実用アプリケーションへの GPGPU の適用 －デバイスシミュレータ Advance/DESSERT における高速化

桜井　清吾*

Applying GPGPU Technology for a Practical Application －Speed-up Results in Device Simulator “Advance/DESSERT”

Shogo Sakurai*

1. はじめに

近年、単一コア CPU の性能向上が頭打ちになる中、GPU（Graphics Processing Unit）は毎年 2 倍近い性能向上を達成し、CPU よりも格段に優れた演算性能・メモリバンド幅を有している。この GPU をグラフィックス処理以外の汎用計算に応用する取り組み（GPGPU:General-purpose computing on GPUs）は、国内外でここ数年目覚ましく進展しており、数値シミュレーション手法の実装・高速化等の研究も顕著な広がりを見せている。

GPU は性能比で CPU よりもコストパフォーマンスに優れており、科学技術計算を行う際に重要な、本格的な倍精度演算対応のプロセッサ Fermi を載せた GPU コンピューティングボード NVIDIA Tesla C2050 が本年 4 月にリリースされている[1]。これは前世代の Tesla C1060 よりも倍精度演算性能が向上したことに加え、科学技術計算分野で重要な ECC をサポートし、新たに L1/L2 キャッシュが搭載されている。このようなハードウェアの発展と共に、プログラム開発環境も整えられており、誰でも無償で NVIDIA の Web サイトを通じて GPGPU 開発環境 CUDA が入手できる[2]。現在は、これにより C,C++言語による API が提供されているが、サードベンダーからは OpenMP のようにコードにディレクティブを挿入するだけで GPU による高速化が実現できる Fortran 開発環境も提供されている[3]。また、GPGPU に関する書籍[4][5],雑誌記事[6]も充実してきており、展示会[7]や研究会[8]も盛況である。

*アドバンスソフト株式会社　技術第 2 部
2nd Technical Division, AdvanceSoft Corporation

図 1 HPC 向け倍精度対応ボード (左)NVIDIA Tesla C1060 (右)NVIDIA Tesla C2050[1]

このように 2010 年は GPGPU の一大ブームが生まれた年であった。しかし、このようなブームはまだ科学技術計算分野の研究者やプログラム開発者に留まっており、真にエンドユーザの利益に供するアプリケーションの開発はこれからである。

そこで本稿では、半導体デバイス設計において欠かすことのできないデバイスシミュレーションを例に取り、当社が独自に開発したデバイスシミュレータ Advance/DESSERT への CUDA を利用した GPGPU による高速化について報告する。

2. 線形疎行列解法について

デバイスシミュレータの高速化において重要な点は、計算時間の大半を占める以下の線形疎行列方程式を如何に高速化するかにある。

$$Ax = b \tag{1}$$

ここで、x,b は大きさ N のベクトル、A は $N \times N$ の疎行列である。デバイスシミュレーションにおいてこのような方程式は、電子と正孔の移流拡散及び電場の方程式を離散化して得られる。また、数値流体力学や構造解析などのいわゆる CAE(Computer Aided Engineering)分野でも同様の方程式が現れる。行列サイズが大きくなると、この方程式を代数的な処理によって直接的に解く方法を適用することは難しくなり、代わって共役勾配法(conjugate gradient)に代表される反復法が使われる。しかし、この方法では行列 A の条件数(condition number)が悪い場合では安定に収束させることが難しく、それに対処する工夫が必須である。これは特に行列の前処理として 1 つの研究分野となっている[9]。この前処理付き共役勾配法は、(1)式を解く代わりに、前処理行列 M を用いて、

