社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

GPUによるモーメント法の高速化

勝田　肇† 今野　佳祐† 陳　強† 澤谷　邦男†
横川　佳†† 袁　巧微††

† 東北大学大学院 工学研究科 電気通信工学専攻 〒 980-8579 仙台市青葉区荒巻字青葉 6-6-05
†† 仙台高等専門学校 〒 989-3128 宮城県仙台市青葉区愛子中央 4-16-1
E-mail: †katsuda@ecei.tohoku.ac.jp

あらまし　モーメント法を用いた大規模なアンテナの電磁界数値解析には, 多大な演算時間がかかるという問題がある. そこで, 本来は画像処理を担うプロセッサである GPU (Graphics Processing Units) を利用し, モーメント法を高速化する試みが多く行われている. しかし, GPU の各プロセッサへの演算の割り当てやモーメント法のアルゴリズムの差異がどの程度高速化に影響するのかを定量的に示した文献は少ない. 本報告では, これらの要素が GPU による高速化の効果に与える影響を定量的に検証したので報告する.

キーワード　モーメント法, GPU

A Study of Accelerated MoM Computation Using GPU

Hajime KATSUDA†, Keisuke KONNO†, Qiang CHEN†, Kunio SAWAYA†, Kei YOKOKAWA††,
and Qiaowei YUAN††

† Electrical and Communication Engineering, Graduate School of Engineering, Tohoku University 6-6-05 Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi, 980-8579, Japan
†† Sendai National College of Technology 4-16-1 Ayashichuuou, Aoba-ku, Sendai-shi, Miyagi, 989-3128 Japan
E-mail: †katsuda@ecei.tohoku.ac.jp

Abstract　In method of moments (MoM), cost of too much computing time becomes a serious problem for numerical analysis of large-scale antennas. In recent years, Graphics Processing Units (GPU), which is originally used for graphics processing, have been used for acceleration of the MoM computation. However, there has been few reports on allocation of numerical operation to each processor in GPU and algorithm of MoM affect computing time of GPU. In this report, relation among computing time of GPU, allocation of numerical operation to each processor in GPU and algorithm of MoM is quantitatively evaluated.

Key words　MoM, GPU

1. はじめに

電磁界数値解析の有力な手法の一つとしてモーメント法 (Method of Moments, MoM) が知られている [1], [2]. モーメント法では, アンテナや散乱体表面の電流分布を求める問題を, N 個の電気的に小さなセグメント上における, 未知の電流係数を求める問題に置換する. そして, 未知の電流係数を求めるために, 電界積分方程式を離散化して得られる行列方程式を解く. しかし, $N \times N$ 行列の演算に $O(N^2)$, 行列方程式を解いて未知の電流係数ベクトルを求めるために $O(N^3)$ の演算時間がかかる. したがって, 大規模なアンテナをモーメント法で解析するには高速化が必要である.

一方, 近年では, PC や WorkStation の画像処理を担うプロセッサである GPU (Graphics Processing Units) の高い並列演算能力を数値解析に応用する試みが盛んに行われている. 特に, GPU を汎用目的で利用するための統合開発環境である CUDA (Compute Unified Device Architecture) [3], [4] が 2006 年に NVIDIA 社より発表されて以降, GPU によってモーメント法を含む様々な数値解析法が高速化されてきた [5]- [8].

これまで, GPU によるモーメント法の高速化について, 様々な研究が行われてきた. 例えば, CPU 側の処理と GPU 側の処理をオーバーラップさせることによるさらなる高速化 [9], HDD を利用することによる取り扱い可能な未知数の上限の拡大の検討 [10], 複数の GPU を用いることによるさらなる高速化 [11]

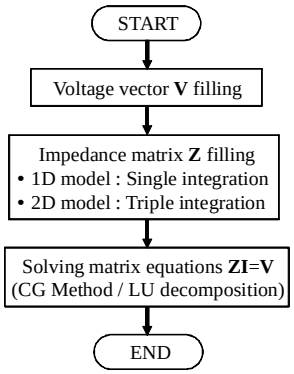

図 1　モーメント法のアルゴリズム.

などが挙げられる.

しかし, GPU の各プロセッサへの演算の割り当てを決定する Threads/Block というパラメータや, モーメント法におけるインピーダンス行列演算のアルゴリズムの差異が, GPU を用いたモーメント法の高速化に与える影響はあまり検討されていない. 本報告では, 自己・相互インピーダンスの演算に必要な積分次数が異なる線状導体, 面状導体において, Threads/Block 及びインピーダンス行列演算のアルゴリズムの違いが GPU によるモーメント法の高速化にどの程度影響するのかを定量的に検証したので報告する.

