光ファイバーの特性の効率的計算

量子・物質工学専攻　植田研究室　柳谷 貴紀

平成 23 年 3 月 4 日

1 Introduction

光（電磁波）とは、狭義には Maxwell's equations(ME) に従う波である。その特性の解析には様々な手法が用いられる。例えば、特殊な光ファイバーである Photonic Bandgap Fiber(PBGF) の解析には、ME を PBGF の誘電率の周期性等で簡略化する Plane Wave Expansion method(PWE) 等が用いられる。この手法は ME を最終的に行列の固有値問題に変え、計算は比較的高速である。しかし、解析の対象となる物体の誘電率分布は無限に周期的であること等を仮定しており、そうでない場合には解析結果の精度が低いか全く解析できないことになる。

一方、より柔軟な解析手法として Finite Difference Time Domain method(FDTD) がある。こちらは、解析の対象が有限であれば、誘電率分布がランダムであっても計算可能である。PWE と異なり解析対象に要求される条件（周期性、一様性等）が緩く、代わりに計算に時間のかかる（精度が要求される場合は特に）手法である。

FDTD の計算は、単純に実装する場合は四則演算のうち加減算および乗算が大部分を占める。計算装置として Graphics Processing Unit(GPU) を用いる場合、それら３種の演算は大部分がハードウェア上に native に実装されており (native instruction, NI)、倍精度であれば 2 clocks 程度で計算可能である。この装置を用いる場合、「計算」にかかる時間の内、上記のような純粋な「計算」にかかる時間よりも、「計算」に必要な情報の「転送」により多くの時間を要する。つまり、長時間の計算の内、演算器が「演算」している時間よりも、演算器が「演算」に必要な情報の「転送」を待っている時間の方が長い。

このため、FDTD のような転送速度により全体の計算時間が支配されるような計算では、演算器の性能を上げても「効率[1]」はさほど向上しないため、以下のような手法をとる必要がある。

- 演算器が情報の転送を待っている間、演算器の消費電力のみを削減する
- 演算器が情報の転送を待っている間、情報の転送量を削減するために情報の可逆圧縮[2]を行う

[1]ここでは、単位消費電力当りの演算能力 (MFlops/W) とする。
[2]圧縮・展開の際に、情報が欠損しない。

表 1: The machine spec.

CPU	
Product Name	Intel Core i7 920
Frequency [GHz]	2.66
#Cores(real, virtual)	4, 8
Memory Size [GB]	3
Memory Bandwidth [GB/s]	32
GPU	
Product Name	nVidia Geforce GTX 480
Frequency [GHz]	1.40
#Cores	480
Memory Bandwidth [GB/s]	177.6
Memory Size [GB]	1.5
Max GFlops [GFlops]	672

前者の手法は、GPU メーカー側に可能な手法で、利用者側にできることではないが、将来的に行われることとなっている。後者の手法は、さらに以下のように分類できる。

1. 特定の情報の圧縮をハードウェアで自動的に行う
2. GPU 間の情報の転送[3]の際に情報の圧縮を行う
3. GPU 内部での情報の転送の際に情報の圧縮を行う

最初の手法は、やはり利用者側にできることではないが、すでに行われていることである。次の手法は、利用者側で既に行われた例がある。FDTD でも、規模がある程度以上（GPU 単体の記憶容量以上）であれば行う必要がある。最後の手法は、FDTD では行われた例がない。

本論文ではこの手法を用いた場合の性能向上率を評価する。

尚、圧縮を用いる場合の演算量の増大は現在の装置では全く無視できないものであり、できる限り圧縮を用いる場合がより優位になるようにした。よって、実用的には圧縮を用いた場合はより不利になる。

