GPU においてリダクション演算のアトミック処理競合を抑える方法と評価

田原司睦† 中平直司†

複数スレッドを用いてリダクション演算を並列に計算する場合，アトミック処理を用いる方法がある．個々のデータ毎に足し込む先が異なるプログラムを並列化する場合は，特に有効である．スレッド間で競合が発生した場合，アトミック処理は不可分な処理部分を逐次実行することでデータ破壊を防ぐ．一方，多数のスレッドが一斉にアトミック処理を行うと，競合が多発して処理速度が著しく低下する場合がある．この論文では，GPU でアトミック処理を用いた場合の競合を抑え，高速にリダクション処理を実行する手法を紹介する．また，実用的なプログラムに適用し，有効性を示す．

Algorithm to Reduce Contention of Atomic Operations for Reduction Operations on GPU

TSUGUCHIKA TABARU† and TADASHI NAKAHIRA†

An atomic operation can be applied to parallel reduction operations. It is useful especially in the case when multiple sums are calculated in parallel, and each datum is determined to which sum it belongs to. An atomic operation keeps consistency of memory data using inseparable operations without disturbance from other memory operations. This mechanism, however, slows down the program seriously under highly contented situation of atomic operations. In this paper, we introduce our method which reduces contention of atomic operations on graphics processing unit, describe the implementation, and show the performance of three practical programs applying our method.

1. はじめに

2007 年頃から, Graphics Processing Unit (GPU)で汎用計算を行う動きが活発になった．GPU がハードウェアとして汎用計算をサポートしたことと共に，GPU のプログラミング環境が充実したことも要因となっている．

汎用計算向け GPU では，NVIDIA 社と AMD 社が有名であり, どちらも GPU 向けプログラムの開発環境を提供している．NVIDIA 社の開発環境は，GPU 向け専用プログラミング言語 CUDA 1)である．この他，The Poland Group Inc. 社や CAPS entreprise 社は，汎用言語で GPU 向けプログラムを開発できるコンパイラを製品化している．GPU は Central Processing Unit (CPU)とは異なった特徴をもっており，GPU の特徴を生かす言語やコンパイラ，性能評価の研究も多数行われている 2)3)4)5)6)7)8)9).

GPU では，同じプログラムを多数のスレッドで実行することで性能を引き出す．主に，ループ処理をスレッドに割り振る粗粒度の並列化が行われる．このため，逆依存，出力依存を含めたデータ依存がループの繰返しにまたがって存在すると，並列化が阻害され，GPU の性能を引き出せない．リダクション演算にも依存があるが，ループ内でリダクション演算の結果を利用しない限り例外的に並列化できる．

ここでいうリダクション演算とは，結合法則と交換法則が成り立つ二項演算子 1 種類だけからなる演算である．リダクション演算の例として，数式 1 に実数の数列 a_i の加算を挙げる．

数式 1
$$s=\sum_{i=0}^{n-1}a_i=a_0+a_1+a_2+\ldots+a_{n-1}$$

結合法則と交換法則の両立から，リダクション演算は演算順序によらず計算可能である a). 計算順序を変えた具体例を，数式 2 に示す.

数式 2
$$\begin{aligned}s&=a_0+a_1+a_2+a_3\\&=((a_0+a_1)+a_2)+a_3\\&=(a_0+a_1)+(a_2+a_3)\\&=(a_0+a_2)+(a_1+a_3)\end{aligned}$$

リダクション演算の演算子をノード，数列を葉として二分木で表現した場合，同じレベルのノードは計算機で並列に演算できる．数列の長さを n とすると，一番低い二分木

* †（株）富士通研究所
Fujitsu Laboratories Ltd.

a) 有限精度の浮動小数点演算の加算等では，演算順序によって丸め誤差の分だけ答えが変わる．

グラフの高さ h は$\lceil \log_2 n \rceil$である．この時，可能な限り並列に演算を行うために必要な最小の演算器の個数 p は，数式 3 で表される．p 個の演算器で並列計算を行えば，グラフの高さ分の演算時間で演算が完了するため，高速に演算できることがわかる．演算器が豊富な GPU では，この方法が重要である 10).

数式 3
$$p = \begin{cases} 2^{h-2} & \left(n : 2^{h-1} < n \le 2^h - 2^{h-2}\right) \\ n - 2^{h-1} & \left(n : 2^h - 2^{h-2} < n \le 2^h\right) \end{cases}$$

計算を正しく行うためには，子ノードの演算結果が得られることを保証するための同期が必要となる．一方，GPU では，限られたスレッド間でのみ同期が可能である．GPU 内の任意のスレッド間で同期をとるには，GPU のプログラム（カーネル関数）を終了させる必要がある．このため，ブロック内でリダクション演算を行うカーネル関数を複数回起動することで，全体のリダクション演算を実現する．カーネル関数の起動を繰り返す間に必要となる中間バッファを管理するコードも必要となる．逐次実行であればループ内の加算 1 つで簡潔に表現できる処理が，並列化により複雑なプログラムになる．

