Vol.2012-HPC-133 No.27
2012/3/27

OpenCL による四倍精度行列積の高速化

中　村　光　典[†1]　　中　里　直　人[†1]

四倍精度演算の実装方法として二つの倍精度数を用いてエミュレートするものがあるが，このエミュレーションによる実装を利用した場合，一度の四倍精度演算に多くの倍精度演算が必要となる．そのため，常用するには高速化が望ましい．本研究ではOpenCL を利用した並列処理による四倍精度行列積の高速化を目指し，その応用として LU 分解の高速化も行った．mpack ライブラリと比較すると我々の OpenCL による行列積は GPU では約 350 倍，マルチコア CPU では約 21 倍の性能を評価でき，また LU 分解においては GPU，マルチコア CPU ともに約 10 倍に近い高速化を実現した．

Acceleration of Matrix Multiplication in Double-Double Precision by OpenCL

Kosuke Nakamura[†1] and Naohito Nakasato[†1]

A method to implement quadruple precision operations is to emulate it with two double precision values, but this emulation scheme requires many double precision calculations for one double-double (quadruple) precision calculation. So the acceleration is desirable for regular usage. The purpose of this research is the acceleration of matrix multiplication in double-double precision by parallel processing using OpenCL and we have applied the accelerated operation to LU factorization. Compared with the mpack, our OpenCL matrix multiplication routine is about 350 times faster on GPU and about 21 times faster on multi-core CPU and in LU factorization it is about 10 times faster on each device.

†1 会津大学
The University of Aizu

1. はじめに

科学や工学における数値計算を行う上で，演算精度は大変重要である．コンピュータの算術において実数を扱うことは不可能であり，浮動小数点数を使用した演算を行うことになる．頻繁に利用されるのは単精度や倍精度による演算であるが，場合によっては精度が足りないことがある．そこで数値的に不安定な問題に対して四倍精度演算のような高精度演算の利用が考えられる．しかし，四倍精度演算の利用には倍精度演算より 10 倍から 20 倍の計算時間が掛かってしまう．本研究では四倍精度演算による行列積を高速化するために，CPU と GPU による複数レベルでの並列化を行った．行列積は物理や工学等で頻繁に使われるので高速化することは非常に有益である．近年ではマルチコア CPU や GPU など，ハードウェアの著しい進化によりますます並列処理が重要になってきている．並列化の実現のためには MPI や OpenMP，POSIX thread のようなプログラミング手法があるが，本研究ではマルチコア CPU や GPU での並列計算を実装するためのフレームワークである OpenCL[1] を使用した．

2. 四倍精度演算

2.1 誤差を考慮した倍精度演算と四倍精度演算の実装

IEEE754 規格で規程された倍精度変数は 64 ビットのデータで構成されており，10 進数において約 16 桁まで正しく表現することができる．IEEE754 における倍精度データ型の構造を図 **1** に示す．

通常，浮動小数点数の演算では桁落ちや丸め誤差などが生じ，正しいビットデータが失われてしまう．従って演算を繰り替えしていくうちにこの誤差が蓄積され正しい計算結果と大きく異なってしまうことがある．これを避けるために誤差を考慮した安定的な演算手法を採用した．この演算によって本来誤差によって失われるデータを保持しながら計算が行える．安定的な倍精度演算は Knuth[3] と Dekker[4] によって考案された．具体的には以下の 3 つのアルゴリズムを利用する．

- *quick_two_sum* (*double a*, *double b*)
- *two_sum* (*double a*, *double b*)
- *two_prod* (*double a*, *double b*)

加算に使用する **quick_two_sum** アルゴリズムは 3 回の倍精度演算，**two_sum** アルゴリズムは 6 回の倍精度演算を行う．**quick_two_sum** の方が 3 演算少ないが $|a| \geqq |b|$ の場

© 2012 Information Processing Society of Japan

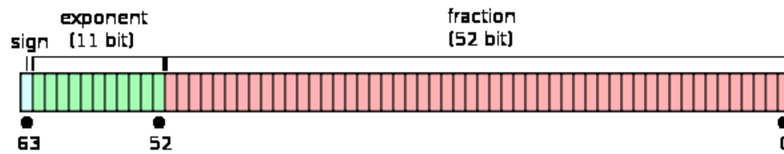

図 1　IEEE754 における倍精度データ型の構造

合のみに適用できる．同様に倍精度数同士の乗算には **two_prod** アルゴリズムが用いられる．**two_prod** アルゴリズムは 17 回の倍精度演算を行う．これらのアルゴリズムは，その特徴から，エラーフリー演算[5] と呼ばれる．