2 装置

計算機の概要を**表 1** に記載した。

[3]現在、この種の転送時間は GPU 内部の最も遅いものと比べても数倍〜十倍程度遅い。

計算は主に GPU で行うため、CPU については詳説しない。

前述のように、GPU は倍精度浮動小数点演算の加減算、乗算を 2 clocks で実行できる。また、乗算＋加算 ($a = bc + d$, MAD) も 2 clocks で実行できる。int の乗算や乗算＋加算、シフト演算、SAD($a = |b - c| + d$, Sum of absolute difference) も同様である。加算については 1 clock で実行できる。型変換には 2 clocks を要する。

上記に無い除算や初等関数等は、基本的に複数の NI によって構成されており、計算負荷が高い。このため、圧縮等にはできるかぎり NI を直接使うような手法をとった。

次に GPU の記憶装置 (Memory) について説明する。Memory には以下のようなものがある。

1. Global Memory: CPU の Memory に対応する GPU の Memory。GPU 内部では最も遅い。後述するより高速な L1、L2 Cache Memory に要求する情報がある場合はそちらから読み込まれ、転送時間は短縮される。L1 Cache Memory と同様に、1 度に 128 Bytes (1024 bits) 読み込む。

2. L2 Cache Memory: Global Memory より高速で、L1 Cache Memory より低速な Cache Memory. 容量は Global Memory の 0.1% 程度。ハードウェアにより自動的に Global Memory の転送を効率化する。L1 Cache Memory 等と異なり、一度に 32 Bytes のみ読み込むため、情報が散逸している場合はこちらのみを使う方が効率的な場合がある。

3. L1 Cache Memory: L2 Cache Memory より高速だが、一度に 128 Bytes 読み込むため情報が散逸している場合は無駄な読み込みが発生する。また容量は後述するように可変ではあるが、L2 の 10% 程度であり、さらに小容量である。その他の点については L2 Cache Memory と同様。後述の Shared Memory と同一の装置であり、L1 Cache Memory と Shared Memory とで割合を変更することができる。詳細は後述。

4. Shared Memory: 明示的に Global Memory から読み書きをする高速・小容量メモリ。即ち、ハードウェアにより自動的に使用されることはない。明示的に使用する際、読み込む順序等に制限があり、その場合は速度が低下する（最大で数十倍程度）。事前に頻繁に使用する情報が予測できる場合はこちらを用い、できない場合は L1 Cache Memory を用いる。その場合は L1 Cache Memory を 48 kB に、Shared Memory を 16 kB に、そうでない場合はその逆に設定する。一度に 4 Bytes のみ読み込むため、情報がミクロに散逸していて、しかしマクロにはそうでない場合は一度まとめてこの Memory に転送しておくと転送が効率化できる。

5. Registers: 計算の並列度が十分に高い場合、この記憶装置の読み書きに必要な時間は隠蔽（不可視）される。演算に大量に必要とされる場合は Global Memory が代用され、速度低下の要因となる。

上記仕様で重要なのは、Global Memory 等は一度に 128 Bytes もの情報を扱うということである。これは倍精度浮動小数点 16 個分に相当し、またそれら上位 8 bits の 128 個分に相当する。圧縮の目的は情報の転送の削減であるが、128 Bytes 未満の削減では結局転送回数は 1 回も減らないため、大幅な転送時間短縮はできない。

GPU には CPU と比較して大量の Core がある。前者のものは単純であり、最大の「効率」は高いが、複雑な処理（条件分岐等）をする場合は最大で 数十倍程度遅くなる。後者のものは複雑であり、複雑な処理でも性能低下の度合いは低い。

前者のものには以下のような階層構造がある。

1. 1 GPU = 15 Multiprocessors: Multiprocessor の数はデバイスに依存する。最新のものでは 1 GPU = 16 Multiprocessors である。L2 Cache Memory を 1 つ持つ（すべての Multiprocessors で共有）。

2. 1 Multiprocessor has up to 8 blocks: Block の数はデバイスに依存する。Multiprocessor は 処理中の 1 個の block で「同期」が発生した場合、待ち時間を隠蔽するために他の block の処理に切り替える。$16 \times 2 = 32$ 個の Core を持ち、後述する 1 half-warp を同時に 2 つ実行する。L1 Cache Memory と Shared Memory を 1 つ持つ。その容量は block の数で等分される。