数式 4 はリダクション演算の応用であり，足しこみ先を項毎に実行時に決定している．この場合にも，上記の方法は利用できない．

数式 4
$$s_j = \sum_i a_i \quad \left(i : a_i \text{の足しこみ先が} s_j\right)$$

コンパイラ等でリダクション演算を並列化する場合，複数のリダクション演算に対応し，それぞれのスレッドから足し込む項数も，実行時に決定できる必要がある．このような複雑な場合にも対応できる手法として，メモリ上のデータをアトミックに処理する方法が挙げられる．アトミック処理は，以下の 3 つの処理を他のスレッドに邪魔されることなく行う操作のことである．

(1) **メモリのあるアドレス *A* から値 *V* を読む.**
(2) ***V* を用いて何らかの処理や計算を行い，結果 *R* を得る. 処理の種類はアトミック命令による.**
(3) ***R* を *A* に書き込む.（*A* に別の値 *S* を書きこむか，書き込まないかを *R* で決めて処理するアトミック命令もある.）**

アトミック処理は，CPU や GPU に備わっている専用の命令を利用して実現する.アトミック処理を用いることで，明示的な同期が不要になり，書き込み先が項毎に変わる場合にも対応できる．更に，並列化のためのプログラムの変更を少量に抑えられるため，コンパイラ等で利用しやすい．

アトミック処理のデメリットとして，競合時の速度低下が挙げられる．アトミック処理は，競合が起こった場合に 1 つずつ処理を行うため，単純には競合数分の時間がかかる．GPU のように数百以上の並列度を持つプロセッサでは，競合の多発により，実行時間が数十倍に延びることもある．

このように速度低下は重要な問題であるため，GPU を活用してリダクション演算を行う際，速度低下を防止する方法を考案した．以下，具体的にこの方法を述べ，評価結果を紹介する．２章では，競合によるアトミック処理の速度の低下を防止するアルゴリズムを述べる．また，ハイパフォーマンスコンピューティングの分野で使われることの多い NVIDIA 社の GPU のアーキテクチャ, Fermi 11)と Kepler 12)に向けた実装例を紹介する．３章では，実装例を使った 3 つの評価結果と効果について述べる． ４章で結論と今後の研究課題についてまとめる．

2. アルゴリズムと実装

アトミック処理の競合を避けるには，アトミック処理数を削減すればよい．リダクション演算を行うアトミック処理であれば，スレッド内である程度リダクションを行ってからアトミック処理を行うことで，アトミック処理数を削減することができる．この方法では時間方向にリダクションを行うため，スレッド毎に合計を保持する変数（バッファ）を必要とする．

数式 4 の場合，バッファは配列変数等で確保することになる．バッファの総容量は，配列変数の大きさと，スレッド数の積で決まる．どちらか，または両方がプログラム実行時に決定する場合，実行時までバッファを確保できるかどうかが判らない．これは，特にコンパイラによる並列化に応用する際に障壁となる．

他に考えられる方法として，アトミック処理で競合する値をスレッド間でリダクションし，アトミック処理を削減する方法が挙げられる．これはバッファが不要な空間方向の集約であり，スレッド数の増減に依らず利用可能な方法である．特に多数のスレッドが同時に走るアーキテクチャで有効な手法である．

Fermi アーキテクチャと Kepler アーキテクチャでは，膨大な数のスレッドを扱うことができる．これらのアーキテクチャでは，スレッドは共通の階層構造をもった集合に分けられて管理される．スレッドの集合の概念には，ワープとブロックの 2 つがある．

ワープは，32 スレッドを束ねたもので，ワープに 1 つの命令ポインタを持ち，32 スレッドで同じ命令を実行する．このため，ワープ内スレッドは常に同期している．条件分岐はプレディケートを用いて，分岐先まで命令を実行しな

いことで実現される．以下，命令を実行しているスレッドを有効スレッドと呼ぶことにする．

ブロックは，ワープをいくつか集めたものである．ブロック毎に共有メモリが割り当てられ，ブロック内では高速にデータを共有できる．また，ブロック内では，スレッド間のバリア同期も可能である．

以下，Fermi アーキテクチャ向けの例として，ワープ単位でリダクションを行うアルゴリズムと，ブロック単位でリダクションを行うアルゴリズムを紹介する．また，前者を Kepler アーキテクチャ向けに改良し，CUDA の第 4.2 版で実装した例も示す．

