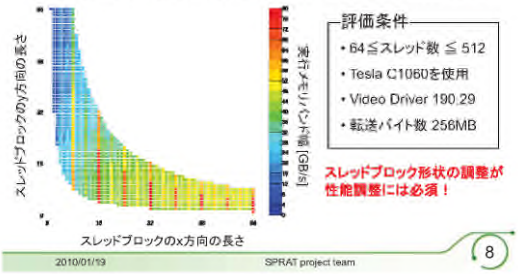
最適化1：共有メモリの利用

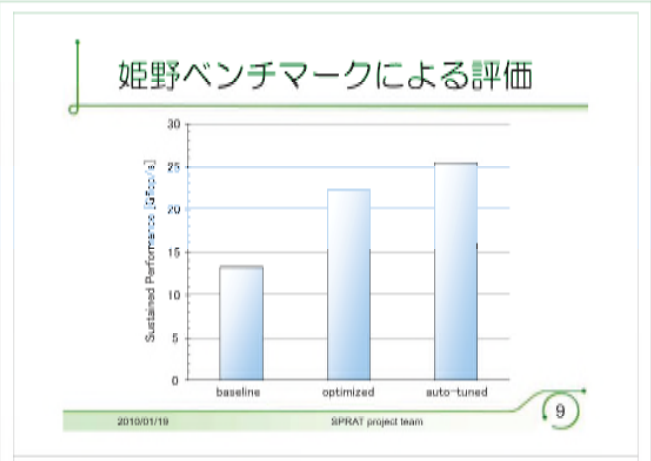
- 共有メモリ
  - 低遅延時間（レジスタと同等）
  - 明示的な読み書きが可能
- 再利用性の高いデータの再配置
  - 複数回利用されるデータを抽出
  - 共有メモリからデータを供給
    → 実効メモリバンド幅を向上

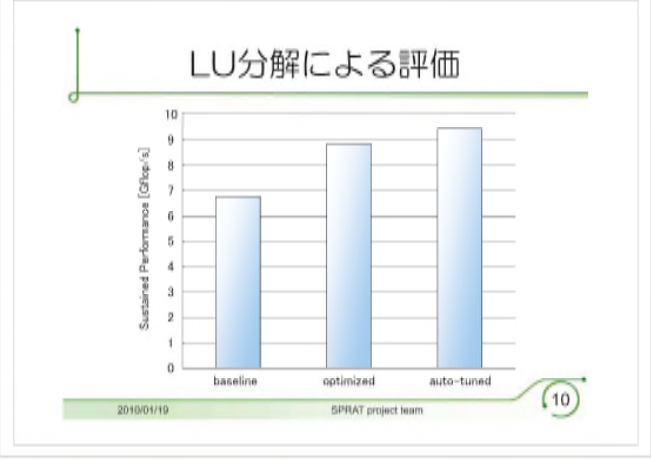


スレッドブロックサイズの影響

- データコピー処理における実行メモリバンド幅の変化




姫野ベンチマークによる評価
Sustained Performance [GFlop/s]
30
25
20
15
10
5
0
baseline
optimized
auto-tuned
2010/01/19
SPRAT project team
9


LU分解による評価
Sustained Performance [GFlop/s]
10
9
8
7
6
5
4
3
2
1
0
baseline
optimized
auto-tuned
2010/01/19
SPRAT project team
10

[添付資料 6]

THE POSSIBILITIES ARE INFINITE FUJITSU

GPGPUプログラミングの状況
－ PGI Accelerator －

2009.1.20
次世代TC本 ソフトウェア開発統括部
堀田 耕一郎

TESLAアーキテクチャ

FUJITSU

- ベクトル長256，30コア
  - ベクトル並列とthread並列のハイブリッド
- Device Memory 4GB

http://www.pgroup.com/lit/articles/insider/v1n1a1.htm より

Copyright 2010 FUJITSU LIMITED

GPGPUを使用するために

FUJITSU

- プログラミングの要件
  - CPUからアクセラレータへのデータ転送
    - 細い転送路
  - GPUへの実行制御の引渡し
  - (GPU内での)並列化
    - SIMD(ベクトル)化
    - マルチスレッド
- 要件の実現方法
  - 詳細記述：CUDA, OpenCL
  - (半)自動記述：PGI Accelerator

Copyright 2010 FUJITSU LIMITED

PGIコンパイラ：Accelerator（1）

FUJITSU

- 「やるべきこと」は基本的に自動実行
  - データ転送，GPUの呼び出し，並列化
    - 1ストリームのプログラムとして記述可能
  - 実行制御の引渡し
    - 一連の手続きとして記述
  - 自動並列化
    - GPU化したい場所のみ指定：Compiler directive
      cf. CUDA：すべてプログラマが記述
    - *すべて自動化すると，過剰にGPU化し，却って遅くなる？*

Copyright 2010 FUJITSU LIMITED

PGIコンパイラ：Accelerator（2）

FUJITSU

- チューニングポイント
  - GPU化部分の選択
  - データ転送の最適化
    - 転送量最小化アプローチ→転送起動回数の縮減
    - 不要な転送？：現行PGIコンパイラの問題点
  - GPU内でのデータアクセス最適化
    - 連続アクセスへの組み換え
  - “scheduling problem”
    - Thread並列とベクトル演算の組み合わせ

Copyright 2010 FUJITSU LIMITED

Region指定

FUJITSU

```
!$acc region [clause[,clause]…]
    Structured block
!$acc end region
```

- 指定されたregion内のループをアクセラレータで実行
  - データの copy in/out自動生成
- アクセラレータで実行できない部分
  ⇒ ホストで実行
  - 並列化できないところを指定しても実行は保証

Copyright 2010 FUJITSU LIMITED

Data Region指定

FUJITSU

!$acc data region [clause[,clause]…]

structured block

!$acc end data region

- 指定されたデータをdevice memoryに配置し，copy in/outを実行
  - Clause
    - copy, copyin, copyout, local, mirror (Fortran)

Copyright 2010 FUJITSU LIMITED

Loop mapping

FUJITSU

- thread並列とSIMD(vector)並列の選択

  !$acc do [clause[,clause]…]

  do loop
  - Clause
    - host, parallel, seq, vector, unroll, kernel, shortloop, private, cache
  - 指定がなければ，コンパイラが自動的に選択
    - host : ホストプロセッサでの実行
    - parallel : アクセラレータでのdo-all型並列実行(thread並列)
    - seq : アクセラレータでの逐次実行
    - vec : アクセラレータでのベクトル実行
    - unroll : アクセラレータでの逐次実行時のアンローリング
    - kernel : アクセラレータでの実行
    - shortloop :
    - private :
    - cache : アクセラレータ上のキャッシュに乗せるヒント

Copyright 2010 FUJITSU LIMITED

PGIコンパイラの評価

FUJITSU

- まだまだ，並列化認識，命令生成は弱いようだ
  - PGIのチューニング例

    http://www.pgroup.com/lit/articles/insider/v1n2a1.htm より

```
for ( i = 1; i < n ; ++ i ){
  a[i] = b[i] + c[i] ;
  d[i] = a[i-1] + 2*a[i] ;
}
```

ソースチューニング

```
for ( i = 1; i < n ; ++ i )
  a[i] = b[i] + c[i] ;
for ( i = 1; i < n ; ++ i )
  d[i] = a[i-1] + 2*a[i] ;
```


Copyright 2010 FUJITSU LIMITED

PGIコンパイラの評価

FUJITSU

- データ転送の認識

  http://www.pgroup.com/lit/articles/insider/v1n2a1.htm より
  - 必要なデータは「間違いなく」copy-in/out
  - Region内で代入されたデータは（その後使われなくても）copy-out
    - データフローの認識に弱点？
  - 転送量の最小化のポリシー
    - 連続転送に変換することでオーバヘッド削減
    - 転送を統合する最適化に弱点？

Copyright 2010 FUJITSU LIMITED

まとめ　PGIアクセラレータ

FUJITSU

- CUDA, OpenCLと比較した使いやすさ
  - 簡単な指示行でソコソコの性能
- 最適化機能には未熟な点も
  - チューニングポイント
  - 今後の最適化機能の強化点になりそう

Copyright 2010 FUJITSU LIMITED

[添付資料 7]

FUJITSU

OpenCLの評価
－ 最適化の観点から －

2010.4.7
次世代TC本 ソフトウェア開発統括部
堀田 耕一郎

shaping tomorrow with you

TESLAアーキテクチャ

FUJITSU

- ベクトル長256，30コア
  - ベクトル並列とthread並列のハイブリッド
- Device Memory 4GB

http://www.pgroup.com/lit/articles/insider/v1n1a1.htm より

2 Copyright 2010 FUJITSU LIMITED

アクセラレータを使用するために

FUJITSU

- プログラミングの要件
  - CPUからアクセラレータへのデータ転送
    - 細い転送路
  - GPUへの実行制御の引渡し
  - (GPU内での)並列化
    - SIMD(ベクトル)化
    - マルチスレッド
- 要件の実現方法
  - 詳細記述：CUDA, OpenCL
  - (半)自動記述：PGI Accelerator

3 Copyright 2010 FUJITSU LIMITED

OpenCL プログラムモデル

FUJITSU

- データ並列
  - データの配置を指定
  - メモリモデル：local, global, host と転送
    - 明示的な local ⇔ global ⇔ host
- プログラマにとっての負荷？
  - 最適なデータ配置，転送が容易に記述できるか？
    - 冗長な転送？

4 Copyright 2010 FUJITSU LIMITED

最適化，ベクトル化，並列化

FUJITSU

- プログラマによる完全なメモリマネジメント
  - どこまでコンパイラによる最適化で崩せるか？
    - 配置の最適化？
    - 転送の結合(coalescing)，削除
- ベクトル化
  - ループは素直に自動ベクトル化
  - 高速化のために，threadを意識したプログラミング
- マルチスレッド
  - プログラムに記述

5 Copyright 2010 FUJITSU LIMITED

まとめ：評価

FUJITSU

- PGIアクセラレータと比較した難解さ
  - データ並列プログラム：データ配置，転送の記述
  - threadの中のベクトルを意識？
- 将来性
  - 最適化との相性，プログラミングを簡単に
    - 完全な記述が最適化の妨げにならないか
    - グローバルメモリアクセスの最適化
  - オープン，マルチプラットフォーム？
    - 普及への切り札？
    - でも，マルチプラットフォームの必要があるのか？
      - nVidiaとAMDを本当に同じプログラムで走らせる？

6 Copyright 2010 FUJITSU LIMITED

[添付資料 8]

FUJITSU

SS研アクセラレータWG 2010/4/7

CUDA and OpenCL

富士通研究所
サーバテクノロジ研究部
成瀬

shaping tomorrow with you

Copyright 2010 FUJITSU LABORATORIES LTD.

OpenCL

FUJITSU

- Apple提唱、Chronosで仕様化
  - ポータビリティ (CUDAへの不安?)
- プラットフォーム
  - GPUだけではない
  - 各種アクセラレータ、CPU
- 開発環境
  - nVidia: nVidia GPU用
  - ATI: ATI GPU用、x86 CPU用
  - Apple: x86 CPU用
  - IBM: Cell用 (正式版は出ない?)

Copyright 2010 FUJITSU LABORATORIES LTD.

概要

FUJITSU

- CUDAとOpenCLの比較
  - nVidia環境上
  - プログラミング・デバッグ・性能
- OpenCL間の比較 (GPU間の比較)
  - nVidia(GT285) and ATI(Radeon HD 5870)
  - ポータビリティ、性能
- 3D-FDTD(電磁場解析)を事例

Copyright 2010 FUJITSU LABORATORIES LTD.

開発環境・測定環境

FUJITSU

- 開発環境

	nVidia	ATI
CUDA	RHEL5.3(64bit) CUDA 2.3 driver: 190.53	×
OpenCL	RHEL5.3(64bit) CUDA 2.3(w/ opencl) driver: 190.29	OpenSUSE 11.0(64bit) ATI Stream SDK v2.0 driver: fglrx-8.682.2

- 測定環境
  - Host: Intel Core-i7 (2.66GHz)、DDR3-1333
  - GPU: nVidia GTX285、ATI Radeon HD5870

Copyright 2010 FUJITSU LABORATORIES LTD.

FUJITSU

- 3D-FDTD

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

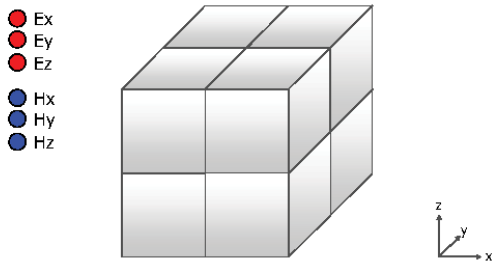
- Maxwell方程式の数値解法の一種
- 時間領域差分法
  - Finite Difference Time Domain method
- 電界・磁界要素を交互に計算
  - t=nΔ:　電界を計算 (Ex,Ey,Ez)
  - t=(n+1/2)Δ:　磁界を計算 (Hx,Hy,Hz)

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

■ 直交格子

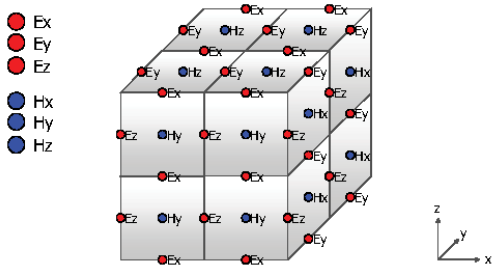


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

■ 電界成分(●)・磁界成分(●)の配置

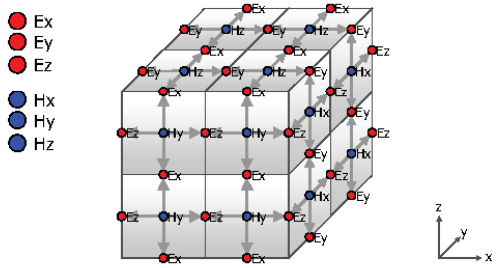


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

■ t=nΔ: 電界成分を更新( )

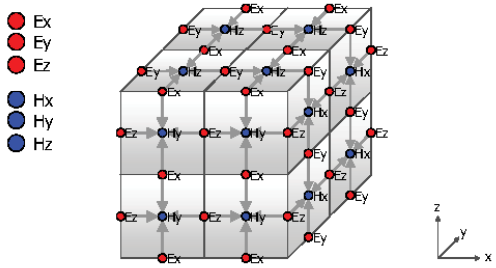


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

■ t=(n+1/2)Δ: 磁界成分を更新( )



Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD(電磁場解析)

FUJITSU

■ Ex値の更新

Ex(z,y,x) = cex(z,y,x) * Ex(z,y,x) + dex(z,y,x) *
(Hz(z,y,x) - Hz(z,y-1,x) - Hy(z,y,x) + Hy(z-1,y,x))

■ Hx値の更新

Hx(z,y,x) = chx(z,y,x) * Hx(z,y,x) + dhx(z,y,x) *
(Ez(z,y,x) - Ez(z,y+1,x) - Ey(z,y,x) + Ey(z+1,y,x))

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD

FUJITSU

■ 特徴
- 規則的なメモリアクセス
- メモリバンド幅ネック
- 6個の更新ルーチン
  - •Ex,Ey,Ez ←→ Hx,Hy,Hz
  - •それぞれ更新範囲が微妙に違う

■ 条件
- 問題サイズ:(128,128,128)、単精度
  → メモリ使用量: 144MiB
- 境界条件: 固定端 (境界成分は更新しない)

Copyright 2010 FUJITSU LABORATORIES LTD.

FUJITSU

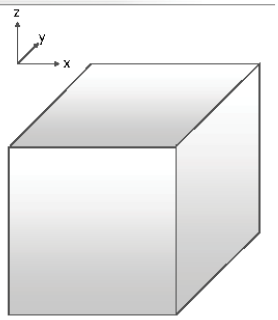
■ nVidia CUDA

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on CUDA

FUJITSU

- 実装: asis
  - 共有メモリは未使用

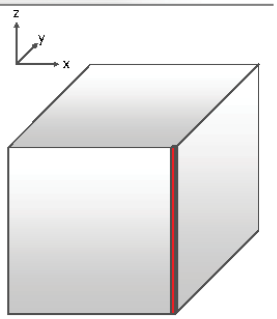


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on CUDA

FUJITSU

- 実装: asis
  - 共有メモリは未使用
- 1スレッド(赤)
  - 担当量: (1,1,128)

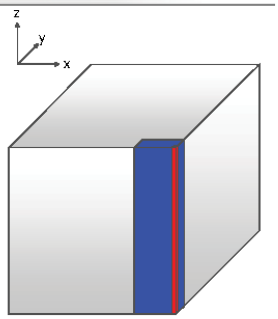


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on CUDA

FUJITSU

- 実装: asis
  - 共有メモリは未使用
- 1スレッド(赤)
  - 担当量: (1,1,128)
- 1スレッドブロック(青)
  - 256スレッド
  - 担当量: (32,8,128)



Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on CUDA: カーネル例

FUJITSU

```
__global__ void gpu_calc_Ex( float *Ex, float *CEx, float *DEx,
                             float *Hx, float *Hy, float *Hz )
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int z;
  int xs = 0;
  int ys = 1;
  int zs = 1;
  int xe = NUM_X-2;
  int ye = NUM_Y-2;
  int ze = NUM_Z-2;
  int flag = 0;
  if ( (x >= xs) && (x <= xe) && (y >= ys) && (y <= ye) ) {
    flag = 1;
  }
  for ( z = zs; z <= ze; z++ ) {
    if ( flag ) {
      EX(z,y,x) = CEX(z,y,x) * EX(z,y,x) +
                  DEX(z,y,x) * (HZ(z,y,x) - HZ(z,y-1,x) - HY(z,y,x) + HY(z-1,y,x));
    }
  }
}
```

スレッドID(x,y)から担当箇所を決定

担当箇所は更新対象?