$$M^{-1}Ax = M^{-1}b \tag{2}$$

を解くことに帰着される。この手法は様々な物理現象の解析に現れる線形疎行列方程式を解く強力な数値解法技術として、数多くの設計支援ソフトウェアに用いられている。

3. CUDA によるプログラミング

具体的な話に進む前に、先ず GPGPU 開発環境 CUDA によるプログラミングについて簡単に触れておきたい。**図 2** には、2 つのベクトル和の計算

$$g_{out} \leftarrow \alpha\, g_{in1} + g_{in2} \tag{3}$$

を GPU 上で行うコードが示されている。このコードでは、CUDA が提供するビルドイン変数 (blockDim,blockIdx,threadIdx)を利用して GPU の各スレッドがベクトルの個々の要素（アドレス）を参照し、ベクトル和の計算を実施している。ここで、CPU のように一つひとつの要素に対し逐次的に和を取っていく訳ではなく、GPU では一度に極めて多数のスレッドを起動することにより、同時に個々の要素の和を取ることができる。

```
template<typename T>
__global__ void axpy_kernel
(
  const int n,
  const T alpha,
  const T *g_in1,
  const T *g_in2,
  T *g_out
){
  int i = blockDim.x * blockIdx.x
          + threadIdx.x;
  if( i < n ) g_out[i] =
       alpha * g_in1[i] + g_in2[i];
}
```

図 2 ベクトル和のカーネル関数

このように GPU 上で実行する関数は、カーネル関数と呼ばれる。一方で、この関数は CPU 側からの指示により起動する。これに該当するコードをホスト関数（もしくはホストコード）と呼ぶ。**図 3** には**図 2** のカーネル関数を呼び出すホスト関数が示されている。CUDA における全てのスレッドは、ブロック単位で管理されており、ホスト関数では、ブロック当たりのスレッド数とブロックの個数が設定される。**図 3** では n をベクトルのサイズとして、それぞれ THREAD マクロと DIV(n,THREAD)マクロがこれに該当している。

```
#define THREAD 256
#define DIV(x,y) (((x)+(y)-1)/(y))

template<typename T>
__host__ cudaError_t axpy
(
  const int n,
  const T alpha,
  const T *g_in1,
  const T *g_in2,
  T *g_out
){
  axpy_kernel<T><<<DIV(n,THREAD),THREAD>>>
      (n, alpha, g_in1, g_in2, g_out);
  return cudaGetLastError();
}
```

図 3 ホスト関数

これらは、2 つのベクトルの和を計算するもので

あったが、共役勾配法では、更に行列・ベクトル積や内積計算などを効率的に行う必要がある。ここで、計算に用いられる行列は多くのゼロ要素を含んだ疎行列であることから、行列・ベクトル積の計算では、メモリ使用量の少ないデータ構造でありつつも同時に GPU 上で効率的な演算ができる必要がある。

本稿では、NVIDIA の研究者が性能評価し[10]、その後の別の研究者が改良した ELLPACK-R フォーマット[11]を利用した。このフォーマットでは、疎行列を以下のデータを用いて表現する。

- **indices:各行の非ゼロ要素カラムインデックス (nrows×width 個の要素)**
- **elems: 非ゼロ要素の係数データ (nrows×width 個の要素)**
- **rl: 各行の非ゼロ要素数 (nrows 個の要素)**