2.1 Fermi アーキテクチャ向けアルゴリズムと実装例

Fermi アーキテクチャにおいて，ワープ内で競合をまとめ，アトミック処理で加算を行うアルゴリズムを図 1 に示す．

図 1 では，まず，ワープ内の全有効スレッドが，共有メモリを介して，各スレッドの書き込み先とデータを共有する．次に，ワープ内で互いの書き込み先をスキャンしていく．各スレッドは，自分の書き込み先と同じアドレスをスキャン中に見つけた場合，競合回避のためにリダクションを行う．この時，自身のブロック内スレッド番号（threadID）よりも大きい番号を持つスレッドのデータのみをリダクションする．スキャンが終わると，競合スレッド内の一番小さいブロック内スレッド番号（writerID）を持つスレッドがアトミック処理を行う．こうすることで，ワープ内の競合は全て解消される．

```
Atomic_operation (addr, data, addrBuf, dataBuf)
     addr: 書き込み先アドレス
     data: 足し込む値
     addrBuf: ワープ内で addr を共有するための共有メモリ
     dataBuf: ワープ内で data を共有するための共有メモリ
 1 threadID ← ブロック内のスレッド番号
 2 addrBuf[threadID] ← addr
 3 dataBuf[threadID] ← data
 4 writerID ← threadID
 5 sum ← 0

 6 for ワープ内の各有効スレッド a
 7   do
 8     if addr = (a の addr)
 9       then
10         if threadID < (a の threadID)
11           then sum ← sum + dataBuf[a の threadID]
12         if (a の threadID) < writerID
13           then writerID ← (a の threadID)

14 if writerID = threadID
15   then
16     アトミック処理で
         v ← * addr
         * addr ← (v + sum + data)
17     dataBuf[threadID] ← v

18 戻り値 ← dataBuf[writerID] + sum
```

図 1　共有メモリを使用して，ワープ内のアトミック加算の競合をなくすアルゴリズム．

図 1 のアルゴリズムで時間と共にリダクションが進んでいく様子を示した例を，図 2 に挙げる．図中，ループは threadID の小さなものから処理するとした．ワープは 8 スレッドで構成され，スレッド 0 と 7 がアドレス A に，スレッド 1，4，6 がアドレス B に，スレッド 3 と 5 がアドレス C に足し込むと仮定する．ループ繰返しの 0 から 2 のⓌで，最終的にアトミック処理を行うスレッドが，スレッド 0, 1, 3 に決定する．ループ繰返しの 3 から 6 の⊕で，競合するスレッド間でリダクションを行う．スレッド 2 は非アクティブなため，処理に参加していない．

図 1 で示したアルゴリズムを CUDA で記述したものが図 3 である．ワープ内データ共有は，共有メモリ上の配列変数 dataBuf と addrBuf で行う b)．これらの長さは，1 ブロック当りのスレッド数である．また，インラインアセンブラを用いて，ワープ内スレッド番号（laneID）を取得している．

CUDA の組み込み関数 4 つを利用して，有効スレッドの取得と，処理スレッドの選択を行っている．__ballot(true) は，ワープ内スレッドが有効な場合 1 を，無効な場合 0 を，ワープ内スレッド番号に対応するビットの値として返す．ワープは 32 スレッドで構成されているため，32 ビットの整数値としてワープ内のスレッドの状態を一括して得ることができる．__brev() は，32 ビット整数のビットの並びを上下で反転した値を返す関数である．__popc() は，32 ビット整数を引数にとり，値が 1 のビットを数える関数である．__clz() は，32 ビット整数を引数にとり，値が 0 のビット

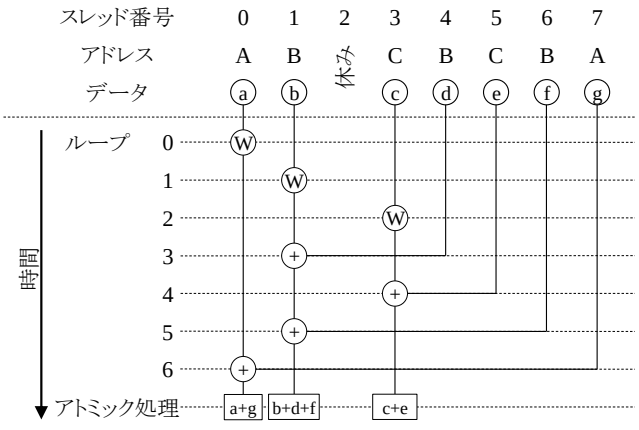

図 2　図 1 のアルゴリズムで，時間と共にリダクションが進んでいく様子を示した例．

b) dataBuf と addrBuf の型を工夫して，他のデータ型のアトミック処理関数とバッファを共有する方法があるが，ここでは割愛する．

が，最上位ビットからいくつ連続しているかを返す．

図 3 は，関数内で共有メモリを確保することにより，関数の引数から addrBuf と dataBuf を省略できる．これにより，CUDA のビルトイン関数(atomicAdd())と同じ引数にできる．しかし，省メモリ化のために，複数関数で共有メモリの同じ領域を一時バッファとして利用できるよう，これらの引数は削除していない．

次に，ブロック内で競合をまとめる方式のアルゴリズムを図 4 に示す．ワープ間通信のために，バリア同期を利用する．この方式では，ワープ内でまとめる方式に比べ，競合の解消がより進むと期待できる．一方，ブロック内の全ワープがほぼ同時にこのアルゴリズムを実行できない場合，正しい実行結果を得られない可能性がある．

