Vol.2010-HPC-124 No.2
2010/2/22

GPUを用いた多倍長整数演算法の設計

田 中 雄 大[†1]　村 尾 裕 一[†2]

本稿ではGPUを多倍長整数演算のためのアクセラレータとして利用する方法を検討する．多倍長整数演算そのものを実装しようとした場合，桁数に応じた動的なメモリ確保や様々な分岐処理が必要であり，これら処理はGPU上での実装には適さない．そこで，GPU上での多倍長整数演算にモジュラー算法を適用し，中国人剰余定理による復元処理をCPU上で実行する方法を提案する．具体例として行列積を取り上げ，NVIDIAが提供するGPGPU向け統合開発環境であるCUDAを用いて実装し，性能評価を行った．GPU上での数の表現に用いるビット幅の違いによる性能の評価も行った．結果，CPU上での4スレッド並列処理に対して最大で4倍の性能を得た．

An Efficient Method for Multiple-Precision Integer Arithmetics Using GPU

Takehiro Tanaka [†1] and Hirokazu Murao[†2]

This paper describes a method to use GPU in order to accelerate the multiple-precision integer(bigint) arithmetics. In general, the bigint arithmetics require a dynamic memory allocation and cause a various style of branching, which can be a problem for the implementation on GPU. In order to adapt to GPU processing, we apply the modular algorithm to bigint arithmetics on GPU, and allot the restoration process by the Chinese remainder theorem on CPU. We implemented matrix multiplication with modular arithmetics using NVIDIA CUDA, an integrated development environment for GPGPU. Our empirical studies include the investigation of performance dependency on the bit width of integers used on GPU. According to our performance evaluation, our method using GPU is 4 times faster at most than 4 thread parallel processing on CPU.

†1 電気通信大学電気通信学研究科情報工学専攻
Department of Computer Science, The University of Electro-Communications
†2 電気通信大学電気通信学部情報工学科
Department of Computer Science, The University of Electro-Communications

1. はじめに

近年，GPU（Graphics Processing Unit）の高性能化に伴い，GPUの処理性能をグラフィックス以外の汎用処理に適用するGPGPU（General Purpose computing on GPUs）が注目されている．GPUは演算コアひとつあたりの処理性能はCPUに劣るが，演算コアを大量に搭載することで単純なデータ並列処理では高い処理性能を発揮する．特に，本来の用途がグラフィックス処理であるため，単精度浮動小数点演算が大変高速であり，このため，GPGPUに関する既存の研究の多くも演算精度が単精度で十分とされる数値シミュレーションなどの一部の分野に限られてきた．

一方で，演算精度が単精度では十分でない科学分野や計算問題も多く存在する．このような分野では，倍精度浮動小数点を用いるのが一般的であるが，さらに厳密な計算を行うという観点から多倍長整数による有理数表現を用いる場合もある．ただし，多倍長整数を用いたことにより，桁数に応じた演算量の増加や多倍長整数を格納するためのメモリ領域の増大などが問題となる．これらの問題に対処する方法として，モジュラー算法を利用することが考えられる．モジュラー算法とは，共通因子をもたないいくつかの法（通常は素数）を使って，多倍長整数を扱う代わりにその剰余について間接的に計算を行う手法である[1)]．

本研究では，このアルゴリズムに注目し，GPUを多倍長整数演算のアクセラレータとして利用する方法を検討する．通常の多倍長整数演算は，桁数に応じた動的なメモリ確保や演算時の分岐処理などが必要であり，GPU上での実装には適さない．しかし，モジュラー算法を用いれば，多倍長整数演算を複数の単一精度の演算として扱えることから，GPU上での処理にも適合しうると予想される．また，GPU上で扱うモジュラー表現からの復元処理をCPUに割り当てることで，CPUとGPUの並列処理による高速な多倍長整数演算が実現できるのではないかと考えた．

本稿では，行列積を対象として適用実験を行い，提案手法を評価した結果を報告する．GPU上での処理の実装には，NVIDIAが提供するGPGPU向け統合開発環境であるCUDA[2)]を用いた．

以下，2章では本研究で注目した多倍長整数演算のアルゴリズムとそのアルゴリズムを計算機上での処理に応用した関連研究について紹介する．3章では注目したアルゴリズムに基づいたGPUを利用する多倍長整数演算の設計について述べる．4章，5章では具体例として行列積を取り上げ，モジュラー算法を適用するための実装法を検討する．また，復元処理まで含めた実行例としてCPUとGPUの並列処理を導入した多倍長整数による行列積の性