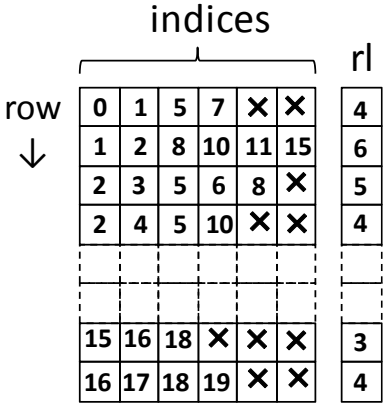


図 4 ELLPACK-R フォーマット

このデータ構造イメージを**図 4** に示す。図中の×は、ゼロ要素を示した padding 要素であり、これが多いと無駄にメモリを消費してしまうが、デバイスシミュレーションやそれ以外の多くの例では、一行当たりの非ゼロ要素数のバラツキはそれほど大きくはない。このデータ構造の利点は、各列のデータがアドレス空間で隣接しており、データアクセスの際に一塊として転送されやすい点にある。このように疎行列そのもののデータアクセスは効率的に行えるが、それに掛け合わされるベクトルは、ランダムアクセスとなる。これは、ELLPACK-R 以外の疎行列用データフォーマットでも同様である。GPU では CPU よりもキャッシュの利用に制限があり、Fermi 以前のプロセッサでは、テクスチャーキャッシュと呼ばれる CG レンダリング用のキャッシュを明示的に使用することである程度効率的なランダムアクセスが実行できる。Fermi は、テクスチャーキャッシュとは別に L1/L2 キャッシュを備えており、プログラマーが意識せずとも、効率的なランダムアクセスが計られている。

GPU 上でのベクトルの内積計算については、既に CUDA のサンプルコード[12]や書籍[4]で既に詳しく解説されていることからここでの説明は省略する。

4. デバイスシミュレーションで求められる前処理の性質[1]

デバイスシミュレーションでは、デバイス端子の電圧を少しずつ変化させて電流を計算し、一本の電圧電流曲線を求める。この際、電子と正孔と静電ポテンシャルはそれぞれ強く連成しており、電圧電流曲線中の各点において非線形連成方程式を解く必要がある。通常は、Gummel 法による外部反復を行うことでこの方程式を解くことが行われる。この反復は少なければ数回で収束するが、多いときには数百回のオーダーとなる。この 1 つの反復において、電子と正孔、静電ポテンシャルに対応する方程式が共役勾配法を用いて順番に解かれる。この時、各外部反復において、疎行列の非ゼロ要素の値が変動し、前処理行列の再生成が必要となることから、この時間が短いことが求められる。但し、デバイスシミュレーションでは、アダプティブメッシュのような場合を除いて、計算中にメッシュ点は変化しないため、各疎行列の非ゼロ要素の場所は変わらない。

[1] ここでは、定常問題で必要とされる性質について議論する。

5. 様々な前処理手法

一概に前処理付き共役勾配法といっても**表 1** に示すように様々な前処理手法があるが、収束性と GPU との親和性は区々である。最も簡単で GPU との親和性が高いのは、対角スケーリングであり、これは前処理行列を

$$M^{-1} \approx D^{-1} \tag{4}$$

で与える方法である。ここで D は A の対角行列である。この方法はしかし、収束性において十分とは言い難く、多くの問題で殆ど収束しない。やや実用的なのは、SOR や多項式前処理である。特に多項式前処理は、古くからベクトルプロセッサで使われており、次のように与えられる。

$$M^{-1} = (D+N)^{-1} = D^{-1}(I+D^{-1}N)^{-1}$$
$$= D^{-1}\sum_{k=0}^{\infty}(-1)^k(D^{-1}N)^k \approx D^{-1}(I-D^{-1}N) \tag{5}$$

ここで、N は A の非対角行列である。この前処理は単純な行列・ベクトル積のみで実施できるため GPU において効率的な演算が可能である。但し、収束性にやや問題があるため、広範囲の計算問題を解くような汎用シミュレータにはあまり適していない。

多項式前処理のように、近似的に逆行列を表現する前処理手法は、それ以外にも AINV や FSAI,bi-conjugation などが知られている[13]。これらの手法は、SOR や不完全 LU とは異なり、明示的に逆行列を表現しており、前処理演算が GPU 上で効率的に行える点から興味深い性質を有している。しかし、不完全 LU よりも fill-in の要素が多くなる傾向があることと、適切な fill-in の場所を特定することが難しいこと、更に近似逆行列自体を生成する処理時間も大きいことから[13]、不完全 LU 前処理が収束しない場合を除いて、それほど応用例が多くないようである。