例として，ブロック内のあるワープ A が，条件分岐によって全くこのアルゴリズムを実行しないまま終了する場合を考える．同じブロックの他のワープは，アルゴリズム中のバリア同期でワープ A を待ち続ける．ワープ A 完了後は，バリア同期でワープ A を待たなくなるため，残ったワープは動作を再開する．この時，共有メモリ上でワープ A に割り振られた領域には，不正な値が残っていると考えられる．残ったワープが不正な値を用いて処理を行うため，結果も不正な値となる．

```
__device__ inline int atomicAddW(int* addr, int data,
        int* dataBuf, int** addrBuf){
    int threadID = (((threadIdx.z * blockDim.y)
            + threadIdx.y) * blockDim.x) + threadIdx.x;
    dataBuf[threadID] = data;
    addrBuf[threadID] = addr;
    int laneID;
    asm("mov.s32 %0, %%laneid;" : "=r"(laneID));
    int warpTop = threadID - laneID;
    int* wData = & (dataBuf[warpTop]);
    int** wAddr = & (addrBuf[warpTop]);
    int writerID = laneID;
    int sum = 0;

    unsigned int active = __brev(__ballot(true));
    for (int aID = __popc(active); aID --;){
        int rID = __clz(active);
        active ^= 1U << (warpSize - 1 - rID);
        if (addr == wAddr[rID]){
            if (laneID < rID){ sum += wData[rID]; }
            if (rID < writerID){ writerID = rID; }
        }
    }
    if (writerID == laneID){
        wAddr[laneID] = atomicAdd(addr, sum + data);
    }
    return wAddr[writerID] + sum;
}
```

図 3　図 1 のアルゴリズムを 32 ビット整数のアトミック加算向けに CUDA で記述した例．インラインアセンブラは， laneID を取得するために利用．

ワープ内で競合をまとめる方式では，このような問題は無く，利用しやすい．以下，ワープ内で競合をまとめる方式を本方式と呼ぶ．

2.2 Kepler アーキテクチャ向けの実装例

NVIDIA の Kepler アーキテクチャでは，シャッフル機能によりワープ内のデータ共有が可能となった．シャッフル機能を利用すると，図 1 の共有メモリへの登録が不要になる．図 5 は，シャッフル機能を利用し，32 ビット整数の加算を行う例を CUDA で記述した例である．この関数の形は，CUDA のビルトイン関数 atomicAdd()と同じ形をしているため，ビルトイン関数からの置き換えが容易である．演算に関係しない引数が不要なため，コンパイラの並列化機能などでも利用しやすい．

3. 評価

本方式の効果を見るために，本方式を利用したアトミック処理と，利用しないアトミック処理の性能について評価した．また，リダクション処理を CPU で行った場合についても調査した．評価は，ヒストグラム生成プログラム，姫野ベンチマーク 13)，及び，量子化学計算プログラムの 3

```
Atomic_operation (addr, data, addrBuf, dataBuf)
     addr: 書き込み先アドレス
     data: 足し込む値
     addrBuf: ブロック内で addr を共有するための共有メモリ
     dataBuf: ブロック内で data を共有するための共有メモリ
 1 threadID ← ブロック内のスレッド番号
 2 ブロック内バリア同期
 3 addrBuf[threadID] ← addr
 4 dataBuf[threadID] ← data
 5 ブロック内バリア同期
 6 writerID ← threadID
 7 sum ← 0

 8 for ブロック内の各有効スレッド a
 9   do
10     if addr = (a の addr)
11       then
12         if threadID < (a の threadID)
13           then sum ← sum + dataBuf[a の threadID]
14         if (a の threadID) < writerID
15           then writerID ← (a の threadID)

16 ブロック内バリア同期
17 if writerID = threadID
18   then
19     アトミック処理で
         v ← * addr
         * addr ← (v + sum + data)
20     dataBuf[threadID] ← v
21 ブロック内バリア同期

22 戻り値 ← dataBuf[writerID] + sum
```

図 4　共有メモリを使用して，ブロック内のアトミック加算の競合をなくすアルゴリズム．

種類で行った．

評価に用いたハードウェアとソフトウェアの構成を表 1 に示す．GPU には，Fermi アーキテクチャの C2075 のみ，または，C2075 と Kepler アーキテクチャの GTX680 の 2 種類を用いて評価した．GPU 用コンパイラには CUDA 4.2 の nvcc を用いた．CPU 用のコンパイラには，G++の第 4.4.7 版を用い，最適化オプションには—O3 を指定した．また，CPU 向けの並列化は OpenMP 14)で行い，並列化のスケジュールには schedule(guided)指示節を用いた．

以下，順に詳細を記す．

3.1 ヒストグラム生成プログラムによる評価

本方式の基本的な振舞を確認するために，単純なヒストグラム作成プログラムで評価を行った．ヒストグラム作成に使用するデータは，0 以上 1 未満に平坦に分布する 10^7 個の乱数で，形式は倍精度浮動小数点とした．ヒストグラムのビン（カテゴリ）数は 1 から 10^7 までとし，10 倍刻みで 8 つのヒストグラムを作った．ビン幅は均等とした．ビンの変数は整数型と倍精度浮動小数点の両方で評価した．

ヒストグラム生成は，データを読み，浮動小数点演算でビン番号を求め，対象のビンに 1 を加える，という処理を繰り返して行った．ビンに 1 を加える処理は，並列化した場合はアトミック処理で行った．

GPU でのアトミック処理は，本方式を採用した場合と採用しない場合の 2 通りで評価した．GPU では倍精度浮動小数点のアトミック加算がサポートされていないため，文献 1)の付録 B.11 にある例を利用した．いずれの場合も，GPU では，1 ブロック当り 128 スレッドとし，1048576 スレッドで処理を行った．