一つの四倍精度変数を表すために倍精度変数を二つ用いる[6),7)]．この四倍精度は 128 ビットのデータで構成され，10 進数において約 32 桁まで正しく表現することができる．このデータにおける仮数部の上半分を上位ビット，下半分を下位ビットとしてそれぞれ倍精度変数に格納する．加算と乗算には倍精度演算と上記のエラーフリー演算を組み合わせて使用する．

2.1.1　四倍精度の加算

二つの四倍精度数の加算のために使用するアルゴリズムを本論文では **DD_TwoSum** と呼ぶ．なお，DD とは Double-Double の略記である．このアルゴリズムは 2 回の倍精度演算と **quick_two_sum** と **two_sum** を組み合わせて，合計で 11 回の倍精度演算を必要とする．

2.1.2　四倍精度の乗算

同様に乗算のためのアルゴリズムを **DD_TwoProd** と呼ぶ．4 回の倍精度演算と **two_prod**，**quick_two_sum** を組み合わせる．従って合計で 24 回の倍精度演算となる．また，後に記述する FMA を使用する場合は **two_prod** の演算数が 17 から 3 に減り，**DD_TwoProd** の倍精度演算回数は 10 回に減る．

なお，ここまでに示したアルゴリズム等は Bailey らによる **QD** という四倍精度演算のライブラリ[7)] の一部を移植・改良して使用している．

2.2　OpenCL

OpenCL は Open Computing Language の略称である．主に Khronos group によって標準化されており，C 言語に近い書き方でプログラムを組むことができる．OpenCL の主な特徴は多くのプラットフォームやデバイスでプログラムを書き換えることなく使用できる

```
First, initialize a block of matrix C with 0.

                  for(i=0;i<N;i++)
                     for(j=0;j<N;j++)
                       for(k=0;k<N;k++)
                         c(i,j) += a(i,k) * b(k,j)
```

図 2　行列積アルゴリズム 1

```
First, initialize a block of matrix C with 0.

                   for(i=0,i<N,i+=Ib)
                     for(j=0,j<N,j+=Jb)
                       for(k=0,k<N,k+=1)
                         for(ii=i,ii<i+Ib && ii<N,ii+=1)
                           for(jj=j,jj<j+Jb && jj<N,jj+=1)
                             c(ii,jj) += a(ii,k) * b(k,jj)
```

図 3　行列積アルゴリズム 2

ことである．OpenCL を利用するためにはホスト側（CPU）とデバイス側（GPU 等）の最低二つのプログラムが必要である．デバイス側のプログラムにそれぞれのスレッドに対する処理を書くことにより並列プログラムを作ることができる．

3.　四倍精度演算による行列積

本論文では，**DD_TwoSum** と **DD_TwoProd** を組み合わせることで，四倍精度演算による行列積を OpenCL により実装した．なお，今回扱った行列積は N 次正方行列のみという制限がある．行列積のアルゴリズムには一般的な三重ループのものとそれをブロック化したものを使用した．それらを**図 2** と**図 3** に示す．Ib，Jb はそれぞれのブロック数を示す．2×2 ブロックなら（Ib，Jb）＝（2，2）となる．

3.1　OpenCL における四倍精度行列積の実装

並列プログラムを作るために各スレッドの処理を決める必要がある．また，OpenCL ではホスト側のプログラムでスレッド数を指定する．行列積のアルゴリズムとして二つを示したが，並列化の方法としては**行列積アルゴリズム 1** では (図 2) の 5，6 行目を各スレッドに割り当てた．**行列積アルゴリズム 2** では (図 3) の 5~8 行目を各スレッドに割り当てた．GPU はデータ並列処理に適しているのでそれぞれのスレッドにほぼ同じような計算をさせることは効率が良いのでこのような割り当てにした．

