Vol.2018-HPC-164 No.4
2018/5/7

Volta GPU におけるテンソルコアを用いた多倍長整数演算の高速化

土井　淳[†1]

概要：ディープラーニングや画像処理などのように高い精度を必要としない計算をより高速に行うために単精度よりもさらに半分の精度の浮動小数点数である，半精度浮動小数点数が注目されつつある．NVIDIA の Volta GPU にはテンソルコアと呼ばれる，半精度浮動小数点数の行列演算を高速に計算する演算器が搭載されており，これを利用することで，単精度浮動小数点演算の 8 倍の演算性能を得ることができる．また，テンソルコアは内部的には単精度で行列同士の乗算の結果の加算を処理できるようになっており，単純に半精度で計算する場合よりも高い精度で演算ができる利点がある．一方で，精密な CAD モデルの作成，金融計算，暗号化等，非常に高い精度あるいは大きな桁数を必要とするような処理では，多倍長整数演算が用いられる．正確に演算結果を保持するために演算を繰り返すごとにデータが大きくなり計算時間が増大する問題がある．本研究では多倍長整数演算について多倍長整数を 8 ビットずつ半精度浮動小数点数に変換し Volta GPU のテンソルコアを用いて高速に乗算を行う方法について述べる．テンソルコアの特性を用いることで桁あふれによる誤差を生じることなく高速化が可能となった．

キーワード：整数演算，多倍長整数演算，GPU，半精度浮動小数点数，テンソルコア

1. はじめに

近年，消費電力や集積度の観点から，GPU のような加速装置を搭載した計算機が主流となりつつある．科学技術計算を加速させるのはもちろん，今まで計算が困難であったディープラーニングのように加速装置によって注目されるようになったものも多い．これらの計算は主に浮動小数点演算が用いられており，また，コンピュータグラフィクスや画像処理のように計算精度があまり要求されない分野では，有効桁数の小さな浮動小数点数を用いることでデータ容量を削減し，メモリアクセスや転送量を節約することができる．半精度浮動小数点数は，単精度浮動小数点数の半分の 16 ビットのデータ型で，このような比較的精度の必要とされない処理に用いられるようになった．また，ディープラーニングにおいても，学習する対象や処理内容によって大きな精度を必要としない場合があり，このような場合に，処理時間とデータ量を減らす目的で半精度浮動小数点数が用いられるようになってきた．そのため，半精度浮動小数点数演算を効率よく行うための演算器を持つ計算機が増えつつある．NVIDIA の Volta 世代の GPU, Tesla V100[1]は，テンソルコア[2]と呼ばれる半精度浮動小数点演算を高速に実行するための演算器を持ち，ディープラーニングで多用される行列演算を高速化するのに利用できる．

一方で，高い精度あるいは厳密な計算を要求するような処理もある．浮動小数点数はその性質上簡単な有理数であっても正確な値を表現できないことがあり，また，演算の過程での丸め誤差のように，誤差を含んだ数となることが多い．例えば，精密な CAD ソフトを開発しようとする場合，浮動小数点数を用いると非常に近い点を含むような形状同士の集合演算を行おうとすると誤差によって処理が破綻してしまうことがある．また，金融計算などのように絶対に計算を間違えてはいけないような処理もある．

計算機において，数値を正確に表現する手法の一つとして，多倍長整数型を用いる方法がある．多倍長整数型はその名の通り，桁数が固定ではなく可変な整数型で，任意の有効桁数を持った有理数（２つの多倍長整数を用いて表す）を正確に表現することができる．厳密な計算結果を必要とするような数値計算処理において有効な表現手法であるが，厳密に計算しようとするとその分大きなメモリ領域と計算量を必要とする．そのため，多倍長整数型の演算も高速化が望まれる．