Z軸方向の全要素を順番に更新

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTDカーネル性能

FUJITSU

- Core-i7 2.66GHz 1コア　79.14 秒
- nVidia GTX285
  - CUDA　3.52 秒

CPU(1コア)と比べて20倍以上高速

Copyright 2010 FUJITSU LABORATORIES LTD.

FUJITSU

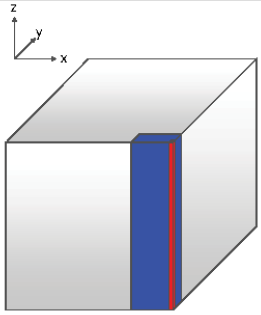
■ nVidia OpenCL

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on OpenCL

FUJITSU

- 実装方針:
  - CUDAと同じ
  - 共有メモリは未使用
- work-item(赤)
  - 担当量: (1,1,128)
- work-group(青)
  - 256 work-item
  - 担当量: (32,8,128)



Copyright 2010 FUJITSU LABORATORIES LTD.

カーネル実行方法

FUJITSU

- CUDA
  - サブルーチン呼び出しと類似

```
dim3 grid(…);
dim3 thread(…);
gpu_kernel<<< grid, thread >>>( … );
```

- OpenCL
  - キューへの投入

```
clEnqueueNDRangeKernel( queue, gpu_kernel, … );
```


Copyright 2010 FUJITSU LABORATORIES LTD.

OpenCLカーネルを実行するには

FUJITSU

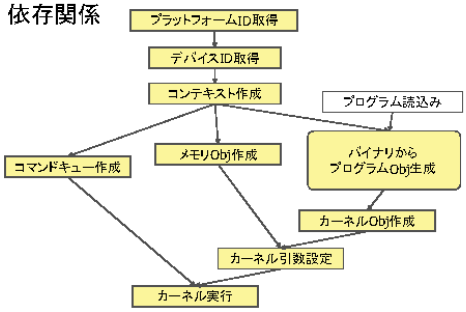
- カーネル実行　　dEnqueueNDRangeKernel()

Copyright 2010 FUJITSU LABORATORIES LTD.

OpenCLカーネルを実行するには

FUJITSU

- 依存関係



Copyright 2010 FUJITSU LABORATORIES LTD.

OpenCLでのプログラミング

FUJITSU

- カーネルルーチンのデバッグが大変
  - コンパイルエラー時、メッセージ無し
  - 開発環境の問題、OpenCLの問題ではない?

```
__kernel void hello(__global char* s)
{
  s[0] = "H";
  s[1] = "e";
  s[2] = "l";
  s[3] = "l";
  s[4] = "o";
  s[5] = "!";
  s[6] = "¥0";
}
```

コンパイルエラー

```
__kernel void hello(__global char* s)
{
  s[0] = 'H';
  s[1] = 'e';
  s[2] = 'l';
  s[3] = 'l';
  s[4] = 'o';
  s[5] = '!';
  s[6] = '¥0';
}
```


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on OpenCL: カーネル例

```
__kernel void gpu_calc_Ex( __global float *Ex, __global float *CEx, __global float *DEx,
                           __global float *Hx, __global float *Hy, __global float *Hz )
{
  int x = get_global_id(0);
  int y = get_global_id(1);
  int z;
  int xs = 0;
  int ys = 1;
  int zs = 1;
  int xe = NUM_X-2;
  int ye = NUM_Y-2;
  int ze = NUM_Z-2;
  int flag = 0;
  if ( (x >= xs) && (x <= xe) && (y
    flag = 1;
  }
  if ( flag ) {
    for ( z = zs; z <= ze; z++ ) {
      EX(z,y,x) = CEX(z,y,x) * EX(z,y,x) +
                  DEX(z,y,x) * (HZ(z,y,x) - HZ(z,y-1,x) - HY(z,y,x) + HY(z-1,y,x));
    }
  }
}
```

If文とfor文の順番を入替え。
入替えないと、配列値が更新されず。
仕様的には元順序で問題ないはず…。
共有メモリ使用時に問題。

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTDカーネル性能

- Core-i7 2.66GHz 1コア　79.14 秒
- nVidia GTX285
  - CUDA　3.52 秒
  - OpenCL　6.36 秒

CUDA生成オブジェクトとOpenCLオブジェクト、ptxレベルで違い
OpenCLでコンパイルしたオブジェクト、最適化されていない

荒井,佐藤,滝沢,小林 (第124回HPC研究会)
OpenCLによるGPUコンピューティングの性能評価

コンパイル時の最適化オプション → 効果無し
clBuildProgram()の第4引数
-cl-mad-enable, -cl-fast-relaxed-math, ...

Copyright 2010 FUJITSU LABORATORIES LTD.

- ATI OpenCL

Copyright 2010 FUJITSU LABORATORIES LTD.

ポータビリティ

- nVidia OpenCL環境で開発したプログラム
  - ATI OpenCL環境で動作
    - Makefileを修正してrebuild
  - ポータビリティはある(ようだ)
- 要因:
  - SDK付属のサンプルプログラムを使わなかった
    - サンプルプログラム、Utilityライブラリ使用 (nVidia/ATI)
    - 便利だが、ポータビリティ優先なら使わない方が無難

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTDカーネル性能

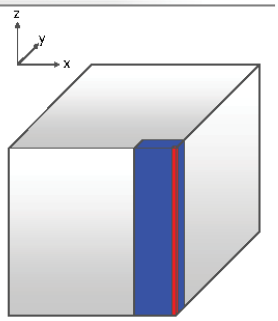
- Core-i7 2.66GHz 1コア　79.14 秒
- nVidia GTX285
  - CUDA　3.52 秒
  - OpenCL　6.36 秒
- ATI Radeon HD 5870
  - OpenCL　843.3 秒

何かの間違い?
nVidia向けカーネルでは駄目?

Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on OpenCL

- work-item(赤)
  - 担当量: (1,1,128)
- work-group(青)
  - 担当量: (32,8,128)
  - work-item数: 256

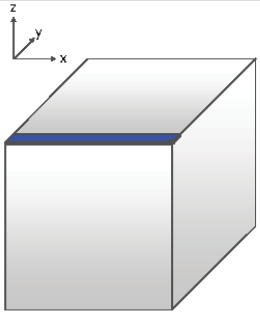


Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on ATI OpenCL

FUJITSU

- ストリーム化
- work-item(赤)
  - 担当量: (1,1,1)
  - 1要素/1work-item
- work-group(青)
  - 担当量: (128,8,1)
  - work-item数: 1024



30 Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTDカーネル性能

FUJITSU

- Core-i7 2.66GHz 1コア　79.14 秒
- nVidia GTX285
  - CUDA　3.52 秒
  - OpenCL　6.36 秒
- ATI Radeon HD 5870
  - OpenCL　843.3 秒
  - (1要素／スレッド)　87.92 秒

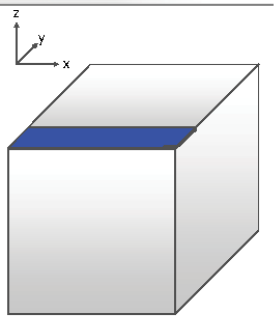
かなり改善、でもCPUより遅い

31 Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTD on ATI OpenCL

FUJITSU

- ベクトル化
  - float -> float4
- work-item(赤)
  - 担当量: (4,1,1)
  - 4要素/1work-item
- work-group(青)
  - 担当量: (128,32,1)
  - work-item数: 1024



32 Copyright 2010 FUJITSU LABORATORIES LTD.

3D-FDTDカーネル性能

FUJITSU

- Core-i7 2.66GHz 1コア　79.14 秒
- nVidia GTX285
  - CUDA　3.52 秒
  - OpenCL　6.36 秒
- ATI Radeon HD 5870
  - OpenCL　843.3 秒
  - (1要素／スレッド)　87.92 秒
  - (ベクトル化:float4)　47.18 秒

33 Copyright 2010 FUJITSU LABORATORIES LTD.

OpenCLはポータブルか

FUJITSU

- 動作
  - 同一プログラム: nVidia GPUとATI GPUでOK
- 性能
  - 同一カーネル: 極端な性能差
    - nVidia向カーネル: nVidia 6秒強、ATI 800秒以上
  - 各GPUの内部特性に見合うカーネルが必要
    - 手間は減らない?

34 Copyright 2010 FUJITSU LABORATORIES LTD.

まとめ

FUJITSU

- CUDAとOpenCLの相違、OpenCL間の相違を調査
  - 事例: 3D-FDTD
- CUDAとOpenCLの相違
  - カーネルコード、ほぼ同じ記述が可能
  - カーネル呼び出し部分、とても煩雑
  - 性能: CUDA > OpenCL
- OpenCL間の相違
  - 動作するが、性能は全然ポータブルではない
  - GPU毎にカーネルチューニング必要
- CUDAの良さを痛感

35 Copyright 2010 FUJITSU LABORATORIES LTD.

[添付資料 9]

FUJITSU

SS研 アクセラレータ技術WG

nVidia新GPUの調査報告
(FermiのCache評価)

2010年7月14日
(株)富士通研究所
成瀬 彰

shaping tomorrow with you

Copyright 2010 FUJITSU LABORATORIES LTD

はじめに

FUJITSU

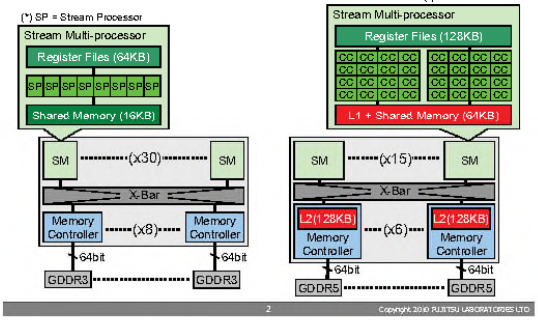
- Fermiの特徴
  - 倍精度演算 … 性能UP
    - DP/SP: 1/12 → 1/2
  - ECCメモリ … 信頼性UP
    - 未サポート → サポート
  - Cache … 使い易さUP
    - Read Only(用途限定) → Read Write(高い汎用性)
- 新旧GPU比較
  - GTX285(旧) vs. GTX480(新,Fermi)
  - 事例: himenoBMT

Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU:アーキ比較

FUJITSU

- GTX285
- GTX480



Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU:メモリlatency比較

FUJITSU

- ランダムアクセス時のメモリlatencyを測定

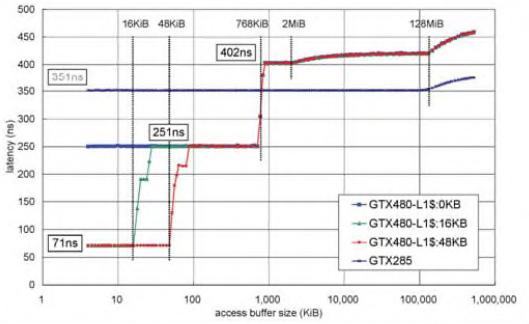
```
int index = (スレッドID);
int num = (アクセス回数);
while ( num > 0 ) {
    index = buf[ index ];
    num--;
}
```

- スレッド数:32 (スレッドブロック数:1, ワープ数:1)
- バッファサイズ: 4KiB ～ 1GiB
- Coalesced Memory Access

Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU:メモリlatency比較

FUJITSU



Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU:メモリバンド幅比較

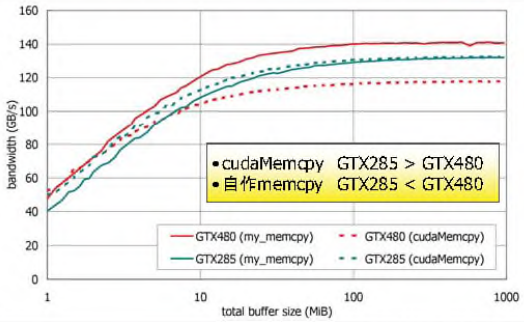
FUJITSU

- ピークバンド幅、ほぼ同じ (GTX480がやや上)
  - データ転送レート: 増 (2.484GHz → 3.696 GHz)
  - Bit幅: 減 (512bit → 384bit)

	GTX285	GTX480
スペック		
メモリタイプ	GDDR3	GDDR5
データ転送レート(GHz)	2.484	3.696
ビット幅	512 bit	384 bit
ピークバンド幅(GB/s)	159.0	177.4

Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU: 実効メモリバンド幅比較



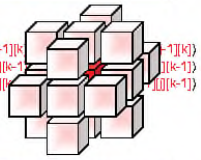
himenoBMT

- himenoBMT概要
  - 流体アプリのカーネルルーチン
    - Poisson方程式解法時の性能を測定
  - メモリアクセス特性
    - 14個の3次元配列
      - 13配列は再利用性なし
      - 1配列は再利用性あり → 共有メモリ・キャッシュが重要
    - メモリバンド幅ネック
  - サイズMを使用 (129x129x257)

himenoBMTのコア部分(jacobi)

```
for (i=1; i<imax-1; i++)
  for (j=1; j<jmax-1; j++)
    for (k=1; k<kmax-1; k++) {
      s0 = a0[i][j][k] * p[i+1][j][k]
         + a1[i][j][k] * p[i][j+1][k]
         + a2[i][j][k] * p[i][j][k+1]
         + b0[i][j][k] * (p[i+1][j+1][k] – p[i+1][j-1][k
         + b1[i][j][k] * (p[i][j+1][k+1] – p[i][j+1][k-1
         + b2[i][j][k] * (p[i+1][j][k+1] – p[i+1][j][k-1
         + c0[i][j][k] * p[i-1][j][k]
         + c1[i][j][k] * p[i][j-1][k]
         + c2[i][j][k] * p[i][j][k-1]
         + wrk1[i][j][k];
      ss = (s0 * a3[i][j][k] – p[i][j][k]) * bnd
      wrk2[i][j][k] = p[i][j][k] + omega * ss;
    }
```

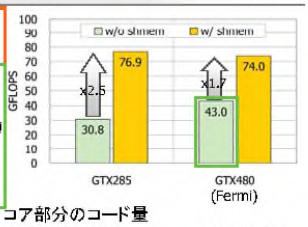
- 13配列: dot access
  - 再利用性無し



- 配列p: 19p-stencil
  - 再利用性有り
  - 共有メモリ利用

新旧GPU: himenoBMT性能比較

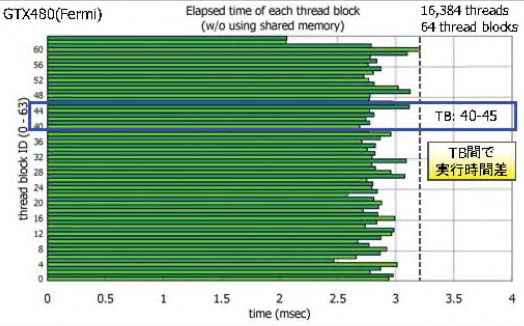
- 共有メモリ使用(*)
  - 高性能、複雑なコード
- 共有メモリ未使用
  - 簡潔なコード、低性能
  - Fermiの性能はそれなり
    - Cacheの効果
  - でも、共有メモリ使用時より性能低い、何故?