この他に、実用的な前処理手法として、不完全 LU や Multigrid 法が知られている。特に不完全 LU は最もポピュラーな前処理手法であり、本稿でも後にこの前処理を用いた事例について報告する。一方で、Multigrid 法は粗い格子を階層的に用いてオーダーN の収束性を実現した興味深いアルゴリズムである。但し、非構造格子の場合には、構造格子のように幾何学的に粗い格子が求まる訳ではなく、疎行列 A の隣接グラフと非ゼロ要素の値から求められる[14]。一般にこの処理には時間が掛かる。特にデバイスシミュレーションでは Gummel の外部反復毎に非ゼロ要素の値が変動するため、毎回の格子生成が必要となる。このため、デバイスシミュレーションへの適用は処理時間の面で難しいことが考えられる。

表 1 前処理手法の種類

前処理手法		スレッド並列化	収束性
対角スケーリング		○	弱い
Gauss-Seidel, SOR		△	ある程度弱い
不完全 LU		△	強い
Multigrid		△	強い
近似逆行列	Polynomial, Chebyshev	○	SOR より弱いか同程度
	AINV	○	ILU より弱いか同程度。
	FSAI	○	
	bi-conjugation	△	

6. 混合精度演算を用いた結果

数年前は倍精度演算器を備えた GPU ボードがなく、一方で GPU の単精度演算性能が高いことから、GPU 側で単精度演算を行い、CPU 側で倍精度演算を行う混合精度演算を用いた研究が行われている[15]。理論的なアイディアは CPU でもまだ倍精度演算が高価だった時代に既に提案されており[16]、近年再びその概念が注目されている。

$r = b - Ax$ (double precision)
do while($\|r\|/\|b\| > \varepsilon$){
  solve $Ay = r$ (single precision)
  $x = x + y$ (double precision)
  $r = r - Ay$ (double precision)
}

図 5 混合精度演算アルゴリズム[16]

表 2 多項式前処理･混合精度演算による比較

演算精度	ポワソン方程式 自由度:76,727 (CG)		電流連続方程式 自由度:40,080 (BiCGSTAB)	
	反復回数	計算時間	反復回数	計算時間
CPU 倍精度	423	4.92 秒	380	4.17 秒
CPU-GPU 混合精度	465	0.81 秒	1,341	0.86 秒

(CPU:Core2Duo6300, GPU:GeForce9800 GT)

現在においても廉価な GPU では倍精度に対応しておらず､単精度演算のみが可能である｡また、倍精度対応のボードでも単精度演算の方が演算性能は高い。そこで、試しに Advance/DESSRT から生成される疎行列と 1 次の多項式前処理を用いて､GPU-CPU による単精度/倍精度の混合精度演算と CPU のみによるフル倍精度演算との比較を行った｡その結果が**表 2** に示されている｡ここで、GPU として単精度演算のみが可能な NVIDIA Geforce 9800GT を用いた｡また CPU もごくありふれた Intel Core2 Duo 6300 を利用している｡この結果では、ポワソン方程式の反復回数が倍精度と混合精度で概ね同じであるのに対して、電流連続方程式では、約 3 倍の反復回数を要していることがわかる。このような違いは、行列の条件数に依存しており、高い条件数では、単精度/倍精度の混合精度アルゴリズムを用いた場合、フル倍精度よりも収束性が大きく低下することが知られている[17]。**表 2** に示すように反復当たりの処理時間は混合精度演算が有利であり、安価な GPU でも演算速度自体は速いことがわかる。但し、収束性の観点でいえば、この方法は汎用シミュレータ向けのアルゴリズムではない。

7. 不完全 LU 前処理を用いる上での困難と Fermi によるベンチマーク

次に収束性と効率性に優れ、実用的なアプリケーションに広く使われている不完全 LU 分解を用いた計算事例について紹介する。この手法はしかし、前処理の過程において並列化の難しい逐次的な前進代入・後退代入を含んでいるという欠点を持っている。これを克服する 1 つの方法として、行列を生成する元になる解析領域を複数の小領域に分割し、個々の小領域の演算を各 CPU で計算する領域分割法と呼ばれる方法がある。この方法は並列化の効率を上げるため、ある程度の領域サイズが必要であり、クラスター型ワークステーションのような個々のコアのメモリリソースが大きな計算機環境に向いている。