2.　モーメント法のアルゴリズム

モーメント法とは, アンテナや散乱体表面における電界積分方程式を行列方程式に離散化し, その行列方程式を解いて未知の電流係数を求めることで, アンテナや散乱体表面の電流を求める手法である. モーメント法のアルゴリズムを図 1 に示す. 図に示したアルゴリズムのうち, 未知の電流係数の数を N とすると, インピーダンス行列の演算には $O(N^2)$, 行列方程式を解くためには $O(N^3)$ の時間がかかる. したがって, N が非常に大きな問題では, 行列方程式を解く演算を高速化することが重要になる. しかし, 面状導体や誘電体など, インピーダンス行列の要素の演算に多重積分が含まれる問題では, インピーダンス行列の演算の高速化も非常に重要である.

インピーダンス行列の各要素はアンテナを分割して得られた N 個のセグメント間の自己・相互インピーンダンスを求めることによって得られる. 線状導体において, m 番目のセグメントと n 番目のセグメントの相互インピーダンスは一般的に以下の式で表される.

$$Z_{mn} = j\omega\mu_0 \iint_S \iint_S \mathbf{f}_m(\mathbf{r}) \cdot \vec{G}_0(\mathbf{r}, \mathbf{r}') \cdot \mathbf{f}_n(\mathbf{r}')\, d\mathbf{r}' d\mathbf{r} \tag{1}$$

ここで, $\mathbf{f}_m$, $\mathbf{f}_n$ はそれぞれ m 番目と n 番目のセグメントの基底関数, $\vec{G}_0$ は自由空間のダイアディックグリーン関数を表す. (1) 式より明らかなように線状導体のセグメント間の相互インピーダンスを求めるためには 4 重積分が必要となる. しかし, 細線近似を導入することで単積分にすることができる [2].

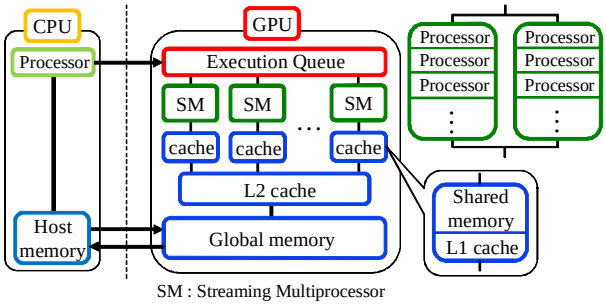

図 2　GPU の構造.

面状導体の場合のセグメント間の相互インピーンダンスは以下の式で表される.

$$Z_{mn} = \frac{1}{W^2} \int_0^W \int_0^W Z(u, v)\, du dv \tag{2}$$

ここで, W はセグメントの幅, $Z(u, v)$ は線状導体間の相互インピーダンスを表す. (1), (2) 式から明らかなように, 線状導体と面状導体ではインピーンダンス行列を求める際に必要な積分次数が異なっており, 線状導体では単積分であるのに対し, 面状導体では 3 重積分が必要となる. このため, 面状導体のインピーダンス行列を求めるには, 線状導体に比べ, 多大な時間がかかる.

本報告では, 線状導体, 面状導体の 2 種類のモデルのインピーダンス行列の演算の高速化を行い, 積分次数の違いが GPU による高速化にどの程度影響するのか検証する.

3.　GPU の構造と演算

3.1　GPU の構造

本報告で使用した GPU は Tesla C2075 であり, 図 2 に示した構造を持つ. 図 2 において, 左側は CPU 内のプロセッサと CPU が直接利用するメモリ (Host memory) を示している. そして右側は GPU 内のプロセッサ群と多層構造を持つメモリを示している.

GPU 内では, 多数のプロセッサが SM (Streaming Multiprocessor) という単位で管理されている. Tesla C2075 の場合は, 448 個のプロセッサが 14 個の SM に均等に振り分けられている. また, GPU 内には数種類のメモリがある. Host memory とデータのやり取りを行うメモリを Global memory, GPU のプロセッサに最も近く, 2 種類の機能を持つメモリを Shared memory / L1 cache, Global memory と Shared memory / L1 cache の中間に位置するメモリを L2 cache と呼ぶ. 容量は Global memory が最も大きく, Shared memory / L1 cache が最も小さい. データの転送速度は Shared memory / L1 cache が最も速く, Global memory が最も遅い.