本論文では，GPU を用いて多倍長整数演算を高速化することを考える．特に演算量が多く多用される乗算について，高速化を行う．多倍長整数演算の処理では，整数の配列の要素間で繰り上げ，繰り下げの処理が必要となる．そのため，基本となる整数型よりも大きな整数型で厳密な計算を行う必要がある．ところで，テンソルコアでは，半精度浮動小数点数の入力を単精度浮動小数点数で積和演算を行う仕組みで計算精度を高めている．この仕組みをうまく利用すれば，整数演算を厳密に行うことができ，さらに普通の整数演算器で計算するよりもはるかに高速に演算ができると考えた．本来精度の必要のない計算に利用する半精度浮動小数点数を，あえて，厳密計算が必要な多倍長整数演算に利用することで高速化を実現する．

2 章では多倍長整数演算の概要を説明し，3 章では半精度浮動小数点数とテンソルコアについて簡単に説明する．4 章でテンソルコアを用いた多倍長整数演算の高速化について提案し，5 章で NVIDIA Tesla V100 を用いて多倍長整数の乗算と，実アプリケーションに近い行列ベクトル積について性能評価を行った結果を示す．

†1 日本アイ・ビー・エム株式会社　東京基礎研究所
IBM Research – Tokyo

2. 多倍長整数演算概要

2.1 多倍長整数型

多倍長整数型は，図 1 に示すように，固定精度の整数型の配列に，有効桁数を保存するのに必要なサイズを確保して，数値を保存することで表現する．符号無しの整数型の配列を用い，別途符号を表す値と，配列の長さを保存する整数型の値の３つの要素によって構成できる．図 1 のように 8 バイト整数の配列を用いると，式(1)に示すような数値を表現できる．多倍長整数同士の演算を行う場合は，それぞれの配列要素について互いに演算を行い，繰り上げ繰り下げ処理を行いながら，新しい多倍長整数に結果を格納する．本論文では，四則演算それぞれの手法についての詳細は割愛し，乗算についてのみに着目する．

図 1　多倍長整数型の実装例．この例では 1 バイトの符号無し整数型の配列で数値を保存する

2.2 多倍長整数の乗算

多倍長整数演算の四則演算の中でも，乗算と除算の二つの演算は，比較的計算量が多く，高速化が求められる処理である．ここでは，多倍長整数同士の乗算について簡単に説明する．

多倍長整数の乗算を最も単純に行う手法として，筆算によって乗算を行う方法がある．二つの多倍長整数 A，B が式(1)のように，それぞれ n 個，m 個の 8 ビット整数で表現されるとき，式(2)のように，筆算で計算を行うように，B の成分を下位から，A に乗じていき，それらを加算したものが，A と B の乗算の結果となる．この場合の乗算結果を保存するには n+m の長さの配列が必要となる．筆算による方法では，$O(n^2)$の計算量が必要となる．

$$A = a_{n-1} \cdot 2^{8(n-1)} + \cdots + a_2 \cdot 2^{16} + a_1 \cdot 2^8 + a_0$$
$$B = b_{m-1} \cdot 2^{8(m-1)} + \cdots + b_2 \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \quad (1)$$

$$a_{n-1} \cdot b_0 \cdot 2^{8(n-1)} + \cdots + a_2 \cdot b_0 \cdot 2^{16} + a_1 \cdot b_0 \cdot 2^8 + a_0 \cdot b_0 +$$
$$a_{n-1} \cdot b_1 \cdot 2^{8(n)} + \cdots + a_2 \cdot b_1 \cdot 2^{24} + a_1 \cdot b_1 \cdot 2^{16} + a_0 \cdot b_1 \cdot 2^8 +$$
$$\cdots$$
$$a_{n-1} \cdot b_{m-1} \cdot 2^{8(n+m-2)} + \cdots + a_1 \cdot b_{m-1} \cdot 2^{8(m)} + a_0 \cdot b_{m-1} \cdot 2^{8(m-1)} \quad (2)$$