その一方で GPU では、極めて多数のスレッドを同時に起動でき､超並列計算が可能である反面、各スレッドに割り振られるメモリ量に限りがあり、領域分割法による各小領域の演算を GPU のスレッドに割り当てることには適していない。この他に、行列要素の並び替えを変更することで前進・後退代入の並列化を実現する手法がある。この手法では、行列の節点を複数のグループに分割し、ループ処理を用いて順次各グループ内で並列計算を実行する（**図 6** 参照)。CUDA ではカーネル関数を用いてグループ内の並列化が実現できる。ここで各グループ内における計算は、行列・ベクトル積に相当する。グループ数が少ない程（つまり同じグループに含まれる節点数が多い程)､並列化の同期点が少なくなるため計算効率は上昇する。その一方で一般的に、グループ数が少ない程、共

役勾配法の収束性が悪化することが知られており[18]、その分多くの演算が必要になる。

ここで取り上げる並び替えの手法は、修正された Cuthill-Mckee オーダリング法[18](以下 MCM オーダリングと略記。) であり、収束性が最適となるグループの分割を自動で行うことができる。ここで重要な点は、この分割には行列 A の隣接グラフだけが使われる点にあり、メッシュの変化がないデバイスシミュレーションにおいては、初回のメッシュデータ読み込みだけこの並び替えを行えばいい点にある。

この並び替え手法を用いて倍精度演算対応ボード Tesla C1060 と Tesla C2050(Fermi)による計算を行った。その結果を**図 8** に示す。比較のため、単体の CPU による結果も載せておく。CPU のみの計算では、特に並び替えを行っておらず、メッシュデータの読み込みによって定まる自然な順序づけが使われている。ベンチマークに用いたダイオードの例では自然な順序づけと、MCM オーダリングによる共役勾配法の収束性は殆ど変わっていない。ベンチマークで用いた不完全 LU の共役勾配法は Advance/DESSERT のリリース版に搭載している次の対角 ILU(0)を用いた。

$$M = (D' + E)D'^{-1}(D' + F) \qquad (6)$$

ここで D' は、不完全 LU 分解の過程で容易に得ることができる対角行列であり、E,F はそれぞれ行列 A の上三角、下三角行列である。この形を用いると、Eisenstat trick により、共役勾配法の反復における前処理時間を隠蔽することができる[9]。また、D' を求める際に次の対角シフト処理

$$A \leftarrow A + \alpha\,\mathrm{diag}(A) \qquad (7)$$

を行うことで収束が加速される[19]。ここで α～0.01 程度の定数である。CPU として、8MB ものキャッシュを積んだ Intel Xeon 5540 を使用した（**表 3** 参照)。

CPU と GPU の結果を比較すると、メッシュデータの読み込みや内部幾何処理、行列要素生成、GPU へのデータ転送などのオーバーヘッドを含めた全体の計算時間は、CPU のみよりも GPU を用いた方が数倍速い結果が得られている。また、各方程式の計算時間を抜き出して比較した結果を**図 9** に示す。共役勾配法の 1 ステップ当たりの前処理回数は、ポワソン方程式で 1 回、電流連続方程式で 2 回であり、後者の方が GPU による並列化の恩恵を受けやすい結果となっている。

```
forward_substitution () {
  for( i=0; i < ngroups; i++ ){
    parallel_group_kernel ( i );
  }
}
```

