DEIM Forum 2023 2a-7-2

GPU用データ並列プリミティブの大規模データへの拡張

中野　雅史† 　三浦　真矢† 　常　　穹† 　宮崎　　純†

† 東京工業大学情報理工学院情報工学系　〒 152–8550 東京都目黒区大岡山二丁目 12 番 1 号
E-mail: †{nakano,miura}@lsc.c.titech.ac.jp, ††{q.chang,miyazaki}@c.titech.ac.jp

あらまし　本研究では，既存の GPU 用データ並列プリミティブを使用し，CPU と GPU を併用して GPU のメモリサイズを超えるデータを処理する手法を提案する．提案手法では，GPU 上の計算と CPU-GPU 間のデータ転送のパイプライン化とストリームの使用，および CPU 上の計算と GPU に関する処理のオーバーラップにより，処理全体の効率化を図る．評価実験では，比較手法として，CPU 上で POSIX スレッドを用いてデータ並列プリミティブと同様の処理を実装し，実行時間を比較した．また，プロファイラを使用して GPU 上での処理の様子を調べた．評価実験の結果から，本研究で扱った全てのデータ並列プリミティブにおいて，比較手法よりも高速な処理が可能であることが判明した．

キーワード　先進ハードウェア活用・GPU, 大規模データ処理, データ並列プリミティブ

1 はじめに

社会で収集・集積されるデータは年々増加しており，それらは様々な用途に利用される．特に，地震や気象の予測・シミュレーションやディープラーニングなどの高度な技術には，膨大な計算量やデータ量が必要とされる場合が多い．大規模データに対する処理の高速化や効率化には，高性能なプロセッサの開発のようなハードウェア的手法のほか，既存のハードウェアを組み合わせて高性能なアーキテクチャを構成する手法がある．後者の手法の 1 つに，ドメインスペシフィックなプロセッサを他のドメインや汎用計算に活用する，GPGPU（General-purpose computing on Graphics Processing Units）がある．

GPGPU は，画像処理を目的として設計されたプロセッサである GPU(Graphics Processing Unit) を，汎用計算に利用する技術である．汎用的な処理に適した CPU に対し，単純な設計のコアを多数搭載する GPU は並列計算に優れており，GPGPU では，GPU 上での並列処理を組み合わせて高速な汎用計算を実現する．また，入力配列に対する scan [1] や sort [2], reduce などの汎用計算を GPU 上で実装したものとして，データ並列プリミティブがある [3].

データ並列プリミティブを実装するライブラリとして，CUDPP [4] や ModernGPU [5] などが存在する．これらのライブラリは，NVIDIA 社の CUDA のような GPGPU のための開発環境で使用することができる．しかし，既存のデータ並列プリミティブは，入力配列として GPU のメモリサイズを超えないデータを対象としているため，処理できるデータの大きさに制限がある．既存の GPU の多くは，CPU のメモリよりも小さなメモリを持つため，この制限は大規模データの処理を行う際に問題となる．

本研究では，データの分割と CPU 上での処理，および GPU 上での既存のデータ並列プリミティブを使用した処理を組み合わせ，データ全体に対して演算を行う手法を提案する．提案手法では，GPU 上での計算と CPU-GPU 間のデータ転送をパイプライン化して，CUDA Stream を用いてオーバーラップさせる効率化と，CPU 上の計算と GPU に関する処理をオーバーラップさせる効率化を行う．評価実験では，比較手法として，CPU 上で POSIX スレッドを用いてデータ並列プリミティブと同様の処理を実装し，実行時間を比較した．評価実験の結果から，本研究で扱った reduce，hashtable，segmented sort の全てで，比較手法よりも高速な処理が可能であることが判明した．

2 GPGPU

画像処理を目的とする GPU は，単純な計算処理を行う数千個のコアを使った並列処理が特徴である．GPGPU は，この特徴を汎用計算に利用する技術であり，GPU の構造に適したプログラムを動作させた場合，汎用計算の逐次処理を得意とする CPU よりも高速な処理が期待できる．

GPGPU では，GPU の性能を最大限に活かすプログラムを作成することが重要である．GPGPU プログラムの処理の基本的な流れを次に示す．

（1） GPU で扱うデータのために GPU 上のメモリ（以下，デバイスメモリと呼ぶ）を確保する
（2） CPU から GPU にデータを転送する
（3） GPU で処理を行う関数（以下，カーネルと呼ぶ）を起動し，処理を実行する
（4） GPU から CPU に計算結果のデータを転送する