実際の演算では，同じ桁数（バイト数）の位置の値同士を加算し，値があふれた場合は繰り上げの処理を行う必要がある．

多倍長整数型の計算量を減らす手法の一つに，Karatsuba 法[3]がある．Karatsuba 法では，式(3)のような，多倍長整数を上位と下位の部分多倍長整数で表せるとし，それらを乗算するとすると，乗じた数は式(4)のように表せるが，これを式(5)のように変形する．式(4)では部分多倍長整数同士の乗算は 4 回なのに対し，式(5)では，乗算結果を再利用できるため，3 回の乗算に減らすことができる．

$$A = a_h \cdot 2^n + a_l$$
$$B = b_h \cdot 2^n + b_l \quad (3)$$

$$A \cdot B = a_h \cdot b_h \cdot 2^{2n} + (a_l \cdot b_h + a_h \cdot b_l) \cdot 2^n + a_l \cdot b_l \quad (4)$$
$$A \cdot B = a_h \cdot b_h \cdot 2^{2n} + \left(a_h \cdot b_h + a_l \cdot b_l - (a_h - a_l) \cdot (b_h - b_l)\right) \cdot 2^n + a_l \cdot b_l \quad (5)$$

Karatsuba 法は，このような部分多倍長整数の乗算を再帰的に変形していくことによって，計算量を$O\left(n^{log_2 3}\right)$まで減らすことができることが知られている．

さらに，多倍長整数演算の乗算の計算量を減らす手法として，フーリエ変換を用いる方法[4]が知られており，高速フーリエ変換（FFT）を用いることで，計算量を$O(n \log n)$まで減らすことができる．しかしながら，FFT を用いる手法では，桁数が巨大な多倍長整数同士の乗算を行う場合に精度が足りずに正確な演算ができなくなる場合がある．

3. 半精度浮動小数点演算概要

多倍長整数演算を実装するには，通常は整数演算器を利用して実装し，桁あふれする分を正確に保持し繰り上げ処理を行うために，保存している単位の整数型よりも大きな整数型を用いて演算を行う．本論文では，テンソルコアの特徴を用いて桁あふれを処理する．そこで，実装の詳細を説明する前に，テンソルコアで用いられる半精度浮動小数点とテンソルコアについての概要を説明する．

3.1 半精度浮動小数点数

半精度浮動小数点数は，その名の通り単精度浮動小数点数の半分のサイズである 16 ビットの浮動小数点数で, FP16 とも呼ばれる．コンピュータグラフィクスや画像処理のようにあまり高い演算精度を必要としない処理において，半分のメモリアクセスで，SIMD 演算器等の対応があればより速く演算が可能となる．近年では，ディープラーニングにおいても有効な形式であることが知られており，GPU などのアクセラレーターでも半精度浮動小数点数をサポートしたり，高速な演算のできる演算器を搭載するなどの動きがある．

半精度浮動小数点数は，図 2 に示すように，10 ビットの仮数部（暗黙的な 1 ビットを追加した 11 ビットの有効桁）と，5 ビットの指数部，1 ビットの符号ビットから構成される．[5]

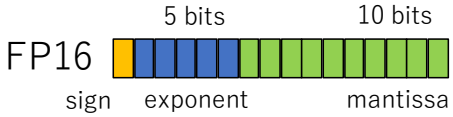

図 2 半精度浮動小数点数（IEEE754-2008, binary16）の格納形式

3.2 テンソルコア

テンソルコアは，NVIDIA の Tesla V100 に搭載された，半精度浮動小数点数の行列積を高速に演算するための特殊な演算器で，主にディープラーニングの処理に多用される行列演算を高速化するために実装された．テンソルコアは図 3 のように 4x4 の行列同士の行列積を別の 4x4 の行列に加算する積和演算（FMA 演算）を行い，これにより単精度浮動小数点数演算を行う場合の GPU あたり 8 倍のピーク演算性能を持つ．

テンソルコアでは，半精度浮動小数点数演算の精度を高めるため，内部的には行列成分同士の乗算結果を単精度浮動小数点数で保持し，さらに乗算結果の加算も単精度浮動小数点数で行うため，単純に半精度浮動小数点数演算を行う場合に比べて非常に高い精度で演算ができる特徴がある．本論文では，この特徴を利用して，8 ビットの整数同士の乗算を厳密に行い，さらに演算結果の加算も単精度浮動小数点演算で行うことで誤差無しの高速な計算が可能となる．