3. 1 block has up to 6 warps: Warp の数はデバイスに依存する。異なる warp でのみ分岐が発生し、warp 内で分岐が発生しない場合は前述の分岐による性能低下は起こらない。

4. 1 Warp = 32 threads = 2 half-warps: 現在 thread の数はデバイス非依存である。本研究で用いるデバイスでは、1 Multiprocessor は 2 warp schedulers を持ち、それぞれが 1 half-warp を実行する。ただし、倍精度浮動小数点演算においては 1 Multiprocessor は 1 warp scheduler のみ作動でき、単精度浮動小数点演算の半分の性能となる。

まとめると、1 Multiprocessor は一度に 1 warp = 32 threads を処理し、それが 15 個ある。すなわち、$15 \times 32 =$

480 threads を同時に処理できる。1 Multiprocessor に最大の thread 数を割り当てる場合、すなわち最大の blocks / multiprocessor と最大の warps / block を割り当てる場合、thread 数は 32 threads / warp、6 warps / block、8 blocks / multiprocessor の積 $32 \cdot 6 \cdot 8 = 1536$ となる。

圧縮を伴う FDTD では、情報は散逸しないように格納することもできるため、前述した Shared Memory の方の容量を優先して 48 kB とする。

この場合、1 Multiprocessor 内では 48 kB の Shared Memory が 8 blocks で等分され、1 block あたり 6 kB となる。これは倍精度浮動小数点数 768 個分に相当する。2 つの 3 次元ベクトルと 2 つのスカラーを 1 セットとすると、96 セット分に相当する。

3 原理

3.1 一般

Maxwell's equations の空間微分や時間微分には除算が含まれているが、これらは誘電率や透磁率を予め除しておくことで回避できる。

前述のように圧縮はできるかぎり集中的に行う必要があるため、また「効率」の比較において圧縮を行う方をより優位にするため、情報の格納の順番を圧縮用に最適化し、計算領域はなるべく 128 の整数倍となるようにした。

情報の圧縮方法には様々な手法があるが、多くの実用化されている圧縮法は圧縮率優先で圧縮の際の計算負荷が高く、それを FDTD の計算中に行うと Memory の転送速度の遅さよりもさらに遅くなってしまうため、本研究では極めて単純で計算負荷の軽い方法を用いた。

FDTD に限らず、多くの物理のシミュレーションの問題では隣接する情報は急激には変化しない（微分が有限である）。そのため、隣接する倍精度浮動小数点数の上位 bits は似た値をとり、その差分を integer の減算により求めるとその integer の上位 bits には 0 が多く含まれるようになる。差分には様々な求め方があるが、例として倍精度浮動小数点数の sign bit を除く上位 16 bits の差分を 32 bit integer の減算により求めると、その integer の上位 24 bits は 0 となりやすい。単純な Waveguide の例では、ほぼ 100 % の領域で上位 24 bits は 0 となった。元の情報量は 16 bits であるから、この場合 倍精度浮動小数点数 1 個あたり 8 bits 分だけ圧縮により情報量を減らせることになる。本研究で用いる装置では、倍精度浮動小数点数 64 bits の読み書きに 30 clocks 程度要するため、8 bits では 3.75 clocks 時間を短縮できることになる。すなわち、圧縮・展開に要する余分な演算量の増加を無視したときの性能向上の理論最大値は $30/(30-3.75) = 114\%$ となる。実際には情報転送以外の演算や、圧縮・展開に要する演算に要する時間も有限であるため、性能向上はこれより低くなる。特に圧縮・展開により生じる余分な演算量は次に述べるように倍精度浮動小数点数 1 個あたり $3.75 \times 2 = 7.5$ clocks を超えているため、現在の情報転送速度と演算速度との比率（およそ 15:1）では性能はむしろ下がる。