GPU では，メモリからのロードのコストが高いため，高性能を引き出すためには，ロー

© 2012 Information Processing Society of Japan

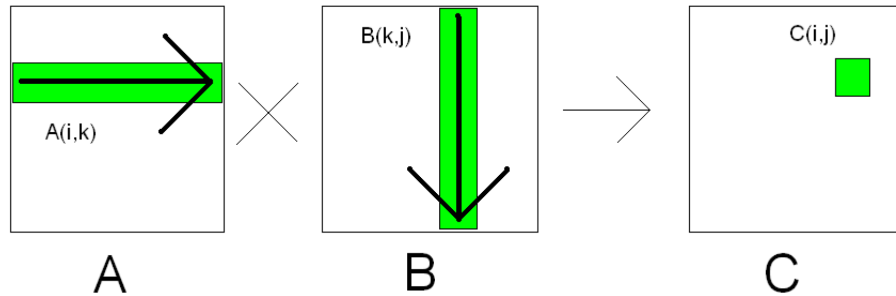

図 4　DD-GEMM カーネルにおける各スレッドの処理イメージ（行列積アルゴリズム 1）

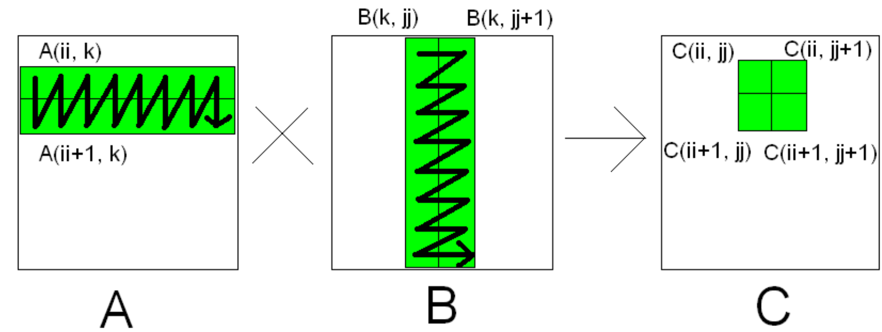

図 5　DD-GEMM カーネルにおける各スレッドの処理イメージ（行列積アルゴリズム 2）

ドするデータの量を最小限にする必要がある．そのため，ロードしたデータをできるだけ再利用するアルゴリズムが向いている．CPU でも，キャッシュヒット率を上げるためには，ブロッキングを行うことが重要な最適化手法であり，GPU でも同様の手法を用いるほうが性能が高い．よって，**行列積アルゴリズム 2** はキャッシュヒットによりデータロードの量が減り，大きなパフォーマンスが期待できる．ブロックサイズは（Ib，Jb）＝（2，2）から始めて，（Ib，Jb）＝（2，4），（4，4）と徐々に増やし，パフォーマンスが下がったところで止めた．

3.2　四倍精度行列積のデバイスプログラム

3.1 で述べたスレッド割り当てによるデバイスプログラム（カーネルコード）における各スレッドの処理イメージを**図 4** と**図 5** に示す．なお，**行列積アルゴリズム 2** のブロックサイズは（Ib，Jb）＝（2，2）の場合と仮定する．積和には 2.1.1 と 2.1.2 で示した四倍精度演算をカーネルに移植し，それを利用する．また，四倍精度行列積を DD-GEMM（DD GEneral Matrix Multiplication）と表記するが，本研究では現時点で，BLAS（Basic Linear Algebra Subprograms）において定義された GEMM の一部である行列積のみを実装した．今回実装した DD-GEMM の配列の格納方向には Row Major 方式を採用しており Fortran 等で使われている Column Major 方式とは異なる．さらに，行列 A，B の各要素は乱数で初期化してある．

3.3　アルゴリズムの最適化

行列積の性能を上げるためデータのブロッキングに加えて，我々は以下の演算処理レベルでの最適化を行った．FMA（Fused Multiply Add）とベクトル化の導入である．OpenCL にはこの二つを使用するための関数や変数が用意されている．

FMA は加算と乗算を一命令で行うことができる[6]．この命令を使えば，**DD_TwoProd** アルゴリズムの演算回数が 24 から 10 に減る．ただし，FMA 命令がハードウェア側でも実装されている必要がある．

もう一つの最適化の手法はベクトル化であるが，本論文におけるベクトル化とは OpenCL のベクトル型変数を明示的に使用することである．この変数を使用することにより，明示的にハードウェアに SSE（Streaming SIMD Extensions）等の並列命令を行わせることができる．例として，double 型のベクトル型変数には **double4** や **double8** がある．

4.　四倍精度行列積高速化の結果

本研究では，表 1 に示す計算機システムを利用して，演算性能を計測した．
Core i7-2600K では動作周波数が 3.4GHz，4 コア 8 スレッドの構造をしており，AVX（Advanced Vector Extensions）命令は 1 コアあたり 4 回の倍精度加算と 4 回の倍精度乗算を実行できるので，倍精度演算での理論ピーク性能は $3.4 \times 2 \times 4 \times 4 = 108.8$（Gflop/s）となる．AVX 命令は SSE 命令の 2 倍の長さのレジスタを扱えるので 1 演算で扱える浮動少数点数の数が倍になる．HD7970（Tahiti アーキテクチャ）においては動作周波数が 0.925GHz，SP（Stream Processors）の数が 2048 であり FMA 使用時に加算と乗算を同時に行うので倍精度演算での理論ピーク性能は $0.925\times$（$2048 \div 2$）$= 947$（Gflop/s）である．

今回使用した CPU は FMA 命令をサポートしていないので，GPU でのみ FMA 命令を用いた結果を示す．また，比較対象として四倍精度演算による BLAS を実装した **mpack**[8]