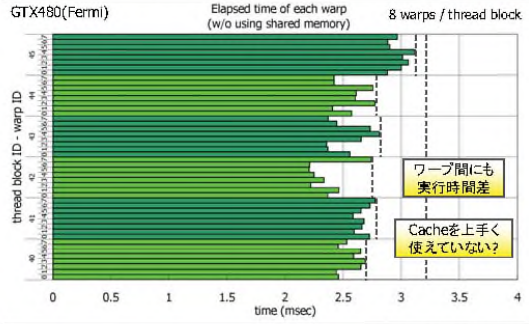
- コア部分のコード量
  - オリジナル(C言語) 44行 (1.00)
  - CUDA 69行 (1.57)
  - CUDA(共有メモリ) 232行 (5.27)

(*) 成瀬ら: GPGPU上での流体アプリケーションの高速化手法～1GPUで姫野ベンチマーク60GFLOPS超～、情報処理学会HPC研究会(第117回)

実行時間解析(全体)



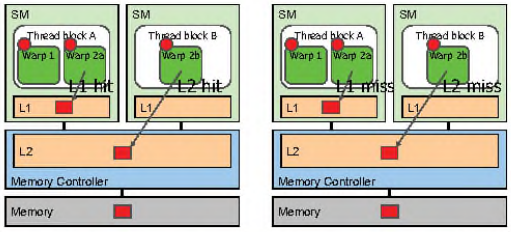
実行時間解析(一部)



ワープ進行差とCacheの関係

FUJITSU

■ ワープ進行差:小　■ ワープ進行差:大

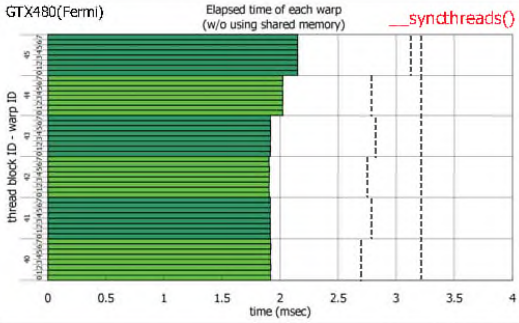


Cacheを有効活用するには、ワープ進行差を広げないことが重要

Copyright 2010 FUJITSU LABORATORIES LTD

実行時間解析(一部)

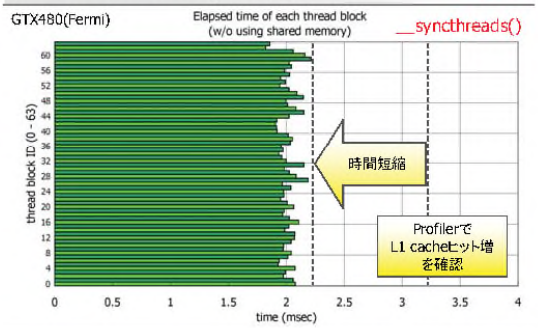
FUJITSU



Copyright 2010 FUJITSU LABORATORIES LTD

実行時間解析(全体)

FUJITSU

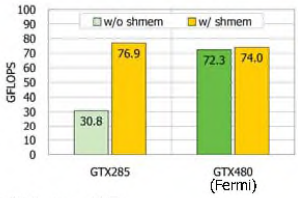


Copyright 2010 FUJITSU LABORATORIES LTD

新旧GPU: himenoBMT性能比較

FUJITSU

- Cacheチューニング
  - __syncthreads() +α
  - Cache活用で高性能
  - コード量は微増
    - 増分は5行!
  - 簡潔なコード、高性能

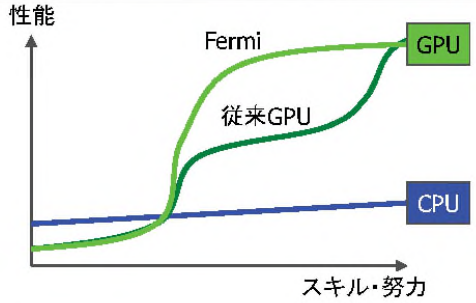


- コア部分のコード量
  - オリジナル(C言語)　44行 (1.00)
  - CUDA　69行 (1.57)
  - CUDA(cache tuning)　74行 (1.68)
  - CUDA(共有メモリ)　232行 (5.27)

Copyright 2010 FUJITSU LABORATORIES LTD

スキル・努力と性能の関係

FUJITSU



Copyright 2010 FUJITSU LABORATORIES LTD

まとめ

FUJITSU

- FermiのCache評価
  - himenoBMTで確認
  - 従来、高性能実現には共有メモリ使用が必須
    - 複雑なコード (メンテナンス困難)
  - Fermi、Cache使用で高い性能
    - 簡潔なコード (メンテナンス容易)
- Fermiはより「GP」GPUになった
  - 従来より簡単に高性能プログラミングが可能
  - 最低限のCacheチューニングは必要

Copyright 2010 FUJITSU LABORATORIES LTD

[添付資料 10]

PGI コンパイラでの自動並列化制限 ならびに CUDA によるアプリケーション性能評価

2010/07/14
九州先端科学技術研究所
本田宏明

PGI Accelerator Workstation

- ディレクティブにより，CPUならびにGPU 向けオブジェクトコードが同一ソースコードから生成可能
  #pragma acc region, !$acc region, …
- 10/07/13 現在: version 10.6
  - CUDA 3.0 までのサポート
  - Nvidia GPU のサポート

概要

1. 開発環境，測定環境
2. 行列積
3. PGI コンパイラプログラミングモデル制限
4. 3D-FDTD (成瀬のプログラム 10/04/07)
   1. Nvidia CUDA 3.0 環境における OpenCL
   2. PGI での性能評価
5. ERI (DRK法, s 軌道限定版)-MO
   1. 異なったアルゴリズムによる ERI-MO (RIKEN-BMT 2009) 計算 (限定版)
      - 原子 s 軌道に関わる計算のみ可能

1. 開発，測定環境

- 開発環境
  - OpenSUSE 11.2 (64bit)
  - CUDA 3.0 (195.36.15)
  - OpenCL (CUDA 3.0)
  - GCC 4.1.2
  - PGI Accelerator Workstation 10.6
- 測定環境
  - HOST: Xeon E5520 2.27GHz, DDR3-1333
  - GPU: Tesla C1060

2. 行列積の計算

オリジナルコード

```
for ( i = 0 ; i < in ; i++ ) {
    for ( j = 0 ; j < N ; j++ ) {
        c [ i ][ j ] = 0 ;
    }
    for ( k = 0 ; k < N ; k++ ) {
        for ( j = 0 ; j < N ; j++ ) {
            c [ i ][ j ] += a [ i ][ k ] * b [ k ][ j ] ;
        }
    }
}
```

実行時間比較

オリジナル:	457ms
GPU 使用:	23ms

#pragma, {} の挿入のみ

PGI 用コード

```
#pragma acc region
{
    for ( i = 0 ; i < in ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
            c [ i ][ j ] = 0 ;
        }
        for ( k = 0 ; k < N ; k++ ) {
            for ( j = 0 ; j < N ; j++ ) {
                c [ i ][ j ] += a [ i ][ k ] * b
            }
        }
    }
}
```

カーネルコード部へ

コンパイル時情報

```
pgcc -fast -Minfo=all -ta=nvidia -o matmul_t.o
main:
     35, Generated an alternate loop for the loop
         Generated vector sse code for the loop
     46, Generating copyout(c[0:1048575])
         Generating copyin(b[0:1048575])
         Generating copyin(a[0:1048575])
         Generating compute capability 1.0 binary
         Generating compute capability 1.3 binary
     48, Loop carried dependence of 'c' prevents parallelis
         Loop carried backward dependence of 'c' prevents v
         Sequential loop scheduled on host
         Loop not vectorized/parallelized: contains call
     49, Loop is parallelizable
         Accelerator kernel generated
         49, #pragma acc for parallel, vector(256)
             CC 1.0 : 3 registers; 24 shared, 44 constant,
             CC 1.3 : 3 registers; 24 shared, 44 constant,
     52, Loop is parallelizable
         Accelerator kernel generated
         52, #pragma acc for parallel, vector(256)
```

3. GPU オフロードのための PGI プログラミングモデルの制限[1)]

- 多重ループでは rectangular 形式
- ループの中において配列アクセスに使用されるポインタには C99 'restrict' 属性を指定
- ループ間ではデータ並列（synchronization も実行不可）
- ループ内でのポインタ演算は不可
- （現実装）ループ内での計算による配列添字使用不可
- （現実装）ループ内関数呼び出し不可
- （現実装）構造体への計算は可能だがネストされた構造体への演算は不可

1) "PGI Fortran & C Accelerator Programming Model", Ver.1.2 March 2010

制限例（1／2）

- 多重ループでは rectangular 形式

```
#pragma acc region
{
for ( i = 0; i < N ; i++ ) {
    for ( j = 0 ; j < i ; j++ ) {
        <loop_body>
```

→ オフロード不可

```
#pragma acc region
{
for ( i = 0; i < N ; i++ ) {
    for ( j = 0 ; j < N ; j++ ) {
        <loop_body>
```

→ GPU へオフロード可能
j ループ以下を GPU で

ループの中において配列アクセスに使用されるポインタには C99 'restrict' 属性を指定

```
float *A = (float *) malloc ( size ) ;
float *B = (float *) malloc ( size ) ;
```

→ オフロード不可

```
float *restrict A = (float *) malloc ( size ) ;
float *restrict B = (float *) malloc ( size ) ;
```

→ A[], B[] を使用する箇所はオフロード可能
（他の制限も満足する必要あり）

制限例（2／2）

- ループ間ではデータ並列

```
#pragma acc region
{
for ( i = 0; i < N ; i++ ) {
    for ( j = 0 ; j < N ; j++ )
        C [ i ][ j ] = 0 ;
    }
    for ( k = 0 ; k < N ; k++ ) {   /// 並列化不可 !!!
        for ( j = 0 ; j < N ; j++ ) {   /// 並列化可能 !!!
            C [ i ][ j ] += a [ i ][ k ] * b [ k ][ j ]
        }
    }
}
}
```

GPU カーネルへオフロード可能

4. 3D FDTD 性能評価
CUDA 3.0 環境での OpenCL

		Naruse[1]	Present[2]
CPU (sec.)		79.14	47.12
GPU (sec.)	CUDA	3.52	5.57
	OpenCL	6.36	5.70

大きな差は無い

1) GTX285,　CUDA 2.3 OpenCL, 10/04/07 SS 研発表資料より
2) Tesla C1060, CUDA 3.0 OpenCL

OpenCL における PTX レベルでの最適化が進んだ？

3D FDTD 性能評価
~PGI コンパイラ使用~

成瀬 GPU コード

```
CPU->GPU データ転送
loop: 計算繰り返し
  gpu_calc_Ex<<grid, thread>>()
  …
  gpu_calc_Hz<<grid, thread>>()
```

CPU ⇔ GPU：
一度だけデータ転送，
その後は GPU 計算のみ

gpu_calc_* 関数内部での展開

PGI による実装

```
#pragma acc data region copy (…)
{
loop:計算繰り返し
#pragma acc region local (…)
 {
  for (z)
    for (y)
      for (x)
        Ex (z,y,x) = …
 }
 …
```

3D FDTD 性能評価結果

	Present
CPU (sec.)	47.12
GPU (sec.)	505.87

大きすぎる実行時間
GPU 計算のたびに
CPU -> GPU データ転送を行っている？
データ転送用の PGI ディレクティブの
適切な使用が出来ていない？

5. 分子軌道法
~理研ベンチ ERI-MO~

- （非経験的）分子軌道法
  - 電子が分子の周りをどのように運動し、どのような エネルギーを持っているのかを計算により求める

※経験的パラメータを内包せず，計算規模に対応して精度が向上　※膨大な計算時間が必要

- 応用例
  - 分子物性の解析
  - タンパク質の構造解析
  - →創薬、新素材（ナノ材料等）の開発

より大きな系での精度のよい計算が求められている

分子軌道計算のボトルネック

- 各部分の実行時間（単位：秒）

入力	sw01-sto (3)	sw02-sto (9)	sw03-sto (24)	sw-04sto (42)
入力処理	0.04925	0.047703	0.05451	0.049123
H行列計算	0.00017	0.00144	0.021473	0.109347
初期D行列設定	0.000563	0.000667	0.002497	0.00972
G行列計算	0.185947	2.820427	172.8649	1590.996
F行列計算	[illegible]	0.00008	0.000627	0.00194
行列対角化	[illegible]	[illegible]	[illegible]	0.106237
D	[illegible]	[illegible]	[illegible]	0.04624
収	[illegible]	[illegible]	[illegible]	0.001013
展	[illegible]	[illegible]	[illegible]	0.00057
合計時間	[illegible]	[illegible]	[illegible]	1591.381

G行列の計算が
実行時間の大部分を占める！

2010/6/29　Xeon E5520 @ 2.27GHz Memory size 12GB　12

ERI(DRK)-MO計算のコード

$$G_{ij} = \sum_{k}^{N}\sum_{l}^{N} D_{kl}[2\langle\varphi_i\varphi_j|\varphi_k\varphi_l\rangle - \langle\varphi_i\varphi_k|\varphi_j\varphi_l\rangle]$$

縮約 ERI 計算

G 行列計算

```
for i=1,N
  for j=1,i
    for k=1,i
      if k==i
      for l=1,j, else for l=1,k
        (ij,kl)における2電子積分(ERI) の計算
        部分G行列の計算
      end loop
    end loop
  end loop
end loop
```

どちらもイタレーション間に依存関係がない

(縮約) ERI計算

```
for a=1,M_i
  for b=1,M_j
    for c=1,M_k
      for d=1,M_l
        初期積分計算
        漸化式計算
      end loop
    end loop
  end loop
end loop
```

原始ERI計算

今回はココを GPU 化

多段階の並列化可能性:

G 行列全体データ転送量 ：～100MB

縮約 ERI 積算データ転送量：～数TB

本来は G 行列計算を GPU 化する必要あり

DRK 積分法

$$(ij|kl) = \sum_{m=0}^{L} C_m F_m(X)$$

$$F_m(X) = \int_0^1 t^{2m}\exp(-Xt^2)dt$$

$$X = \rho(r_A - r_B)^2$$
$$r_A = (\alpha_i r_i + \alpha_j r_j)/A$$
$$r_B = (\alpha_k r_k + \alpha_l r_l)/B$$

$$\rho = AB/(A+B)$$
$$A = \alpha_i + \alpha_j$$
$$B = \alpha_k + \alpha_l$$
$$L = L_a + L_b + L_c + L_d$$

$$(ij|kl) = \int_0^1 \exp(-Xt^2)P_L(t)dt$$

P_Lは、Cmを係数とした区間[0,1] の L 次の直交多項式

$$(ij|kl) = \sum_{\omega=1}^{N} W_\omega P_L(t_\omega)$$

Gauss 型区分求積法による計算

理研ベンチでは $F_m(X)$ をテイラー展開により求値（今回のコードとの違いは上記値の評価法）

ERI(DRK)-MO 性能評価

		G行列の実行時間[1] (sec.)
CPU[2]		1.96
GPU	CUDA[3]	57.8
	PGI[4]	0.71

実行時間の増加 ＝データ転送のオーバヘッド

1. プログラム内に EXP, SQRT, 2. 大量のポインタ演算（とても効率できず）＝オフロードが不可（1, 2, 両方について コンパイル時メッセージが表示）

1) H_2 分子 22 量体, STO-3G 基底, (s 軌道のみの計算)
2) GCC4.4.1 での計算
3) 原始 ERI 計算のみ GPU 化
4) PGI では, GPU へのオフロードはされていない