CPU では，単一コアでの逐次処理と，OpenMP を用いた 6 コア 6 並列処理での評価を行った．

図 6 は，ビンの変数が倍精度浮動小数点の場合の結果である．縦軸はヒストグラム生成部の処理時間を，横軸はヒストグラムのビン数を表している．上記の 4 通りで処理した結果をプロットした．

本方式の効果は，競合頻度の高いビン数の少ないところに現れている．ビン数が 1 の時，アトミック加算処理に比べ C2075 で 747 倍高速化された．ワープ中のスレッドをまとめるだけなので，速度向上は 32 倍以下となるはずだが，実測では大幅に上回った．原因は，倍精度浮動小数点のア

```
__device__ inline int atomicAddW(int* addr, int data){
    int addrH = (unsigned long long int)addr >> 32;
    int addrL = (unsigned long long int)addr;
    int laneID;
    asm("mov.s32 %0, %%laneid;" : "=r"(laneID));
    int writerID = laneID;
    int sum = 0;

    unsigned int active = __brev(__ballot(true));
    for (int aID = __popc(active); aID --;){
        int rID = __clz(active);
        active ^= 1U << (warpSize - 1 - rID);
        if ((addrL == __shfl(addrL, rID))
                && (addrH == __shfl(addrH, rID))){
            int dataI = __shfl(data, rID);
            if (laneID < rID){ sum += dataI; }
            if (rID < writerID){ writerID = rID; }
        }
    }
    int oldValue = 0;
    if (writerID == laneID){
        oldValue = atomicAdd(addr, sum + data);
    }
    oldValue = __shfl(oldValue, writerID);
    return oldValue + sum;
}
```

図 5　シャッフル機能を利用して，図 1 のアルゴリズムを 32 ビット整数加算向けに CUDA で記述した例．インラインアセンブラは，laneID を取得するために利用．

表 1　評価に用いたハードウェアとソフトウェアの構成．

部品	項目	値
CPU	型	Intel Core i7 3960X
	コア数	6
	コア周波数	3.3 GHz
	メモリ容量	32 GiB
GPU 0	型	NVIDIA GTX680
	CUDA コア数	1536
	コア周波数	1.06 GHz
	メモリ容量	2 GiB
GPU 1	型	NVIDIA C2075
	CUDA コア数	448
	コア周波数	1.15 GHz
	メモリ容量	5 GiB
ソフトウェア	オペレーティングシステム (OS)	GNU / Linux
	OS のカーネルの版	3.2.0
	CPU 開発環境	G++ 4.4.7
	GPU ドライバ	NVIDIA 製 304.32
	GPU 開発環境	CUDA 4.2

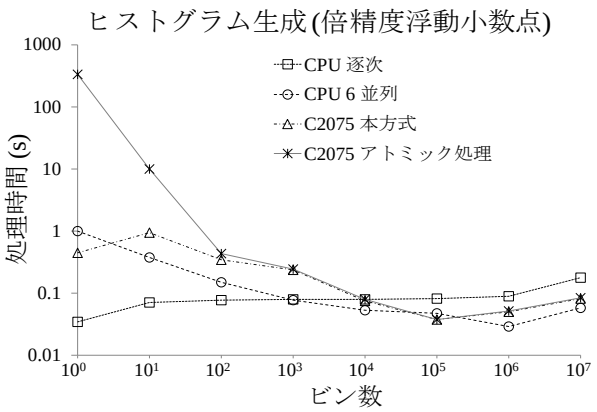

図 6　倍精度浮動小数点のヒストグラム生成にかかる時間．

トミック加算を 64 ビット整数のアトミック比較交換命令で実現しており，リトライが多発するためである．

CPU の 6 並列版でもアトミック処理を評価した．この場合でも，特に衝突が多いところで実行速度が遅くなっていることが判る．衝突が無くなるにつれて高速化し，100—1000 ビンで CPU での逐次実行よりも早くなっている．

ビン数が 1000 以下では，CPU で逐次処理を行った場合が最速だった．計算量が少なく，アトミック比較交換命令でリトライが多発する場合には，本方式を含めて，並列化の恩恵は少ないことが分かる．

また，GTX680 でも C2075 と同様の傾向がみられた．ビン数が 1 の場合に，アトミック加算処理に比べ 176 倍高速化された．

図 7 は，ビンの変数が整数の場合の結果である．C2075 ではビン数が 10 以下で，本方式の競合抑制による優位性が見られた．尚，GPU 利用時の処理時間には，CPU—GPU 間のデータ転送と GPU の起動にかかる合計の時間 15 ms 程度が含まれており，ヒストグラム作成部の実行時間は最短で数 ms であった．

ハードウェアのアトミック加算命令が利用でき，リトライが発生しない環境では，CPU で逐次処理した場合，および，並列処理した場合のいずれと比べても，ビン数によらず本方式で高速化できている．

