Technical Reports on Information and
Computer Science from Kochi
Vol. 4 (2012), No. 7

GPGPU を用いた高速統計データ計算方法

A Fast Statistical Data Calculation Method based on GPGPU

西村 直樹 1） 豊永 昌彦 2）

Naoki Nishimura 1） Masahiko Toyonaga 2）

1）高知大学理学部 2)高知大学 情報講座

Information Science Division, Faculty of Science, Kochi University

あらまし

本論文で，並列計算環境 GPGPU を利用した最大最小検索アルゴリズムと統計データ計算アルゴリズムを提案し，従来アルゴリズムが時間複雑度 O(N)に対して，提案アルゴリズムが O(log(N)) であることを実装したプログラムを用いた実験より示す．実験では，100～100 万レコードのデータについて，単一スレッドと並列スレッドで比較を行った．480 コアの並列化した最大最小検索では，単独処理に比べて最大で 59.9 倍の速度改善されることが判明した．また，同 480 コアで求めた統計データでは，単独処理に比べて最大で 31.5 倍の高速化が判明した．

キーワード:最大値検索，最小値検索，平均値，GPGPU

1. はじめに

インターネットや電子機器の普及により，情報化が進み，データが膨大な量になってきている．米国 IDC によると，2009 年に作成されたデータ量は，前年比 62%増の 0.8 ゼタバイト(8,000 億 GB)だったと発表しており，2020 年には，世界で作成されるデータ量は 35 ゼタバイトに達すると予測している[1].

データ量の増加に伴いデータベースや，情報検索等の処理の高速化が課題となってくる．これらのデータベースや情報検索等の高速化の基本機能として，最大最小検索や平均値・分散・標準偏差等の統計データ計算が挙げられる．

一方，近年並列計算環境として GPGPU が普及してきた．GPGPU とは，高度な並列計算を得意とする GPU を，画像処理という本来の目的ではなく，より一般的な並列計算に利用するというものである．従来の単一処理(単一スレッド)に比べて，高い処理能力を持ち，安価で入手することが出来る．

GPGPU を利用した並列処理の先行技術としては，文献[2]では GPGPU を用いて実装した基数ソートが，最新のマルチコア CPU ソートルーチンより 23%速いという結果が得られている．[3]では VLSI 回路最適化の高速化のために，ゲートサイジングおよびしきい値電圧の割り当てに GPGPU を適用している．その結果，従来の単一スレッドの計算時間と比較して，56 倍の高速化を提供している．また，[4]では GPGPU を用いて暗号アルゴリズム DES に対するパスワードクラックを実装した結果について報告しており，従来の計算時間と比較して約 9 倍の高速化が確認されている．

本論文では，並列計算環境 GPGPU を利用した最大最小検索アルゴリズムと統計データ計算アルゴリズムを提案し，従来アルゴリズムが時間複雑度 O(N)に対して，提案アルゴリズムが O(log(N)) であることを実装と実験結果から示す．実験では，100～100 万レコードのデータについて，単一スレッドと並列スレッドで比較を行った．480 コアの並列化した最大最小検索では，単独処理に比べて最大で 59.9 倍の速度改善されることが判明した．また，同 480 コアで統計データ計算では，単独処理に比べて最大で 31.5 倍の速度改善が得られることが判明した．

以下，本論文の構成を説明する．

次の第 2 章において，従来の最大最小検索と統計データ計算方法，GPGPU 環境 CUDA について述べる．第 3 章では，GPGPU を用いて並列化した最大最小検索と実験結果について述べる．第 4 章では，GPGPU を用いて並列化した統計データ計算方法と実験結果について述べる．第 5 章でまとめを行う．

2. 統計データ

大規模データの統計値として，本論文では，データ列の最大値・最小値，および平均値と分散，標準偏差について扱う．それぞれの計算法について以下説明する．

(1) 最大値・最小値

最大最小検索とは，配列上に登録されたデータから最大値，あるいは最小値をもつデータレコードを検索する処理をいう．以下，従来のアルゴリズムについて説明する．

```
[最大最小検索アルゴリズム]
  Step1)VmaxとVminの初期値としてDATA[1]を代入
  Step2)残りのDATA[j](j = 2,3,...,N)について以下を繰り返す
    Step2-1) DATA[j] > VmaxならばVmax = DATA[j]とする
    Step2-2) DATA[j] < VminならばVmin = DATA[j]とする
  Step3)Step2の処理後に得られたVmaxとVminを出力
  Step4)終了
```

図 2.1 最大最小検索のアルゴリズム

以後ではデータレコードが配列 DATA[i](i=1,2,3,…,N)

で与えられるものとして，図 2.1 に示す最大最小検索アルゴリズムを説明する．最大値・最小値の検索は，まず Step1)において，最大値 *Vmax* と最小値 *Vmin* の初期値を DATA[1]とする工程から始まる．次に Step2)において，先ほど定義した最大値 *Vmax*，最小値 *Vmin* と j 番目のレコードのデータ DATA[j] (j=2,3,…,N)について，DATA[j]>*Vmax* を満たすなら最大値 *Vmax* として代入する(*Vmax*= DATA[j])，これとは違って，DATA[j]<*Vmin* を満たすなら最小値 *Vmin* に代入する(*Vmin*=DATA[j])．次に Step3)において，以上の処理で得られた *Vmax* と *Vmin* を最大値・最小値として出力する工程で実装されている[5]．計算時間複雑度は，データ数 N に対してそれぞれにたかだか 1 度程度の処理となることから，O(N)である．