GPU を用いた演算の流れを説明する. まず, Host memory から, 必要なデータを Global memory に転送する. 次に, CPU から, GPU へ実行させる演算を指示する. そして, GPU は与えられた演算を分割し (マルチスレッド化), 各 SM で並列に実行する. この際, 必要なデータは適宜メモリから呼び出す. 各プロセッサの演算が全て終了した後, Global memory から演算結果を Host memory に転送し, GPU を用いた演算は完了とな

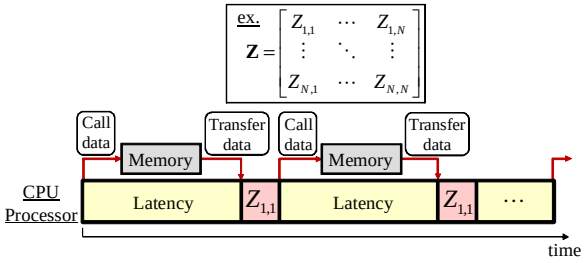

図 3　CPU による演算.

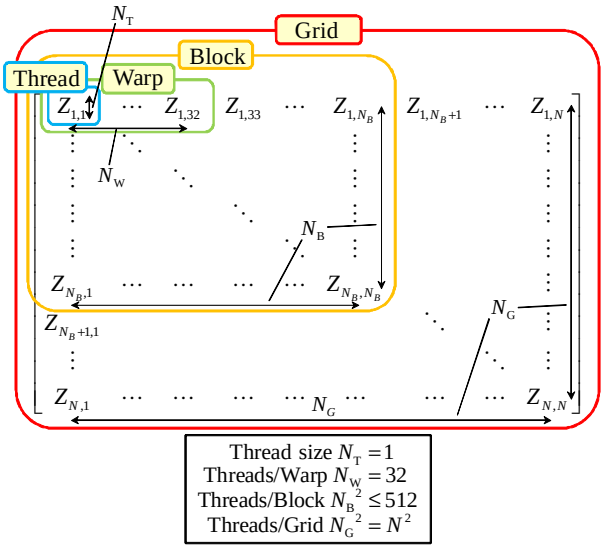

図 4　Grid, Block, Warp, Thread の関係.

る. なお, 以上の命令はすべて CUDA で書かれたコードによって行う.

GPU による演算では, プロセッサが無駄なく動作するよう演算をマルチスレッド化すること, データ転送が高速なメモリを有効に活用することが重要となる.

3.2　マルチスレッド化

CPU による演算の様子を図 3 に示す. CPU による演算では, まず $Z_{1,1}$ の演算のために必要なデータをメモリから呼び出し, それを用いて演算を行う. 演算の途中に別のデータが必要になれば, 一時的に演算を中断し, メモリからデータを呼び出してから演算を再開する. この際, 演算自体にかかる時間は非常に短いが, メモリからのデータ転送に多大な時間を消費する. メモリからのデータ転送等によるこのような無駄な時間をレイテンシといい, メモリのデータ転送速度が CPU のクロック周波数に対して非常に遅いために生じる. CPU による演算に対し, GPU による演算では, レイテンシを減らすために, 高速なメモリの利用やマルチスレッド化が行われる.

GPU を用いたインピーダンス行列の演算を例に, GPU 内での演算のマルチスレッド化の様子を図 4 に示す. インピーダンス行列の演算を Grid, Grid を各 SM に割り当てるために分割したものを Block, SM 内で同時に実行される演算の括りを Warp, SM 内の 1 プロセッサに割り当てられる演算を Thread と呼ぶ. Grid 及び Block は 2 次元で定義する. 本報告では, 1 Thread がインピーダンス行列の 1 要素を求める演算に当たる.

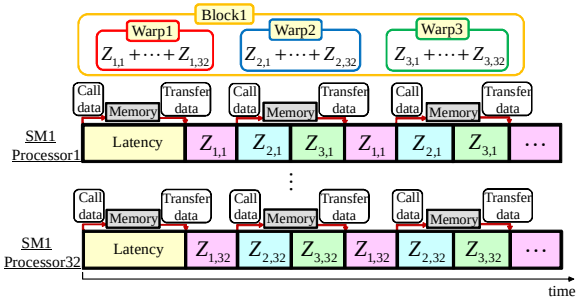

図 5　GPU 内の SM による演算.