図 6 オーダリング手法による前進代入の並列化擬コード

表 3 倍精度ベンチマークの内容

項目	内容
ベンチマーク対象	N-I-P ダイオード(図 7)
解変数	電場、電子・正孔濃度分布
解法	ICCG(ポワソン方程式)、ILU-BICGSTAB(電流連続方程式)
オーダリング法	CPU:ナチュラルオーダリング、 GPU:修正された Cuthill-Mckee オーダリング
計算環境*	CPU：Xeon E5540 (2.53GHz), GPU：NVIDIA Tesla C1060, C2050
演算器	CPU,GPU 共に倍精度演算器を使用
線形方程式の収束条件	デバイスシミュレーションの実用精度に合わせた（ポワソン方程式：1E-15, 電流連続方程式：1E-20）。

(*ハードウェア提供：ソルテック・ソリューションデザイン株式会社 殿、株式会社エルザジャパン 殿)

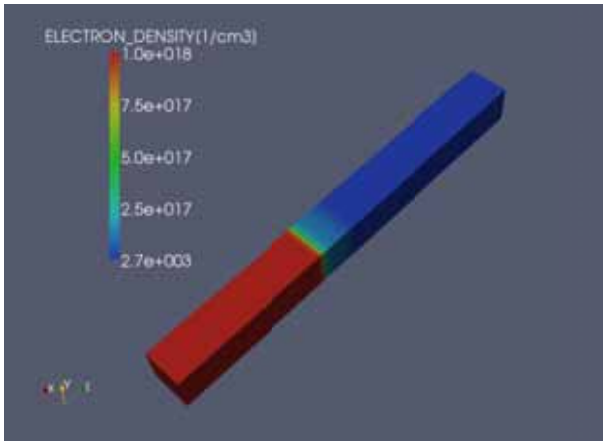


図 7 N-I-P ダイオード

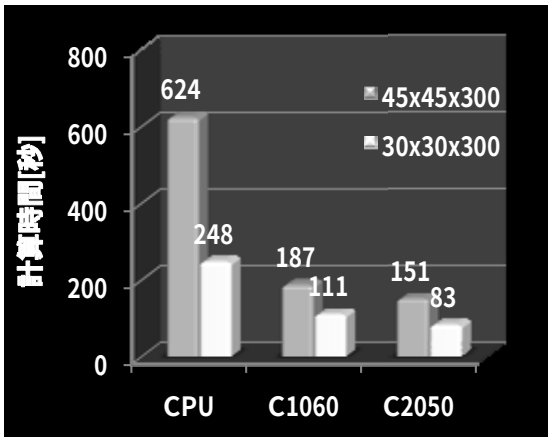
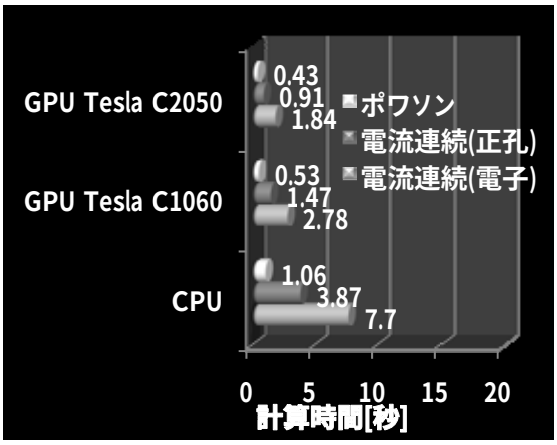


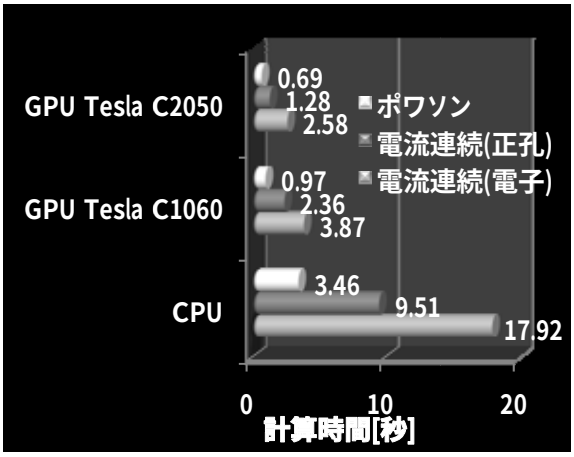
図 8 初回のバイアスステップの全計算時間（但し、幾何形状読み込み等の起動時間を含む。）