図 2.2 の 8 レコード(N=8)のランダムデータ配列に最大最小検索を適用した例を図 2.1 のアルゴリズムに沿って説明する．

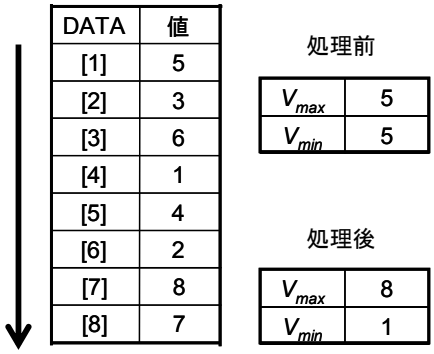


図 2.2 従来の最大最小検索

まず，最初に Step1)の *Vmax* と *Vmin* の初期値として配列の先頭 DATA[1]の値 5 とする．次に Step2)として *Vmax* の値，および *Vmin* の値と配列 DATA[2]の値 3 とを比較する．この時，配列 DATA[2]の値が *Vmax* より大きければ配列 DATA[2]の値を *Vmax* に格納する．また配列 DATA[2]の値が *Vmin* より小さければ配列 DATA[2]の値を *Vmin* に格納する．今の場合配列 DATA[2]の値 3 は *Vmin* の値より小さいので，*Vmin* に 3 を代入することになる．この作業を全データ DATA[1]～DATA[8]まで行う．最後に Step3)として *Vmax* と *Vmin* に得られた *Vmax* = 8，*Vmin* = 1 を出力する．

(2) 平均，分散，標準偏差の計算式

平均値とは，全データ DATA[i] (i=1,2,3,…,N)の総和を総数 N で割った値 *Va* として式(2.2.1)で定義される[5].

$$V_a = \frac{\sum_{i=1}^{N} DATA[i]}{N} \qquad (2.2.1)$$

また，分散とは，全データ DATA[i] (i=1,2,3,…,N)から平均値 *Va* の 2 乗を引いた値の総和を全数 N で割った値 *Vb* として次の式(2.2.2)で定義される[5].

$$V_b = \frac{\sum_{i=1}^{N} (DATA[i] - Va)^2}{N} \qquad (2.2.2)$$

さらに，標準偏差は，全データ DATA[i] (i=1, 2, 3, …, N)から平均値 *Va* の 2 乗を引いた値の総和を全数 N で割った値の平方根 *Vh* として次の式(2.2.3)で定義される[5].

$$V_h = \sqrt{\frac{\sum_{i=1}^{N} (DATA[i] - Va)^2}{N}} \qquad (2.2.3)$$

(3) 平均値計算アルゴリズム

図 2.3 に平均値計算のアルゴリズムを示す.

```
[平均値計算アルゴリズム]
  Step1)Vaの初期値として0を代入
  Step2) DATA[i](i = 1,2,3,...,N)について以下を繰り返す
    Step2-1) Va += DATA[i]とする
  Step3)Step2の処理後に得られたVaをNで割る
  Step4)Step3の処理後に得られたVaを出力
  Step5)終了
```

図 2.3 平均値計算のアルゴリズム

データが配列 DATA[i](i=1,2,3,…,N)として与えられたとき，まず Step1)において，平均値を初期値 0 として，*Va*＝0 を設定する．次に Step2) は，平均値 *Va* に i 番目のデータレコードの値 DATA[i] (i=1,2,3,…,N)をすべて加算していく工程である．最後に Step3)において Va を全数 N で割って再定義する．以上により得られた平均値 *Va* を Step4)において，平均値として出力する [5]．計算複雑度について，これらの処理は，データ数 N に対して 1 回程度の処理なので O(N)となる．

平均値計算アルゴリズムについて，図 2.4 に示した 8 レコード(N=8)のランダムデータを持つ配列に適用した処理の例を示す.

まず, 最初に Step1)として *Va* の初期値を 0 とする．次に Step2)として配列 DATA[1]から順番に *Va* に加算していく．この場合，配列 DATA[1]の値は 5 なので *Va* に 5 を加算して *Va* の値は 5 になる．次に配列 DATA[2]の値を加算すると *Va* の値は 8 になる．この作業を全データについて行う．Step3)として配列 DATA[1]から配列 DATA[8]まで加算された *Va* を全データ数 8 で割る．Step4)として *Va* に得られた *Va* = 4.5 を出力する.

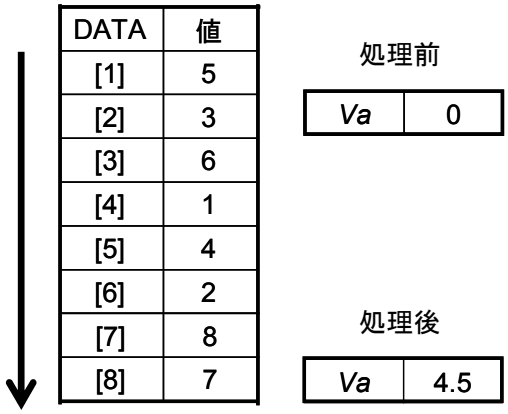


図 2.4　　従来の平均値の計算

(4) 分散と標準偏差

```
[分散・標準偏差計算アルゴリズム]
  Step1)VbとVhの初期値として0を代入
  Step2) DATA[i](i = 1,2,3,...,N)について以下を繰り返す
    Step2-1) Vb += DATA[i] * DATA[i] / Nとする
  Step3)Step2の処理後に得られたVbから平均値の2乗を引く
  Step4)Step3の処理後に得られたVbの平方根がVhとなる
  Step5)Step4の処理後に得られたVbとVhを出力
  Step6)終了
```
図 2.5 分散・標準偏差計算のアルゴリズム

図 2.5 に分散・標準偏差計算のアルゴリズムを示す．注意すべきは *Vb* を式 2.2.2 のままでなく，次のような変形式(2.2.2b)を行い，平均値 *Va* から独立した計算で行う．

$$V_b = \frac{\sum_{i=1}^{N}(DATA[i]-Va)^2}{N} = \frac{\sum_{i=1}^{N}(DATA[i]^2 - 2DATA[i]\cdot Va + Va^2)}{N}$$
$$= \frac{\sum_{i=1}^{N}DATA[i]^2 - 2Va\sum_{i=1}^{N}DATA[i] + Va^2\sum_{i=1}^{N}}{N} = \frac{\sum_{i=1}^{N}DATA[i]^2 - NVa^2}{N}$$
$$= \frac{\sum_{i=1}^{N}DATA[i]^2}{N} - Va^2 \qquad (2.2.2b)$$