圧縮は次のように行う。

1. 演算結果の倍精度浮動小数点数のうち、上位 32 bits を 32 bit integer として register に代入する: 6 clocks
2. integer の絶対値をとる: 2 clocks
3. integer を右に 16 bits シフトする: 2 clocks
4. 上の手順を隣接する倍精度浮動小数点数についても行い、差分をとる: 2 clocks

sign bit の扱いを省略しても、圧縮だけで 12 clocks 要する。

展開は以下のように行う。

1. 倍精度浮動小数点数の下位 32 bits を 32 bit integer i_1 に読み込む: 0 clocks
2. 倍精度浮動小数点数の上位 32 bits の内の下位 16 bits を 32 bit integer i_2 に読み込む: 0 clocks
3. 差分の基準となる上位 16 bits を 32 bit integer i_3 に読み込む: 0 clocks
4. 差分の基準以外において差分の下位 8 bits を 32 bit integer i_4 に読み込む: 0 clocks
5. $i' = i_3 + i_4$ を新たな差分の基準とする: 1 clock
6. $i'' = i'$ を左に 16 bits シフトする: 2 clocks
7. i_2 を i'' に加える: 1 clock
8. unsigned long long int i_2' に i_2 を代入する: 0 clocks
9. i_2' を左に 32 bits シフトする: more than 2 clocks, assuming 4 clocks
10. i_2' に i_1 を加える: more than 3 clocks, assuming 4 clocks

ここでも sign bit の扱いについては省略したが、それでも展開だけで 12 clocks かかる。なお、以上において圧縮できるかどうかのチェックは一切していないが、Waveguide のような例では全領域において圧縮可能であるためチェックに要する時間については無視する。

以上のように、圧縮・展開あわせて最低でも 24 clocks 必要であり、これは 8 bits の読み書きに必要な 7.5 clocks のおよそ 3 倍となるため、前述のように性能は低下する。

この方法でわずかにでも性能向上を求めるためには、現在よりもさらに 4 倍程度情報転送速度と演算速度との比率が開く必要がある。

以上の方法は、結局のところ FDTD にも光学分野にも限る必要はなく、必要な条件は隣接する情報間の差のオーダーが一定値以下であることのみである。

3.2 誘電率の圧縮

FDTD で光ファイバーを扱う場合は、誘電率の取りうる値が限られているためより効率的な圧縮が可能である。

比誘電率 ε_{r} は

$$1.0 \leq \varepsilon_{\mathrm{r}} \leq 32768.0 = 2^{15} \tag{1}$$

を満たす場合が多い。この範囲を倍精度浮動小数点で表すと

`0x3ff0-0000-0000-0000` から `0x40e0-0000-0000-0000`

となる。これに 2 を乗じると `0x4000-0000-0000-0000` から `0x40f0-0000-0000-0000` となり、上位 8 bits はともに `0x40` となる。つまり、$2\varepsilon_{\mathrm{r}}$ については上位 8 bits は `0x40` となるから、その部分については格納する必要も読み込む必要もなくなり、それ以外の 56 bits のみ読み込めば良いこととなる。$2\varepsilon_{\mathrm{r}}$ ではなく ε_{r} が必要な場合は、計算に定数が含まれる場合はそれに 2 を乗じるか除すれば $2\varepsilon_{\mathrm{r}}$ から ε_{r} を復元するのに余分な演算は発生しない。そうでなくても、2 の累乗の乗算や除算は倍精度浮動小数点の指数部分の加減算で代用できるため 2 clocks ではなく 1 clock で実行できる。同様にして、$\varepsilon \equiv \varepsilon_0\varepsilon_{\mathrm{r}}$ は `0x3da3-7876-f14d-ed31` から `0x3e93-7876-f14d-ed31` の範囲であり、それに 2^6 を乗じると `0x3e03-7876-f14d-ed31` から `0x3ef3-7876-f14d-ed31` となり、ともに上位 8 bits は `0x3e` となる。ε^{-1} は `0x423a-4bcd-78b3-d090` から `0x414a-4bcd-78b3-d090`、これに 2^{-4} を乗じて `0x41fa-4bcd-78b3-d090` から `0x410a-4bcd-78b3-d090`、上位 8 bits は `0x41` となる。