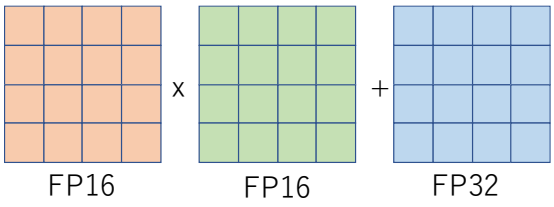

図 3 テンソルコアによる半精度浮動小数点数の 4x4 行列の積和演算

なお，テンソルコアは 4x4 の行列の積和演算を行うが，プログラミングを行う上での API 的には，16x16 の行列の積和演算を行うような仕様[2][6]となっているため，この要素単位で処理を行う必要がある．また，テンソルコアによる行列積の計算は warp 単位（32 スレッド単位）で実行することになっている．

4. テンソルコアによる多倍長整数の乗算

4.1 筆算による多倍長整数の乗算

テンソルコアは，16x16 の行列積を計算するため，筆算による多倍長整数積を行列積計算に置き換える必要がある．また，半精度浮動小数点数を用いるため，使用する多倍長整数型の最小構成単位は 8 ビット整数型とする．これは，半精度浮動小数点数の仮数部 11 ビットで厳密に表現できる数値である．

まず，二つの多倍長整数 A, B について，図 4 のように，8 ビット整数の配列として定義されているとする．

図 4 二つの多倍長整数 A,B の例

このとき，A と B の積を筆算の方法で計算するには次の疑似コードのような処理を行う．

```
for i < B の長さ
  for j < A の長さ
    C[j+i] = C[j+i] + A[j+i]*B[i]
```

よって，この処理を 16x16 の行列積による計算に置き換えれば良い．これを行列で計算するには，配列 A を行方向の成分に持った行列を考え，行が進むごとに列方向に 1 要素ずらしたような配列を考えることで，上の処理の+i の部分が処理できる．また，配列 B の成分は行列に順番に格納する．

図 5 は説明を簡単にするために 2x2 の行列積を用いて多倍長整数の乗算を筆算の方法で計算する方法を示している．A を行方向に 2 成分ずつ，2x2 行列に格納していく．次の行には，1 つ左の列にずらして格納していく．この時，空いた部分には 0 を入れておく．一方，B は，2x2 行列に順番に格納していく．図 5 で $x2^n$ と書かれているのはそれぞれの要素の多倍長整数の元の位置を表す．

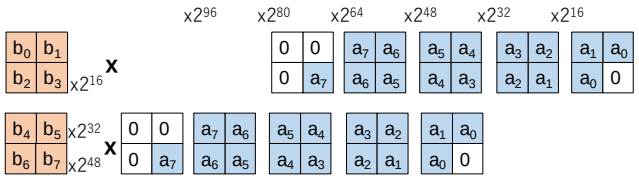

図 5 2x2 行列積による多倍長整数の乗算の例

図 5 で，B の 1 つの 2x2 行列に対して，A の 5 つの 2x2 行列との積を，右から順番に（下位から順番に）計算をしていく．図 6 のように，2x2 の行列積の結果として行列 C が計算されるが，一番上の行（一番下位に当たる部分）は，後段の計算には利用されないので，結果として書き出す．実際の実装では，結果は図 7 のように 4 成分分（32 ビット分）をまとめて繰り上げ処理をしたあと，64 ビット整数として図 8 のような 64 ビットの整数型の配列を 32 ビットずつずらした配列に加算する．ここで加算結果は 32 ビットからあふれるため，後に 64 ビット整数の繰り上げ処理をする必要がある．次に図 6 の真ん中の行列 D のように，一番上の行を取り除いたので，残りの行を一つずつ上の行に移動し一番下の行には 0 を入れておく．さらに，次の行列 A と B の積を行列 D に加算することで同じ桁の部分が加算されていく．この処理を A の行列全部について繰り返し行い，最後に残った行列 D の成分を 4 成分ずつ 64 ビット整数として結果の配列に加算する．