GPU では, 各プロセッサが各 Thread を並列に演算するが, 各プロセッサは逐次的な動作しかできないため, (2) 式中の積分などを並列に演算にすることはできない. また, 1 Block 当たりの Thread 数 (Threads/Block) のみ, CUDA 上で任意の値を設定することができるが, 1 Block 当たり 512 Thread 以下という制約がある. 1 Warp は原則 32 Thread である.

マルチスレッド化された演算を, GPU 内の SM がどのように実行しているのか説明する. ここでは例として, SM1 に Block1 が割り当てられた場合の SM1 内のプロセッサの様子を図 5 に示す. Processor1 はまず, Warp1 内の Thread1, すなわち $Z_{1,1}$ のためのデータ呼び出し及び演算を行う. このとき, 同一 SM 内の他のプロセッサも同様に Warp1 の演算を行なっている. 演算の途中に別のデータが必要になれば, その都度データを呼び出す. ここで GPU では, レイテンシが生じないようにするために, Warp1 のためのデータ転送の間に, データ呼び出しが完了している Warp2 を実行する. Warp2 にもデータ転送が必要になれば, 次は Warp3 を実行する. このように, GPU では演算をマルチスレッド化し, 1 つの SM が多数の Warp を確保することで, プロセッサが常に動作するようにし, 高速化を実現する.

以上より, GPU を効率良く稼働させるには, 1 つの SM が多くの Warp を確保するように Threads/Block の値をなるべく大きな値に設定する必要があると考えられる. また, 1 Warp は 32 Thread であるから, Threads/Block を 32 の倍数にすることも重要であると考えられる.

4.　数 値 解 析

本報告における数値解析環境は表 1 の通りである.

表 1　数値解析環境.

CPU	Intel Xeon @ 2.27 GHz
GPU	NVIDIA Tesla C2075
Compiler	PGI Visual Fortran 12.4
CUDA Driver	CUDA 4.1
OS	Windows 7 Professional 64 bit

図 6 と図 7 に線状, 面状導体の数値解析モデルをそれぞれ示す. この 2 つのモデルについて, GPU によるインピーダンス行列の演算の高速化を検証する. なお, 本報告中では特に断りがない場合は, (2) 式中の積分は 2 重ガウス積分によって行い, 積分点数は 8×8 点としている.

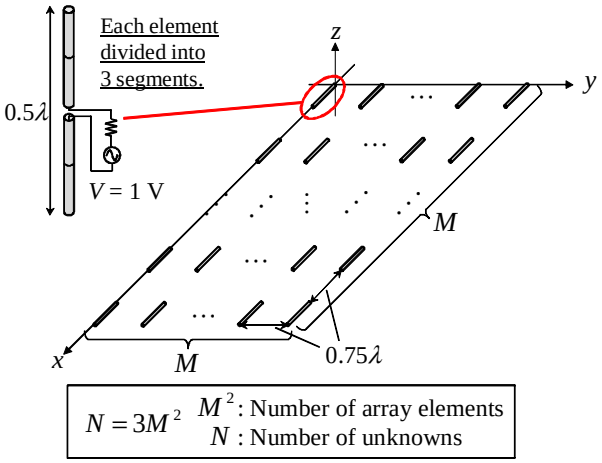

図 6　2 次元半波長ダイポールアレーアンテナ.

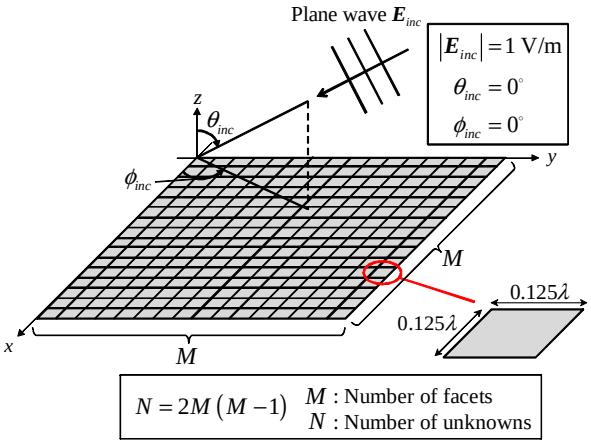

図 7　平面アンテナ.