行列サイズに対する計算時間の依存性を比較した結果を**図 10** に示す。共役勾配法の演算量は、概ね行列サイズ Nの 1.5 乗に比例するといわれており、図中の破線は、計算時間を $\alpha \times N^{1.5}$ と推定し、α を節点数 30x30x300 における計算時間で求めた曲線である。CPU のみによる計算では、45x45x300 の結果も概ね推定曲線上にあり、演算量と計算時間との対応が取れている。一方、GPU を用いた 45x45x300 の計算時間は、推定曲線よりも下回っており、単純に演算量と計算時間との対応が成りたっておらず、行列サイズが増大するとパフォーマンスが向上する結果が得られている。この要因は、ベンチマークに用いられた MCM 法に起因しており、このオーダリングにおけるグループ数の分布がパフォーマンスに重要な影響を与える。

GPU のカーネル関数の実行には、①カーネルの起動、②本体の計算、③スレッド同期処理が必要であるが[20]、③の処理はオーバーヘッドとして避けがたく、とりわけ MCM オーダリングを用いた場合では、必要なグループ数の多さから共役勾配法の 1 ステップ当たり多数のカーネル関数の実行が必要になる。ベンチマークで用いられた対象に対するグループ数の分布は、**図 11** に示す通りであり、節点数 30x30x300 と 45x45x300 に対するグループ数はそれぞれ 356 と 385 である。行列のサイズが 2 倍以上になっているが、グループ内の節点数が増えているため、グループ数自体はさほど増大していない。これによりカーネルの実行回数の増加が抑えられ、それほど計算時間を要しないことがわかる。