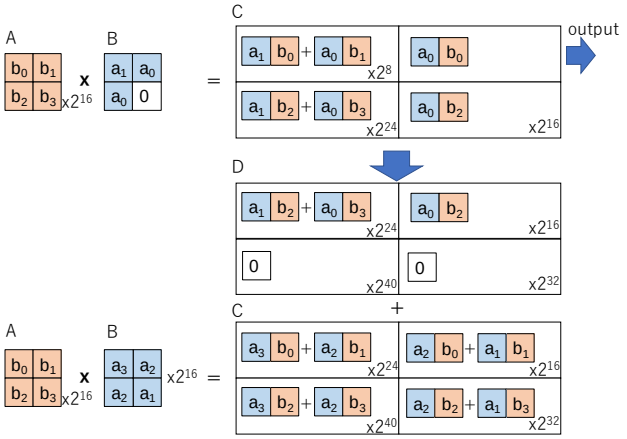

図 6 乗算結果の行列への加算処理

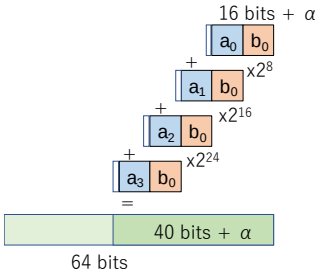

図 7 隣接する 4 成分の繰り上げ処理．8 ビット整数同士の乗算（といくつかの加算）の結果は 16 ビット+α になるため，8 ビットずつずれた 4 成分を足し合わせるには 40 ビット以上必要になる

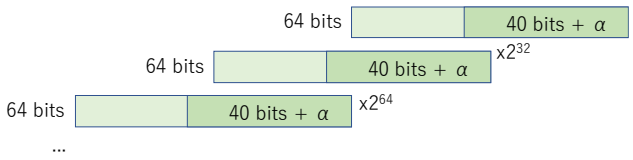

図 8 計算結果を一時的に保存するための 64 ビット整数型配列．32 ビットずつずらした位置の数値を保存する

ところで，16x16 の行列として考えた場合，テンソルコアで 8 ビット整数の乗算と 15 回の加算が行われ，その結果を厳密に表現するには 20 ビット必要である．テンソルコアでは加算は単精度浮動小数点数で計算されるため，その仮数部の有効ビット数 24 ビット内に収めることが可能である．また，図 6 の行列 D で加算を行う回数は最大で 15 回であり，これを加えて厳密に演算結果を保持するのに必要なビット数は 24 ビットとなり，これも単精度浮動小数点数の仮数部に収めることができる．

実際のテンソルコアでの実装では，それぞれの配列はメモリから読み込んで使用する．このメモリには共有メモリも含まれるため，図 5 に示す各行列と図 6 の行列 C および行列 D は，共有メモリ内で処理を行う．また，図 5 で，2 つの B の行列と A の乗算は独立に処理できるが，結果を図 8 の配列に加算するときだけ注意が必要である．整合性を得るために，アトミック演算を用いて結果を加算している．このとき，16x16 の行列を共有メモリに作成するので，複数のスレッドで共通の部分（図 5 の A の行列等）を共有することでさらに最適化が可能となる．

最後に，図 8 の 64 ビット整数に一時的に保存された計算結果の繰り上げ処理を行い，8 ビット整数の配列に正規化する．この処理は，下位の 64 ビット整数から順番に 32 ビット以上の値を 32 ビット右にシフトしたものを，一つ上位の 64 ビット整数に足していく処理を繰り返すことで実装できる．この処理は下位から順番に足していくことが必要なので並列化が困難であるため，並列化が可能な乗算の処理本体とは別の処理として，乗算後に行う．

4.2 Karatsuba 法による多倍長整数の乗算

計算量を減らしてさらに高速化するには，Karatsuba 法を組み合わせるのが有効である．Karatsuba 法では，多倍長整数を再帰的に半分に分割していき，分割された多倍長整数がある程度小さくなった場合に筆算による計算方法で乗算を行う．このとき，多倍長整数の積は式(5)の減算部分にも用いられるため，計算結果は図 8 の配列の 2 か所に，それぞれ加算，減算される．また，式(5)で上位と下位の多倍長整数を加算してできる新しい多倍長整数同士の乗算も筆算の方法で計算し，結果に加算する．最後に，8 ビット整数の配列に正規化を行う．