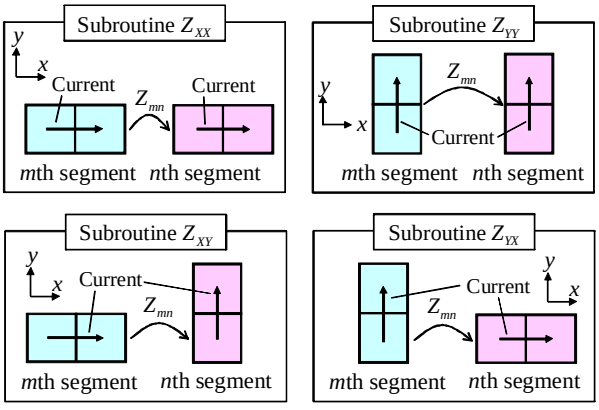

図 8　インピーダンス行列を求めるサブルーチン $(Z_{XX}, Z_{YY}, Z_{XY}, Z_{YX})$.

4.1　面状導体の自己・相互インピーダンス演算のアルゴリズム

本報告における面状導体の数値解析モデルでは, セグメントによって電流の流れる方向が異なる. 電流の流れる方向が同じセグメント間の相互インピーダンスは, 座標変換により, 積分次数を一つ下げることが可能である. したがって, 本報告では, 面状導体のインピーダンス行列を求めるサブルーチンを図 8 のように 4 つに分けた.

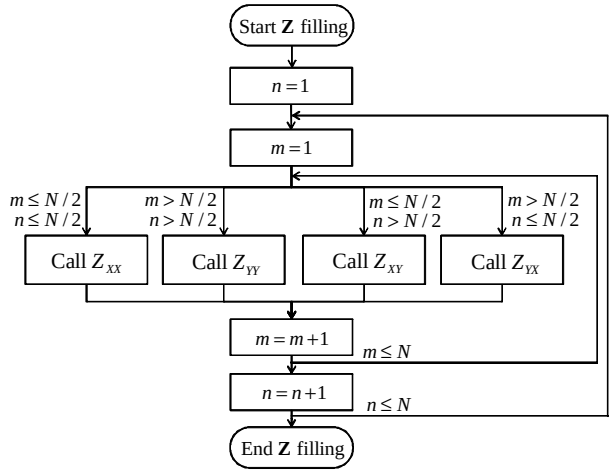

図 9　インピーダンス行列を求めるアルゴリズム (ループ内に分岐有).

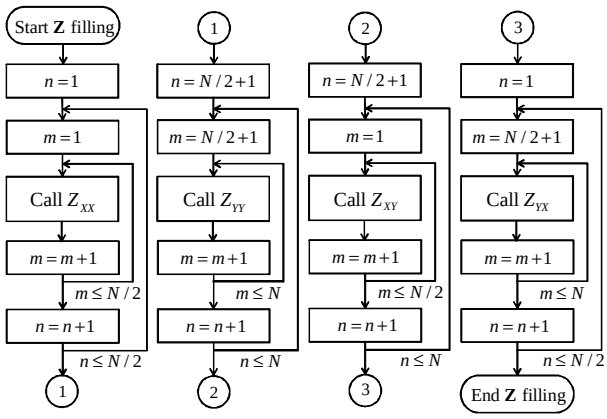

図 10　インピーダンス行列を求めるアルゴリズム (ループ内に分岐無).

この場合, 通常は図 9 に示したフローチャートのようにループ内でセグメントの番号に応じて分岐を行い, 演算を実行する. GPU で同様の演算を行う場合は, ループを展開し, 並列に演算を行うが, 基本的な考え方は同じである. しかし, GPU では, 一つの Warp 内ではすべての Thread がほぼ同一の演算を行うことが前提となっているため, ループ内に分岐を含む問題は不得手であると考えられる.

そこで, ループ内に分岐を含むコードと図 10 に示したフローチャートのようにループ内に分岐を含まないコードをそれぞれ作成し, 未知数の数に対する演算時間を測定した. その結果を図 11 に示す. ループ内に分岐を含むコードの方が, 含まないコードよりも演算時間が長くなることが分かる. これより, GPU を用いた高速化では, ループ内に分岐を含む複雑なコードは避けた方が良いと言える. 特に, 線状セグメント, 面状セグメントの両方を含むモデルの解析では, セグメント形状に合わせたサブルーチンを複数用意することになるため, ループ内に分岐を含まないよう心掛ける必要がある.

4.2　Threads/Block の最適化

$N \simeq 10^4$ 程度において, Threads/Block に対する演算時間を図 12, 13 に示す. 本報告では, 図 4 に示したように Block を 2 次元で定義しているので, Threads/Block は 1 から 22 までの整数値の 2 乗の値しかとることができない.