(a)30x30x300



(b)45x45x300

図 9 初回のガンメルステップにおける各方程式の線形解法時間（但し、各線形ソルバーの係数行列生成等の起動時間を除く。）

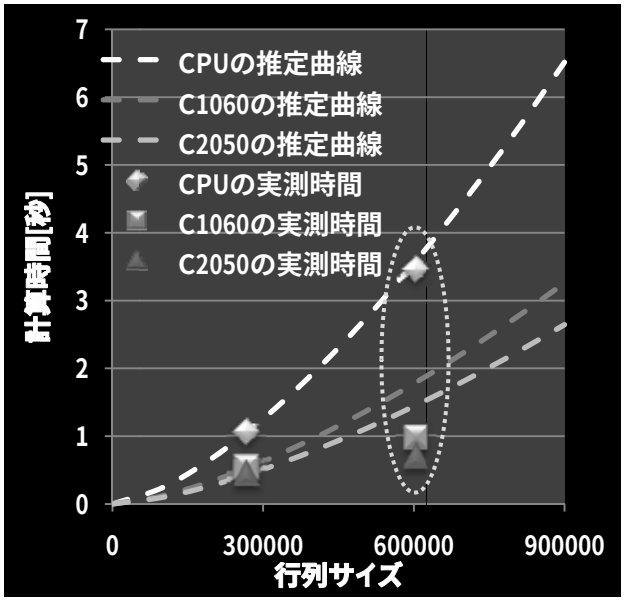


図 10 ポアソン方程式の行列サイズと計算時間の依存性

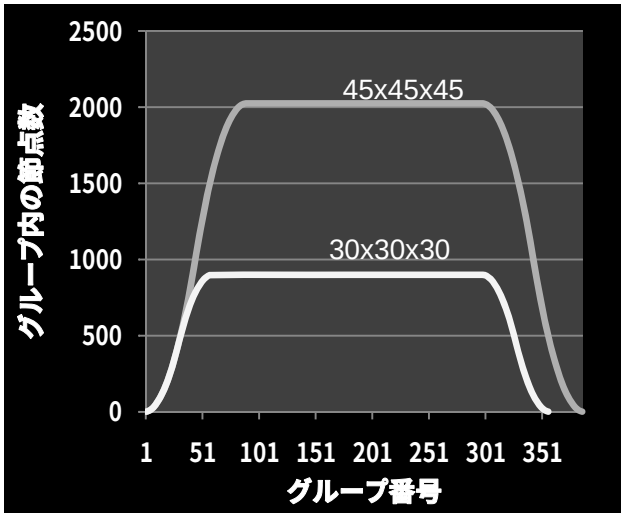


図 11 グループ内の節点数分布

8. おわりに

今回の計算結果では、Tesla C1060 に対する C2050 の計算時間は、その倍精度演算処理性能の向上と比べ、それほど低減していないように見える。これは、スレッド同期処理のオーバーヘッドが 1 つのボトルネックと考えられ、性能向上の阻害要因となっている。その意味で、今回用いたアルゴリズムは Fermi 本来の性能を十分に引き出している訳ではなく、更なる検討が必要である。勿論、実アプリケーションにおける GPU の適用は始まったばかりであり、今後、前処理付き共役勾配法の分野において、GPU に適した新しいアルゴリズムの研究・開発が進められるものと期待している。

参考文献

[1] NVIDIA TESLA C2050, URL:http://www.elsa-jp.co.jp/products/hpc/index.html

[2] NVIDIA GPU Computing Developer Home Page, URL:http://developer.nvidia.com/object/gpucomputing.html

[3] PGI CUDA Fortran Compiler, URL:http://www.pgroup.com/resources/cudafortran.htm

[4] 青木尊之, 額田彰, “はじめての CUDA プログラミング”, 工学社, 2009.

[5] David B. Kirk, Wen-mei W. Hwu, “Programming Massively Parallel Processors - A Hands-on Approach”, Morgan Kaufmann, 2009.

[6] アスキー・メディアワークス刊, ASCII.technologies, "特集：GPGPU プログラミング", 2010 年 8 月号.

[7] GPU コンピューティング 2010, URL:http://www.nv-event.jp/gpu-computing/

[8] GPU コンピューティング研究会, URL:http://gpu-computing.gsic.titech.ac.jp/index-j.html

[9] Yousef Saad, "Iterative Methods for Sparse Linear Systems", 2nd ed. SIAM, 2003.

[10] Nathan Bell, Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA", CUDA ZONE, 2008.

[11] F. Vazquez, et al. "The sparse matrix vector product GPU", 2009.

[12] NVIDIA, "GPU Computing SDK code samples", URL:http://developer.nvidia.com/object/cuda_3_1_downloads.html

[13] Michele Benzi, "Preconditioning Techniques for Large Linear Systems: A Survey", Journal of Computational Physics, vol. 182, pp. 418-477, 2002.

[14] Ulrich Trottenberg, Cornelis Oosterlee, Anton Schuller, “Multigrid”, Academic Press, 2001.

[15] Dominik Goddeke, Robert Strzodka, Stefan Turek, “Accelerating Double Precision FEM Simulations with GPUs”, In Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique, Sept. 2005.

[16] 戸川隼人, "共役勾配法", 教育出版, 1977.

[17] Serban Georgescu, Hiroshi Okuda, "Conjugate Gradients on Graphic Hardware: Performance & Feasibility", PARA 2008 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing.

[18] 中島研吾, "OpenMP によるプログラム入門(II)", 東京大学情報基盤センター・スーパーコンピューティングニュース , Nov. 2007.

[19] T. A. Manteuffel, "An Incomplete Factorization Technique for Positive Definite Linear Systems", Mathematics of Computation, vol. 34, pp. 473-497, 1980.

[20] Wu-chun Feng, Shucai Xiao, "To GPU Synchronize or Not GPU Synchronize?", NVIDIA GPU Computing Theater, 2010.