4.3 多倍長整数の行列ベクトル積の実装

GPU で一組の多倍長整数の積を求めるのは，長さが短い場合並列度の点から考えてあまり効率が良いとは言えない．逆に，サイズの大きな多倍長整数の乗算では，FFT による実装が必要となり，精度の観点から倍精度整数演算が必要となるためテンソルコアによる実装はできない．また，実アプリケーションにおいては単一の多倍長整数の積を求めるケースは少なく，複数の多倍長整数の組を処理する場合がほとんどである．

そこで，実アプリケーションに近い例として，多倍長整数による行列ベクトル積を実装した．複数の多倍長整数の積を同時に計算することで，比較的小さいサイズの多倍長整数でも，GPU で計算するのに十分な並列度が確保できることが期待でき，また，行列ベクトル積や行列行列積では，共通の多倍長整数を参照する計算が現れるため，メモリアクセスや 16x16 行列の生成処理において効率よく処理できることが期待される．本論文では，次の二つの実装を試した．

(1) 多倍長整数の乗算を各成分に独立に行う方法
(2) 多倍長整数の行列ベクトル積を行う専用のカーネルを用意する方法

(2)の実装では，NxM の行列と長さ N のベクトルの積の場合，同じベクトルの成分は M 回参照されるので，ベクト

ル成分をを図 5 の A として，共有メモリを利用して複数のスレッドで共有することで，効率よく処理ができる．

5. Volta GPU による性能評価

5.1 計算機環境

本論文で開発および性能評価に使用した計算機環境を表 1 にまとめる[7]. Tesla V100 が 4 枚搭載されているが, 1GPU のみを使用した．

表 1 性能評価に計算機環境の概要

	IBM Power System AC922 (Newell)
CPU	IBM POWER9
CPU コア数	20
CPU 周波数	2.80 GHz
CPU ソケット数	2
CPU メモリバンド幅	120 GB/s
GPU	NVIDIA Tesla V100
GPU 数	4
Interconnect	NVLink2
コンパイラ	IBM XL C/C++ Compiler version 13.1.7
CUDA Toolkit	9.1

5.2 多倍長整数の乗算の性能評価

まず，一組の多倍長整数の乗算をテンソルコアを用いて行うときの処理時間を評価する．比較として，CPU 上で同じ組の多倍長整数型の乗算を行った場合の処理時間を測定した．CPU 上での乗算には GNU Multi-Precision Library(GMP) [8]を利用した．GMP は多くの計算機に最適化された多倍長整数ライブラリで，乗算には Karatsuba 法の他に FFT による実装もされており，桁数が大きくなると自動的に FFT が使用される．単体の乗算では並列化はされていないので，CPU の 1 コアを用いた測定を行う．GPU での処理時間の測定には, CPU-GPU 間のデータ転送の時間は含めず，GPU のメモリ上にある多倍長整数を処理する前提で測定を行った．また，用意する多倍長整数は乱数で生成されたもので，二つの桁数はおおよそ同じくらいとする．

図 9 は，多倍長整数型の桁数（ビット数）を変えていった場合の，それぞれの実装による乗算の処理時間を示す．まず，GPU と書かれたものは筆算による方法でテンソルコアを用いた場合の処理時間を表す．桁数が大きくなると指数関数的に処理時間が増大するため，4,194,304 ビットを超えると, CPU に逆転されてしまう. 一方の CPU の方は GMP の実装では 262,144 ビットを超えたあたりからグラフの傾きが変化し，FFT が使用されたことが分かる．よって，計算量の違いから処理時間が逆転する．それよりも小さい桁数の場合，テンソルコアによる実装によって数倍～十倍程度高速化できていることが分かる．ただし，桁数が比較的小さな場合は GPU で十分な並列度が得られていないため十分な加速ができず，グラフ上では処理時間がほぼ横ばいとなっている．また，Karatsuba 法による実装では，筆算による方法に比べてグラフの傾きが小さく，計算量が減少できていることが確認できているが，FFT による実装と比較すると計算量がまだまだ多く，これ以上桁数が大きくなると CPU と逆転してしまう．