ⓒ 2012 Information Processing Society of Japan

表 1 使用した CPU，GPU と OpenCL SDK のバージョン

デバイス	CPU	GPU
名称	Intel Core i7-2600K	AMD Radeon HD7970 （Tahiti アーキテクチャ）
倍精度理論ピーク性能	108.8 Gflop/s (AVX)	947 Gflop/s (FMA)
OpenCL SDK	OpenCL ver. 1.5(Intel)	OpenCL ver. 2.6(AMD)

ライブラリの演算性能を採用する．mpack での四倍精度 BLAS は QD ライブラリを利用して実装されており，通常のコンパイルオプションでは SSE や AVX 命令は利用されず，またマルチスレッド処理も行わない．

CPU におけるベクトル化を使わないパターンと使ったパターンの結果を**図 6** と**図 7** に示す．次に，GPU におけるベクトル化を使わないパターンと使ったパターンの結果を**図 8** と**図 9** に示す．

Rgemm（mpack）による非並列化のピーク性能は約 0.2Gflop/s であった．一方でベクトル化なしの OpenCL（CPU）による並列化のピーク性能は約 0.8Gflop/s（図 6）であり，OpenCL（GPU）においては約 70Gflop/s（図 8）である．OpenCL（CPU）では**行列積アルゴリズム 2** の 4×4 ブロックのときにピークに達するが，N が大きくなるにつれて 8×8 ブロックが良い性能を維持している．また，OpenCL（GPU）では**行列積アルゴリズム 2** の 2×2 ブロックかつ FMA を使用したときにピークに達し，N が大きくなっても他のパターンに比べ大きな性能を維持している．OpenCL を使用した CPU，GPU の双方において**行列積アルゴリズム 1** より**行列積アルゴリズム 2** の方が性能が上であることは明らかである．さらに GPU においては FMA による性能上昇も大きい．

次に，ベクトル化ありの OpenCL（CPU）のピーク性能は約 4.2Gflop/s（図 7）であり，OpenCL（GPU）のピーク性能は約 70Gflop/s（図 9）である．OpenCL（CPU）においてはベクトル化によって格段に性能が上昇していることが分かる．ピークに達するのは**行列積アルゴリズム 2** の 4×8 ブロックのときであり，**double8** によってベクトル化されている．他のパターンもベクトル化なしのときより高い性能を維持している．GPU を使用した場合，ピークを示しているのはベクトル化なしのときと同様に**行列積アルゴリズム 2** の 2×2 ブロックかつ FMA 使用のパターンだが，他のパターンにおいても全て 40Gflop/s に達しておりベクトル化による性能上昇が見られる．OpenCL によるピーク性能を Rgemm（mpack）のピーク性能と比べると，CPU では約 21 倍，GPU では約 350 倍となる．

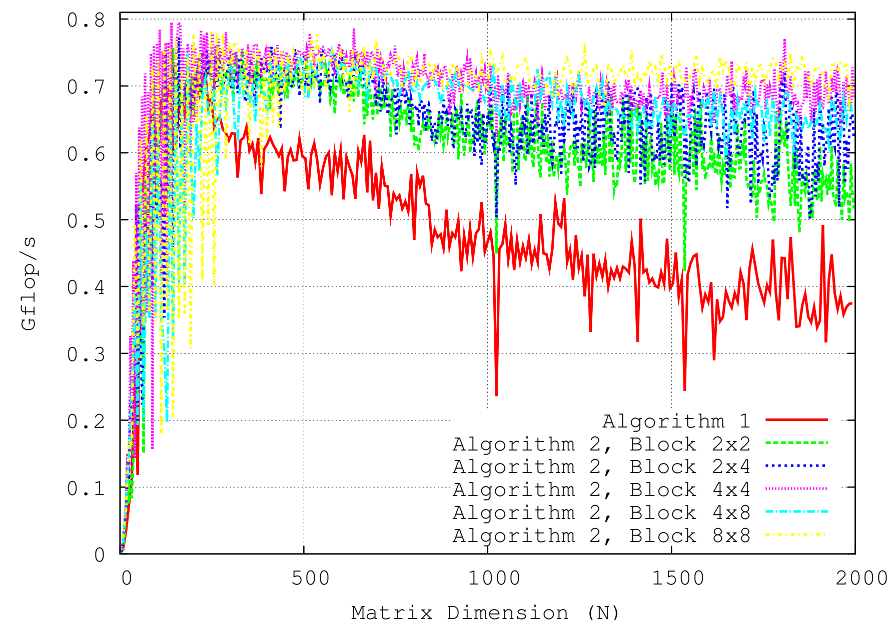

図 6 OpenCL CPU における DD-GEMM の性能 (ベクトル化なし)