GTX680 では，アトミック処理命令を直接使った場合と本方式の間に，有意な速度差は見られなかった．Kepler アーキテクチャでは，単体のアトミック処理命令が，本方式と同程度に高速な場合があることが判った．また，シャッフル機能を用いた場合と，共有メモリを用いた場合の速度についても，有意な差は認められなかった．

3.2 姫野ベンチマークによる評価

姫野ベンチマークは，ポアッソン方程式解法をヤコビの反復法で解く際の，主要なループの処理速度を求めるものである．計算部分では，複数の 4 次元配列の要素の積と和を求め，解となる 3 次元行列を更新していく．収束の度合いを見るため，行列要素の値と期待値の差の自乗和を誤差として求める．変数は単精度浮動小数点で，計算も単精度浮動小数点でおこなった．

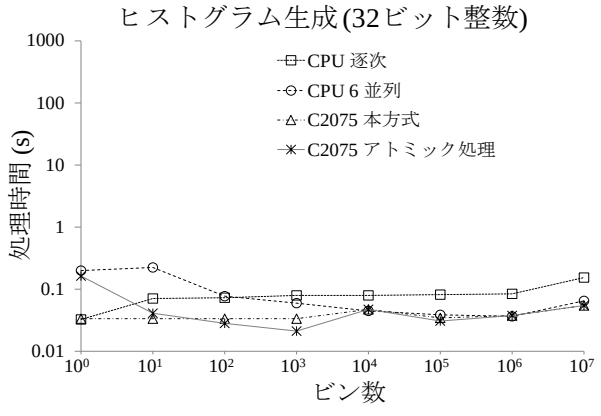

図 7　整数のヒストグラム生成にかかる時間．

評価は，C 言語で書かれたソースプログラム（himenoBMTxpa.c）を CUDA に変更して行った．

GPU 向けの修正では，カーネルループである関数 jacobi() 中の最外ループを CPU で制御し，中の 2 つのループを GPU で順に実行するようにした．この 2 つのループは，どちらも 3 重ループであるが，3 重ループを全て GPU のスレッドで展開した．ブロック当りのスレッド数は *x* 方向で 128 とし，*y* 方向，*z* 方向は 1 とした．

誤差を累積するための変数 gosa へのリダクション部は，以下の 4 つのパターンで計算した．

(1) **本方式を採用．**
(2) **CUDA のアトミック処理をそのまま利用．**
(3) **ループ内で求められる個々の要素をバッファ経由で CPU に渡し，CPU 1 コアで逐次実行で集計．**
(4) **ループ内で求められる個々の要素をバッファ経由で CPU に渡し，OpenMP を用いて CPU 6 コアで並列化して集計．**

CPU 向け修正は，以下の 2 通りを行った．

(5) **元のソースプログラムそのままで，CPU 1 コアで逐次実行．**
(6) **OpenMP を用いて CPU 6 コアで並列化．**

リダクション先はスカラー変数であるため，パターン(4)と(6)では OpenMP の reduction 指示節を用いて処理している．この場合，アトミック処理のような競合は発生しない．また，並列化は，外側から 2 番目，3 番目のループに対して，OpenMP の collapse(2) 指示節を用いて行った．

GPU 版，CPU 版いずれも，書き換え以外のプログラムの最適化は行っていない．

問題サイズは XS と L の 2 通りを採用した．

評価結果を図 8 に示す．この図は，姫野ベンチマークにより得られた浮動小数点演算性能（FLOPS）が，パターン(5)の性能の何倍になるかを問題サイズ毎に見たものである．縦軸が性能比で，横軸はハードウェアとリダクションの処理方法を表している．

図中，パターン(6)は，パターン(5)の 5.5 倍以上の FLOPS を出しており，高い並列性を得られることがわかる．問題サイズ L では，処理時間の約 96%を行列計算部で費やしており，リダクション処理では 3.6%程となった．

GTX680 を利用した結果を図 8 の中央に示した．いずれもパターン(6)の結果より高速化された．本論文の方法を採

用したパターン(1)ではパターン(2)よりも高い FLOPS が得られており，問題サイズ L においては 2 倍以上高速化された．また，演算部を比較すると，パターン(6)に比べて，GPU では 3.8 倍高速化された．

リダクション処理のみを CPU で行った場合も高速化された．パターン(4)では，問題サイズ L の全処理時間の約 13%がリダクション処理であった．

C2075 を利用した結果は，図 8 の右側に示してある．本方式を使用したパターン(1)では，問題サイズ L でパターン(5)の 12 倍まで高速化されたが，パターン(2)は 1.4 倍にとどまった．

また，二分木のリダクションを行った場合の実行性能は，GTX680 においてパターン(2)と同程度から 1.5 倍に高速化された．C2075 においては，パターン(2)の 1.5 倍から 2 倍に高速化された．

3.3 量子化学計算プログラムによる評価

実際の数値計算プログラム例として，量子化学計算プログラムを用いて評価を行った．このプログラムは，OpenFMO プロジェクト 15)の一部として本田，稲富，真木らにより作成されたものである．

このプログラムの実行時間の殆どは，類似した 6 つの関数中のカーネルループで費やされている．カーネルループは，外側に 2 重ループがあり，ループ中で，倍精度浮動小数点の演算処理のループと，倍精度浮動小数点の演算結果を配列変数に足しこむループを順番に実行するように構成されている．