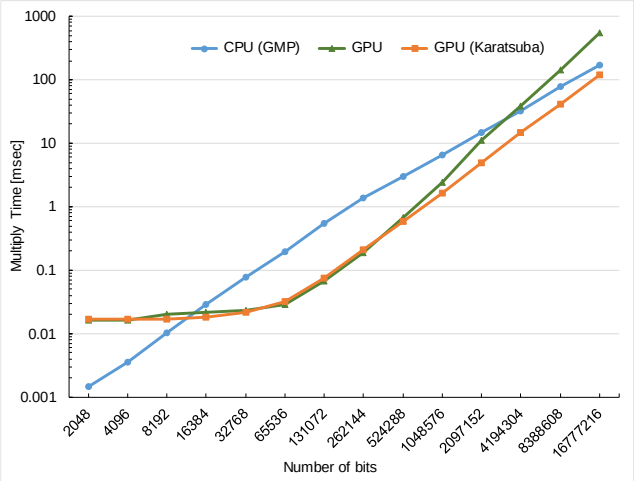

図 9 一組の多倍長整数の乗算の処理時間の実装による比較（単位はミリ秒）．GPU は筆算による方法

5.3 多倍長整数の行列ベクトル積の性能評価

より実用的な評価を行うために，複数の多倍長整数の乗算を行う処理として，行列ベクトル積を評価する．また，図 9 に示すような極端に大きい桁数を持つ多倍長整数演算を行うことはメモリ容量を考えてあまり実用的ではないので，比較的小さな多倍長整数を用いる場合の性能を評価した. 前述の通り, GPU による行列ベクトル積の実装では，一組の多倍長整数演算の乗算を行うカーネル（separate kernel）を組み合わせる方法と，多倍長整数の行列ベクトル積を行うカーネル（integrated kernel）の二つの実装を試した．また，CPU での測定では，NxM 行列と N 成分のベクトルの乗算について M 方向に OpenMP を用いて並列化を行った．それぞれの成分の乗算の計算は GMP の多倍長整数の積和演算を用いた．

図 10 と図 11 は, それぞれ異なるサイズの行列を用いた行列ベクトル積の処理時間の比較を示す．多倍長整数の行列ベクトル積のカーネルを用いることで，多倍長整数の桁数が小さい場合でも十分な加速が得られていることが分かる．しかしながら，単一の乗算を組み合わせただけの実装では，大きな桁数でないと効果が得られていない．GMP による実装ではキャッシュが有効に効くのか，特別な処理をしていないのにも関わらず，非常に高速である．また，行列ベクトル積のカーネルを使用する場合，図 10 のように比較的小規模な行列の場合と比べて，図 11 のような比較的大きな行列の場合の方が桁数が小さいときの加速率は大きい．