ここで，倍精度における浮動小数点数の演算回数を見積もると次のようになる．一度の **DD_TwoSum** と **DD_TwoProd** の使用でそれぞれ 11 回，10 回（FMA が使われていれば）の倍精度演算を必要とする．従って四倍精度における 70Gflop/s は倍精度換算で約 735（$= 70 \times ((11 + 10) \div 2)$）Gflop/s となる．今回使用した GPU の倍精度理論ピーク性能（表 1）は 947Gflop/s（FMA 使用）であるので，その約 77%に該当する．また，OpenCL のベクトル化によって急激に性能が上がったのは非ベクトル化に比べ SSE や AVX 等による処理がより最適になったからであると思われる．今回使用した CPU においては倍精度理論ピーク性能が 108.8Gflop/s であった．四倍精度における OpenCL（CPU）のピーク性能は 4.2Gflop/s であったが，これを倍精度換算すれば約 74（$= 4.2 \times ((11 + 24) \div 2)$）Gflop/s であり，理論ピーク性能の約 68%に等しい．

© 2012 Information Processing Society of Japan

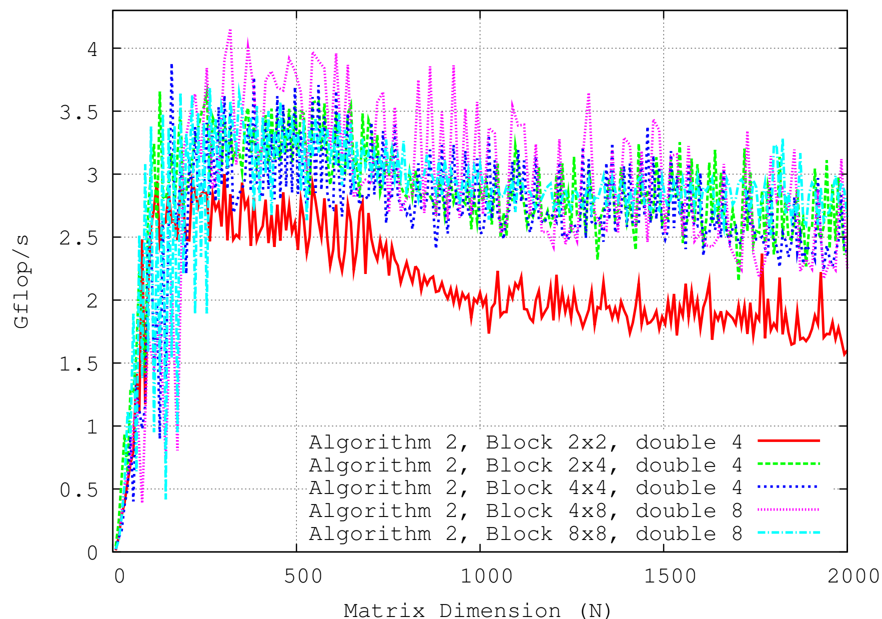

図 7 OpenCL CPU における DD-GEMM の性能 (ベクトル化あり)

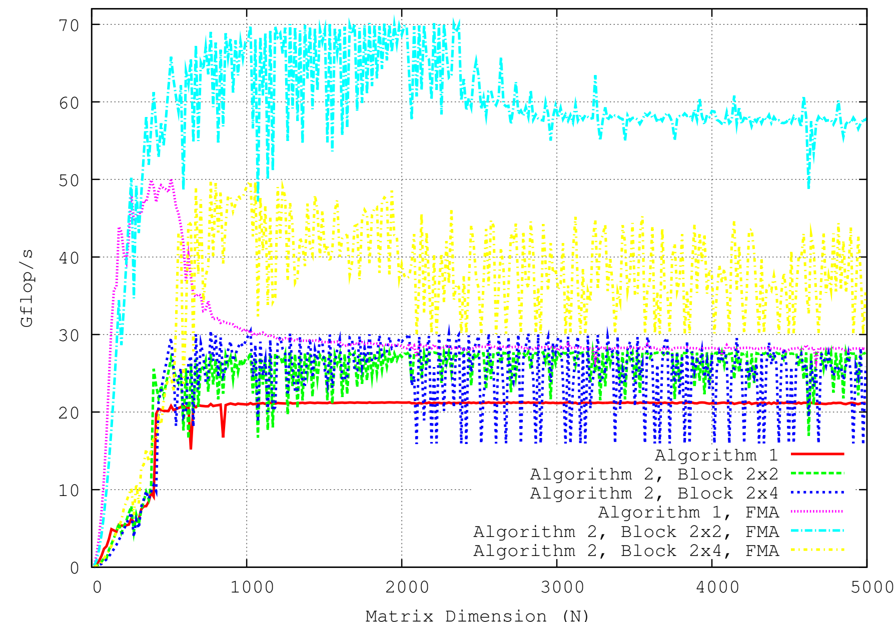

図 8 OpenCL GPU における DD-GEMM の性能 (ベクトル化なし)