© 2010 Information Processing Society of Japan

能を示す．6 章では本研究のまとめを行い，今後の課題について述べる．

2. 多倍長整数演算のアルゴリズムと関連研究

2.1 モジュラー算法

モジュラー算法とは，共通因子をもたないいくつかの法 $p_1, p_2, ..., p_r$ を使って，数 u を直接扱う代わりに剰余 $u \bmod p_1, u \bmod p_2, ..., u \bmod p_r$ について間接的に計算を行う手法である．ここでは，簡単のため

$$u_1 = u \bmod p_1, u_2 = u \bmod p_2, ..., u_r = u \bmod p_r \tag{1}$$

のように定め，剰余の組 $(u_1, u_2, ..., u_r)$ をモジュラー表現とする．

モジュラー表現において注目すべき特徴は，加算，減算，乗算が大変簡単に行えることである．モジュラー表現における演算を以下に示す．

$$(u_1, ..., u_r) + (v_1, ..., v_r) = \big((u_1 + v_1) \bmod p_1, ..., (u_r + v_r) \bmod p_r\big) \tag{2}$$

$$(u_1, ..., u_r) - (v_1, ..., v_r) = \big((u_1 - v_1) \bmod p_1, ..., (u_r - v_r) \bmod p_r\big) \tag{3}$$

$$(u_1, ..., u_r) \times (v_1, ..., v_r) = \big((u_1 \times v_1) \bmod p_1, ..., (u_r \times v_r) \bmod p_r\big) \tag{4}$$

ただし，除算については法のもとでの逆数の乗算を必要とし，他の演算と比較してコストがかかるため注意が必要である．

式 (2)，(3)，(4) は法 $p_1, p_2, ..., p_r$ について独立であり，一般的な多倍長整数演算に見られる carry や borrow の概念が存在しない．また，p_i のそれぞれが計算機のワード長に近ければ $r \approx n$ のとき n 桁の数を扱うことができ，加算，減算，乗算の計算量はそれぞれ $O(n)$ である．乗算については，通常のアルゴリズムが $O(n^2)$ であるため，モジュラー算法を用いる上でかなりの利点となる．そして，法の数 r が十分に大きければ，次に述べる中国人剰余定理により，モジュラー表現 $(u_1, u_2, ..., u_r)$ から数 u が一意に再計算できる．

2.2 中国人剰余定理

定理 1 (中国人剰余定理) $p_1, p_2, ..., p_r$ を，どの 2 つをとっても互いに素である正の整数であるとする．すなわち，

$$p_j \perp p_k \quad (j \neq k \text{ の場合}) \tag{5}$$

が成り立つ．ここで $P = p_1 p_2 ... p_r$ として，$a, u_1, u_2, ..., u_r$ を整数とする．このとき次の条件を満たす整数 u が 1 つだけ存在する．

$$a \leq u < a + P \quad \text{かつ} \quad u \equiv u_j \pmod{p_j} \quad \text{ただし } 1 \leq j \leq r \tag{6}$$

たとえば，正の整数のみを扱う場合は $a = 0$ とし，$0 \leq u < P$ の範囲で正の整数 u を一意に再計算できる．負の整数も扱いたい場合は，$a = \lfloor P/2 \rfloor$ とすれば，$-\lfloor P/2 \rfloor \leq u \leq \lfloor P/2 \rfloor$ の範囲で整数 u を一意に再計算できる．このように，最終的な復元結果に対して必要に応じて値域をずらすことで，正負の整数にそれぞれ対応できる．中国人剰余定理の詳しい証明については文献 1) を参照されたい．

モジュラー表現 $(u_1, u_2, ..., u_r)$ から数 u を再計算するには以下の式を利用する．

$$u = \left(\sum_{i=1}^{r} Q_i \cdot u_i\right) \bmod P \tag{7}$$

ここで用いる係数 Q_i は，法 $p_1, p_2, ..., p_r$ から事前に計算しておくことが可能な値であり，次のようにして求める．

$$P_i = P/p_i \tag{8}$$

$$w_i : 0 \leq w_i < p_i, \quad w_i \cdot P_i \equiv 1 \pmod{p_i} \tag{9}$$

$$Q_i = P_i \cdot w_i \tag{10}$$

式 (9) では法 p_i のもとでの逆数の計算を行っている．これには拡張ユークリッドの互除法などを用いればよい．

以上の方法は，モジュラー算法により計算して得られたモジュラー表現 $(x_1, x_2, ..., x_r)$ から未知の数 x を復元する場合にも適用可能である．ただし，何らかの方法で事前に x の上限を見積もり，十分な数の法を用意しておく必要がある．