64 bits をそのまま読み込む場合、転送時間以外の時間は 0 である。56 bits のみ読み込む場合、前述のように 128 bytes 単位以外での読み込みは効率が悪いため、shared memory に 128 bytes 単位で読み込みそこからさらに 32 bits 単位で読み込む。$56 \times 4 = 8 \cdot 4 + 16 \cdot 4 + 32 \cdot 4 = 32 + 32 \cdot 2 + 32 \cdot 4$ であるから、誘電率は 4 つセットで読み込む。この場合の擬似コードは以下のようになる。

```
// %REG* is a 64-bit register.
// %Reg* is a 32-bit register.
// %reg* is a 16-bit register.

// 4 registers.
// Each register has least significant 32
//   bits of a double precision floating
//   point number.
mov.b32 %Reg32A, 0xGHIJ-KLMN;
mov.b32 %Reg32B, 0xUVWX-YZab;
mov.b32 %Reg32C, 0xijkl-mnop;
mov.b32 %Reg32D, 0xwxyz-@?=*;

// 2 registers.
// Each register consists of two 16-bit
//   parts.
//
// A 16-bit part has least significant 16
//   bits of most significant 32 bits of
//   a double precision floating point
//   number.
mov.b32 %Reg16AB, 0xCDEF-QRST;
mov.b32 %Reg16CD, 0xefgh-stuv;

// 1 register with four 8-bit parts.
// An 8-bit part has least significant 8
//   bits of most significant 16 bits of a
//   double precision floating point
//   number.
mov.b32 %Reg8ABCD, 0xABOP-cdqr

// Unpack 2 registers with four 16-bit
//   parts and store the results in 4
//   registers.
// Data movement instruction(mov) only.
//
mov.b32 {%reg16B, %reg16A}, %Reg16AB;
  // 0xQRST, 0xCDEF <- 0xCDEF-QRST
mov.b32 {%reg16D, %reg16C}, %Reg16CD;
  // 0xstuv, 0xefgh <- 0xefgh-stuv

// Unpack a register with four 8-bit parts
//   and store the intermediate results in
//   2 registers, each of which has two
//   8-bit parts.
// These 2 registers also should be
//   unpacked later.
mov.b32 {%reg8CD, %reg8AB}, %Reg8ABCD;
  // 0xcdqr, 0xABOP <- 0xABOP-cdqr

// Unpack 2 registers with four 8-bit
//   parts and store the results in 4
//   registers.
// Since current multiprocessor does not
//   have a native instruction which
//   unpacks a 16-bit register to two
//   8-bit registers, such kind of
//   functionality must be implemented by
//   multiple native instructions.
// The use of add.s32, mad.lo.s32 and
//   shl.b32 was chosen to minimize
//   extra computation time, though this
//   implementation has both of two kinds
//   of functionality -- that is, two
```

```