すなわち，*Vb* を求めるために，配列 DATA[i] (i=1, 2, 3, …,N)の 2 乗を N で割った値の和を求めて，平均の 2 乗を引いて計算する．

分散・標準偏差計算アルゴリズムによれば，まず Step1)において，分散を初期値 0 として，*Vb*＝0，*Vh*=0 を設定する．次に Step2) は，平均値 *Vb* に i 番目のデータレコードの値 DATA[i] (i=1,2,3,…,N)を 2 乗して全数 N で割った値をすべて加算していく工程である．Step3)は Step2 で得られた *Vb* から平均値の 2 乗を引く工程である．最後に Step4)において Step3 で得られた分散の値 *Vb* の平方根が標準偏差である．以上により得られた平均値 *Vb* と標準偏差 *Vh* を Step5)において，分散と標準偏差として出力する [5]．計算複雑度について，これらの処理は，データ数 N に対してたかだか 1 度の処理であることから， O(N)であることがわかる．

図 2.6 に示した 8 レコードのランダムデータに適用した処理の例を示す．

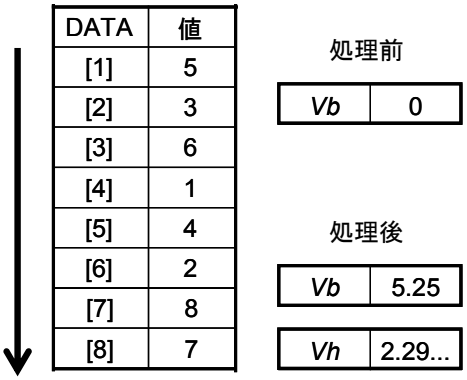


図 2.6 従来の分散と標準偏差の計算

まず，最初に Step1)として *Vb* と *Vh* の初期値を 0 とする．次に Step2)として配列 DATA[1]から順番に，*Vb* に配列の値を 2 乗して全データ数で割った値を加算していく．この場合，配列 DATA[1]の値は 3.125 になるので *Va* に 3.125 を加算して *Va* の値は 3.125 になる．次に配列 DATA[2]の値を加算すると *Va* の値は 4.25 になる．この作業を全データについて行う．Step3)として配列 DATA[8]まで加算し終えた値 25.5 から平均値の 2 乗の値 20.25 を引くと分散 *Vb* の値 5.25 が求まる．Step4)として Step3 で得られた分散 *Vb* の値 5.25 の平方根を計算すると標準偏差 *Vh* の値 2.29…が求まる．最後に Step5)として分散 *Vb* と標準偏差 *Vh* に得られた *Vb* = 5.25 と *Vh* = 2.29…を出力する．

3. 並列化した統計データ計算手法

(1) 並列化した最大最小検索手法

前述したように，グラフィック処理の並列機能を一般処理に利用する環境として GPGPU では，多数のスレッドを利用することが出来る．今，データ数 N の場合，スレッドを N 個利用出来るとして[並列化した最大最小アルゴリズム]を，図 3.1 に示す．

並列化した最大最小検索のアルゴリズムは，dataCopy と calMinMax の 2 つの並列処理で構成されている．CPU の処理は＜Host＞，GPU の処理は＜Device＞として示した．dataCopy は，data[i]を data2[i]に代入する処理，calMinMax は data2[]から並べ替えた結果が大なら data[i]に書き出し，小なら data[N/2+i]に書き出す処理である．このカーネルを log(N)回繰り返す．

なお，最大最小は data1[1], data1[N]として得られる．これは最大のデータが次のようにして，データレコードを変遷していくからである．まず，並ぶ 2 つのデータ data1[2*i], data1[2*i-1] の大きい値を配列の前半分 1~N/2 に書き込む 1 回の処理で，最大値は N/2 より前に整理される．これを logN 回繰り返せば，先頭レコードに

最大値が書き込まれることになる. 反対に並ぶ 2 つのデータ data1[2*i], data1[2*i-1]の小さい値を配列の後半 N/2+1~N に書き込む 1 回の処理で, 最小値は N/2 より後に整理される. これを logN 回繰り返せば, 最終レコードに最小値が書き込まれることになる.

```
<Host(CPU)>
rep=log(N):
for(int i=1; i<= rep ; i++){
        dataCopy<<<blocks, threads>>>(data, data2);
        calMinMax<<<blocks, threads>>>(data, data2);
}

<Device(GPU)>
__global__ void dataCopy(int *data, int *data2){
        i=get-thread-id();
        data2[i] = data[i];
}
__global__ void calMinMax(int *data, int *data2){
        i=get-thread-id();  id = i * 2;
        v1=data2[id]; v2=data2[id-1];
        if(v1>v2){
                data[i] = v1; data[N/2+i] = v2;
        }else{
                data[i] = v2; data[N/2+i] = v1;
        }
}
```

図 3.1 並列化した最大最小検索のアルゴリズム

ホストとデバイスの構成

GPGPU の実装には, CPU で処理するホストプログラムと, GPU 側で処理するデバイスプログラム(カーネル)を作成しなければならない. ホスト側は, 外部入出力の設定やデータの準備を行い, デバイス側は, これを受け取り並列処理し, その後, デバイス側から処理結果をホスト側に転送して終了する. 最大最小検索における各役割は, 図 3.2 の通りである.