評価は，3.2 節の(1)から(6)のパターンで行った．ただし，CPU でのリダクションにもアトミック処理を採用した．

GPU 向けに修正する際は，カーネルループの 2 重ループを全て GPU のスレッドとして展開した．GPU のブロック当りのスレッド数は 128 とし，x 方向と y 方向で比率を調整した．GPU の倍精度浮動小数点のアトミック加算には，文献 1)の付録 B.11 にある例を利用した．

パターン(3)と(4)に関しては，カーネルループの 2 重ループを分割し，前半の演算処理ループのみを GPU で行い，結果を足し込むループを CPU で処理した．

その他は，特にプログラムの最適化を行っていない．

評価には計算サイズの異なる 3 つのデータセットを用いた．

評価結果を図 9 に示す．この図は，量子化学計算プログラムを実行した際の実行速度が，データ毎にパターン(5)の何倍になるかを見たものである．縦軸が速度比で，横軸はハードウェアとリダクションの処理方法を表している．

パターン(1)の結果は，パターン(2)に比べ，5—10 倍の高速化が見られた．また，いずれのデータセットにおいても，パターン(6)よりも高速であり，データ 1 ではパターン(1)が最速だった．ただし，パターン(6)に対する高速化率は，1.6—2.2 倍と小さい．

データ 3 に関して，パターン(1)の処理時間の約 46%が演算処理で，残り 54%がリダクション処理であった．演算処理時間のみを比較すると，データ 3 において GPU は CPU の 6 並列の 4 倍に高速化された．

パターン(6)はパターン(5)の 5 倍程度の速度を出しており，高い並列度が得られている．CPU では，処理時間の 98%が演算処理に，残り 2%弱がリダクション処理に割かれていた．

一方，パターン(4)は並列化によりパターン(3)よりも遅くなっている．パターン(3)と(4)の違いであるリダクション処理をデータ 3 で比較すると，パターン(4)はパターン(3)の 1.6 倍の時間がかかっている．また，パターン(3)では，GPU

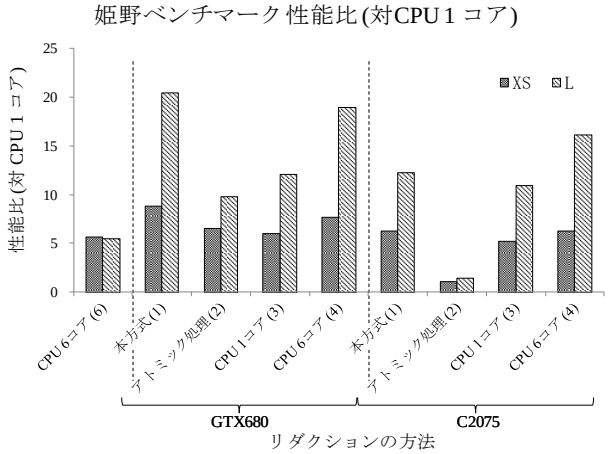

図 8 姫野ベンチマークの結果．縦軸は，問題サイズ毎に(5)の性能を 1 とした場合の値である．

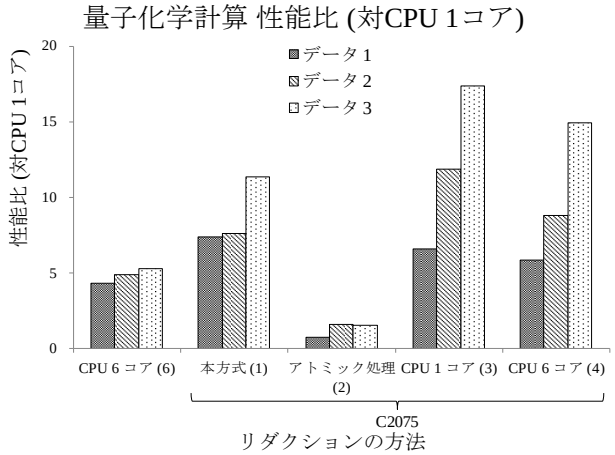

図 9 量子科学計算プログラムによる評価結果．縦軸は，データ毎に(5)の実行速度を 1 とした場合の値である．

による演算処理時間は全体の 71%, CPU によるリダクション処理は，28%であった．

パターン(6)と(4)の違いとして，前者では CPU で演算処理とアトミック処理を交互に繰り返してループを回るのに対し，後者ではアトミック処理のみをループで繰り返すことが挙げられる．このため，前者では演算処理時間がアトミック処理のタイミングを分散させる緩衝材となり，競合が起こりにくくなっている．一方，後者では CPU の全スレッドで一斉にアトミック処理を開始し，その後もアトミック処理を集中して続けるため，競合を起こしやすい．

パターン(1)と(2)でも，上記パターン(4)と同じ処理順序となるため，演算処理時間によりある程度競合が緩和されると期待される．一方，C2075 で同時にアクティブになるワープ数は最大で 224，スレッド数では 7168 であった．これは，CPU の並列度 6 に比べて非常に大きい．このため，実際には競合が残ったと考えられる．