2.3 関 連 研 究

モジュラー算法と中国人剰余定理による多倍長整数演算のアルゴリズムについて，計算機上での応用や研究は古くから行われている．文献 3) では，行列式の計算にこのアルゴリズムを適用した例が示されている．行列式の計算では，浮動小数点を用いた場合，大きい数同士の減算でけた落ちが発生し，しばしば精度の低下が問題となる．このような精度の問題を解決するために係数を有理数として多倍長整数で表現すると，中間結果の係数が膨大な桁数となるが，モジュラー算法を適用することで単一精度での計算が可能となり，中間結果の係数の膨張を防ぐことができる．

また，文献 4) では，モジュラー算法と中国人剰余定理を用いた行列の高速乗算法が提案されている．Pentium 200Mhz 上での評価実験を行い，要素ごとに通常の多倍長整数演算を行った場合と比較して，最大で 8 倍程度の速度向上を達成している．理論的な演算量の面からもモジュラー算法と中国人剰余定理を用いた方法が優れていることを示している．

上記の研究はいずれも CPU 上での実装，評価であり，同様のアルゴリズムを GPU を用いた環境で実装，評価した例はない．

© 2010 Information Processing Society of Japan

3. 設計方針

本研究では，GPU の豊富な計算資源を活かすという観点から要素を多倍長整数とする行列演算を対象として設計を行う．

3.1 GPU 上での多倍長整数演算の問題点

まず，GPU 上で通常の多倍長整数演算を実装しようとした場合に問題となる点について以下にまとめる．

- 多倍長整数の桁数に応じて動的なメモリ確保が必要となる
- carry や borrow を扱う際，また多倍長整数の符号や桁数に応じて，しばしば分岐処理が必要となる

前者の問題については，GPU のデバイスメモリ領域は CPU 側から静的にしか確保できないため対応が難しい．対策としては，あらかじめ最大桁数を見積もりメモリ領域を確保しておく方法が考えられる．しかし，この方法では複数の多倍長整数を計算する場合，桁数が一定でなければ多くのメモリ領域が無駄となる可能性が高く，また，デバイスメモリの容量は CPU のメインメモリと比較して小さい傾向にあるため，扱える問題規模も小さくなってしまうことが予想される．

後者のような分岐処理は，GPU のようなメニーコアアーキテクチャ上で処理する際に性能低下を導く要因となるだろう．本研究で実装に用いる CUDA に対応した GPU では，多数の演算ユニットを搭載しているが，命令ユニットは 8 個の演算ユニットに対して 1 個しか搭載されておらず，CUDA の最小実行単位であるワープ（Warp）内において分岐処理が発生した場合，双方のパスが逐次的に実行されることになる[⋆1]．減算においては数の大小により，演算数と被演算数が入れ替わるなど実行パスが大きく異なる場合があり，分岐処理による性能低下が深刻になると考えられる．また，同一ワープ内における処理はすべて最大桁数の演算に足並みを揃えなくてはならない．

3.2 処理の分割

前節で示した問題の多くは GPU 上での計算にモジュラー算法を適用することで解決できる．モジュラー算法では，演算時に用いる法 p を GPU 上で扱えるワード長の範囲にとれば，すべての演算に対してその剰余を結果とするため，どこまでも単一精度での計算が可能となる．これにより，多倍長整数の桁数に応じた動的なメモリ確保の必要性を回避できる．また，carry や borrow の概念もないため，多くの分岐処理を排除できる．モジュラー表現において扱える値の範囲は，選んだすべての法の積に等しく，法ごとの演算は独立である．よって，より高い精度が必要となった場合，法の数を増やすだけで対応できる．

次に，モジュラー算法を用いたことにより，最終的な演算結果である多倍長整数を中国人剰余定理により復元する処理が必要となる．復元処理では通常の多倍長整数演算を要するため，CPU 上での実装を検討する．復元処理においては，用いる法の数を r としたとき，$n \times n$ 行列に対して $r \times n^2$ 回の積和算を行う．一般に $r \ll n$ であることが予想されるので，行列積や LU 分解などの $O(n^3)$ の行列処理に適用すれば，CPU 上での処理を十分に減らすことができると考えられる．また，復元処理以外の途中計算は GPU 上で実行されるので，全体としては既存の CPU のみの実行環境と比較して高速な実装が可能であると考えられる．