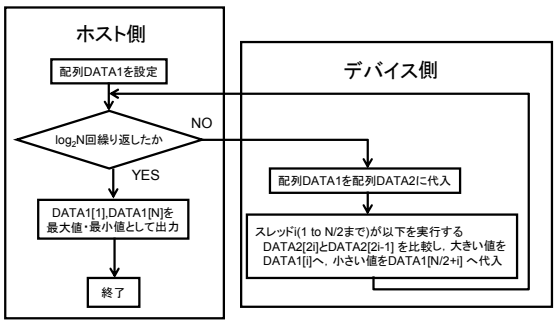


図 3.2 ホストとデバイスの構成

まずホスト側で最大最小検索対象となるデータを配列 DATA1 に設定する. 次にデバイス側の処理を logN 回に達するまで繰り返す. デバイス側の処理は, まず配列 DATA1 を配列 DATA2 に代入し, さらに各スレッド i(i=1,2,3,…,N/2) は配列 DATA2[2*i] と配列 DATA2[2*i-1]の値を比較し, 大きい値を DATA1[i]へ, 小さい値を DATA1[N/2+i]へ代入する. 以上が logN 回に達すれば, ホスト側で DATA1[1],DATA1[N]を最大値・最小値として出力し, 処理を終了する.

並列化した最大最小検索の例

以上の並列化した最大最小アルゴリズムについて, 図 3.3 に示した 8 レコード(N=8)のランダムデータに適用した処理の例を示す.

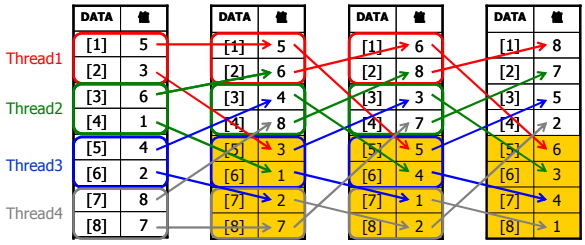


図 3.3 並列化した最大最小検索(1)

ここで並列化した最大最小検索法は 2 レコードずつの比較を N/2 個のスレッドを用いて並列で行う.

配列には 1 から 8 までのランダムな値が格納されている. この配列を並列に 2 つずつ比較するのだが, スレッドの割付は図 3.3 のようになる. スレッドとは1つの処理単位のことである. それぞれのスレッドで比較を行い, 大きければ配列番号がスレッド番号と同じになる位置に格納し, 小さければ配列番号が配列の半分にスレッド番号を足した位置に格納する. この処理を全スレッド同時に並列で行う. この処理をデータ数 N としたとき, log(N)回繰り返す.

一度目の処理を行う場合に Thread1 の赤枠で囲まれた部分に注目してみると, 配列 DATA[1]の値 5 と配列 DATA[2]の値 3 を比較すると, 値 5 の方が大きいので, 値 5 は配列番号 1 に格納される. 一方値 3 は配列の半分にスレッド番号を足した位置, つまり配列番号 5 の位置に格納される. 図ではわかりやすくするために配列の下半分を黄色く色付けしている. この処理をスレッド 2 から 4 に関しても同時に並列で行う.

以上のことを log2 の 8 回つまり 3 回繰り返す.

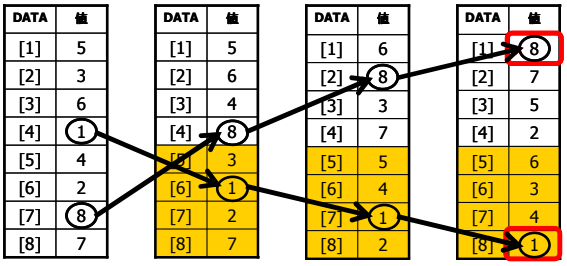


図 3.4 並列化した最大最小検索(2)

この時, 図 3.4 では最大値となる 8 と最小値となる 1 を黒丸で囲み, シフトの様子を太線で示した. 最大値は配列 DATA[1], 最小値は配列 DATA[8]に格納される.

この処理は繰り返し回数が log(N)回なので, オーダーは O(log(N))となる.

(2) 並列化した統計データ計算

データ数 N の場合，スレッドを N 個利用できるとして，[並列化した統計データ計算アルゴリズム]を図 3.5 に示す．

```
<Host(CPU)>
dataSquare<<<blocks, threads>>>(data3, data4,realdata);
rep=log(N):
for(int i=1; i<= rep ; i++){
                dataCopy1<<<blocks, threads>>>(data, data2);
                calAve<<<blocks, threads>>>(data, data2);
                dataCopy2<<<blocks, threads>>>(data3, data4);
                calDis<<<blocks, threads>>>(data3, data4);
}
Va = (float)index[1] / (float)N;
Vb = index[3] - (Va * Va);
Vh = sqrt(Vb);

<Device(GPU)>
__global__ void dataSquare(float *data3, float *data4,int realmax){
                i=get-thread-id();
                data3[i] = data3[i] * data3[i] / (float)realmax[0];
}
__global__ void dataCopy1(int *data, int *data2){
                i=get-thread-id();
                data2[i] = data[i];
}
__global__ void calAve(int *data, int *data2){
                i=get-thread-id();  id = i * 2;
                data[i] = data2[id] + data2[id-1];
                data[N/2+i] = 0;
}
__global__ void dataCopy2(int *data3, int *data4){
                i=get-thread-id();
                data4[i] = data3[i];
}
__global__ void calDis(int *data3, int *data4){
                i=get-thread-id();  id = i * 2;
                data3[i] = data4[id] + data4[id-1];
                data3[N/2+i] = 0;
}
```