パターン(3), (4)のようにリダクション部周辺のみを CPU で行う方法は，GPU 単体で処理するよりも高速になる場合がある．しかし，あるプログラムを CPU と GPU で処理する場合の速度は，データやパラメータ，ハードウェアに大きく依存しており，一概にどちらが速いとは言い難い場合がある．

また，パターン(3)，(4)では，プログラムをループの途中で分割する必要があり，場合によっては中間バッファを必要とする．このため，コンパイラ等でいつでも使える手段というわけではない．一方，本論文の方式ではこのような変更が不要であり，広範囲に適用できる．

GTX680 を使用した場合，アトミック処理による加算でパターン(5)の 4—9 倍，本方式を採用すると 7—13 倍の高速化を実現できた．

4. まとめ

GPU でアトミック処理を用いる場合，競合により著しく処理速度が低下する場合がある．本論文では，リダクション演算に対して競合を削減する方法を提案し，評価結果を示した．この方式では，GPU のワープ内で予め競合する書き込みをまとめ，全体の競合数を削減する．

本方式の Kepler アーキテクチャへの実装は CUDA の built-in 関数と同じ引数を持つように行い, built-in 関数を使ったプログラムへ容易に適用できるようにした．ワープ内でのデータ共有に共有メモリ等を使用せず，逐次実行型のプログラムを並列化する際にも，少ない変更量で適用が可能である．

Fermi アーキテクチャへの実装では，ワープ内の情報共有を共有メモリ経由で行った. Built-in 関数と同じ引数を持つようにすることも可能だが，省メモリ化を優先し，共有メモリ領域を関数の引数として渡す形にした．

いずれの場合も，二分木形式のリダクション処理に比べ，プログラムの変更量を抑え，利用できるケースを増やし，コンパイラ等でも利用しやすい形となっている．

実効性能の評価は，姫野ベンチマークと量子化学計算プログラムで行った．アトミック関数をそのまま使う場合に比べて数倍以上の高速化が見られ，極端な例では数百倍の高速化が実現できた．

本方式の基本性能は，ヒストグラムを生成するプログラムで評価した．競合が多い場合には本方式による高速化が見られたが，明らかに競合が少ない場合には高速化が認められなかった．

CPU との比較のため，本方式を採用した場合と，CPU で逐次実行した場合，6 並列処理した場合との比較も行った．CPU の並列処理でもアトミック処理の競合による実行速度の低下がみられる場合があったが，アトミック処理のタイミングを分散させ，緩衝させるのに十分な量の処理がプログラム中にあれば，競合を回避できることが分かった．CPU の数十から数千倍の並列度を持つ GPU では，競合の緩衝材となる処理の量は膨大になるため，本方式のように積極的に競合を削減する方式が必要である．

CPU，GPU ともに，プログラムの実行速度は，プログラムだけでなく，データとパラメータ，ハードウェアに依存する．これらの要因の変動により CPU よりも GPU の処理時間が長くなると，GPU を利用する価値が半減する．このような場合でも，処理時間の遅延を少なく抑える方法の研究を今後行っていく．

謝辞 姫野ベンチマークを公開してくださった姫野龍太郎博士と，量子化学計算の評価と結果の公表を快く承諾してくださった本田宏明博士に, 謹んで感謝の意を表する．

参考文献

1) CUDA_C_Programming_Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
2) Andrew Stromme, Ryan Carlson and Tia Newhall: Chestnut: a GPU programming language for non-experts, PMAM '12 Proceedings, pp.156—167 (2012).
3) Yi Yang, Ping Xiang, Jingfei Kong and Huiyang Zhou: A GPGPU compiler for memory optimization and parallelism management, PLDI '10 Proceedings, pp.86—97 (2010).
4) Mehrzad Samadi, Amir Hormati, Janghaeng Lee and Janghaeng Lee: Paragon: collaborative speculative loop execution on GPU and CPU, GPGPU-5 Proceedings, pp.64—73 (2012).

5) Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim and Richard Vuduc: A performance analysis framework for identifying potential benefits in GPGPU applications, PPoPP '12 Proceedings, pp.11—22 (2012).
6) Yooseong Kim and Aviral Shrivastava: CuMAPz: a tool to analyze memory access patterns in CUDA, DAC '11 Proceedings, pp.128—133 (2011).
7) Usman Dastgeer, Johan Enmyren and Christoph W. Kessler: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems, IWMSE '11 Proceedings, pp.25—32 (2011).
8) Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye and Wen-mei W. Hwu: Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors, PPoPP '12 Proceedings, pp.23—34 (2012).
9) Sunpyo Hong and Hyesoon Kim: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, ISCA '09 Proceedings, pp.152—163 (2009).
10) Optimizing Parallel Reduction in CUDA, http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf
11) NVIDIA Fermi Compute Architecture Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
12) NVIDIA-Kepler-GK110-Architecture-Whitepaper, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
13) 姫野龍太郎: 姫野ベンチマーク, http://accc.riken.jp/secure/4502/himenobmtxpa.lzh
14) The OpenMP® API specification for parallel programming, http://openmp.org/wp/
15) OpenFMO プロジェクト, http://www.openfmo.org/OpenFMO/