//   kinds of functionality was combined:
//   (1) unpack
//   (2) recover most significant 16 bits
//   by packing 0x41 and the unpacked 8
//   bits.
mov.b16 %regZero, 0x0000;
mov.b32 %Reg8zzAB, {%reg8AB, %regZero};
  // 0x0000-ABOP <- 0xABOP, 0x0000
mov.b32 %Reg8zzCD, {%reg8CD, %regZero};
  // 0x0000-cdqr <- 0xcdqr, 0x0000
shl.b32 %Reg8zABz, %Reg8zzAB, 8;
  // 0x00AB-OP00 <- 0x0000-ABOP0
shl.b32 %Reg8zCDz, %Reg8zzCD, 8;
  // 0x00cd-qr00 <- 0x0000-cdqr
mov.b32 {%reg8Bz, %reg8zA}, %Reg8zABz;
  // 0xOP00, 0x00AB <- 0x00AB-OP00
mov.b32 {%reg8Dz, %reg8zC}, %Reg8zCDz;
  // 0xqr00, 0x00cd <- 0x00cd-qr00
mov.b32 %Reg8zAzC, {%reg8zC, %reg8zA};
  // 0x00AB-00cd <- 0x00cd, 0x00AB
add.s32 %Reg8fAfC, %Reg8zAzC, 0x4100-4100;
  // 0x41AB-41cd <- 0x00AB-00cd
mov.b32 {%reg8fC, %reg8fA}, %Reg8fAfC;
  // 0x41cd, 0x41AB <- 0x41AB-41cd
mov.b32 %Reg8zzBz, {%reg8Bz, %regZero};
  // 0x0000-OP00 <- 0xOP00, 0x0000
mov.b32 %Reg8zzDz, {%reg8Dz, %regZero};
  // 0x0000-qr00 <- 0xqr00, 0x0000
mad.lo.s32 %Reg8fBzz, 0x0000-0100,\
           %Reg8zzBz, 0x4100-0000;
  // 0x41OP-0000 <- 0x0000-OP00
mad.lo.s32 %Reg8fDzz, 0x0000-0100,\
           %Reg8zzDz, 0x4100-0000;
  // 0x41qr-0000 <- 0x0000-qr00
mov.b32 {%regTrash, %reg8fB}, %Reg8fBzz;
  // 0x0000, 0x41OP <- 0x41OP-0000
mov.b32 {%regTrash, %reg8fD}, %Reg8fDzz;
  // 0x0000, 0x41qr <- 0x41qr-0000

// Recover most significant 32 bits.
mov.b32 %Reg8fAAA, {%reg16A, %reg8fA};
  // 0x41AB-CDEF <- 0xCDEF, 0x41AB
mov.b32 %Reg8fBBB, {%reg16B, %reg8fB};
  // 0x41OP-QRST <- 0xQRST, 0x41OP
mov.b32 %Reg8fCCC, {%reg16C, %reg8fC};
  // 0x41cd-efgh <- 0xefgh, 0x41cd
mov.b32 %Reg8fDDD, {%reg16D, %reg8fD};
  // 0x41qr-stuv <- 0xstuv, 0x41qr

// Recover full 64 bits.
mov.b64 %REG8fAAA-AAAA,
        {%Reg32A, %Reg8fAAA};
  // 0x41AB-CDEF-GHIJ-KLMN
  //   <- 0xGHIJ-KLMN, 0x41AB-CDEF
mov.b64 %REG8fBBB-BBBB,
        {%Reg32B, %Reg8fBBB};
  // 0x41OP-QRST-UVWX-YZab
  //   <- 0xUVWX-YZab, 0x41OP-QRST
mov.b64 %REG8fCCC-CCCC,
        {%Reg32C, %Reg8fCCC};
  // 0x41cd-efgh-ijkl-mnop
  //   <- 0xijkl-mnop, 0x41cd-efgh
mov.b64 %REG8fDDD-DDDD,
        {%Reg32D, %Reg8fDDD};
  // 0x41qr-stuv-wxyz-@?=*
  //   <- 0xwxyz-@?=*, 0x41qr-stuv
```