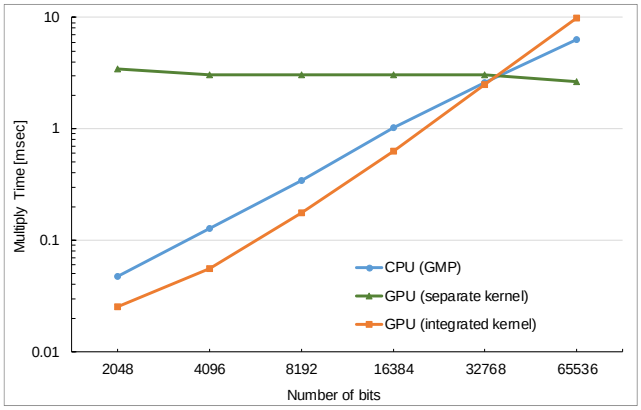

図 10 16x16 の行列を用いた多倍長整数の行列ベクトル積の処理時間の比較（単位はミリ秒）

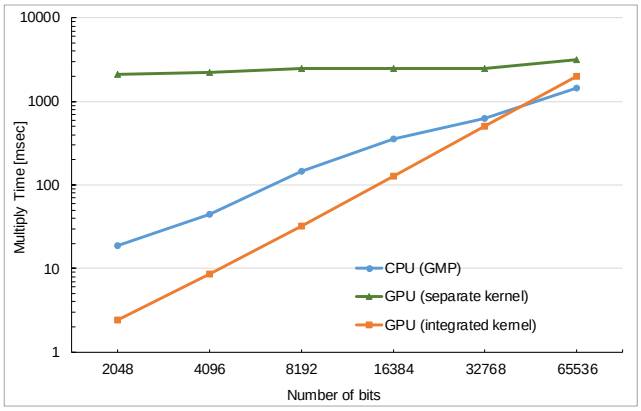

図 11 512x512 の行列を用いた多倍長整数の行列ベクトル積の処理時間の比較（単位はミリ秒）

6. まとめ

多倍長整数型の演算のうち，特計算量が多くよく使われる乗算の処理について，GPU を利用して高速化を行う検討を行った．NVIDIA Tesla V100 GPU に搭載されたテンソルコアは，半精度浮動小数点数の行列積を高速に計算する演算器であるが，計算結果の加算に単精度浮動小数点数を利用することで精度を高める仕組みを，整数演算を厳密にかつ高速に計算することに利用し，多倍長整数の乗算を高速化する手法を提案した．筆算による多倍長整数の乗算と，Karatsuba 法を用いた実装を行い，また，実アプリに近い形で，多倍長整数の行列ベクトル積を計算するカーネルも実装した．

Tesla V100 の実機を用いた性能評価では，単一の多倍長整数の積を計算する処理時間を GMP を用いて CPU で計算する場合と比較し，桁数が大きすぎず，小さすぎない場合に数倍～十倍程度の高速化ができた．しかしながら，桁数が小さい場合は並列度が足りないために十分な加速ができず，逆に桁数が大きいときには GMP は FFT を利用して計算量自体を減らしているため性能が逆転してしまった．

多倍長整数の行列ベクトル積では，小さな桁数の多倍長整数でも十分な並列度が得られるため，GPU による加速率が良いことが確認された．また，専用のカーネルを実装し，ベクトル成分を共有メモリを用いてスレッド間で共有することで処理の効率が改善され，単一の多倍長整数の乗算を組み合わせるよりも高速化された．

今後は，実アプリケーションでどの程度効果があるのかを試したい．その際，他の四則演算での効果や，配列のサイズに関しても考察したい．また，実際のアプリケーションでは，桁数の異なる多倍長整数型が混在する状態になると思われるため，そのような状況でのロードバランスを考慮した実装も求められる．

また，テンソルコアの使い道として，他の処理に応用できないかも検討したい．特に精度を維持したまま積和演算ができる仕組みを利用できるものを整数演算以外にも応用してみたい．

参考文献

[1] NVIDIA Tesla V100, https://www.nvidia.com/en-us/data-center/tesla-v100/
[2] J. Appleyard and S. Yokim, Programming Tensor Cores in CUDA 9, https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/
[3] A. Karatsuba and Yu. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". Proceedings of the USSR Academy of Sciences. 145: 293–294. Translation in the academic journal Physics-Doklady, 7 (1963), pp. 595–596.
[4] A. Schönhage and V. Strassen, "Schnelle Multiplikation großer Zahlen", Computing 7 (1971), pp. 281–292.
[5] IEEE 754: Standard for Binary Floating-Point Arithmetic, http://grouper.ieee.org/groups/754/
[6] CUDA C Programming Guide: Warp matrix functions, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma
[7] Alexandre Bicas Caldeira: “IBM Power System AC922 Introduction and Technical Overview,” An IBM Redpaper publication (2018).
[8] The GNU Multiple Precision Arithmetic Library, https://gmplib.org/