図 3.5 並列化した統計データ計算のアルゴリズム

並列化した統計データ計算のアルゴリズムは，dataSquare ，dataCopy1，calAve ，dataCopy2，calDis の 5 つの並列処理で構成されている．CPU の処理は＜Host＞，GPU の処理は＜Device＞として示した．dataSquare は，data3[i]を 2 乗してデータ数 N で割った値を data3[i]に代入する処理，dataCopy1 は data1[i]を data2[i]に代入する処理，calAve は data1[i]に data2[id]と data2[id-1]を足した値を代入し，data[N/2+i]に 0 を代入する処理，dataCopy2 は data3[i]を data4[i]に代入する処理，calDis は data3[i]に data4[id]と data4[id-1]を足した値を代入し，data3[N/2+i]に 0 を代入する処理である．このアルゴリズムでは dataCopy1，calAve，dataCopy2，calDis の 4 つのカーネルを log(N)回繰り返す．

なお，平均値 *Va* は data1[1]をデータ数 N で割ることで得られる．これは data1[i]が次のようにして，データレコードを変遷していくからである．まず，並ぶ 2 つのデータ data1[2*i]，data1[2*i-1]の加算した値を配列の前半分 1~N/2 に書き込む 1 回の処理で，加算した値は N/2 より前に整理される．これを logN 回繰り返せば，先頭レコードに data1[i]の和が書き込まれることになる．

分散 *Vb* は data3[1]を平均値 *Va* の 2 乗の値を引くことで得られる．これは data3[i]が次のようにして，データレコードを変遷していくからである．まず，並ぶ 2 つのデータ data3[2*i]，data3[2*i-1]の加算した値を配列の前半分 1~N/2 に書き込む 1 回の処理で，加算した値は N/2 より前に整理される．これを logN 回繰り返せば，先頭レコードに data1[i]の和が書き込まれることになる．

標準偏差 *Vh* は分散 *Vb* の平方根を計算すると得られる．

ホストとデバイスの構成

統計データ計算におけるホストとデバイスの構成は，図 3.6 の通りである．

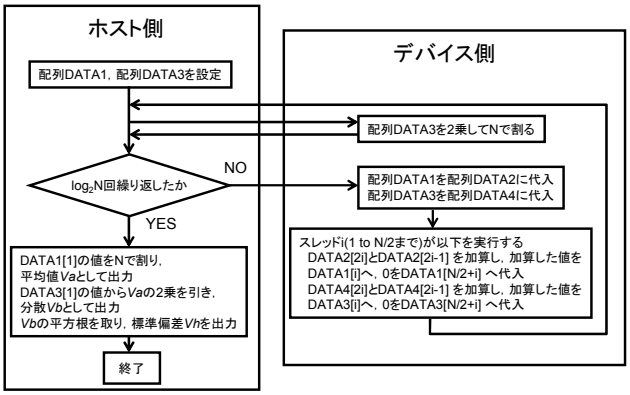


図 3.6 ホストとデバイスの構成

まずホスト側で統計データ計算対象となるデータを配列 DATA1，配列 DATA3 に設定する．

次にデバイス側の処理として，配列 DATA3 を 2 乗して N で割る．それからデバイス側の処理を logN 回に達するまで繰り返す．デバイス側の処理は，まず配列 DATA1 を配列 DATA2 に代入し，配列 DATA3 を配列 DATA4 に代入する．さらに各スレッド i(i=1,2,3,…,N/2) は配列 DATA2[2*i]と配列 DATA2[2*i-1]の値を加算し，加算した値を DATA1[i]へ，0 を DATA1[N/2+i]へ代入する処理，DATA4[2*i]と配列 DATA3[2*i-1]の値を加算し，加算した値を DATA3[i]へ，0 を DATA3[N/2+i]へ代入する処理を行う．以上が logN 回に達すれば，ホスト側で平均値は DATA1[1]の値を N で割り，値 *Va* として出力し，分散は DATA3[1]の値から *Va* の 2 乗を引き，値 *Vb* として出力し，標準偏差は *Vb* の平方根を計算し，値 *Vh* として出力し，処理を終了する．

4. 実験

(1) 最大最小検索の実験

100 から 100 万レコードまで昇順に並べたデータを実験データとした．表の処理時間は Compute Visual Profiler の出力結果を用いた．単位は μs である．Ts は単

一スレッドでの最大最小検索法による処理時間，Tm は並列スレッドでの最大最小検索の処理時間である．Ts/Tm は並列化の高速された倍数を示す．プログラム開発環境には Core 数が 16cores，48cores，480cores の 3 種類を用いた．プログラム開発環境を表 4.1 に示す．

表 4.1 プログラム開発環境

CPU	GeForce 9400 GT	GeForce GT 220	GeForce GTX 480
プロセッサ	Intel Core i7 CPU 950 @ 3.07GHz 3.07GHz	Intel Core i5 CPU 760 @ 2.80GHz 2.80GHz	Intel Core2 Duo CPU E8400 @ 3.00GHz 2.99GHz
実装メモリ	2.99GB	4GB	2.00GB
CUDAプロセッサーコア	16	48	480
グラフィッククロック(MHz)	550	625	700
プロセッサークロック(MHz)	1400	1360	1401
メモリクロック(MHz)	400	790	1848
標準メモリ設定	512MB	1 GB DDR3	1536MB GDDR5
メモリインターフェース幅	128-bit	128-bit	384-bit
メモリバンド幅(GB/sec)	12.8	25.3	177.4

実験結果

GeForce 9400 GT(16cores)，GeForce GT 220(48cores)，GeForce GTX 480(480cores)の並列化した最大最小検索の実験結果を表 4.2 に示す．

表 3.1 によれば，GeForce 9400 GT(16cores)では 1000 レコードにおいて，Ts は 652.9(μs)で Tm は 452.0(μs)となり，単一スレッドと比較して並列スレッドでは 1.4 倍の速度改善が確認された．GeForce GT 220(48cores)では 1 万レコードにおいて，Ts は 7,615,5(μs)で Tm は 556.4 (μs)となり，単一スレッドと比較して並列スレッドでは 13.7 倍の速度改善が確認された．GeForce GTX 480(480cores)では 10 万レコードにおいて，Ts は 14,863.2 (μs)で Tm は 248.1(μs)となり，単一スレッドと比較して並列スレッドでは 59.9 倍の速度改善が確認された．