本研究では，NVIDIA 社の GPU と GPGPU 用の開発環境である CUDA を用いて提案手法の実装と評価実験を行うため，以下では CUDA を用いたプログラミングについて説明する．

2.1 CUDA

2.1.1 CUDA プログラムの処理の流れ

CUDA プログラミングでは，GPU で処理するデータの入力や生成，使用するデバイスメモリの確保と解放，CPU メモリ

（以下，ホストメモリと呼ぶ）とデバイスメモリの間のデータ転送などは，CPU 上で呼び出され実行される関数によって行われる．そのため，CUDA プログラムは，CPU 上で実行されるホストコードと，CUDA カーネルのように GPU 上で実行されるデバイスコードで構成される．NVIDIA の CUDA コンパイラである nvcc（NVIDIA CUDA Compiler）は，コンパイル時にホストコードとデバイスコードを分離する．基本的な CUDA プログラムの処理の流れは，2 で示した GPGPU プログラムの処理の流れと同様で，最後にデバイスメモリの解放を行うことが加わる．(3) を除く 4 つの操作は，CUDA Runtime API が提供する関数で行うことができる．また，(3) の操作で実行されるカーネルは，デバイスコードに記述された CUDA カーネルである。

2.1.2 CUDA ストリーム

一般にデバイスメモリサイズは，ホストメモリサイズよりも小さい場合が多い．したがって，GPU を使用する処理では，デバイスメモリより大きなデータを処理する場合，両者の間のデータ転送を何度も行う必要がある．これは処理の実行時間に大きく影響し，ボトルネックとなる可能性がある．

そこで，CUDA は，マルチストリームによるデータの非同期処理を実現する API を提供している．CUDA Runtime API の cudaStreamCreate( ) 関数で生成したストリームを cudaMemcpyAsync( ) 関数や GPU カーネルの引数に渡すと，これらの関数によるデータ転送と計算処理は，発行された順序に従い，CUDA ランタイムが判断したタイミングで実行される。複数のストリームを使用する場合，あるストリームに割り当てられたデータ転送とカーネルによる処理計算は，他のストリームに割り当てられた異なる種類の処理とオーバーラップされる．また，非同期なデータ転送を行う場合，ページングできない領域（ピンメモリ）をホストメモリとして使用する必要があり，ピンメモリの確保は cudaMallocHost( ) 関数を用いて行うことができる．ストリームを使用したパイプライン処理の様子を図 1 に示す．

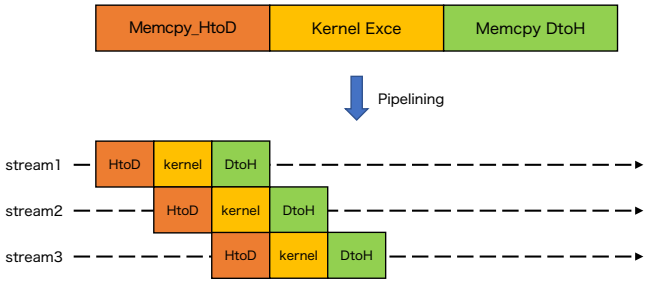

図 1　ストリームを用いたパイプライン処理の例

3 関 連 研 究

3.1 データ並列プリミティブ

データ並列プリミティブは，汎用計算を GPU 上で実装したものであり，提供しているライブラリには CUDPP や ModernGPU などがある．本研究で使用したデータ並列プリミティブについて以下に述べる．

reduce

このデータ並列プリミティブは，要素数 n の配列 $[a_0, a_1, ..., a_{n-1}]$ と二項演算子 $\oplus$ を受け取り，$a_0 \oplus a_1 \oplus ... \oplus a_{n-1}$ を計算して出力する．

hashtable

このデータ並列プリミティブは，GPU 上でハッシュテーブル [6] を実装したものである。CUDPP の hashtable は，入力として同じ長さの key と value の配列が与えられると，key 配列から重複する key を取り除いてソートし，配列 $[key_0, key_1, ..., key_{n-1}]$ を作成する．その後，この配列の各 key とそのインデックスを，式 (1) のハッシュ値を用いてハッシュテーブルに格納する．一方で，value 配列は key の順にソートして保持される．