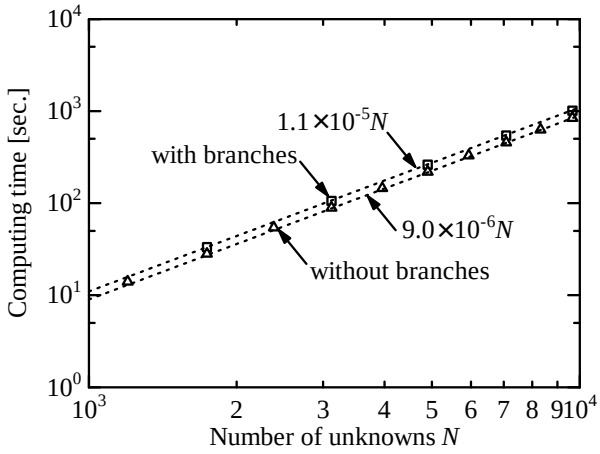

図 11　ループ内の分岐の有無に対する GPU を用いた演算時間.

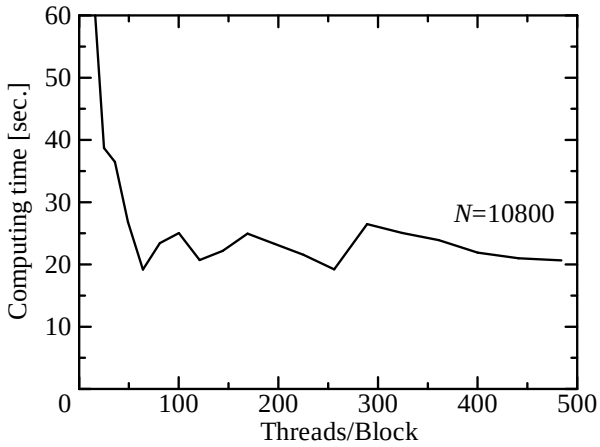

図 12　線状導体における Threads/Block に対する演算時間 ($N = 10800$).

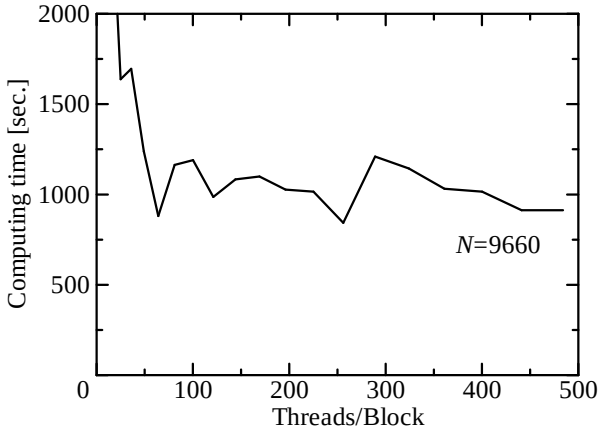

図 13　面状における Threads/Block に対する演算時間 ($N = 9660$).

図 12, 13 では共に Threads/Block が 256 のときに最も演算時間が短くなった．また, Threads/Block が 64 のときにも同じ程度演算時間が短くなった．これより, 前章で述べたように, Threads/Block は 32 の倍数かつなるべく大きな値, すなわちこの場合は 256 が望ましいということが確認された.

また, 図 14, 15 に, N に対する最も演算時間が短くなる

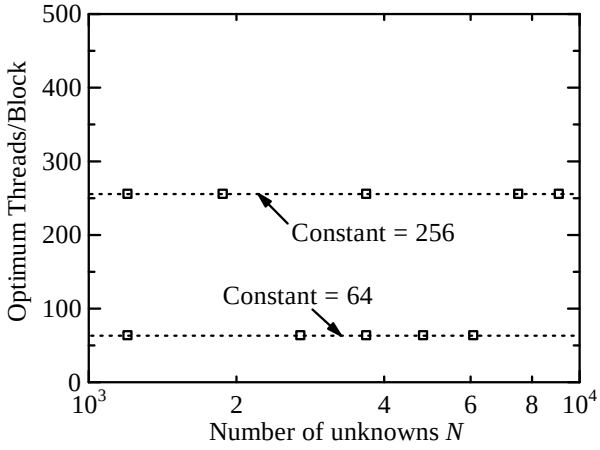

図 14　線状導体における最適な Threads/Block.