表 4.2 最大最小検索の実験結果

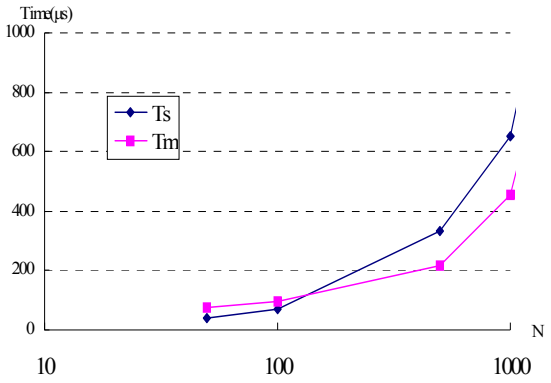
レコード数	GeForce 9400 GT(16cores)			GeForce GT 220(48cores)			GeForce GTX 480(480cores)		
	Ts(μs)	Tm(μs)	Ts/Tm	Ts(μs)	Tm(μs)	Ts/Tm	Ts(μs)	Tm(μs)	Ts/Tm
100	73.2	93.9	0.8	82.1	51.9	1.6	18.4	34.5	0.5
1000	652.9	452.0	1.4	766.3	91.1	8.4	152.0	53.4	2.8
10000	6,431.0	5,576.1	1.2	7,615.5	556.4	13.7	1,488.4	87.6	17.0
100000	63,978.3	64,548.6	1.0	76,150.1	6,045.1	12.6	14,863.2	248.1	59.9
1000000	641,346.0	742,639.0	0.9	761,405.0	69,633.2	10.9	153,374.0	2,686.9	57.1

速度比較

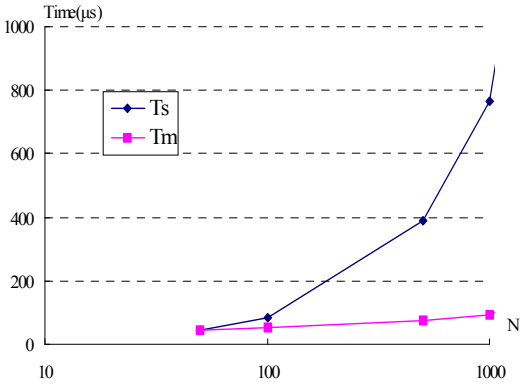
先ほどの実験結果で(1)レコード数 N=1000 までの時間比較および，(2)レコード数 N=100 万までの時間比較をそれぞれ(a) GeForce 9400 GT(16cores)，(b) GeForce GT 220(48cores)，(c) GeForce GTX 480(480cores)についてグラフで表した．縦軸は処理時間，横軸はレコード数を表している．黒色の線は単一スレッド，ピンク色の線は並列スレッドを示している．このグラフでは，横軸を log で目盛りを打っている．

(1-1) レコード数 N=1000 までの速度比較

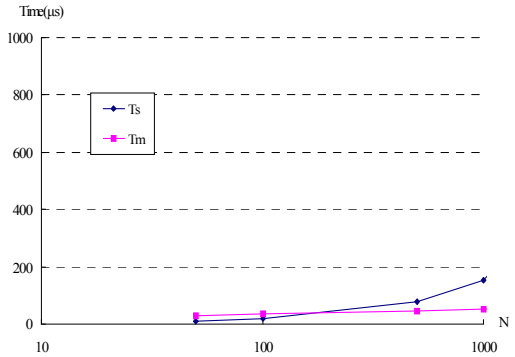
レコード数 N=1000 までの速度比較を図 4.1 に示す．横軸を log で目盛りを打っているので，GeForce GT 220(48cores)と GeForce GTX 480(480cores)の処理結果で並列スレッドである Tm が直線に見えていることから，計算複雑度がほぼ O(log(N))となっていることが確認出来る．また，提案手法を用いた GeForce 9400 GT(16cores)の処理結果では計算複雑度がほぼ O(log(N))となっていることが確認出来なかった．