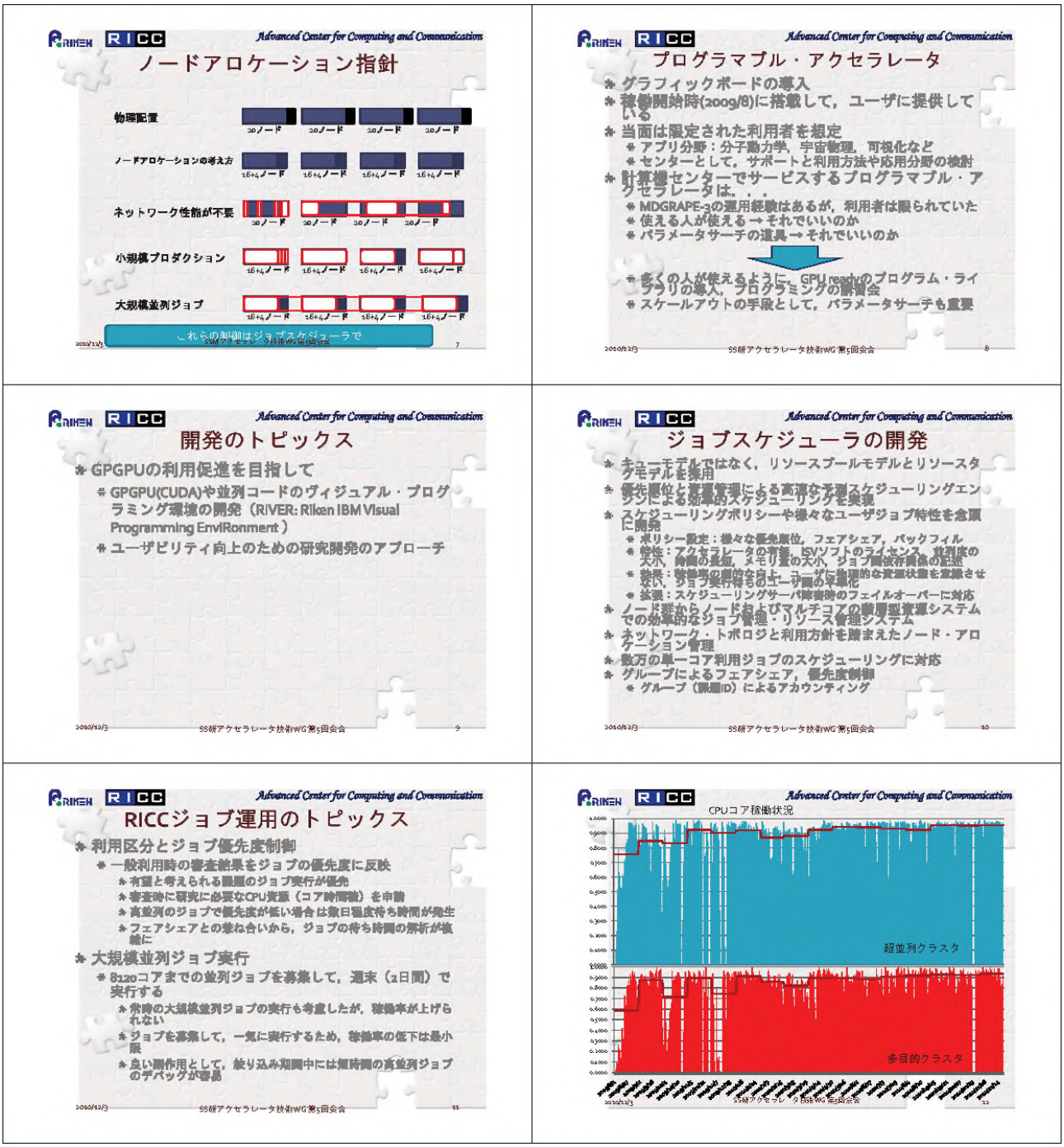
更に大きな計算ブロックでの GPU 計算が必要

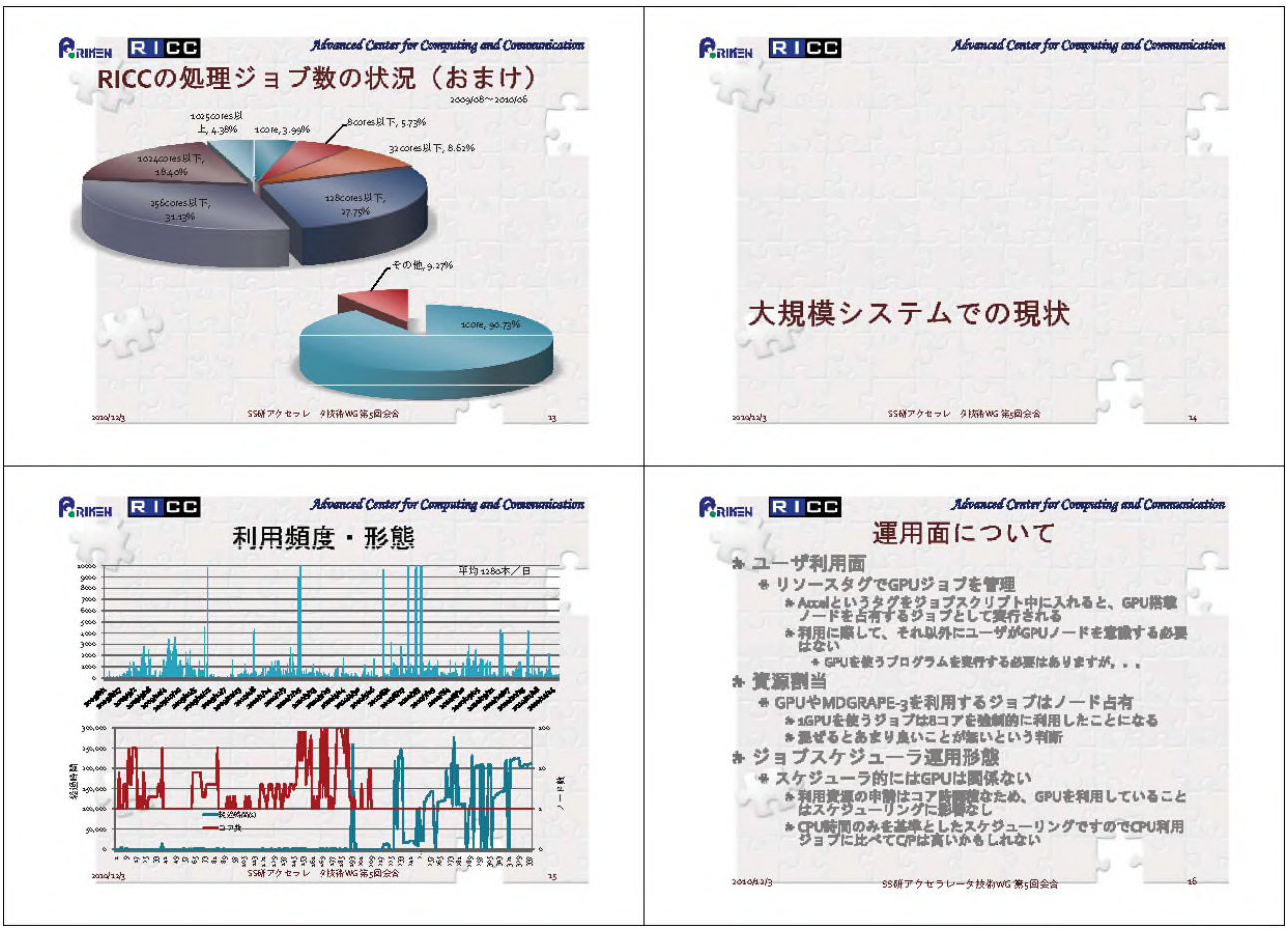
まとめ，今後の予定

- 行列積，3D-FDTD, ERI(DRK)-MO の計算を例として PGI コンパイラを利用したGPGPU プログラミングを試みた.
  - 行列積計算については容易に加速が可能
  - 3D-FDTD 計算については適切なデータ転送方法を選択できず実行時間の低下
  - ERI(DRK)-MO については最内ループボディについてのみGPUへオフロード，より上位のコード部分については変更点が多すぎ，現在コーディング中
- 今後の予定：
  - 3D-FDTD, ERI(DRK)-MO 計算について，適切なコード生成やPGIコンパイラの使用，加速の試み

[添付資料 11]

RIKEN RICC
Advanced Center for Computing and Communication
アクセラレータを使った大規模システム
導入にあたっての課題
"RICCの場合"
黒川 原佳
情報基盤センター
独立行政法人 理化学研究所
本日の目次
RICCシステムの構成について
システム構成
運用ソフトウェア構成
稼働状況
大規模システムでの現状
利用頻度
形態（単一ノード or マルチノード）
運用面について
ジョブスケジューラ運用形態や資源割当方法など
RICCシステム構成について
RICCシステム構成
【システム構成】2009年8月から
PCクラスタ＋大容量メモリ計算機＋アクセラレータ
実験データ
【超並列PCクラスタ】 1040Nodes
97.5TFLOPS, 12TB(mem), 435TB(hdd)
12GB/Node, DDR IB×1/Node
【多目的PCクラスタ】 100Nodes
9.3TFLOPS, 2.3TB(mem), 25TB(hdd)
24GB/Node, DDR IB×1/Node
PCI-ex16レーン×1
アクセラレータ：
Tesla C1060×100
Ethernet, IB
アーカイブ装置4PB,
HPSS, 10GbE
共有ディスク装置
550TB, SRFS, DDR IB
【大容量メモリ計算機】1Node
0.24TFLOPS, 512GB(mem)
PCI-E, 10GbE
【分子動力学専用計算機】
64TFLOPS
ホストノード:32Nodes
32GB/Node, DDR IB×1/Node
アクセラレータ：
MDGRAPE-3 16ユニット
2.1TB/s
400GB/s
IB DDR
10GbE
GbE
外部ネット
ネットワーク
26GB/s
264GB/s
フロントエンド・システム
大容量メモリ
InfiniBandネットワーク構成
FBB構成よりもLeaf 2/3,
Spine 1/5の構成
ファイルサーバ
IBスイッチ（Spine） 144ポート×2
20Nodes 20Nodes 20Nodes 20Nodes 20Nodes 20Nodes 20Nodes 20Nodes 12Nodes FE
超並列PCクラスタ 1040ノード
多目的PCクラスタ 132ノード
IBスイッチ（Leaf）
24ポート×59


RIKEN RICC Advanced Center for Computing and Communication
ノードアロケーション指針
物理配置
ノードアロケーションの考え方
ネットワーク性能が不要
小規模プロダクション
大規模並列ジョブ
これらの制御はジョブスケジューラで
プログラマブル・アクセラレータ
GPGPUの利用促進を目指して
GPGPU(CUDA)や並列コードのヴィジュアル・プログラミング環境の開発（RIVER: Riken IBM Visual Programming EnviRonment）
ユーザビリティ向上のための研究開発のアプローチ
ジョブスケジューラの開発
RICCジョブ運用のトピックス
利用区分とジョブ優先度制御
大規模並列ジョブ実行
CPUコア稼働状況
超並列クラスタ
多目的クラスタ

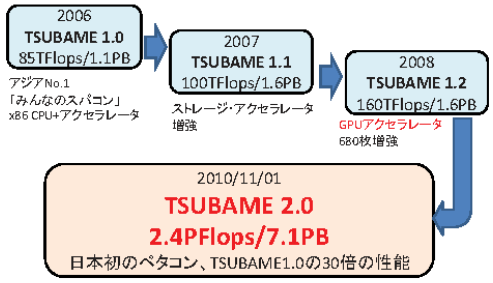

RICCの処理ジョブ数の状況（おまけ）
2009/08～2010/06
1025cores以上, 4.38%
1core, 3.99%
8cores以下, 5.73%
32cores以下, 8.62%
1024cores以下, 18.40%
256cores以下, 31.13%
128cores以下, 27.79%
その他, 9.27%
1core, 90.73%
大規模システムでの現状
利用頻度・形態
平均1280本／日
運用面について
ユーザ利用面
リソースタグでGPUジョブを管理
Accelというタグをジョブスクリプト中に入れると、GPU搭載ノードを占有するジョブとして実行される
利用に際して、それ以外にユーザがGPUノードを意識する必要はない
GPUを使うプログラムを実行する必要はありますが，．．
資源割当
GPUやMDGRAPE-3を利用するジョブはノード占有
1GPUを使うジョブは8コアを強制的に利用したことになる
混ぜるとあまり良いことが無いという判断
ジョブスケジューラ運用形態
スケジューラ的にはGPUは関係ない
利用資源の申請はコア時間単位なため、GPUを利用していることはスケジューリングに影響なし
CPU時間のみを基準としたスケジューリングですのでCPU利用ジョブに比べてGPUは貴いかもしれない

[添付資料 12]

TSUBAME 2.0におけるGPU運用 (とLinpack)

東京工業大学
情報理工学研究科
遠藤敏夫

TSUBAMEの歴史



- TSUBAME初の完全リプレース

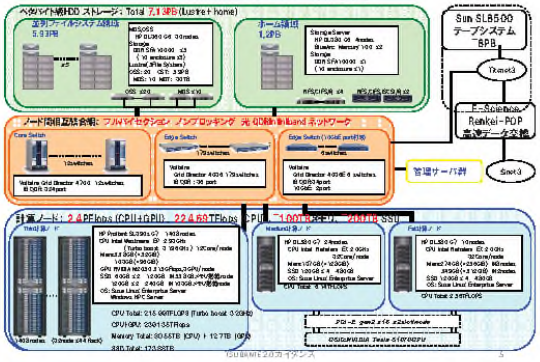
TSUBAME2.0の特徴(1)

- 理論値2.4PFlopsのばく大な演算性能
  - CPU合計性能: 220TFlops ・・・4.4倍 (TSUBAME1.2比)
  - GPU合計性能: 2.18PFlops ・・・20倍
- 合計容量7.1PByteの巨大ストレージ
  - T1.2の4.5倍の容量
- バイセクションバンド幅200Tb/sの高速光ネットワーク

TSUBAME2.0の特徴(2)

- ソフトウェア資産の継続性と新規運用
  - 既存のMPI, OpenMP, CUDAプログラムの利用
  - 既存ISVアプリの大部分の利用
  - SUSE Linux Enterprise 11 (T1.2はSUSE10)
  - 新たにWindows HPC Serverの運用 (今回は説明対象外)
- GPU対応アプリも採用
  - CPUよりも計算が短時間で済む⇒課金も少なくてすむ
  - 現在はAMBER、今後も続々対応予定

TSUBAME2.0 システム概念図



電力について

- GPUを含めたビジー時システム電力：約1400kW
  - ノード，電源，スイッチ含む
  - 空調，ストレージ含まず
  - Green500提出値は1257kW
- Thinノード
  - ビジー時950W，アイドル時300W
- 冷却装置
  - チラー：冬季280kW，夏季370kW (NEC予測)
  - MCSラック：80kW (42ラック分)
  - 他，一部ノードは空調使用

TSUBAME2 計算ノード (1)

- Thinノード，Mediumノード，Fatノードの三種類
- Thinノード：1408台 [一番良く使われる計算ノード]
  - HP Proliant SL390s G7
  - CPU: Intel Xeon 2.93GHz 6コア×2=12コア
    - Hyperthreadingのために24コアに見える
  - GPU: NVIDIA Tesla M2050 3GPU
  - Memory: 54GB (一部は96GB)
  - SSD: 120GB (一部は240GB)
  - ネットワーク：QDR InfiniBand x 2 = 80Gbps

CPU 140GF + GPU 1545GF = 1685GF

TSUBAME2 計算ノード(2)

- Medium/Fatノード：M24台 + F10台
  [大容量メモリが必要なジョブ向け]
  - HP Proliant DL580 G7
  - CPU: Intel Xeon 2.0GHz 8コア×4 = 32コア
    - Hyperthreadingのために64コアに見える
  - Memory: 128GB (Medium), 256/512GB(Fat)
  - SSD: 480GB
  - ネットワーク：QDR InfiniBand x 1 = 40Gbps
  - GPUとして、TSUBAME 1.2で使っていたTesla S1070を利用可能

新旧ノードの性能比較

TSUBAME 1.2ノードとTSUBAME 2.0 (Thin)ノードの性能比較

理論性能

	TSUBAME1.2	TSUBAME2.0
CPU	デュアルコアOpteron 2.4GHz×8－16コア 76.8GFlops(*1)	Westmere-EP2.93GHz×2－12コア[24スレッド] 152.4GFlops (*3)
RAM	32GB	54GB
IB	SDR×2 (20Gbps)	QDR×2 (80Gbps)
Disk	HDD	SSDRAID-0
GPU	S1070×2GPU 172GFlops(*2)	M2050 ×3 1545GFlops(*4)
メモリバンド幅	20GB/s以下 (*5)	64GB/s

*1　2.4(GHz) * 2 (FP) * 2 (Core) * 8(Socket) = 76.8GFlops
*2　86(GFlops) * 2 = 172GFlops
*3　2.196(GHz) * 4 (FP) * 6(Core) * 2(Socket) = 152.4GFlops
*4　515(GFlops) * 3 = 1545GFlops
*5　CPUの制約のためメモリ単体の各数値より劣る

アプリ性能・ベンチマーク性能参考値 (*6)

	TSUBAME1.2	TSUBAME2.0相当
SpecCINT2006	約 120～140	358.0
SpecCFP2006	(情報無し)	244.0
Gaussian (bzeysich)	197.227 [秒]	97.55 [秒]
Gaussian (cysi2 b)	396.550 [秒]	98.20 [秒]
Gaussian (cysi2c)	98.993 [秒]	32.60 [秒]
AMBER (hw)	82.732 [秒]	39.5 [秒]
AMBER (pyp)	1131.835 [秒]	811.1 [秒]

*6　実際のTSUBAME2.0ノードと異なる測定環境なので参考値

TSUBAME2.0のストレージ

- ホーム
  - NFS, CIFS, iSCSI
    - BlueArc Mercury 100 (一部GridScaler)
    - DDN SFA 10K × 1, SATA × 600 disks
- 並列ファイルシステム
  - Lustre
    - MDS : HP DL360 G6 × 6
    - OSS : HP DL360 G6 × 20
    - DDN SFA 10K × 3, 2TB SATA × 3550 disks, 600GB SAS × 50 disks
  - GPFSによる提供の予定あり

System Software

	TSUBAME 1.2	TSUBAME 2.0
Linux OS	SUSE Linux Enterprise Server 10 SP2	SUSE Linux Enterprise Server 11 SP1
Windows OS	-	Windows HPC Server 2008 R2
Job Scheduler for Linux	Sun N1 Grid Engine	PBS Professional
Job Scheduler for Windows	-	Windows HPC Server

- Windows OSを新規にサポート
- ジョブスケジューラが変更されたため、バッチジョブ投入オプションが変更

Compilers & Libraries

	TSUBAME 1.2	TSUBAME 2.0
Compiler	Intel Compiler 10 & 11 PGI CDK 6,7,8,9 & 10 gcc 4.1	Intel Compiler 11.1.072 (標準) PGI CDK 10.6 gcc 4.3.4
MPI	Voltaire MPI (標準) OpenMPI 1.2.6	OpenMPI 1.4.2 (標準) MVAPICH2 1.5.1
CUDA	2.3	3.1

- 標準のコンパイラ・バージョンが変わっているので、独自アプリはTSUBAME 2.0用に再コンパイルが必要
- コンパイラの切り替えは環境変数の設定で可能
- T1.2同様、CUDA C/FortranによるGPUプログラミング可能

TSUBAME 2.0ガイダンス 13

バッチキュー運用

- 基本的にはバッチキュー(PBSPro)を介して利用
- 複数の異なる性質のキュー
  - CPUむけ、GPUむけ…
- Gangliaなどによる利用率公開
  - tsubame.gsic.titech.ac.jpより
  - GPUジョブの統計は困難なためまだ