5. LU 分 解

今回高速化した DD-GEMM の応用として四倍精度における LU 分解の高速化を行った．LU 分解は行列（例として A）を $A \longrightarrow L \times U$ のように下三角行列（L）と上三角行列（U）に分解することである．この分解をすることによって線形方程式の解を求めることや A の逆行列，行列式等を得るのが容易になる．

5.1 Right-looking アルゴリズム

行列積を LU 分解に適用するためには，LU 分解のブロック化が必要となる．そのアルゴリズムとして Left-looking，Right-looking，Crout の 3 つが存在する[9] が今回は **Right-looking** と呼ばれるアルゴリズムを使用する．本来，このアルゴリズムでは Remaining matrix のアップデートのみに行列積が使われる．しかし文献 10) で提案された手法を使うと，他の処理においても GEMM を利用できるのでアルゴリズムにおいて行列積が占める割合を増やせる．つまり LU 分解の性能が DD-GEMM の性能に大きく依存することになる．

元の行列 A のサイズが $N \times N$ であり，ブロック行列のサイズが $b \times b$ ならばアルゴリズム中のブロック行列の数は $n = N/b$ として，$n \times n$ 個である．我々は LU 分解における $b \times b$ のブロック行列同士の積に高速化した DD-GEMM を適用した．行列積以外の命令（Remaining matrix の LU 分解，逆行列の計算）には mpack を使用する．また，DD-GEMM で使用するパターンは一番高速であったパターンを使用する．OpenCL（CPU）ではベクトル化ありの行列積アルゴリズム 2 において 4×8 ブロックのパターン，OpenCL（GPU）ではベクトル化あり，FMA ありの行列積アルゴリズム 2 において 2×2 ブロックのパターンである．

問題となるのはブロックサイズである．DD-GEMM はある程度大きいサイズでピーク性能を発揮するが mpack の命令は小さいサイズの方が処理が速く，サイズが大きくなるにつれて LU 分解の総時間に負担をかける．そこで実行時間に関して以下の簡単な性能モデルを考える．

© 2012 Information Processing Society of Japan

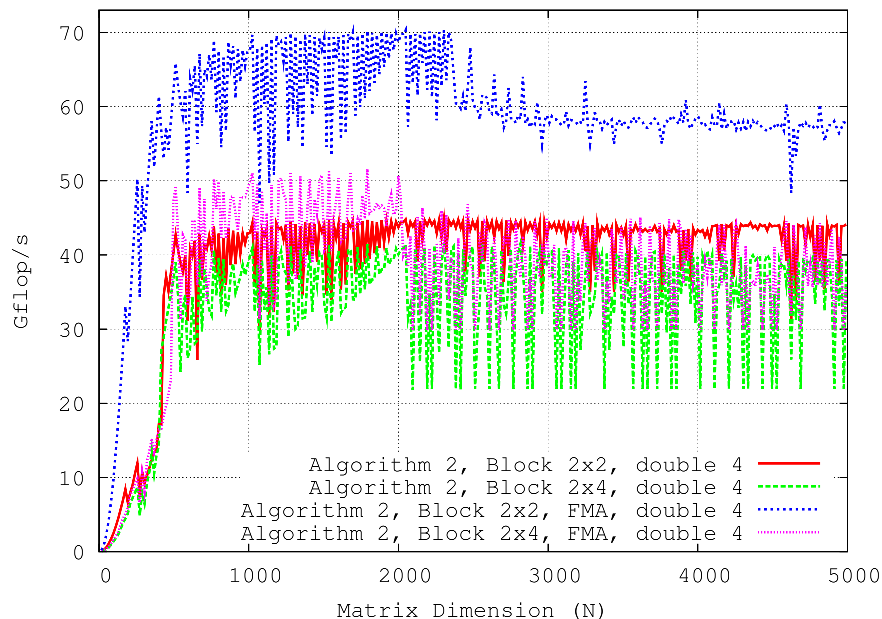

図 9 OpenCL GPU における DD-GEMM の性能 (ベクトル化あり)