3.3 処理の流れ

GPU を用いた多倍長整数による行列演算の処理の流れを以下に示す．

(1) GPU のデバイスメモリ領域を確保する
(2) 入力行列を GPU のデバイスメモリへ転送する
(3) 法の数の繰り返し処理を行う（$1 \leq i \leq r$）
  (a) 入力行列の各要素に対して $\mathrm{mod} p_i$ を GPU 上で並列に計算する
  (b) 法 p_i のもとでの行列演算を GPU 上で並列に実行する
  (c) 法 p_i のもとでの演算結果を CPU のメインメモリへ転送する
  (d) 中国人剰余定理による復元処理を CPU 上で実行する
(4) 各要素に対して $\mathrm{mod} P$ を CPU 上で計算し，最終的な演算結果とする

(3) の処理は，各法のもとでの演算が独立であることから並列処理も含めて様々な処理手順が考えられるが，本研究では，ある法のもとでの行列演算を GPU 上で並列に処理し，その演算結果を CPU のメインメモリへ転送してから，次の法のもとでの行列演算に移行することとした．主に以下の理由による．

- 各法のもとでの演算結果を独立に保持しなければならないため，通常の演算の r 倍のメモリ領域が必要である．演算結果を随時 CPU のメインメモリへ転送することで GPU のデバイスメモリを節約することができ，より大きな規模の問題に対応できる．
- 復元処理に用いる式 (7) に注目すると，Q_i と u_i の積和算となっており，Q_i は事前に計算しておくことが可能な値である．よって，モジュラー算法による演算結果である u_i が決まる度に復元処理を進めることができる．また，GPU と CPU の処理は非同期に

⋆1 ワープは CUDA における最小実行単位であり，ID が連続する 32 個のスレッドの集合である．また，ワープ内での分岐処理の逐次化をワープ・ダイバージェントと呼ぶ．

© 2010 Information Processing Society of Japan

行われるため，復元処理の時間を隠蔽できる可能性がある．

一方で，ひとつの演算結果に対してさらに別の演算を適用したい場合も考えられる．このような場合は法のもとでの演算結果を GPU のデバイスメモリ上に残しておき，再利用するのが望ましいが，このような場合の処理手順については具体的な応用とともに今後検討していきたい．

4. GPU 上でのモジュラー算法による行列積の実装

本章からは具体例として行列積を取り上げ，実装法の検討を行う．

4.1 CUDA による行列積の実装

CUDA による行列積の実装はすでに様々な方法が考案されている[5)6)]．最も手軽な方法としては，NVIDIA が提供する CUBLAS ライブラリ[⋆1] を用いることが考えられる．しかし，CUBLAS ライブラリにおける SGEMM 関数（単精度行列積）をモジュラー算法による行列積に利用しようとした場合，法として使える数が小さくなりすぎるという問題が生じる．また，DGEMM 関数（倍精度行列積）に関しては，GPU の倍精度演算のピーク性能が単精度演算と比較して 1/8 程度であり[⋆2]，転送時間も加えると CPU 上での実装に対して十分に高速な実装は不可能である．よって，本研究では CUDA による行列積関数を一から実装する．

行列積 $C = AB$ について考える．行列 A, B は GPU のデバイスメモリ上に行メジャー形式で格納されているものとする．

CUDA による行列積の実装では，高速なオンチップメモリを活用するためにブロッキングを用いたアルゴリズムが有効である．ブロッキングの方法は複数考えられるが，本研究では 16×16（**図 1**）と 16×64（**図 2**）のブロッキングによる実装を採用した．

16×16 のブロッキングによる実装では，1 スレッドが行列の 1 要素を計算する．図 1 中の行列 A, B の斜線部をブロック単位で共有されるシェアードメモリに適宜コピーすることで，デバイスメモリへのアクセス回数を減らしている．

16×64 のブロッキングによる実装では，1 スレッドがブロック内の 16 要素を列方向に計算する．この方法では，図 2 中の行列 A の斜線部をシェアードメモリにロードし，行列 B の斜線部をレジスタにロードする．なぜならば，各スレッドは列方向に計算を進めるため，行列 B の要素はスレッド間で共有する必要がないからである．主な狙いは，一方の計算要素をレジスタに置くことで最内ループにおけるシェアードメモリへのロード命令数を削減し，演算密度を高めることである．