ノード占有系：Sキュー・Lキュー

- Sキュー：12CPUコア, 3GPU, 54GBメモリを持つノードを利用
  - 従来のSLAキューに相当
  - 多数CPUまたはGPUによる並列性や、I/O(ディスク・通信)性能が必要なジョブ向け
  - ノード内のジョブ混在は起こらない
  - 従量制課金
- 大容量メモリが必要なジョブには、S96, L128, L256, L512キュー
  - 数字はメモリ容量(GB)
  - Sに比べ1.5倍、2倍…の課金
  - L系はMeduim/Fatノードなので、CPUが多く、GPUが古い

TSUBAME 2.0ガイダンス 15

仮想マシン内共有系：Vキュー

- ノードあたり8CPUコアを利用
  - 従来のBESキューに近い
  - 逐次ジョブや比較的小規模なジョブ向け
  - KVM仮想マシン技術により、以下のようなノードに見える
    - 8CPUコア (hyperthreadingで16コアに見える)
    - 32GBメモリ
    - TSUBAME 1.2ノード相当、GPUは無し
  - I/O速度は他キューより下がるので注意

TSUBAME 2.0ガイダンス 16

GPU系：Gキュー

- ノードあたり3GPU+4CPUコアを利用
  - GPUジョブに適している
  - 以下のようなノードに見える
    - 4CPUコア
    - 3GPU
    - 22GBメモリ
  - Vキュージョブと仮想マシン技術によりノードを共有

TSUBAME 2.0ガイダンス 17

予約系：Hキュー

- 予約した期間ノードを占有して利用
  - 従来のHPCキューに相当
  - 1000CPUコアレベルの並列性が必要なジョブ向け
  - Webから日程・ノード数を予約
  - バッチキューを介さない利用も可
  - 従来よりも、柔軟な予約が可能
    - ノード数は16以上自由、期間は一日単位で最大7日

TSUBAME 2.0ガイダンス 18

主要サービス比較

TSUBAME1.2			TSUBAME2.0			
性能保証サービス(SLA) (sla1, sla2,ram64,ram128)	254台	S ノード占有系 S96, L128など	従量	300台	並列度・I/O速度重視 演算性能2倍,メモリバンド幅3倍(T1.2比)を占有 GPUジョブもOK
ベストエフォートサービス(BES)	234台	V 仮想マシン内共有系	定額	440台 (Linux) 40台 (Windows)	比較的小規模ジョブ向け T1.2に近い性能、ただしI/Oはやや弱め
予約制大規模サービス	98台	H 予約系	従量	420台	大規模並列向け 1日単位1ノード単位で予約が可能に
		G GPU系	従量	480台 (Vと共有)	GPUジョブ向け GPU+MPIもOK
		超大規模		700~1000台	超大規模ジョブ向け 審査制、年数回予定

※各キューへの配分ノード数は今後の利用状況に応じて調整します
※ノード占有Windowsなどは検討中です

TSUBAME2.0ガイダンス　19

ISVアプリケーションのGPU対応状況

Application	状況
PGI CDK	対応
AMBER	対応
Maple	対応
Total View Debugger	次期バージョンにて対応
ABAQUS Standard/Explicit	Nov. 2010 より対応予定

Application	状況
MD NASTRAN	Jun 2011 より対応予定
Gaussian	Q3 2011 より対応予定
Molpro	Q3 2011 より対応予定
Mathematica	Nov. 2010 より対応予定
ANSYS FLUENT	Nov. 2011 より対応予定

ISVアプリケーションでのGPU使用例: AMBER

- AMBERでは、pmemdがGPU対応

t2sub <グループ, キュー指定> ./job.sh

job.sh
#!/bin/sh
pmemd.cuda<pmemdと同等のオプション>

TSUBAME2.0ガイダンス　20

TSUBAME2.0上のLinpack

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT YH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
3	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
4	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61
5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00

1.192PFlops in Linpack, 世界4位!

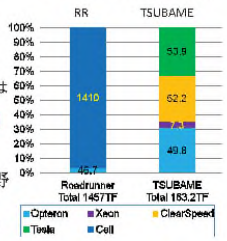
LinpackとHPL

- Linpack: 密行列連立一次方程式を直接法で解くベンチマーク --- スパコンのランキングTop500に利用
  - 行列サイズN, 計算量O(N^3)
  - 倍精度
- HPL (High-performance Linpack) by A. Petitet
  - 著名なMPI並列実装
  - ブロック化されたLU分解+部分ピボッティングに基づく
  - Top500ではNの選択は自由
    - メモリにおさまる範囲でNが大きいのが有利
    - TSUBAMEでは Nは約1,000,000
  - DGEMMカーネルの占める時間がほとんど

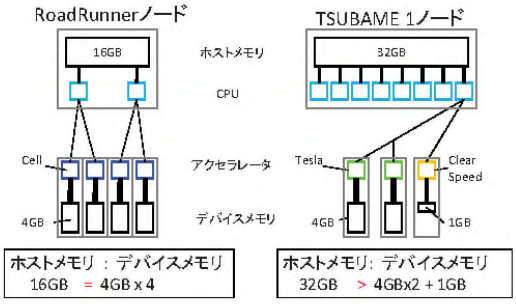
誰が計算する?

- カーネル計算
  - RoadRunnerでは, Cellが性能の96%を占める
    - CPUsは4%
  - ⇒ Cellのみ利用
  - TSUBAME1では, CPUが35%
    - いずれかのプロセッサ種類を省くのは性能低下が大きい
  - ⇒ CPU, GPU, ClearSpeedの全てを利用
  - TSUBAME2では、GPUが90%
  - ⇒ GPUを利用・ただしCPU併用も視野
- カーネル以外の処理
  - MPI通信、ピボッティングにはCPU利用

ピーク性能の内訳 (倍精度)



行列データをどこにおく? (1)



行列データをどこにおく?(2)

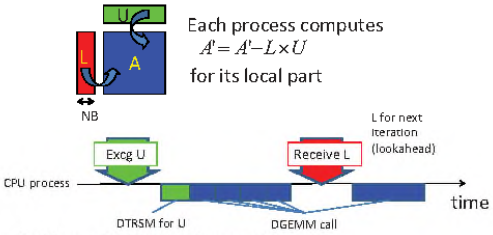
- Linpackの効率のためには行列サイズNが大きいのが望ましい
  - ⇒ ホストメモリサイズに近付けたい
- RRでは,
  - (1) デバイスメモリ容量 = ホストメモリ容量
  - (2) カーネル計算はCell上
  - ⇒ 行列データはCellデバイスメモリに
- TSUBAME1/2では,
  - デバイスメモリ容量 < ホストメモリ容量
  - ⇒ 行列データは通常、ホストメモリに

Overlapping Communication and Computation

- On GPGPU systems, overlapping communication is the key
  - Comm. overhead is *relatively* heavier than on x86 clusters
  - Both interconnect comm. (MPI) and PCIe comm. (cudaMemcpy) have to be considered

	Typical x86 clusters	TSUBAME 2.0
Peak FLOPS per node	100—300 GF	1685 GF
	x5--x15	
Interconnect BW per node	1 – 8 GB/s	8 GB/s
	x1—x8	
PCIe BW per node	--	8 GB/s x 3GPUs

An Iteration in Original HPL



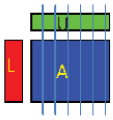
Each process computes
$A' = A' - L \times U$
for its local part

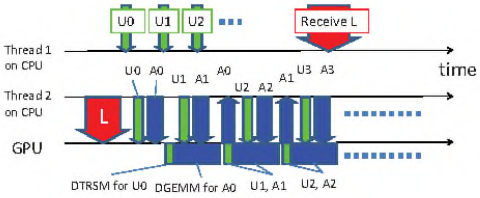
Problems of original implementations are:

- MPI comm and kernel are NOT overlapped
- DGEMM calls are fragmented

An Iteration of Our Implementation

- Kernels on GPU, MPI comm, PCIe comm are overlapped!
- Matrix A, U are split into A0, A1, A2... and U0, U1, U2...





TSUBAME2のLinpack速度

7:06 – 9:33, October 18

T/V	N	NB	P	Q	Time	Gflops
WR15R2R16	2490368	1024	69	69	9639.84	1.192e+06
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=					0.0003911	PASSED

- 運用前の10月中旬に測定
- OpenMPI 1.4.2, CUDA 3.1, gcc 4.3.4
- 1357nodes, 4071GPU利用により，日本初の1PF越え
- RoadRunner(1.04PF)以上、Nebulae(1.27PF)未満
  - 4600GPUのNebulaeは抜けず、無念

SC10におけるランキング発表

- 演算性能1.192PFlops ⇒ 世界4位!!
  - 日本のスパコンで10位以内は4年ぶり，5位以内は5年ぶり
- 省エネ性能958MFlops/W⇒世界2位!!
  - Greenest Production Supercomputer in the World賞獲得!!
- 計算世界観の一環として開発したヘテロ対応Linpackソフトウェアによる成果

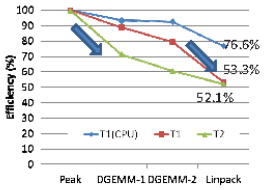
Analyzing Overhead (1)

	Peak (TFlops)	Linpack (TFlops)	Efficiency
RoadRunner	1376	1042	75.7%
TSUBAME 1.0 CPU-Only case	49.87	38.18	76.6%
TSUBAME 1.2	163.2	87.01	53.3%
TSUBAME 2.0	2288	1192	52.1%

- Why is the efficiency is lower?
  - DGEMM performance is important

Analyzing Overhead (2)

- DGEMM-1: 各GPU/CPU coreのDGEMM性能の合計
  - PCIe通信コスト含まない
- DGEMM-2: 各ノード単位の DGEMM性能合計
  - PCIe通信コスト・バス衝突含む



- On T2, most important factor is limitation of DGEMM speed on Fermi GPU
  - DGEMM: 360GF
  - Peak: 515GF
- On T1, MPI communication cost, node-heterogeneity heavily affect

[添付資料 13]

Hartree-Fock-Roothaan 量子計算プログラム概説

2011/03/02
九州大学 システム情報科学研究院
本田宏明

1

非経験的分子軌道法（*Ab initio* MO法）とは？

- 原子、分子における電子分布の様子（電子状態）を求めるための計算手法の1つ
  - 分子物性の解析
  - 創薬、新素材の開発
  - ex) プリンタのカラーインク、液晶ディスプレイ
- ミクロな物理現象を扱うために、量子力学（原子核と電子に対するSchrödinger方程式）に基づいて計算を行う
- 実験値などのパラメータは用いない
- 電子間相互作用の記述の程度や仮定する波動関数の形などにより様々な理論レベルが存在する

Ab initio MO法の解法の標準的な方法として、Hartree-Fock-Roothaan (HFR) 方程式を解く方法があり、Gaussian, GAMESS 等の計算パッケージにて使用されている。

2

HFR方程式の特徴

$$\mathbf{FC} = \mathbf{SC}\varepsilon$$

エネルギーに対応

MO 係数行列：波動関数に対応

Fock 行列：Hamiltonian に対応

S は方程式の直交化されていない基底に関する重なり行列

- Fock行列Fと重なり行列Sを入力とした一般化固有値問題である。
- 分子系と対象とした多体のSchrödinger方程式に対し、1粒子模型近似、Born-Oppenheimer 近似等を施すことにより得られる。
- 与えるべき Fock 行列 F の計算には、この方程式の解である MO 係数行列 C が必要であるため、この方程式は非線型方程式であり、反復解法（Self Consistent Field, SCF法）によって解かなければならない。（次頁参照）
- 計算のボトルネックは行列対角化処理部分ではなく、Fock 行列要素計算の1処理である、電子同士の反発エネルギーを計算する2電子ハミルトニアン行列（G 行列）計算部分である。（後述）

繰り返しによる HFR 固有値問題解法

$$\mathbf{F}[\mathbf{c}']\mathbf{c} = \mathbf{S}\mathbf{c}\varepsilon$$

初期値の入力

対角化

$$\mathbf{F}[\mathbf{c}_0]\mathbf{c}_1 = \mathbf{S}\mathbf{c}_1\mathbf{e}_1$$

$$\mathbf{F}[\mathbf{c}_1]\mathbf{c}_2 = \mathbf{S}\mathbf{c}_2\mathbf{e}_2$$

$$\mathbf{F}[\mathbf{c}_2]\mathbf{c}_3 = \mathbf{S}\mathbf{c}_3\mathbf{e}_3$$

$$\vdots$$

$$\mathbf{F}[\mathbf{c}']\mathbf{c} = \mathbf{S}\mathbf{c}\varepsilon \quad (\mathbf{c}' \approx \mathbf{c} \text{ まで繰り返し})$$

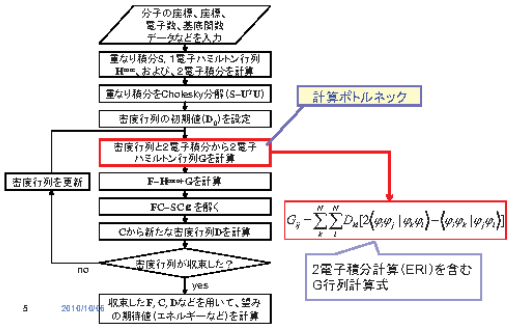
$$\sum_{l=1}^{N} F_{ij} C_{la} - \sum_{l=1}^{N} S_{ij} C_{la} \varepsilon_a$$

- Fは答えであるcを利用して求められる。
- 前回の c を使い、F を求め、対角化を行い新しい c を求める。
- 実際には c の代わりに c により求められる密度行列 D を F に作用させる。

繰り返しによる方法（SCF法）

4

分子軌道法プログラムの全体構造



$$G_{ij} = \sum_{k}^{N}\sum_{l}^{N} D_{kl}\left[2\{\varphi_i\varphi_j|\varphi_k\varphi_l\} - \{\varphi_i\varphi_k|\varphi_j\varphi_l\}\right]$$

5　2010/10/06

分子軌道計算のボトルネック

水分子多量体計算における各部分の実行時間（単位：秒）、

入力	sw01-sto(3 原子)	sw02-sto(9)	sw03-sto(24)	sw-04sto(42)
入力処理	0.04925	0.047703	0.05451	0.049123
H行列計算	0.00017	0.00144	0.021473	0.109347
初期D行列設定	0.000563	0.000667	0.002497	0.00972
G行列計算	0.185947	2.820427	172.8649	1590.996
F行列計算	0.00004	0.00008	0.000627	0.00194
行列対角化	0.00081	0.001697	0.02379	0.106237
D行列更新	0.00006	0.000387	0.008873	0.04624
収束判定	0.000613	0.000663	0.000883	0.001013
最終処理	0.00039	0.00039	0.000473	0.00057
合計時間	0.23835	2.874407	172.9903	1591.381

Xeon E5520 @ 2.27GHz Memory size 12GB

G 行列計算が実行の大部分を占める

6

G行列の計算の大変さ

$$G_{ij} = \sum_{k}^{N}\sum_{l}^{N} D_{kl}\left[2\left\langle \phi_i\phi_j, \phi_k\phi_l \right\rangle - \left\langle \phi_i\phi_k, \phi_j\phi_l \right\rangle\right]$$

巨大分子では
$N = 1000 \sim 10000$

密度行列

2電子積分(ERI)

$N = 5000$ → G, D ～10^8 要素計算
(～500原子分子) $\langle \phi_i\phi_j, \phi_k\phi_l \rangle$ ～10^{15} 要素計算

- G は大量の2電子積分の和により計算
  - i,j,k,l の4重ループ構造
- $O(N^4)$のため、小さな分子でも莫大な計算量に

7

G行列並びに ERI のコード

全体のG 行列計算

G 行列の初期化

(ss,ss) 型積分に関するG行列計算
(ps,ss) 型積分に関するG行列計算
(ps,ps) 型積分に関するG行列計算
(pp,ss) 型積分に関するG行列計算
(pp,ps) 型積分に関するG行列計算
(pp,pp) 型積分に関するG行列計算

後処理

この G 行列計算コードは '08 理研ベンチマーク ERI-MO とほぼ同等(簡易版)。
G行列計算では、ERI 計算のコストが最も大きいために ERI の重複計算が発生しないようループ制御を行う。1 つのERI 計算毎に ERI が寄与するG行列要素のみ更新してゆくアルゴリズムをとる。
(積分駆動型アルゴリズム)

各G 行列計算(積分駆動型アルゴリズム)

データテーブル作成
Loop: p_sp
Loop: q_sp
Loop: p_pp
Loop: q_pp
初期積分計算
垂直漸化式計算
End:
End:
水平漸化式計算
部分 G 行列計算
End:
End:

1組のERI 計算:
ERI は積分の型に応じて、1～81の要素を持つ

計算したERI が寄与する G 行列要素のみの更新

入力：G, D行列、各種係数
出力：G 行列

添付プログラムについて

- 解凍して make を行うことで、実行可能プログラムである rhf_seq_sort_time が出来ます。
- リンクの際には -llapack, -lblas が必要です。
  (準備をお願いします)
- Debian lenny (5.0), GCC 4.3.2 で開発しました。
- 倍精度計算をしています。
- 実行の際、入力ファイル：sw01-sto.inp ~ sw04-sto.inp、sw01-631.inp ~ sw03-631.inp をプログラム引数として渡します。
- 上の 631, sto の2種類の入力は、sw??- が同じであれば同じ分子なのですが、化学精度の異なる計算を行なります。具体的にはG 行列計算内部の多重ループ長が異なります。
- 答えの正否については、出力に現れる "TOTAL ENERY: " の値の確認のみで十分です。
  (次ページ参照)
- 実行時間測定については、1回のG行列計算部分のみでも結構です。

9

計算結果例

sw02-sto.out

```
...
RESULTS: *** CONVERGED ***
        ONE ELECTRON ENERGY =   -478.7829259482
        TWO ELECTRON ENERGY =    170.4454644483
 NUCLEAR REPULSION ENERGY =     83.4462930029
 ----------------------------------------------
                TOTAL ENERGY =   -224.8911684969
...
```

倍精度計算をしています。収束が同じであれば 12 桁程度一致するはずです。
また、化学的に正しい答えでも微妙に収束結果が異なる場合があります。
その場合でも 9桁程度はエネルギーが一致するはずです。

10

[添付資料 14]

PGI アクセラレータコンパイラによる Fortran 2D-FDTD の性能評価

2011/03/25
九州大学
本田宏明

Agenda

- PGI アクセラレータコンパイラ
- 2通りの CUDA 利用プログラミングモデル
  - ディレクティブ
  - PGI CUDA Fortran
- 性能評価
  - Fortran 2D-FDTD アプリケーションを用いた性能評価

PGI アクセラレータコンパイラ

- Fortran、C/C++ からの GPU 利用
  - コンパイラディレクティブの使用により、シリアルプログラムを気軽に GPU 化可能（Fortran/C/C++）
  - CUDA と同様なプログラミング環境を Fortranに対しサポート（CUDA Fortran）
- 11/03/24 現在: version 11.3
  - CUDA 3.2 までのサポート
  - Nvidia GPU のサポート

コンパイラディレクティブによる方法

```
do it = 1, N
!$acc region
      do i = 1, MEMHY-2
         do j = 0, MEMHY-2
            Hx ( j, i ) = Hx ( j, i )
&                 - Dhy * ( Ez ( j + 1, i ) - Ez ( j, i ) )
         enddo
      enddo
      do i = 0, MEMHY-1-1
      ...
      enddo
      do i = 1, MEMHY-1-1
      ...
      enddo
!$acc end region
   enddo
```

- 逐次プログラムに対しディレクティブを挿入するだけで CUDA が半自動的に利用可能
- HOST <-> GPU 間のデータ転送については制御可能
  - GPU に関わるデータを3 種類に区別
    - HOST-> GPU
    - HOST <- GPU
    - GPU 内部のみ
  - 同一データを利用しGPUによる計算を複数回行う際にはその旨を記述
- Fortran/C/C++ に対し使用可能
- 手軽

- HOST<->GPU データの転送方法
- カーネルコード呼出し方法の指定

⇒逐次プログラムから、ただちに CUDA 利用コードへと変更可能

CUDA Fortran による方法

```
   grid = dim3 ( MEMHY/32, 1, 1 )
   block = dim3 ( 32, 1, 1 )
   allocate ( d_Ez ( 0:MEMHY-1, 0:MEMHY-1 ) )
   allocate ( d_Hx ( 0:MEMHY-1, 0:MEMHY-1 ) )
   allocate ( d_Hy ( 0:MEMHY-1, 0:MEMHY-1 ) )
   d_Ez = Ez
   d_Hx = Hx
   d_Hy = Hy
   do it = 1, N
      call fdtdkernel_Hx <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_Ez, d_Hx, d_Hy )
      call fdtdkernel_Hy <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_Ez, d_Hx, d_Hy )
      call fdtdkernel_Ez <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_Ez, d_Hx, d_Hy )
   enddo
...
module cuda_kernel
contains
attributes (global) subroutine fdtdkernel_Hx ( CEx, CEy, CHx, CHy, Ez, Hx, Hy )
   implicit none
   integer ix, xs, xe, ys, ye
   real4, value :: CEx, CEy, CHx, CHy
   real4, device :: Ez ( 0:MEMHY-1, 0:MEMHY-1 ), Hx ( 0:MEMHY-1, 0:MEMHY-1 ),
   ix = ( blockidx - 1 ) * blockdim + threadidx
...
```

HOST 側コード
- grid, block 定義、
- Device メモリ確保
- Host ⇔ GPU データ転送
- GPU 関数呼び出し
- 必要な手続きは従来の CUDA と同様

GPU 側コード

CUDA C/C++ に対する Fortran 版

2D-FDTD プログラムに対する性能評価

- 2種類の CUDA 利用手法の比較
  1. ディレクティブによる CUDA 利用
  2. CUDA Fortran による記述
- 電磁界方程式 2D-FDTD プログラム

測定、開発環境

- 測定環境
  - HOST: Intel Xeon X5650, 2.67GHz
    5GB Mem.
  - GPU: Tesla M2050, 1.03TFlop/s (single)
    Mem.BW: 148GB/s
- 開発環境
  - Cent OS 5.5
  - CUDA 3.2
  - PGI Accelerator Compiler 11.3

差分式に対する逐次プログラムの構造

Hx, Hy, Ez の時間発展(n ⇒ n+1)

Hx

$$H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2}\right)-H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2}\right)+C_{Bx}\left\{E_z^n(i,j+1)-E_z^n(i,j)\right\}$$

$$H_x^{n+\frac{1}{2}}\left(i,j-\frac{1}{2}\right)-H_x^{n-\frac{1}{2}}\left(i,j-\frac{1}{2}\right)+C_{Bx}\left\{E_z^n(i,j)-E_z^n(i,j-1)\right\}$$

Hy

$$H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j\right)-H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2},j\right)+C_{By}\left\{E_z^n(i+1,j)-E_z^n(i,j)\right\}$$

$$H_y^{n+\frac{1}{2}}\left(i-\frac{1}{2},j\right)-H_y^{n-\frac{1}{2}}\left(i-\frac{1}{2},j\right)+C_{By}\left\{E_z^n(i,j)-E_z^n(i-1,j)\right\}$$

Ez

$$E_z^{n+1}(i,j)=E_z^n(i,j)$$

$$+C_{Ex}\left\{H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2}\right)-H_x^{n+\frac{1}{2}}\left(i,j-\frac{1}{2}\right)\right\}$$

$$+C_{Ex}\left\{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j\right)-H_x^{n+\frac{1}{2}}\left(i-\frac{1}{2},j\right)\right\}$$

```
      do it = 1, N
c     ------
        do i = 1, NMESHX-1-1
          do j = 0, NMESHY-1-1
            Hx ( j, i ) = Hx ( j, i )
     &                  - CHy * ( Ez ( j + 1, i ) - Ez ( j, i ) )
          enddo
        enddo
        do i = 0, NMESHX-1-1
          do j = 1, NMESHY-1-1
            Hy ( j, i ) = Hy ( j, i )
     &                  + CHx * ( Ez ( j, i + 1 ) - Ez ( j, i ) )
          enddo
        enddo
        do i = 1, NMESHX-1-1
          do j = 1, NMESHY-1-1
            Ez ( j, i ) = Ez ( j, i )
     &                  + CEx * ( Hy ( j, i ) - Hy ( j, i - 1 ) )
     &                  - CEy * ( Hx ( j, i ) - Hx ( j - 1, i ) )
          enddo
        enddo
c     ------
      enddo
```

2D-FDTD ディレクティブの記述

```
!$acc region                                                    &
!$acc copy (Ez ( 0.NMESHX-1, 0.NMESHY-1 ))  &
!$acc copy (Hx ( 0.NMESHX-1, 0.NMESHY-1 ))  &
!$acc copy (Hy ( 0.NMESHX-1, 0.NMESHY-1 ))  &
!$acc copyin (CEx, CEy, CHx, CHy)

!$acc do hst1
      do it   0, N - 1
!
        do i   1, NMESHX-1-1
!$acc do vector (256)
          do j   0, NMESHY-1-1
            Hx (j, i )   Hx (j, i ) - CHy * ( Ez (j + 1, i ) - Ez (j, i ) )
          enddo
        enddo
        do i   0, NMESHX-1-1
!$acc do vector (256)
          do j   1, NMESHY-1-1
            Hy (j, i )   Hy (j, i ) + CHx * ( Ez (j, i + 1 ) - Ez (j, i ) )
          enddo
        enddo
        do i   1, NMESHX-1-1
!$acc do vector (256)
          do j   1, NMESHY-1-1
            Ez (j, i )   Ez (j, i ) + CEx * ( Hy (j, i ) -
                                    - CEy * ( Hx (j, i ) -

          enddo
        enddo
!
      enddo
!$acc end region
```

GPU オフロード化箇所の指定

do it ループの前後のみ
データ転送
copy:　該当データは入出力
copyin:　該当データは入力のみ

do it ループは HOST で処理

CUDA スレッド・ブロックサイズ
の指定
(256 の引数や記述自身を省略可能,
コンパイラが自動指定、
修正が必要な場合に記述)

プログラマは
1.どの場所を GPU で計算するか
2.どのデータを GPU とやりとりするか
のみを記述できれば良い

2D-FDTD CUDA Fortran 化 (1/2)

Host 側コード

```
real*4, dimension (..), allocatable, device ..d_Ez, d_Hx, d_Hy
type(dim3) grid, block
grid    dim3 (NMESHX/32, 1, 1 )
block   dim3 ( 32, 1, 1 )

allocate ( d_Ez ( 0.NMESHX-1, 0.NMESHY-1 ) )
allocate ( d_Hx ( 0.NMESHX-1, 0.NMESHY-1 ) )
allocate ( d_Hy ( 0.NMESHX-1, 0.NMESHY-1 ) )
!
d_Ez   Ez
d_Hx   Hx
d_Hy   Hy
!
  do it   1, N
    call fdtdkernel_Hx <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
    call fdtdkernel_Hy <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
    call fdtdkernel_Ez <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
  Enddo
!
Ez   d_Ez
Hx   d_Hx
Hy   d_Hy
```

Device メモリ領域指定

Grid, Thread 指定

Device メモリ領域確保
cudaMemAlloc と同機能
(cudaMemAlloc を記述してもよい)
行列アロケートの文法は f90 と同様

Host ⇔ Device メモリデータ転送
cudaMemcpy と同機能
(cudaMemcpy を記述してもよい)
行列代入の文法は f90 と同様、
Host メモリ領域行列からDevice メモリ領域行列への代入とみなす

GPU コード呼び出し

2D-FDTD CUDA Fortran 化 (2/2)

GPU 側コード

```
module cuda_kernel
contains
attributes (global) subroutine fdtdkernel_Hx ( CEx, CEy, CHx, CHy, Ez, I
  implicit none
  include "fdtd2dt.h"
  integer i, j, it, N, tg
  integer ix, xs, xe, ys, ye
  integer c1, c2, chks1
  real*4, value .. CEx, CEy, CHx, CHy
  real*4, device .. Ez ( 0.NMESHX-1, 0.NMESHY-1 ), Hx ( 0.NMESHX-
            Hy ( 0.NMESHX-1, 0.NMESHY-1 )

  i   ( blockidx%x - 1 ) * blockdim%x + threadidx%x  -1
  xs   1
  xe   NMESHX-2
  tg   0
  if ( ( xs <  i ) .and. ( i <  xe ) ) then
    tg   1
  endif
  do j   0, NMESHY-1-1
    if ( tg      1 ) then
      Hx ( j, i )   Hx ( j, i ) - CHy * ( Ez ( j + 1, i ) - Ez ( j, i ) )
    endif
  enddo
end
```

Host から呼出される
kernel 関数の指定

Call by value 対象引数の指定

Device メモリ領域引数の指定

blockidx%x, threadidx%x は 1 から

.cuf の拡張子のプログラムの場合、
PGI コンパイラはCUDA Fortran コード
とみなし、オプション等を適切に設置

CUDA C/C++ のプログラミング手法を知っている場合、
移行することは容易

結果

	HOST	GPU		
		ディレクティブ	CUDA Fortran	CUDA[1)]
実行時間 (msec.)	470	196 計算: 125 データ転送: 67	1476	
HOST /GPU	1	2.4	3.1	
GFlop/s	4.3	10.3	1.4	31.3

1) GeForce GTX 280, 1 枚使用の場合：高田直同, 下馬場朋禄, 塩田伝志, 伊藤智義, "GPUによる共有メモリを効率的に用いたFDTD法差分計算の高速化", 第8回情報科学技術フォーラム, C-013, Sep.2009.

PGI CUDA Profiler による結果

ディレクティブ

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M2050
# TIMESTAMPFACTOR fffff6ca1e42db0
method,gputime,cputime,occupancy
method [ memcpy HtoD ] gputime [ 12190.656 ] cputime [ 12339.000 ]
method [ memcpy HtoD ] gputime [ 11676.240 ] cputime [ 11936.000 ]
method [ memcpy HtoD ] gputime [ 11456.480 ] cputime [ 11629.000 ]
method [ cpucalc_48_gpu ] gputime [ 4664.960 ] cputime [ 14.000 ] occupancy [ 1.000 ]
method [ cpucalc_55_gpu ] gputime [ 4974.496 ] cputime [ 4.000 ] occupancy [ 0.833 ]
method [ cpucalc_62_gpu ] gputime [ 7293.472 ] cputime [ 4.000 ] occupancy [ 0.667 ]
method [ cpucalc_48_gpu ] gputime [ 4662.752 ] cputime [ 3.000 ] occupancy [ 1.000 ]
...
```

プログラム実行時にシェル環境変数を設定
Export CUDA_PROFILE=1
⇒ cuda_profile_0.log にログが出力される

Host ⇒ GPU データ転送

kernel 関数計算

Kernel 関数実行時間がおおよそ 7 倍
⇒ 実行時間の違いに対応

CUDA Fortran

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla M2050
# TIMESTAMPFACTOR fffff6d50e535a8
method,gputime,cputime,occupancy
method [ memcpy HtoD ] gputime [ 11850.272 ] cputime [ 12112.000 ]
method [ memcpy HtoD ] gputime [ 11422.656 ] cputime [ 11613.000 ]
method [ memcpy HtoD ] gputime [ 11396.608 ] cputime [ 11546.000 ]
method [ fdtdkernel_hx ] gputime [ 35311.008 ] cputime [ 14.000 ] occupancy [ 0.167 ]
method [ fdtdkernel_hy ] gputime [ 35414.273 ] cputime [ 4.000 ] occupancy [ 0.167 ]
method [ fdtdkernel_ez ] gputime [ 69443.711 ] cputime [ 4.000 ] occupancy [ 0.167 ]
method [ fdtdkernel_hx ] gputime [ 35306.977 ] cputime [ 3.000 ] occupancy [ 0.167 ]
...
```

Host ⇒ GPU データ転送

kernel 関数計算

CUDA Fortran では各 kernel 計算処理対し、
なんらかのオーバヘッド？

まとめ

- Fortran 2D-FDTD プログラムに対し、PGI アクセラレータコンパイラがサポートする 2通りの CUDA 利用プログラミングモデルにより、GPU の利用を図った。
- ディレクティブを使用する場合では、必要分のデータを適切に GPU へ転送する方法をとることで加速実行することが出来た。
- CUDA Fortran を使用する場合では、ディレクティブを使用する方法に比較し各カーネル関数処理の時間が大きく、逐次実行プログラムに比較し性能低下となった。

CUDA GPU エミュレーター等による原因調査

[添付資料 15]

PGI アクセラレータコンパイラによる Fortran 2D-FDTD の性能評価（2）ならびに HMPP によるテスト計算

2011/08/19
九州大学
本田宏明

PGI CUDA Fortran による 2D-FDTD 計算（2）

Agenda

- 性能評価
  - Fortran 2D-FDTD アプリケーションを用いた性能評価（2）
- HMPP の紹介
- HMPP による 2D-FDTD 計算、性能評価

PGI アクセラレータコンパイラ

- Fortran、C/C++ からの GPU 利用
  - コンパイラディレクティブの使用により、シリアルプログラムを気軽に GPU 化可能（Fortran/C/C++）
  - CUDA と同様なプログラミング環境を Fortranに対しサポート（CUDA Fortran）
- 11/08/19 現在: version 11.8
  - CUDA 4.0 までのサポート
  - Nvidia GPU のサポート

2D-FDTD プログラムに対する性能評価

- 3(4) 種類の CUDA 利用手法の比較

1. ディレクティブによる CUDA 利用 PGI Accelerator Fortran (C++)
2. PGI CUDA Fortran
3. NVidia CUDA C

前回との違い
- CUDA Fortranについて、前回は dim3 の grid, block 指定に対する関数内の thread 番号指定が適切でなく、大幅な実行時間低下。今回は修正
- C, Fortran ともに 1 次元配列のみを用い、ほぼ同じ実装による比較