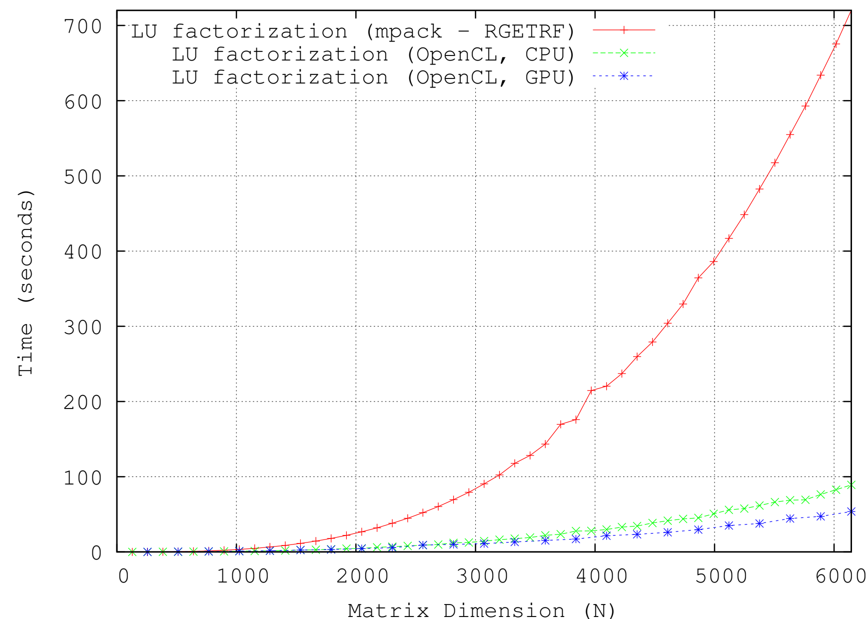

図 10 四倍精度 LU 分解の総実行時間 (mpack，OpenCL CPU，OpenCL GPU)

$$T_{\mathrm{LU}} = T_{\mathrm{mpack}} + T_{\mathrm{DD-GEMM}}$$

T_{mpack} はおおまかに b^3 に比例し，$T_{\mathrm{DD-GEMM}}$ は例えば図 9 においては $b < 2000$ で大きく変化するのでこの範囲から b の最適値が決まる．いくつかのサイズで試してみたところ，OpenCL（CPU）では $b = 128$，OpenCL（GPU）では $b = 256$ が LU 分解全体の実行時間として一番良い性能を出すという結果になり，このブロックサイズを採用した．

5.2 四倍精度 LU 分解高速化の結果

DD-GEMM を使用した LU 分解の処理時間を**図 10** に示す．また性能比較のために非ブロック化（非並列化）による LU 分解を mpack のみで処理した場合の計算時間も図示した．

mpack の結果に対して高速化されたことが図 10 から読み取れる．$N = 6144$ において OpenCL（CPU）では約 9 倍，OpenCL（GPU）では約 12 倍速くなった．しかし，OpenCL の CPU と GPU を使用した実行時間があまり変わらないように見える．DD-GEMM の高速化の結果を見れば明らかに GPU の方が速いが，ブロック化 LU 分解においては DD-GEMM 以外の命令も用いているのでそれがボトルネックになっていると思われる．総実行時間に加えてカーネルのみの実行時間も示したものを**図 11** に示す．

図 11 を見ると，CPU 使用時は DD-GEMM の実行時間が総実行時間の 50%近くを占めるのに対し，GPU 使用時は 5%にも満たない．つまり GPU 使用時に関しては DD-GEMM 以外の命令に大きく時間を取られている．

5.3 性能改良のための考察

GPU 使用時においてさらに処理時間を短くする方法として，再帰的にブロック化するという案が挙げられる．図 9 等を見て頂ければ分かるが，DD-GEMM が最高性能を発揮するのは行列サイズが 1000 から 2000 の間であり，今回ベンチマークから得た 256 というサイズでは性能を最大に利用できていない．しかし再帰的に LU 分解のブロック化を行えば

© 2012 Information Processing Society of Japan

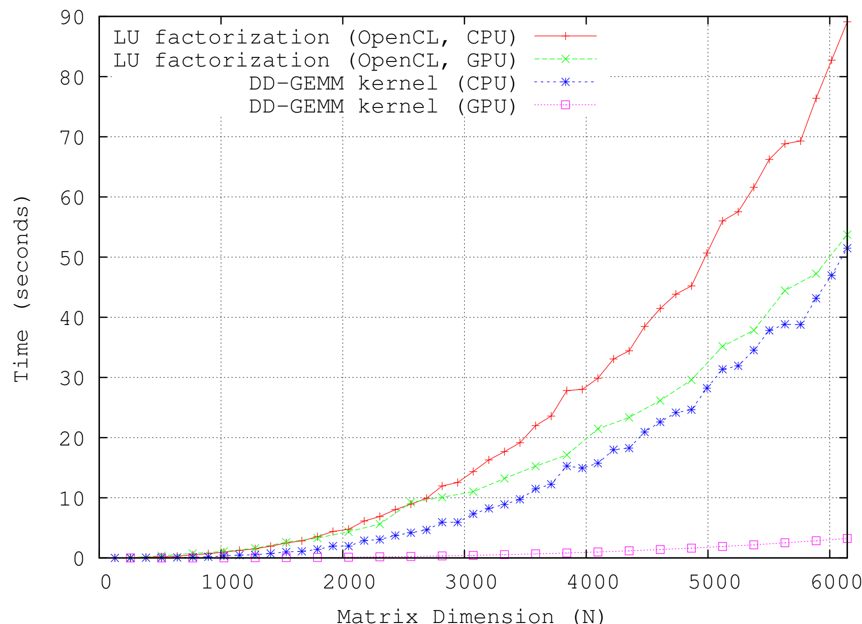

図 11 四倍精度 LU 分解の総実行時間とカーネルのみの実行時間 (OpenCL CPU，OpenCL GPU)