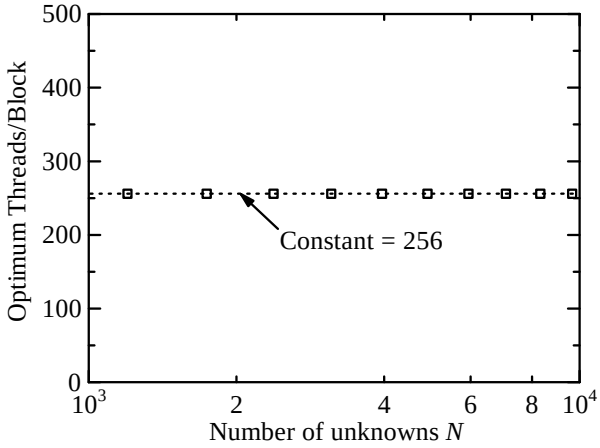

図 15　面状導体における最適な Threads/Block.

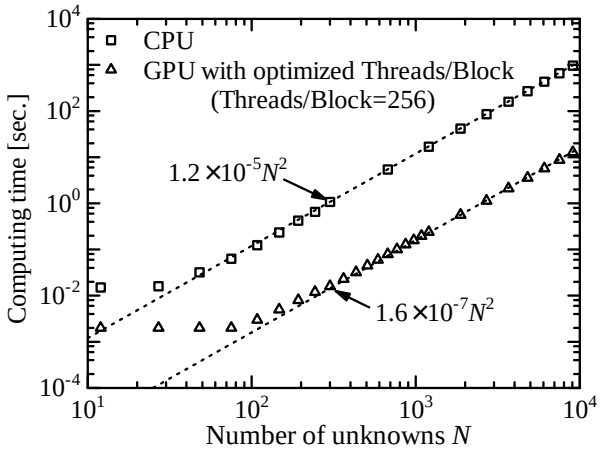

図 16　線状導体において最適な Threads/Block を選んだ場合の高速化の効果.

Threads/Block を示した．線状導体の場合は Threads/Block が 256 ではなく 64 のときに演算時間が最も短くなることもあるが, どちらの場合も演算時間に大きな差はない．従って, Threads/Block は 256 とすれば演算時間をほぼ最小にできることが分かる.

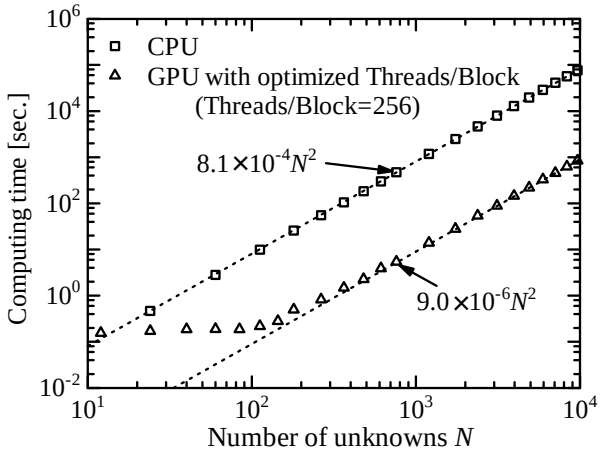

図 17　面状導体において最適な Threads/Block を選んだ場合の高速化の効果.

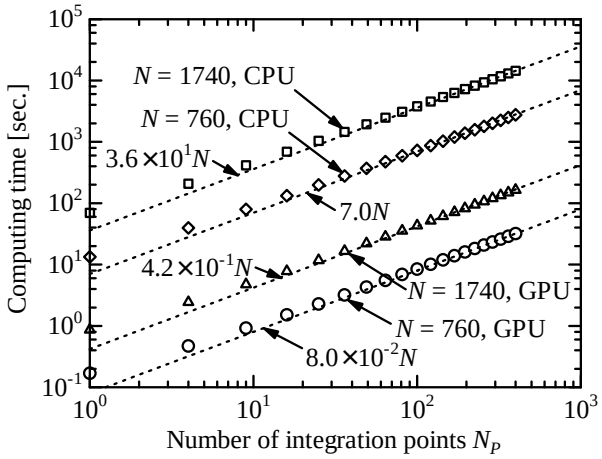

図 18　面状導体における積分点数に対する演算時間の変化.