測定、開発環境

- 測定環境
  - HOST: Intel Xeon X5650, 2.67GHz 5GB Mem.
  - GPU: Tesla M2050, 1.03TFlop/s (single) Mem.BW: 148GB/s
- 開発環境
  - Cent OS 5.5
  - CUDA 3.2
  - PGI Accelerator Workstation 11.8

差分式に対する逐次プログラムの構造

Hx, Hy, Ez の時間発展 (n ⇒ n+1)

Hx

$$H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2}\right)=H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2}\right)+C_{Hy}\left\{E_z^n(i,j+1)-E_z^n(i,j)\right\}$$

$$H_x^{n+\frac{1}{2}}\left(i,j-\frac{1}{2}\right)=H_x^{n-\frac{1}{2}}\left(i,j-\frac{1}{2}\right)+C_{Hy}\left\{E_z^n(i,j)-E_z^n(i,j-1)\right\}$$

Hy

$$H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j\right)=H_y^{n-\frac{1}{2}}\left(i+\frac{1}{2},j\right)+C_{Hx}\left\{E_z^n(i+1,j)-E_z^n(i,j)\right\}$$

$$H_y^{n+\frac{1}{2}}\left(i-\frac{1}{2},j\right)=H_y^{n-\frac{1}{2}}\left(i-\frac{1}{2},j\right)+C_{Hx}\left\{E_z^n(i,j)-E_z^n(i-1,j)\right\}$$

Ez

$$E_z^{n+1}(i,j)=E_z^n(i,j)$$

$$+C_{Ex}\left\{H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2}\right)-H_x^{n+\frac{1}{2}}\left(i,j-\frac{1}{2}\right)\right\}$$

$$+C_{Ex}\left\{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j\right)-H_x^{n+\frac{1}{2}}\left(i-\frac{1}{2},j\right)\right\}$$

```
Subroutine Hx:
      do i = 1, NMESHX-1-1
        do j = 0, NMESHY-1-1
          Hx ( j, i ) = Hx ( j, i )
&                 - CHy * ( Ez ( j + 1, i ) - Ez ( j, i ) )
        enddo
      enddo

Subroutine Hy:
      do i = 0, NMESHX-1-1
        do j = 1, NMESHY-1-1
          Hy ( j, i ) = Hy ( j, i )
&                 + CHx * ( Ez ( j, i + 1 ) - Ez ( j, i ) )
        enddo
      enddo

Subroutine Ez:
      do i = 1, NMESHX-1-1
        do j = 1, NMESHY-1-1
          Ez ( j, i ) = Ez ( j, i )
&                 + CEx * ( Hy ( j, i ) - Hy ( j, i - 1 ) )
&                 - CEy * ( Hx ( j, i ) - Hx ( j - 1, i ) )
        enddo
      enddo
```

2D-FDTD ディレクティブの記述

```
$acc region
$acc copy (Ez (0,NMESH-1)) &
$acc copy (Hx (0,NMESH-1)) &
$acc copy (Hy (0,NMESH-1)) &
$acc copyin (CEx, CEy, CHx, CHy)

$acc do host
      do it   0, N-1
!
        do i   1, NMESHX-1-1
$acc do vector (256)
          do j   0, NMESHY-1-1
            Hx (j, i )   Hx (j, i ) - CHy * ( Ez (j + 1, i ) - Ez (j, i ) )
          enddo
        enddo
        do i   0, NMESHX-1-1
$acc do vector (256)
          do j   1, NMESHY-1-1
            Hy (j, i )   Hy (j, i ) + CHx * ( Ez (j, i + 1 ) - Ez (j, i ) )
          enddo
        enddo
        do i   1, NMESHX-1-1
$acc do vector (256)
          do j   1, NMESHY-1-1
            Ez (j, i )   Ez (j, i ) + CEx * ( Hy (j, i ) - Hy (j, i - 1 ) )
                                    - CEy * ( Hx (j, i ) - Hx (j - 1, i ) )
          enddo
        enddo
!
      enddo
$acc end region
```

GPU オフロード化箇所の指定

do it ループの前後のみ
データ転送
copy:　該当データは入出力
copyin:　該当データは入力のみ

do it ループは HOST で処理

CUDA スレッド・ブロックサイズ
の指定
(256 の引数省略記述自身を実略可能

プログラマは以下の記述を実行
1.Redundant な転送を防ぐために、
同一データを含む加速部を1箇所にマージ
2.どの場所を GPU で計算するか
3.どのデータを GPU　とやりとりするか

2D-FDTD CUDA Fortran 化 (1/2)

Host 側コード

```
real*4, dimension (...), allocatable, device ..d_Ez, d_Hx, d_Hy
type(dim3) grid, block
grid    dim3 (1,NMESHY, 1 )
block   dim3 (1, NMESHY, 1 )

allocate ( d_Ez (0,NMESH-1 ) )
allocate ( d_Hx (0,NMESH-1 ) )
allocate ( d_Hy (0,NMESH-1 ) )
!
d_Ez   Ez
d_Hx   Hx
d_Hy   Hy
!
  do it   1, N
    call fdtdkernel_Hx <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
    call fdtdkernel_Hy <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
    call fdtdkernel_Ez <<< grid, block >>> ( CEx, CEy, CHx, CHy, d_E
  Enddo
!
Ez   d_Ez
Hx   d_Hx
Hy   d_Hy
```

Device メモリ領域指定

Grid, Thread 指定

Device メモリ領域確保
cudaMemAlloc と同機能
(cudaMemAlloc を記述してもよい)
行列アロケートの文法は f90 と同様

Host ⇔ Device メモリデータ転送
cudaMemcpy と同機能
(cudaMemcpy を記述してもよい)
行列代入の文法は f90 と同様、
Host メモリ領域行列から Device　メモリ領域行列への代入とみなす

GPU コード呼び出し

2D-FDTD CUDA Fortran 化 (2/2)

GPU 側コード

```
module cuda_kernel
contains
attributes (global) subroutine fdtdkernel_Hx ( CEx, CEy, CHx, CHy, Ez, Hx, Hy )
  implicit none
  include 'fdtd2dt.h'
  integer i, j, it, N, ttg
  integer ix, xs, xe, ys, ye
  integer c1, c2, chost
  real*4, value .. CEx, CEy, CHx, CHy
  real*4, device .. Ez (0,NMESH-1 ), Hx (0,NMESHX-1 ), Hy (0,NMES

  i   ( blockidx%x - 1 ) * blockdim%x + threadidx%x  -1
  xs   1
  xe   NMESHX-2
  ttg   0
  if ( ys <   j ) .and. ( j <   ye ) then
    ttg   1
  endif
  do i   0, NMESHX-1-1
    if ( ttg    1 ) then
      Hx ( i * NMESHY + j )   Hx ( i * NMESHY + j )
                    - CHy * ( Ez ( i * NMESHY + j + 1 ) - Ez ( i * NMESHY + j ) )
    endif
  enddo
end
```

Host から呼出される
kernel 関数の指定

Hx 計算ループ kernel コード生成

Device メモリ領域引数の指定

blockidx%x, threadidx%x は 1 から

.cuf の拡張子のプログラムの場合、
PGI コンパイラはCUDA Fortran コード
とみなし、オプション等を適切に設置

CUDA C/C++ のプログラミング手法を知っている場合、
移行することは容易

結果

	HOST			GPU		
	GCC	PGI/C	PGI/Fort	CUDA/C	PGI CUDA/Fort	PGI DIRE/Fort
実行時間 (msec.)	136.0	101.0	44.3	6.90	10.8	16.9
高速化 (HOST GCC 基準)	1	1.34	3.06	19.7	12.6	8.04
GFlop/s	2.30	3.15	7.08	45.5	29.0	18.5

今一歩、
CUDA/C に比較し、
性能が発揮出来ていない

1) (x, y): 512x512 : 100 steps 計算, 10 回繰り返し
2) Required Byte/Flops: 3.67byte/sec., BW: 140GB/s
=> 38.3 GFlop/s

Hybrid Multicore Parallel Programming (HMPP) による 2D-FDTD 計算

HMPP の概要

- フランス CAPS 社が提供している GPU コードジェネレーター
- CUDA, OpenCL サポート
- ディレクティブベースプログラミングモデル
  - PGI コンパイラに似たディレクティブベース
  - 基本は関数定義部分のみアクセラレータへオフロード
    - 関数中のループ構造部分のオフロードも可能
  - C, Fortran のサポート
  - PGI より詳細な指示行指定が可能
  - HMPP は CUDA 等のコード生成のみ
    - ディレクティブの翻訳のみ担当
  - GCC 等と組み合わせて使用、PGI, Intel Compiler との組合せも可能

基本コード例

オリジナル

```
void myFunc( int n, intA[n], intB[n] )
{
   int i ;
   for ( i = 0 ; i < n ; ++i ) {
      B[i] = A[i] + 1;
   }
}

int main ( )
{
   int X[10000], Y[10000], Z[10000] ;

myFunc(10000, X, Y) ;
...
}
```

GPU 使用

```
#pragma hmpp call1 codelet, target=CUDA
void myFunc( int n, intA[n], intB[n] )
{
   int i ;
   for ( i = 0 ; i < n ; ++i ) {
      B[i] = A[i] + 1;
   }
}

int main ( )
{
   int X[10000], Y[10000], Z[10000] ;

#pragma hmpp call1 callsite
 myFunc(10000, X, Y)
 ...
}
```

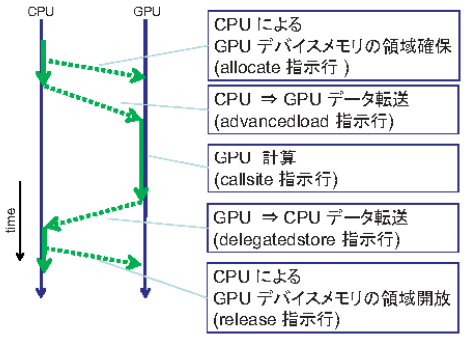
CORDLET 指示行
関数定義に指定

"call1" が identifier

CODESITE 指示行
関数呼出しに指定

GPU 計算に関与する関数定義と関数呼び出しの指定（この2つの指示行はペアとして必要、デフォルトでデータ転送等が行われる）

CPU-GPU 動作と指示行による制御



HMPP による 2DFDTD 実装（1／2）

関数定義：CODELET 側

```
#pragma hmpp <fdtd2d> group, target=CUDA

#pragma hmpp <fdtd2d> gpu_calc_Hx codelet, args[Ez,Hx,Hy].io=inout
void gpu_calc_Hx ( SS_LD CHy, SS_LD Ez [ NMESH ], SS_LD Hx [ NMESH ], SS_LD Hy [ NMESH ] )
{
   SS_LI i, j, is, ie, js, je ;
   is = 1 ;
   is = 0 ;
   je = NMESHX - 2 ;
   je = NMCCIIY - 2 ;
   for ( i = is ; i <= ie ; i++ ) {
      for ( j = js ; j <= je ; j++ ) {
         Hx [ i * NMESHY + j ] = Hx [ i * NMESHY + j ]
                                 - CHy * ( Ez [ i * NMESHY + j + 1 ]- Ez [ i * NN
      }
   }
}

#pragma hmpp <fdtd2d> gpu_calc_Hy codelet, args[Ez,Hx,Hy].io=inout
void gpu_calc_Hy ( SS_LD CHx, SS_LD Ez [ NMESH ], SS_LD Hx [ NMESH ], SS_LD Hy [ NMESH ] )
{
   SS_LI i, j, is, ie, js, je ;
   is = 0 ;
   is = 1 ;
   ....
```

CODELET グループ化：
（グループ名：fdtd2d）

CODELET 指示行：
- 関数引数にはサイズを明示
- デフォルト指定なし引数
  - CPU ⇒ GPU のみ転送
  - CHy
- 指定引数
  - CPU ⇔ GPU 転送制御
  - Ez, Hx, Hy

HMPP による 2DFDTD 実装（2／2）

関数定義：CODELET 側

```
#pragma hmpp <fdtd2d> allocate, args[*::Ez].size={ (512*512) } &
#pragma hmpp &                  args[*::Hx].size={ (512*512) } &
#pragma hmpp &                  args[*::Hy].size={ (512*512) }

#pragma hmpp <fdtd2d> gpu_calc_Hx advancedload, args[ Ez ; Hx ; Hy]

   for ( it = 0 ; it < NUMLOOP ; it++ ) {
      #pragma hmpp <fdtd2d> gpu_calc_Hx callsite, args[Ez;Hx;Hy].noupda
      gpu_calc_Hx ( LCHy,      LEz, LHx, LHy ) ;
      #pragma hmpp <fdtd2d> gpu_calc_Hy callsite, args[Ez;Hx;Hy].noupdate=true
      gpu_calc_Hy ( LCHx,      LEz, LHx, LHy ) ;
      #pragma hmpp <fdtd2d> gpu_calc_Ez callsite, args[Ez;Hx;Hy].noupd
      gpu_calc_Ez ( LCEx, LCEy, LEz, LHx, LHy ) ;
   }

#pragma hmpp <fdtd2d> gpu_calc_Ez delegatedstore, args[Ez ; Hx ; Hy]

#pragma hmpp <fdtd2d> release
```

ALLOCATE 指示行：
- cudaMemAlloc に対応（#define による変数が使えない？）

ADVANCEDLOAD 指示行：
- cudaMemCopyに対応

CALLSITE 指示行：
- GPU コード呼び出し
- CALLSITE に伴うデフォルトデータ転送の抑制

DELEGATEDSTORE 指示行：
cudaMemCopyに対応

RELEASE 指示行：
cudaMemFreeに対応

結果

	HOST			GPU			
	GCC	PGI/C	PGI/Fort	CUDA/C	PGI CUDA/Fort	PGI DIRE/Fort	HMPP/C
実行時間 (msec.)	136.0	101.0	44.3	6.90	10.8	16.9	27.1
高速化 (HOST GCC 基準)	1	1.34	3.06	19.7	12.6	8.04	5.01
GFlop/s	2.30	3.15	7.08	45.5	29.0	18.5	11.57

HMPP/C コードについて更なる最適化の検討が必要
- CORDLET側 関数内での最適化
  - 明示的な grid, block 指示制御
  - 2 重ループに対する明示的な parallel 化、no parallel 化指示
  - Loop unroll 指示

HMPP 関係、その他

- マニュアル、チュートリアルの整備やネット情報が不十分？
- サポートは、こじんまりとしていて丁寧な印象
- OpenHMPP としての規格化、NVidia がサポートするCAPS 以外での実装もあり

[添付資料 16]

SS研 アクセラレータ技術WG

FUJITSU

分子軌道法プログラムのGPGPU化

2011年8月19日
(株)富士通研究所
成瀬 彰

Copyright 2010 FUJITSU LABORATORIES LTD

概要

FUJITSU

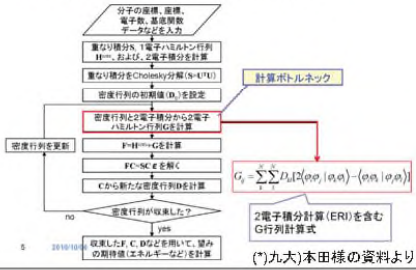
- 前回:
  - 一部関数のGPGPU化検討状況を報告
  - GPGPU化部分: 10倍性能UP (対1コアCPU)
  - atomic操作がボトルネック、もっと性能UPするはず
- 今回:
  - atomic操作が不要な実装方法に変更
  - (ほぼ)全ての主要関数をGPGPU化
  - プログラム全体: 最大13倍性能UP (対1コアCPU)

Copyright 2010 FUJITSU LABORATORIES LTD

分子軌道法プログラムフロー

FUJITSU

分子軌道法プログラムの全体構造



(*)九大)本田様の資料より

Copyright 2010 FUJITSU LABORATORIES LTD

ボトルネックはG行列計算

FUJITSU

	時間(秒)	比率(%)
全体	26.30	100.0
G行列計算	26.22	99.7
(ss,ss)型積分	7.99	30.4
(ps,ss)型積分	9.96	37.9
(ps,ps)型積分	3.73	14.2
(pp,ss)型積分	2.15	8.2
(pp,ps)型積分	1.94	7.4
(pp,pp)型積分	0.45	1.7

- 入力データ
  - sw03-sto
- CPU
  - Core-i7 2.66GHz
  - 1コア使用
- コンパイラ
  - gcc 4.1.2 (-O3)
- 計算ライブラリ
  - Intel mkl 10

- (ss,ss)型積分処理のGPGPU化を検討
  - 関数名: PSI_gen_gmat_ssss

Copyright 2010 FUJITSU LABORATORIES LTD

G行列計算 ((ss,ss)型積分)

FUJITSU

係数行列作成
1st loop begin (528)
2nd loop begin (1~528)
3rd loop begin (9)
4th loop begin (9)
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
部分G配列更新
2nd loop end
1st loop end

係数行列作成
係数行列数:16種
総量:2.4MB (問題依存)

1st & 2nd loop:
ループ回数:139,656 (問題依存)

ERI計算:
ループ回数:81 (問題依存)
ERI値:1種 (積分型依存)