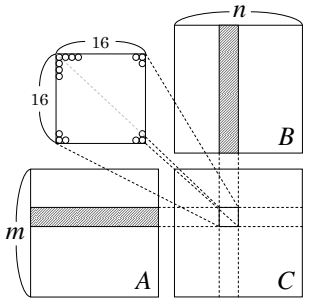

図 1 16×16 ブロッキングによる行列積

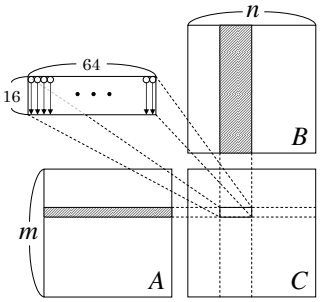

図 2 16×64 ブロッキングによる行列積

両者の実装に関して，単精度浮動小数点による予備実験を行い，16×16 のブロッキングでは 207.1GFLOPS，16×64 のブロッキングでは 343.4GFLOPS の性能を確認した[⋆3]．

4.2 モジュラー算法の適用

つづいて，CUDA による行列積にモジュラー算法を適用する．モジュラー算法を適用することは，複数の法のもとでの演算を行うことである．ここで検討すべきことは，各法をどの程度の大きさとするか，また，法のもとでの演算をどのように実装するかである．

まず，各法の大きさについて考える．前述の通り，各法の大きさを GPU 上で扱えるワード長の範囲に取ることで計算をどこまでも単一精度で行える．データ型としては整数型（unsigned int）か実数型（float）の 32-bit のデータ型を用いることが考えられるが，ここでは，整数型（unsigned int）を用いることにする．なぜならば，整数型が 0 から $2^{32} - 1$ までの整数を正確に表せるのに対し，実数型は 0 から 2^{24} までの整数しか表せず，指数部分の 8bit が無駄となるためである．

法の大きさは積の精度が倍になることを考慮し，$0 < p < 2^{16}$ の場合と $2^{16} \leq p < 2^{32}$ の場合に分けて考える．$0 < p < 2^{16}$ の場合は，入力値から出力値まで，積の中間結果も含めてすべて unsigend int で表現する．$2^{16} \leq p < 2^{32}$ の場合は，入力値及び出力値は unsigned int で表し，積の中間結果は unsigned long long（64-bit）で表現する．ここで，注意すべき

⋆1 http://www.nvidia.com/object/cuda_get.html.
⋆2 NVIDIA GTX200 シリーズの GPU では，搭載される倍精度演算ユニットの数は単精度演算ユニットの数の 1/8 である．
⋆3 CPU-GPU 間通信を含む．

© 2010 Information Processing Society of Japan

点は CUDA の整数乗算命令の性能である．文献 5) によると，整数型に関しては下位 24bit を入力とする高速な`__[u]mul24` 命令を利用することができる．この命令は単精度浮動小数点の乗算命令と同様に 4 クロックサイクルで 1 ワープを処理できる．一方，32bit を入力とする乗算命令は，`__[u]mul24` 命令と比較して 4 倍低速であり，1 ワープの処理には 16 クロックサイクルを要する．(ただし，今後の GPU アーキテクチャの変更に伴ない，整数型の 32bit 同士の乗算命令も高速になる可能性がある．) つまり，現状では $0 < p < 2^{16}$ の場合の方が高速な実装が可能となる．しかし，法の大きさにより復元に必要な法の数が異なり，処理全体の計算量も変わるため，$2^{16} \leq p < 2^{32}$ の場合についても実装し，性能を調べることにする．

次に，法のもとでの演算の実装について述べる．行列積の 1 要素の計算に注目すると，その計算はベクトルの内積である．法のもとでの内積 $\sum_i a_i b_i \bmod p$ の最も単純かつ安全な計算方法は，すべての乗算，加算の後に mod$p$ を計算することである．この方法では，ベクトルの長さを N としたとき総計 $2N$ 回の剰余算命令が必要となる．一般に，剰余算命令は他の算術演算命令と比較しても高価であり，GPU においてもその点は変わらない．そこで，法のもとでの内積における剰余算命令の回数を減らすため，以下に示す 2 つの方法を用いる．

- v, w を中間結果を保存するための同じ型の変数とする．$v = a_i b_i, w = w + v$ を計算していき，$w < v$ の条件で加算におけるオーバーフローを検出した時，補正値を足しこむ．補正値は中間結果の型が unsigned int であれば $adj = 2^{32} \bmod p$ とし，unsigned long long であれば $adj = 2^{64} \bmod p$ とする．
- 内積の結果が中間結果の変数の最大値を超えないように法を指定し，計算を行い，最後に 1 回だけ mod p を計算する．ベクトルの長さを N としたとき，中間結果の型が unsigned int であれば $0 < p < \sqrt{2^{32}/N}$，unsigned long long であれば $0 < p < \sqrt{2^{64}/N}$ とすればよい．