4.3　線状導体と面状導体における高速化の効果

線状導体, 面状導体において, 最適な Threads/Block を選んだ場合に, GPU によってインピーダンス行列の演算がどの程度高速化できたのかを図 16, 17 にそれぞれ示す. GPU を用いた結果, 線状導体では約 75 倍, 面状導体では約 90 倍の高速化が達成されたことが分かる. これより, 面状導体の数値解析の場合の方が, GPU を用いた高速化の効果が大きいと言える. しかし, GPU では, インピーダンス行列演算における数値積分は並列化されず, 各プロセッサで逐次的に演算されるため, 積分次数によって高速化の効果は変化しないと考えられる.

そこで, 数値解析モデルの差異による高速化の効果の変化について検証するために, (2) 式中の数値積分を行う際の積分点数に対する演算時間を図 18 に示す. ここで, 積分点数 $N_P = 1$ の結果は, 線状導体の数値解析結果と等価である.

また, 図 18 における GPU を用いた高速化の効果を表 2 に示す. 表 2 中の S_{GPU} は GPU による高速化の効果を意味し, 以下の式で表される.

$$S_{\mathrm{GPU}} = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}} \tag{3}$$

ここで, t_{CPU} は CPU による演算時間, t_{GPU} は GPU による演算時間である.

表 2　積分点数に対する GPU を用いた高速化の効果.

N_P	1	4	9	100	400
S_{GPU} $(N = 760)$	78.9	84.0	86.2	86.1	86.1
S_{GPU} $(N = 1740)$	79.8	85.3	87.3	87.5	87.4

この高速化の効果は, 積分点数が低いときは約 80 倍であるが, 積分点数が増えるに従って徐々に大きくなり, 約 87 倍程度で飽和する. 以上の結果から, GPU によるインピーダンス行列演算の高速化の効果は積分点数と何らかの関係があることが分かるが, その原因は今後の検証が必要である.

5. むすび

本報告では, Threads/Block の変化が GPU によるインピーダンス行列の演算の高速化の効果に与える影響を定量的に明らかにした. そして最適な Threads/Blcok を選んだ GPU を用いることによって, CPU のみによる演算に対し, 線状導体の場合は約 75 倍, 面状導体の場合は約 90 倍の高速化が達成された. 積分点数と高速化の効果の関係は, 今後の検討を要する. また, GPU を用いた演算では, ループ内に分岐を含む複雑なアルゴリズムは望ましくないということも確認した.

文　献

[1] R.F. Harrington, Field Computation by Moment Methods, New York, Macmillan, 1968.

[2] J.H. Richmond and N.H. Greay, “Mutual impedance of nonplanar-skew sinusoidal dipoles,” IEEE Trans. Antennas and Propag., vol.23, no.5, pp.412-414, May 1975.

[3] NVIDIA Corporation, “CUDA Zone - The Resource for CUDA Developers,” 2012, available at http://www.nvidia.com/cuda.

[4] SofTek Corporation, “ソフテック PGI テクニカル情報・コラム,” available at http://www.softek.co.jp/SPG/Pgi/TIPS/para_guide.html.

[5] S. Peng and Z. Nie, “Acceleration of the Method of Moments Calculations by Using Graphics Processing Units,” IEEE Trans. Antennas and Propag., vol.56, no.7, pp.2130-2133, July 2008.

[6] E. Lezar and D.B. Davidson, “GPU-Accelerated Method of Moments by Example: Monostatic Scattering, ” IEEE Antennas and Propag. Magazine, vol.52, no.6, December 2010.

[7] A. Balevic et al., “Accelerating Simulations of Light Scattering Based on Finite-Difference Time-Domain Method with General Purpose GPUs, ” IEEE International Conference on Computational Science and Engineering, pp.327-334, July 2008.

[8] D.D. Donno et al., “Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD,” IEEE Antennas and Propag. Magazine, vol.52, no.3, June 2010.

[9] T. Topa, A. Noga and A. Karwowski, “Adapting MoM With RWG Basis Functions to GPU Technology Using CUDA,” IEEE Antennas and Wireless Propag. Letters, vol.10, 2011

[10] E. Lezar and D.B. Davidson, “GPU-based LU Decomposition for Large Method of Moments Problems,” Electronics Letters, vol.46, no.17, August 19, 2010.

[11] D.P. Zoric, D.I. Olcan, B.M. Kolundzija, “Solving Electrically Large EM Problems by Using Out-of-Core Solver Accelerated with Multiple Graphical Processing Units,” IEEE International Symposium on Antennas and Propag., pp.3-8, July 2011.