DD-GEMM は大きなブロックサイズに対して直接利用し，mpack 命令は再帰によって小さくなったサイズに対してのみ使用すれば良い．例えばサイズが 1024 のブロック行列を採用した場合，ブロック行列積はそのまま DD-GEMM を使えばピーク性能を発揮でき，mpack 命令の部分はサイズ 1024 のブロック行列をさらにサイズ 256 等のブロック行列に分割して適用するのである．この場合，先述の性能モデルにおける T_{mpack} と $T_{\mathrm{DD-GEMM}}$ に対して互いに独立した最適な b を与えることができる．LU 分解のサイズが大きくなれば図 10 よりもさらに良い性能が見られるだろう．

最後に，$|A - L \times U|$ の各要素の平均値を平均残差として評価した．平均残差は約 10^{-27} であった．四倍精度において丸め誤差は約 10^{-31} 程度であるが，今回はピボッティング操作を行っていないため算出された残差はこの値より大きくなる．

6. 関 連 研 究

椋木ら[11] は本研究と同様に四倍精度の GEMM に加え BLAS におけるいくつかのルーチンにおいて四倍精度での実装を施し，NVIDIA 社の CUDA を使用して GPGPU による評価を行った．ハードウェアの大きな進化もあり，本研究ではさらに良い結果を得ることができた．また，OpenCL の普及により GPU のみでなく CPU における性能評価ができたところは新しい成果である．

7. ま と め

四倍精度行列積を OpenCL によって高速化し，DD-GEMM として実装することができた．また，高速化した DD-GEMM の応用として四倍精度 LU 分解を行った．今回実装した DD-GEMM は正方行列のみを扱い，LU 分解も正方行列のみと制限されていたので今後の展望として，より一般的なサイズの行列を扱えるようにしていきたい．さらに DD-GEMM については CPU，GPU 共にピーク性能に可能な限り近づくように改良したい．LU 分解では先ほど述べた改良に加え，安定性の問題があるのでピボッティングを使用しアルゴリズム等を見直す．

最後に，今回は一種類の CPU と GPU を用いたが OpenCL は多くのプラットフォームやデバイスを使用できるのが最大の特徴であり，プログラムを修正せずに他の環境で試すことができる．そのため今回実装した DD-GEMM の性能を別の CPU や GPU で計測することも行う．

参 考 文 献

1) 土山了士，中村孝史，飯塚拓郎，浅原明広，三木聡，株式会社フィックスターズ：OpenCL 入門，株式会社インプレスジャパン (2010).
2) Double precision floating-point format, http://en.wikipedia.org/wiki/Double-precision_floating-point_format .
3) Knuth D.E. : The art of computer programming Vol:2 Seminumerical algorithms 3rd Edition, Addison-Wesley (1998).
4) Dekker T.J. : A floating-point technique for extending the available precision, Numerische Mathematik, Vol.10, pp.224-242 (1971).
5) K. Ozaki, T. Ogita, S. Oishi, S. M. Rump: Error-Free Transformation of Matrix Multiplication by Using Fast Routines of Matrix Multiplication and its Applications, Numerical Algorithms, 59:1 (2012), 95-118.

© 2012 Information Processing Society of Japan

6) 永井貴博，吉田仁，黒田久泰，金田康正：SR11000 モデル J2 における 4 倍精度積和演算の高速化，情報処理学会研究報告， Vol.48, No.13, pp.214–222 (2007).
7) High-Precision Software Directory, http://crd-legacy.lbl.gov/dhbailey/mpdist/ .
8) The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), http://mplapack.sourceforge.net/ .
9) Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen and Henk A. van der Vorst : Numerical Linear Algebra for High-Performance Computers, Society for Industrial and Applied Mathematics, (1998).
10) 酒井智哉， 松本和也， 中里直人， Stanislav G. Sedukhin：LU-factorization on Cypress GPU，情報処理学会第 73 回全国大会 (2011).
11) 椋木大地，中山星空，越智浩之：GPU による多倍長精度 BLAS の開発，ICT ソリューションアーキテクト育成プログラム：ソリューション型特別プロジェクト最終報告書 (2010).

© 2012 Information Processing Society of Japan