ただし $2^{-4}\varepsilon^{-1}$ を

`0x41AB-CDEF-GHIJ-KLMN`、`0x41OP-QRST-UVWX-YZab`、`0x41cd-efgh-ijkl-mnop`、`0x41qr-stuv-wxyz-@?=*`

とし、読み込みではなく即値代入のように表現した（`[A-Za-z@?=*]` は `0` ～ `f`）。上記のような実装をした場合、`mov` を 0 clock とすれば、圧縮・展開による余分な演算量は `1 add.s32, 2 shl.b32, 2 mad.lo.s32`（それぞれ throughput は 32, 16, 16）合計で $1+2\cdot2+2\cdot2=9$ clocks となる。1 double precision floating point number あたり 8 bits の読み込みの省略であるから、4 つでは 32 bits となり、情報転送量削減による時間の短縮は $30\cdot32/64=15$ clocks となる。よって、全体では $15-9=6$ clocks の短縮となる。`shl.b32, mad.lo.s32` の throughput が 32 となれば、$15-5=10$ clocks の削減となる。

いずれにせよ、情報転送時間に比して演算時間が 0 とみなせるような圧縮に最大限有利な条件を想定しても、誘電率の上位 8 bits のみの圧縮による効率の上昇率は前述の 114 % を下回る。

3.3 電磁場の圧縮

前述の誘電率専用の圧縮方法では、仮定により全領域での圧縮が可能であるため圧縮が可能であるかどうかの判定は不要であり、GPU のような分岐が不得意な計算機には好都合であった。

一方、電磁場の場合、waveguide のような単純なものでも圧縮可能領域は 100 % にはならず（例示にあまり意味は無いが一例をあげれば 90%程度）、圧縮の判定の必要性がある。この場合、演算量の増加を無視しても分岐による性能低下（現在は最大 1/32 倍）は無視できないため、圧縮可能領域の割合が低いか散逸している場合は圧縮による性能向上は期待できない。

仮に 100% の領域で圧縮可能であっても、誘電率の場合と異なり電磁場は基本的に時間の関数であり、読み込み専用ではないため展開だけでなく圧縮も行う必要があり、また指数部分であっても直接の圧縮はできず差分をとる必要があるため、誘電率の場合と比べ圧縮による性能向上はより難しくなる。

そこで、一般的に成り立つ方法ではなく特定の計算機にのみ有効な方法を考える。

実際の計算機では、throughput 16 の演算は throughput 32 の演算の 2 倍の時間で演算が完了しないことがある。例えば、前者の演算を 10 回行った場合、後者の演算を 20 回行った場合と同じ時間で演算が完了されると予想され

るが、実際には 30 回分の時間がかかったとする。このとき、差分の 10 回分の後者の演算を追加で行っても演算時間が微増にとどまるといったことが起こる。すなわち、前者の演算が予想より低い効率となってしまった場合に、後者の演算を追加で行うことで予想通りの効率を実現できる。このような現在の計算機に特有の性質を逆手にとれば、つまり偽の余剰演算能力を圧縮に利用すれば、実用上は計算効率の向上が期待できる。その偽の余剰演算能力が十分であれば、前述のような単純な演算による圧縮（低圧縮率）ではなく、より複雑な演算による圧縮（高圧縮率）を行うことで、前述の 114% の性能向上率を超える可能性もある。この場合は誘電率のような定数でなく、電磁場のような変数でも実用的に圧縮しながらの演算による効率向上が期待できる。

しかし、この種の方法は hardware の改善によって無効になるものであり、将来性に欠けるものである。

4 まとめ

以上より、FDTD 法を情報圧縮により高速化するためには、誘電率を特定の範囲内に制限する方法は有効であることが分かった。また、電磁場等の圧縮や誘電率の高圧縮を行うことによる性能向上のためには、現在の計算機の欠点を利用することが有効であることが分かった。

参考文献

[1] NVIDIA corporation, CUDA C Programming Guide. Version 3.2. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf, (参照 2011-01-15).

[2] NVIDIA corporation, PTX ISA. Version 2.2. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/ptx_isa_2.2.pdf, (参照 2011-01-15).

[3] NVIDIA corporation, CUDA C Best Practices Guide. Version 3.2. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf, (参照 2011-01-15).

[4] NVIDIA corporation, Reference Manual. Version 3.2. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf, (参照 2011-01-15).

[5] NVIDIA corporation, Fermi Compatibility Guide for CUDA Applications. Version 1.3. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/Fermi_Compatibility_Guide.pdf, (参照 2011-01-15).

[6] NVIDIA corporation, Tuning CUDA Applications for Fermi. Version 1.3. 2010. (online), http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/Fermi_Tuning_Guide.pdf, (参照 2011-01-15).