(a) GeForce 9400 GT(16cores)の処理結果



(b) GeForce GT 220(48cores)の処理結果

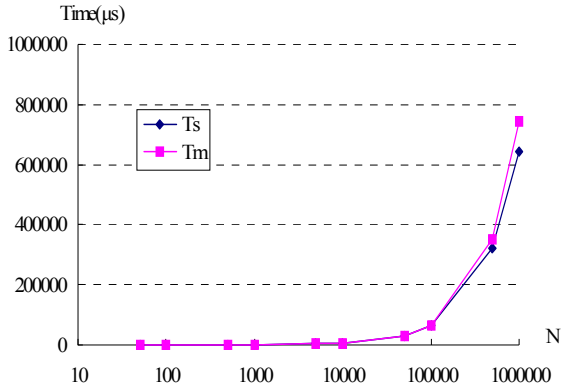


(c) GeForce GTX 480(480cores)の処理結果

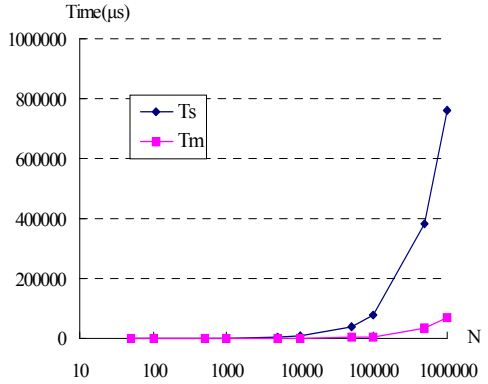
図 4.1 レコード数 N=1000 までの速度比較

(1-2) レコード数 N=100 万までの速度比較

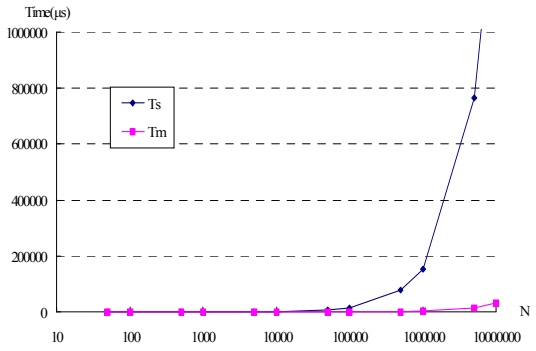
レコード数 N=100 万までの速度比較を図 4.2 に示す．レコード数 N=100 万までの速度比較の処理結果を見てみると，GeForce 9400 GT(16cores)の処理結果では Ts と Tm にはほとんど処理時間に差が無いことがわかる．GeForce GT 220(48cores)と GeForce GTX 480(480cores)の処理結果ではレコード数が増えていくと，並列スレッドである Tm が直線では無くなり始め，計算複雑度が O(log(N))から O(N)に近づいていることが確認出来る．



(a) GeForce 9400 GT(16cores)の処理結果



(b) GeForce GT 220(48cores)の処理結結果



(c) GeForce GTX 480(480cores)の処理結果

図 4.2 レコード数 N=100 万までの速度比較

考察

16 コアの実験結果では，レコード数 N=1000 以上において，1.2 倍，1.0 倍と速度比が下がっているのが確認できる．同様に 48 コアの実験結果ではレコード数 N=1 万以上において，12.6 倍，10.9 倍と速度比が下がり，480 コアの実験結果ではレコード数 N=10 万以上において，59.9 倍から 57.1 倍へと速度比が下がっているのがわかる．これはレコード数が増えてもデバイス側の GPU 数が有限なためと思われる．

(2) **統計データ計算の実験**

100 から 100 万レコードまで昇順に並べたデータを実験データとした．表の処理時間は Compute Visual Profiler の出力結果を用いた．単位は μs である．Ts は単一スレッドで平均値・分散．標準偏差を求めた処理時間，Tm は並列スレッドで平均値・分散・標準偏差を求めた結果である．Ts/Tm は並列化の高速された倍数を示す．プログラム開発環境には Core 数が 16cores, 48 cores, 480 cores の 3 種類で表 4.1 と同じ開発環境を用いた．

統計データ計算の結果

GeForce 9400 GT(16cores), GeForce GT 220(48cores), GeForce GTX 480(480cores)の並列化した統計データ計算の実験結果を表 4.3 に示す．表 4.1 によれば，GeForce 9400 GT(16cores)では単一スレッドと比較して並列スレッドでは速度改善が確認出来なかった．GeForce GT 220(48cores)は 1 万レコードにおいて，Ts は 8,366.1(μs)で Tm は 1,088.4(μs)となり，単一スレッドと比較して並列スレッドでは 7.7 倍の速度改善が確認された．GeForce GTX 480(480cores)では 10 万レコードにおいて，Ts は 17,035.6 (μs)で Tm は 540.8(μs)となり，単一スレッドと比較して並列スレッドでは 31.5 倍の速度改善が確認された．

表 4.3 統計データ計算実験

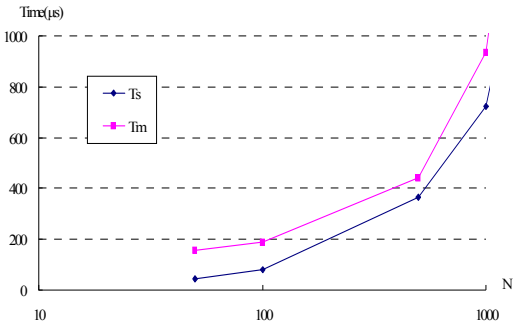
レコード数	GeForce 9400 GT(16cores)			GeForce GT 220(48cores)			GeForce GTX 480(480cores)		
	Ts(μs)	Tm(μs)	Ts/Tm	Ts(μs)	Tm(μs)	Ts/Tm	Ts(μs)	Tm(μs)	Ts/Tm
100	78.6	186.8	0.4	86.9	98.8	0.9	20.3	69.1	0.3
1000	723.3	934.2	0.8	839.2	182.6	4.6	169.3	105.5	1.6
10000	7,253.6	11,665.2	0.6	8,366.1	1,088.4	7.7	1,659.3	163.8	10.1
100000	72,906.6	131,720.7	0.6	83,628.3	11,933.5	7.0	17,035.6	540.8	31.5
1000000	729,020.0	1,506,753.3	0.5	834,849.0	138,309.3	6.0	170,332.0	5,427.7	31.4

速度比較

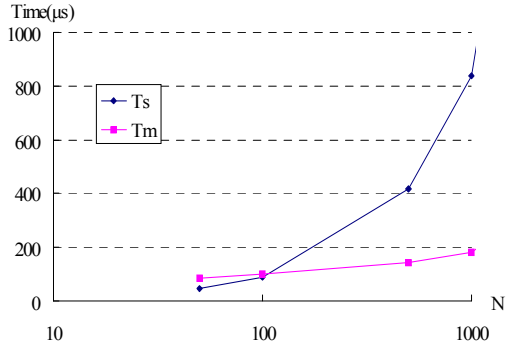
先ほどの実験結果で(1)レコード数 N=1000 までの時間比較および，(2)レコード数 N=100 万までの時間比較をそれぞれ(a) GeForce 9400 GT(16cores), (b) GeForce GT 220(48cores), (c) GeForce GTX 480(480cores)についてグラフで表した．縦軸は処理時間，横軸はレコード数を表している．黒色の線は単一スレッド，ピンク色の線は並列スレッドを示している．このグラフでは，横軸を log で目盛りを打っている．