ひとつめの方法では，オーバーフロー検出のために分岐処理が必要となる．CUDA において，分岐処理は性能低下の要因となりやすいが，総計 $2N$ 回の剰余算命令と比較すれば，そのコストは十分小さいと考えられる．ふたつめの方法では，剰余算命令は最後の 1 回だけとなり，行列積の性能への影響は極めて小さいと考えられる．ただし，行列のサイズに応じて扱える法の大きさが小さくなることに注意が必要である．本研究で実装を試みる以上の方法について**表 1** にまとめる．

表 1 モジュラー算法の実装法

実装	法の大きさ	中間結果	剰余算の回数
I	$0 < p < 2^{16}$	32-bit	$2N$
II	$0 < p < 2^{16}$	32-bit	分岐処理
III	$0 < p < \sqrt{2^{32}/N}$	32-bit	1
IV	$2^{16} \leq p < 2^{32}$	64-bit	$2N$
V	$2^{16} \leq p < 2^{32}$	64-bit	分岐処理
VI	$0 < p < \sqrt{2^{64}/N}$	64-bit	1

表 2 実験環境

CPU	Intel Core i7 920
コアクロック	2.67GHz
メモリ容量	3GB
OS	OpenSUSE 11.1
kernel	2.6.27.7-9
CUDA	version 2.3
コンパイラ	gcc version 4.3.2(-O2)

表 3 GPU 環境

GPU	NVIDIA GeForce GTX 285
コアクロック	1.48GHz
SM 数	30
SP 数	240
単精度ピーク	710.4GLOSP
VRAM 容量	1GB
接続バス	PCI Express x16

4.3 性能評価

本研究の実験環境及び利用する GPU は**表 2**，**表 3** の通りである．表 1 に示した各実装法と CPU 上での実装の処理時間の比較を**表 4** に示す．入力行列は正方行列とし，GPU 上での実装については CPU-GPU 間通信も含めた時間を計測している．CPU 上での実装には，最適化 BLAS のひとつである GotoBLAS [⋆1] の DGEMM 関数を利用した．IEEE 規格の倍精度浮動小数点では，0 から 2^{53} までの整数を正確に表現できるので，法を $0 < p < \sqrt{2^{53}/N}$ の範囲にとれば，GotoBLAS による行列積を適用した後，各要素について mod$p$ を 1 回だけ計算すればよい．CPU 上での実装は 4 スレッド並列処理を行っている．

法のもとでの内積計算の高速化技法の効果を検証する．中間結果を 32-bit とする場合と 64-bit とする場合に分けて考察する．

中間結果を 32-bit とする場合では，16×64 のブロッキングを用いており，実装 III，実装 II，実装 I の順に高速であった．実装 III は剰余算を最後に 1 回しか適用しないため，行列積の性能低下はほとんどなく，FLOPS 換算で 337.8GFLOPS となり，単精度浮動小数点を

⋆1 http://www.tacc.utexas.edu/resources/software/.

© 2010 Information Processing Society of Japan

表 4 表 1 の各実装法と CPU 上での実装に対する行列サイズごとの処理時間（単位：秒）

サイズ	CPU	I	II	III	IV	V	VI
128	0.000241	0.000684	0.000353	0.000216	0.009408	0.000431	0.000315
256	0.001279	0.003482	0.000865	0.000556	0.043459	0.001795	0.001204
512	0.008227	0.025119	0.003696	0.002295	0.331893	0.010361	0.007494
1024	0.063204	0.192298	0.023825	0.009952	2.573477	0.074627	0.044547
2048	0.457446	1.491079	0.144302	0.065045	-	0.528063	0.330858
4096	3.849261	-	1.047754	0.406855	-	4.044150	2.527607

用いた場合とほぼ同等の性能を示した．実装 II は分岐処理によりオーバーフローを検出し，補正値を足しこむ方法であるが，実装 III に対して処理時間は 2.5 倍程度に留まった．一方，総計 $2N$ 回の剰余算を行う実装 I では，行列サイズが 1024 の時点で処理時間が 10 倍以上となり，CUDA における剰余算命令が圧倒的に高価であることを示している．GPU 上での演算にモジュラー算法を適用する上で剰余算命令の削減は必須であることがわかった．