部分G配列更新
前出ERI値と関連する部分を更新
G配列更新箇所:6 (積分型依存)
G配列要素数:1,596 (問題依存)

(*)入力データ: sw03-sto

Copyright 2010 FUJITSU LABORATORIES LTD

G行列計算のGPGPU化

FUJITSU

係数行列作成
1st loop begin (528)
2nd loop begin (1~528)
3rd loop begin (9)
4th loop begin (9)
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
部分G配列更新
2nd loop end
1st loop end

係数行列作成
係数行列数:16種類
総量:2.4MB (問題依存)

◆CPUで係数行列を作成
◆係数行列を転送(CPU→GPU)

(*)入力データ: sw03-sto

Copyright 2010 FUJITSU LABORATORIES LTD

G行列計算のGPGPU化

係数行列作成
1st loop begin (528)
2nd loop begin (1~528)
3rd loop begin (9)
4th loop begin (9)
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
部分G配列更新
2nd loop end
1st loop end

1st & 2nd loop:
ループ回数：139,656 (問題依存)

◆1st & 2nd loopの1ループ内の処理を、1スレッドに割当

(*)入力データ: sw03-sto

G行列計算のGPGPU化

係数行列作成
1st loop begin (528)
2nd loop begin (1~528)
3rd loop begin (9)
4th loop begin (9)
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
部分G配列更新
2nd loop end
1st loop end

ERI計算：
ループ回数：81 (問題依存)
ERI値：1種 (積分型依存)

◆3rd & 4th loop、マルチスレッド対象外
◆81ループ分の計算を、1スレッドで逐次実行

(*)入力データ: sw03-sto

G行列計算のGPGPU化

係数行列作成
1st loop begin (528)
2nd loop begin (1~528)
3rd loop begin (9)
4th loop begin (9)
初期積分計算
垂直漸化式計算
4th loop end
3rd loop end
水平漸化式計算
部分G配列更新
2nd loop end
1st loop end

部分G行列更新
ERI値毎に、関連する箇所を更新
G行列更新箇所：6
(全要素数：1,596)

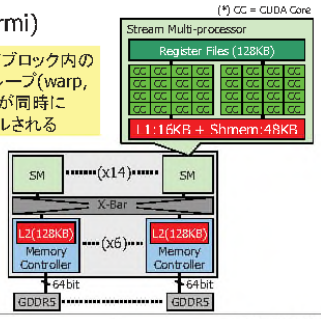
◆更新箇所は不連続
◆他スレッドと同じG行列要素を同時に更新する可能性あり
◆atomicCAS()による排他処理
◆CAS: Compare And Swap

(*)入力データ: sw03-sto

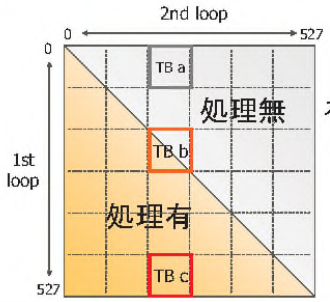
GPU内部構成

■ C2050(Fermi)

同じスレッドブロック内のスレッドグループ(warp, 32スレッド)が同時にスケジュールされる

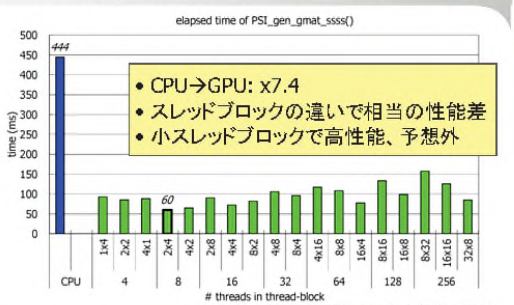


スレッドブロック割当



スレッドブロック(TB):
スレッド数: 4~256
形状: $2^n \times 2^m$

測定結果((ss,ss)型積分 1回の時間)

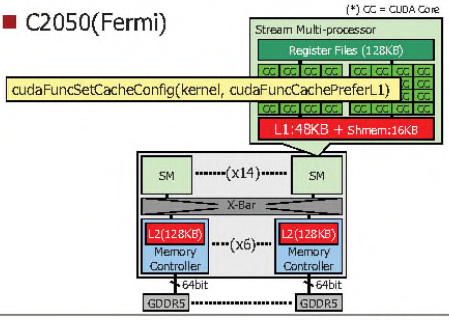


- CPU→GPU: x7.4
- スレッドブロックの違いで相当の性能差
- 小スレッドブロックで高性能、予想外

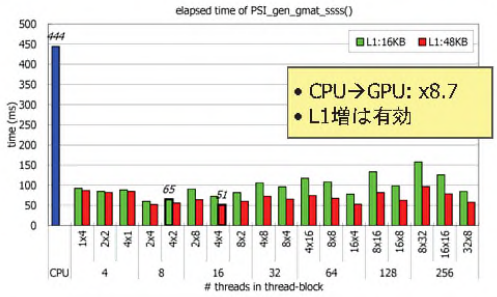
(*) GPU時間にはデータ転送時間含む

L1サイズ

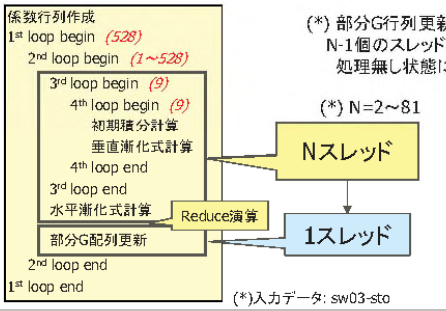
- C2050(Fermi)



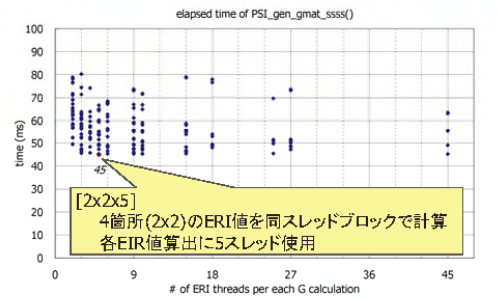
測定結果(L1:16KB→48KB)



ERI計算のマルチスレッド化



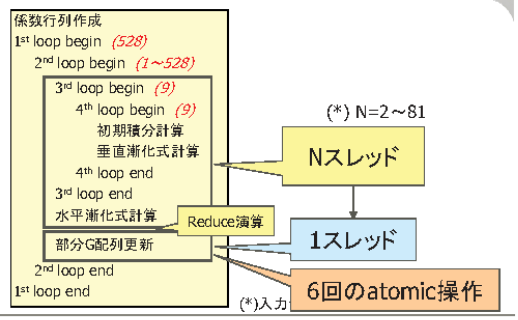
測定結果(ERI計算のスレッド化)



測定結果まとめ ((ss,ss)型 1回)

- CPU(1core)　444 ms
- GPU　L1:16KB　65 ms (x7.4)
- GPU　L1:48KB　51 ms (x8.7)
- GPU　ERIスレッド　45 ms (x9.9)

ボトルネックはどこ?



G行列更新のatomic操作を外すと…

- CPU(1core)　444 ms
- GPU　L1:16KB　65 ms (x7.4)
- GPU　L1:48KB　51 ms (x8.7)
- GPU　ERI計算スレッド化　45 ms (x9.9)
- GPU (w/o atomic操作)　25 ms (x17.8)

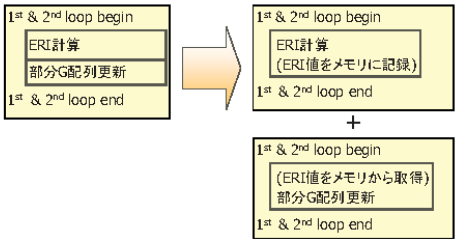
(*)ただし、正しい結果は得られない

- ボトルネックはatomic操作?

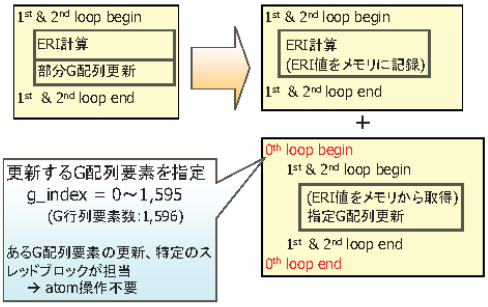
更に高速化するには..

- atomic操作の削減
  - 共有メモリ活用
    - 各スレッドの行列G更新、共有メモリに蓄積
    - スレッドブロック単位で、行列G(グローバルメモリ)に反映
    - → グローバルメモリへのatomic操作回数が削減
  - ERI値計算と部分G配列更新を、別カーネルに分離
    - ERI値計算: 計算したERI値、全てグローバルメモリに記録
    - 部分G配列更新: グローバルメモリからERI値を読み、G配列を更新
    - → atomic操作不要、行列Gへのアクセスが局所化
- メモリアクセスの局所化
  - 行列要素の並び替え (既に実施済み?)

kernel分離

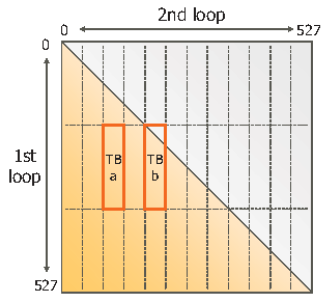


kernel分離



ERI計算kernelの並列化

L1:48KB, Shmem:16KB

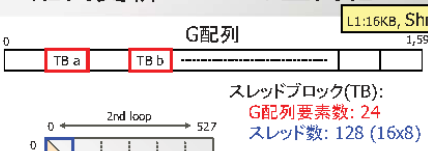


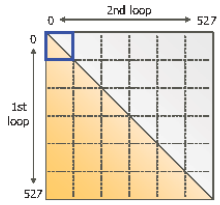
スレッドブロック(TB):
スレッド数: 128
形状: 4 x 32

- 各スレッド、1st & 2ndループの1ループに対応するERI計算を担当
  - 3rd & 4thループは並列化せず
- 算出したERI値をメモリに記録

G配列更新kernelの並列化

L1:16KB, Shmem:48KB





スレッドブロック(TB):
G配列要素数: 24
スレッド数: 128 (16x8)

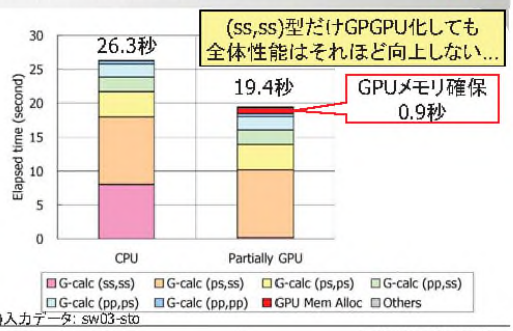
共有メモリ 24KB /TB

- スレッド挙動
  - これから読み込むERI値で更新されるG配列index値を算出
  - index値が自TBの対象範囲内なら、ERI値を読み、G配列の更新内容を共有メモリに記録
- 共有メモリに記録した値を集約し、G配列更新

Kernel分離の効果 ((ss,ss)型 1回)

- CPU(1core) 444 ms
- GPU L1:16KB 65 ms (x7.4)
- GPU L1:48KB 51 ms (x8.7)
- GPU ERI計算スレッド化 45 ms (x9.9)
- GPU (w/o atomic操作) 25 ms (x17.8)
- GPU kernel分離 11 ms (x40.2)
  - ERI計算kernel (9.7 ms)
  - G行列更新kernel (1.3 ms)

全体の実行時間は?



(*)入力データ: sw03-sto

他積分型もGPGPU化

積分型	時間(秒) CPU	時間(秒) GPU	性能UP
(ss,ss)	7.99	0.20	x40.2
(ps,ss)	9.96	0.32	x31.3
(ps,ps)	3.73	0.24	x15.3
(pp,ss)	2.15	0.12	x18.4
(pp,ps)	1.94	0.23	x8.5
(pp,pp)	0.45	...	...

- 入力データ
  - sw03-sto
- CPU
  - Core-i7 2.66GHz
  - 1コア使用
- コンパイラ
  - gcc 4.1.2 (-O3)
- 計算ライブラリ
  - Intel mkl 10

- 性能UP率、積分型次第でかなり違う

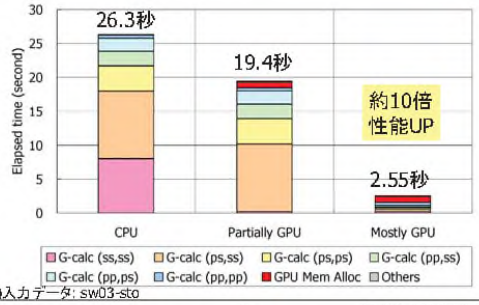
(*)入力データ: sw03-sto

実行時間分析

積分型	実行時間 (CPU)	実行時間 (GPU)	性能UP	1st&2nd ループ数	ループあたりERI数	総ERI数
(ss,ss)	7.99	0.20	x40.2	139,656	1	139,656
(ps,ss)	9.96	0.32	x31.3	135,168	3	405,504
(ps,ps)	3.73	0.24	x15.3	32,896	9	296,064
(pp,ss)	2.15	0.12	x18.4	19,008	9	171,072
(pp,ps)	1.94	0.23	x8.5	9,216	27	248,832
(pp,pp)	0.45	...	...	666	81	53,946

- 実行時間は何で決まるのか
  - CPU実行時間: 1st&2ndループ数と高い相関 (0.97)
  - GPU実行時間: 総ERI数と高い相関 (0.87)
  - → (pp,pp)型で性能UPは厳しそう…

改めて、全体の実行時間



(*)入力データ: sw03-sto

各種入力データでの測定結果

入力データ	実行時間(秒) CPU	GPU		GPU (w/o mem alloc)	
sw01-sto	0.0310	0.961	(0.03)	0.0510	(0.61)
sw02-sto	0.451	1.01	(0.45)	0.0980	(4.60)
sw03-sto	26.3	2.55	(10.3)	1.64	(16.0)
sw04-sto	254	19.5	(13.0)	18.6	(13.7)
sw01-631	0.0590	1.05	(0.06)	0.134	(0.44)
sw02-631	2.72	1.39	(1.96)	0.479	(5.68)
sw03-631	155	27.1	(5.72)	26.1	(5.94)

- データサイズ、大きいほうがGPU向き
- *-631は苦手 (3rd&4thループ回数が不定)

GPU実行時間の内訳

FUJITSU

積分型	GPU実行時間(ms)			性能UP	1st&2nd ループ数	ループあたりERI数	総ERI数
	合計	ERI	G				
(ss,ss)	199	175	24	x40.2	139,656	1	139,656
(ps,ss)	318	201	117	x31.3	135,168	3	405,504
(ps,ps)	244	122	122	x15.3	32,896	9	296,064
(pp,ss)	117	58	59	x18.4	19,008	9	171,072
(pp,ps)	228	139	89	x8.5	9,216	27	248,832
(pp,pp)	…	…	…	…	666	81	53,946

30 Copyright 2010 FUJITSU LABORATORIES LTD

計算精度

FUJITSU

- CPU(gcc)
  - -O3

```
      ONE ELECTRON ENERGY =   -1806.8918323226
      TWO ELECTRON ENERGY =     719.8050450871
 NUCLEAR REPULSION ENERGY =     487.3450878391
 ---------------------------------------------
            TOTAL ENERGY =    -599.7416993964
```

- GPU

```
      ONE ELECTRON ENERGY =   -1806.8918323226
      TWO ELECTRON ENERGY =     719.8050450871
 NUCLEAR REPULSION ENERGY =     487.3450878391
 ---------------------------------------------
            TOTAL ENERGY =    -599.7416993963
```

- CPU(intel)
  - -O3

```
      ONE ELECTRON ENERGY =   -1753.8872399834
      TWO ELECTRON ENERGY = -361844.2578283538
 NUCLEAR REPULSION ENERGY =     487.3450878391
 ---------------------------------------------
            TOTAL ENERGY = -363110.7999804982
```

31 Copyright 2010 FUJITSU LABORATORIES LTD

まとめと今後

FUJITSU

- 分子軌道法プログラムのG行列計算の大部分をGPGPU化
  - GPGPU化した積分型:
    - (ss,ss), (ps,ss), (ps,ps), (pp,ss), (pp,ps) … 時間比率98%
- アプリ全体で、最大13倍の高速化 (対1コア)
  - GPUメモリ確保に固定オーバーヘッド (0.9秒)
  - それを除けば、最大16倍の高速化
- CPUとGPU、適切な実装方法が違う
- 今後:
  - (pp,pp)型積分のGPGPU化
  - 大規模入力データへの対応

32 Copyright 2010 FUJITSU LABORATORIES LTD

FUJITSU

shaping tomorrow with you

【発行者】サイエンティフィック・システム研究会
【編　集】サイエンティフィック・システム研究会
　　アクセラレータ技術ＷＧ
【発行日】2012 年 3 月 26 日

◆　◆　◆

＜連絡先＞
サイエンティフィック・システム研究会 事務局
〒105-7123 東京都港区東新橋 1-5-2 汐留シティセンター
TEL ： 03-6252-2582
E-mail ： office@ssken.gr.jp

※著作権は各原稿の著者または所属機関に帰属します。無断転載を禁じます。