レコード数 N=1000 までの速度比較を図 4.3 に示す．GeForce 9400 GT(16cores)の処理結果では，並列スレッド Tm の処理時間が単一スレッド Ts より多くなっているのがわかる．GeForce GT 220(48cores)と GeForce GTX 480(480cores)の処理結果では，並列スレッドである Tm が直線に見えていることから，計算複雑度がほぼ O(log(N))となっていることが確認出来る．

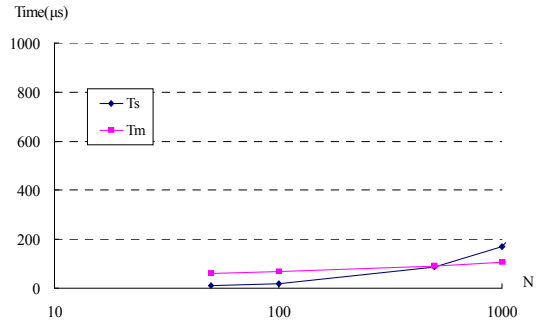
レコード数 N=100 万までの速度比較を図 4.4 に示す．レコード数 N=100 万までの比較でも GeForce 9400 GT(16cores)の処理結果では，並列スレッド Tm の処理時間が単一スレッド Ts よりも遅くなっているのが確認出来る．



(a) GeForce 9400 GT(16cores)の処理結果

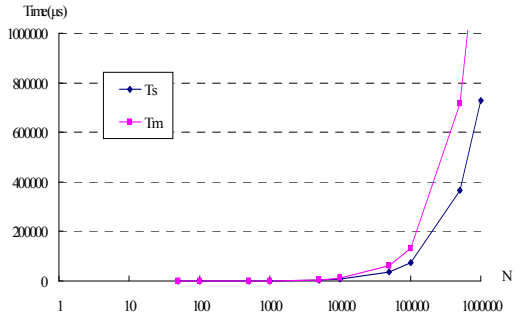


(b) GeForce GT 220(48cores)の処理結果

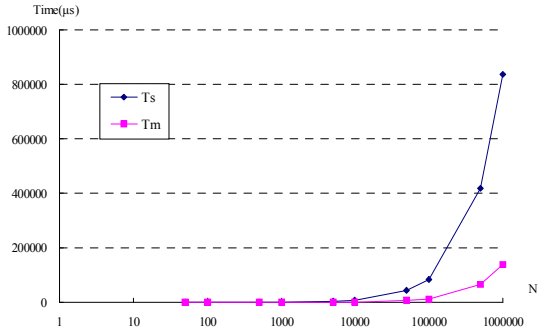


(c) GeForce GTX 480(480cores)の処理結果

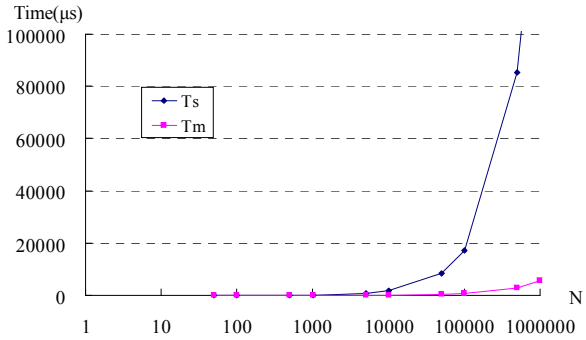
図 4.3　レコード数 N=1000 までの速度比較



(a) GeForce 9400 GT(16cores)の処理結果



(b) GeForce GT 220(48cores)の処理結果



(c) GeForce GTX 480(480cores)の処理結果

図 4.4　レコード数 100 万までの速度比較

GeForce GT 220(48cores) と GeForce GTX 480(480cores)の処理結果ではレコード数が増えていくと. 並列スレッドである Tm が直線では無くなり始め, 計算複雑度が O(log(N))から O(N)に近づいていることが確認出来る.

考察

16 コアの実験結果では, レコード数 N=1000 以上において, 0.6 倍, 0.5 倍と速度比が下がっているのが確認できる. 同様に 48 コアの実験結果ではレコード数 N=1 万以上において, 7.0 倍, 6.0 倍と速度比が下がり, 480 コアの実験結果ではレコード数 N=10 万以上において, 31.5 倍から 31.4 倍へと速度比が下がっているのがわかる. これはレコード数が増えてもデバイス側の GPU 数が有限なためと思われる.

5. まとめ

並列計算環境 GPGPU を利用した最大最小検索アルゴリズムと統計データ計算アルゴリズムについて O(log(N)) で求める方法とその実装, 実験結果を示した. 100～100 万レコードのデータについて, 単一スレッドと並列スレッドで比較を行った. 480 コアの並列化した最大最小検索では, 最大で 59.9 倍の速度改善が得られることが判明した. また, 480 コアの統計データ計算では, 最大で

31.5 倍の速度改善が得られることが判明した.

今後の課題としては, さらなるアルゴリズムの改善を検討している.

参考文献

[1]COMPUTERWORLD
“http://www.computerworld.jp/topics/561/ストレージ/180749/世界のデータ量が今後 10 年間で 44 倍に”

[2] Nadathur Satish, Mark Harris, Michael Garland,” Designing Efficient Sorting Algorithms for Manycore GPUs,”IPDPS(2009).

[3] Yifang Liu, Jiang Hu,”GPU-Based Parallelization for Fast Circuit Optimization”,DAC,pp943-946,(2009).

[4]西川尚紀,岩井啓輔,黒川恭一,”GPU の汎用計算環境 CUDA による暗号アルゴリズムに対するキークラックの高速化”IEICE,pp49-54,(2009).

[5] 戸川隼人著 「ザ・C」 [第 3 版] サイエンス社 (2007 年)

[6] NVIDIA 社 Cuda 3.2 マニュアル