中間結果を 64-bit とする場合も，32-bit の場合と同様の傾向が見られた．同じ高速化技法を施した 32-bit の実装と比較すると，4〜6 倍ほど低速であった．これは，主に CUDA の整数乗算命令の性能がそのまま現れたと考えられるが，中間結果を 64-bit とする実装では，ブロッキングを変更しても性能改善が見られなかった．この事実は中間結果を 64-bit とする実装では演算そのものがボトルネックとなり，ロード命令数を削減する方法ではこれ以上の性能改善を行えないことを意味している．また，中間結果を保存するためのレジスタ数が増えたことにより，最内ループにおけるループアンローリングが効果的に働かなかったことも一因と考えられる．

CPU 上での実装と比較すると，実装 II，実装 III，実装 VI が CPU 上での実装よりも高速であった．実装 VI は扱える法の範囲は CPU 上での実装より大きいが，処理時間は CPU 上での実装の 66 %程度であり，CPU と GPU のピーク性能比を考慮すると有効であるとは言い難い．実装 II，実装 III の処理時間は CPU 上での実装に対してそれぞれ 27 %，10 %ほどであり，復元処理のコストを無視すれば，十分に利用価値があると考えられる．しかし，いずれの実装も CPU 上での実装と比較して扱える法の範囲が小さいため，同じ桁数の多倍長整数演算を行う場合でも，より多くの数の法を必要とする．そのため，さらに議論を進めるためには，実際の利用場面での復元処理まで含めた性能評価実験が必要であろう．

5. 復元処理を含めた実行例

本章では，要素を多倍長整数とする単一の行列積について復元処理まで含めた実装を行い，性能を評価する．

5.1 CPU と GPU の並列処理

本研究では CPU 上での処理と GPU 上での処理が非同期に行われることに注目し，CPU と GPU の並列処理による多倍長整数演算の高速化について検討する．具体的には，CPU 上での復元処理と GPU 上でのモジュラー算法による行列演算を並列に行うことで，どちらか一方の処理時間を隠蔽することを考える．

CUDA では，ひとつの GPU のコンテキストをひとつの CPU スレッドに対応付けるため，CPU と GPU の並列処理を実現しようとした場合，CPU 側ではマルチスレッド処理が必要となる．本研究では OpenMP を用いて，GPU 上での並列処理関数の呼び出しや CPU-GPU 間通信を担当するスレッドと多倍長整数の復元処理を担当するスレッドを生成し，ふたつのスレッド間で同期を取りながら処理を進めていく形で実装した．

5.2 性 能 評 価

正方行列のサイズが 2048 と 4096 のとき，以下の方法について処理時間を計測した．

- CPU による 4 スレッド並列処理（行列演算，復元処理ともに CPU 上で実行）
- CPU と GPU による逐次処理（行列演算と復元処理のオーバーラップなし）
- CPU と GPU による並列処理

多倍長整数の入力は簡単のため 2^{32} を一桁とする固定長の配列で与えた．CPU 上での復元処理の実装には多倍長演算ライブラリの GMP を用いた．入力行列の最大入力桁数を x，サイズを N とすれば，計算要素の最大値は $x^2 \times N$ で求められる．法は各々の実装において取りうる範囲の素数を大きいものから順に選び，その総積が $x^2 \times N$ を超えるように法の数を決定した．入力桁数を一桁ずつ増やしていき処理時間の変化を調べた．それぞれの方法の処理時間の比較を**図 3**，**図 4** に示す．GPU を用いた実装に関しては，GPU を利用するために必要な前処理も含めた時間を計測しており，それぞれの行列サイズにおいて最も処理時間の短かった実装のみを示している．

いずれの方法においても，入力桁数に応じて法の数がほぼ線形に増加するため，処理時間もほぼ線形の増加を見せている．

行列サイズが 2048 のとき，実装 II が最も高速であった．CPU と GPU の並列処理を導入しない場合で 2 倍，導入した場合で 3 倍の速度向上を示した．各ステップの処理時間に

© 2010 Information Processing Society of Japan

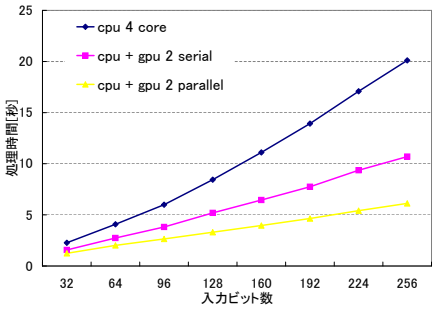

図 3 行列サイズ $N = 2048$ のときの処理時間の比較

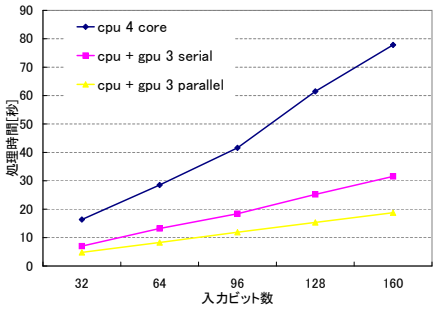

図 4 行列サイズ $N = 4096$ のときの処理時間の比較

注目すると，GPU 上でのひとつの法のもとでの行列積の処理時間が 0.149 秒，CPU 上での復元処理が 0.092 秒であり，CPU と GPU の並列処理により復元処理の時間の多くを隠蔽できていることが確認できた．一方で実装 III を用いた場合，1 ステップの行列積の処理時間が 0.052 秒であり，用いる法の数から 1.5 倍のステップ数を要するため，CPU 上での復元処理の時間のほうが支配的となり，実装 II を用いた場合よりも低速であった．

行列サイズが 4096 のとき，実装 III が最も高速であった．CPU と GPU の並列処理を導入しない場合で 2.4 倍，導入した場合で 4 倍の速度向上を示した．各ステップの処理時間は，行列積が 0.369 秒，復元処理が 0.368 秒であった．

以上の結果から，CPU と GPU に処理を分割することで，ステップ数は増加するが，全体の処理時間を短縮できることがわかった．また，CPU と GPU の並列処理により復元処理の時間を隠蔽することでさらに処理時間の短縮を確認した．その効果は並列処理を導入しない場合と比較して最大で 1.7 倍程度であった．ただし，GPU 上での行列積の実装には，単純に最も高速な実装を選ぶのではなく，問題規模に応じて CPU の処理時間とのバランスを考慮した実装を選ぶ必要がある．

6. ま と め

本研究では，GPU を用いた環境で多倍長整数演算を行う方法として，モジュラー算法と中国人剰余定理にもとづき GPU と CPU に処理を分割する方法を提案した．モジュラー算法による独立かつ単一精度の計算が GPU 上での処理に適合しうると考え，具体例として行列積を取り上げ，モジュラー算法を適用するための実装法を検討した．CPU 上での実装と比較して，単純な処理時間では 9.5 倍高速な実装が可能であることがわかった．一方で，CPU 上での実装に対して優位な性能を得るには，法の大きさが最大でも 2^{16} 程度限られるという問題が浮上した．そのため，復元処理まで含めた実装では，計算に必要なステップ数が増加してしまうが，CPU と GPU の並列処理を導入することで復元処理の時間を隠蔽し最大で 4 倍の速度向上を達成した．

本研究では，提案手法を単一の行列積に適用するという最も単純な例を示したに過ぎない．今後の課題は，提案手法をより現実的な問題に適用することである．例えば，連立一次方程式の反復解法などが挙げられる．

最後に，今後の GPU アーキテクチャの変化と本研究の関連について述べる．NVIDIA の次世代 GPU では，本研究の実験環境である GTX200 シリーズの GPU と比較して，倍精度演算のピーク性能が 8 倍になるとしている[7]．これにより，データ表現として CPU 上での実装と同様に倍精度浮動小数点を用いた場合でも高速な実装が可能となり，CPU 上での実装と同じ法の数で多倍長整数演算を行えるようになるだろう．さらに，ベンダーが提供する CUBLAS ライブラリの倍精度関数を利用することで提案手法をより手軽に様々な行列演算に適用できるようになることが期待される．

参 考 文 献

1) Knuth, D.E.: *The Art of Computer Programming, Volume 2, Seminumerical Algorithms, Third Edition*, Addison-Wesley (1997).
2) NVIDIA: CUDA Zone, http://www.nvidia.com/object/cuda_home.html.
3) 高橋秀俊，石橋善弘：電子計算機による exact な計算の新方法：modulo p 演算の応用，情報処理， Vol.1, No.2, pp.78–86 (1960).
4) 後保範：多倍長精度の値を係数とする行列の高速乗算方式 (偏微分方程式の数値解法とその周辺 II)，数理解析研究所講究録， Vol.1198, pp.170–178 (2001).
5) NVIDIA: NVIDIA CUDA Programming Guide, Version 2.3.1 (2009).
6) Volkov, V. and Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebra, *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, IEEE Press, pp.1–11 (2008).
7) NVIDIA: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009).

© 2010 Information Processing Society of Japan