$$\text{ハッシュ値} = ((random\ 定数\ 1\hat{}key) + random\ 定数\ 2)\%定数素数 \tag{1}$$

ある key に対して計算されたハッシュ値が，値の異なる他の key に対して計算されたハッシュ値と一致する場合があり，この現象を衝突と呼ぶ．CUDPP の hashtable は Cuckoo Hashing [7] を採用しており，衝突が起きたとき，既に格納されている方の key に対して random 定数 1 と random 定数 2 を異なる値に変えてハッシュ値を再計算する，という操作を繰り返す．定めた試行回数を超えた場合，もう 1 つのハッシュテーブルへの格納を試みる．

segmented sort

このデータ並列プリミティブは，配列 A と，m 個のセグメントの境界を定義する要素数 $m-1$ の配列 B を受け取り，配列 A をセグメントごとにソートする．このとき，$B[i], i = 0, 1, ..., m-2$ は，$i+1$ 番目のセグメントの開始インデックスを表す．

merge sort

このデータ並列プリミティブは，配列と比較に用いる関数を受け取り、その関数に従う順に配列をソートする．

3.2 拡張データ並列プリミティブの実装

三浦ら [8] は，既存のデータ並列プリミティブを用い，入力データを分割して適切な順序で処理を行うことで，GPU のメモリサイズよりも大きな入力データの処理が可能な，transform, reduce, scan, scatter, sort を実装した．CPU-GPU 間のデータ転送と計算をパイプライン化し，CUDA Stream を使用してそれらをオーバーラップさせることで処理の効率化も行なった．評価実験では，比較手法として CPU 上で各データ並列プリミティブと同じ処理を実装し，scatter 以外のプリミティブで比較手法より高速な処理が可能であることを示した．

3.3 データ並列プリミティブを活用したデータベース処理の実装

大原ら [9] は，データベースに蓄積されたシーケンスデータに対する行パターンマッチングを標準化した SQL/RPR（Row Pattern Recognition）の処理に，データ並列プリミティブを使用して，GPU 上で処理する手法を提案した．SQL/RPR の主な処理は以下である．

（1） PARTITION BY 句を使用したデータベースのパーティション

（2） ORDER BY 句を使用したデータの並び替え

（3） DEFINE 句を使用した行パターン変数の暫定マッピング

（4） PATTERN 句を使用したパターンの抽出

提案手法では (1), (2), (4) の処理の一部に，それぞれ CUDPP の hashtable，ModernGPU の segmented sort，ModernGPU の reduce を用いた．評価実験では，比較手法として PostgreSQL 上で SQL/RPR を実装し，提案手法がクエリ処理の実行時間を短縮することを示した．

4 提 案 手 法

本節では，既存のデータ並列プリミティブを使用して，GPU のメモリサイズよりも大きなデータを GPU と CPU を併用して処理するための具体的な実装方法について説明する．本研究で扱ったデータ並列プリミティブは reduce，hashtable，segmented sort である．

4.1 reduce

はじめに，入力配列を $m:n$ に分割し，それぞれ GPU 上と CPU 上で同時に処理を行う．GPU 上で処理する部分に対しては，配列を GPU のメモリサイズ以下に分割し，2.1.2 で示した方法でパイプライン処理を行う．カーネルには CUDPP の reduce を使用する．CPU 上で処理する部分に対しては，POSIX スレッドを用いて，配列をスレッド数と同数に分割してスレッドに割り当て，並列処理を行う．最後に GPU と CPU の計算結果に対して CPU 上で計算を行い，出力する．また，事前に行った実験でこれらの GPU 上と CPU 上での処理の速度を測定し，分割比 m，n をそれぞれ 0.65，0.35 とした．reduce の動作例を図 2 に示す．

4.2 hashtable

本研究では，GPU のメモリサイズより大きなデータを扱えるように拡張した，データ並列プリミティブの hashtable を用いてハッシュテーブル作成の高速化を図り，入力データを同一 key ごとにグループ化する処理の効率化を目的とする．

reduce における GPU 上での処理と同様に，key と value の 2 つの入力配列を分割してストリームに割り当て，カーネルには CUDPP の MultivalueHashTable を用いる。

MultivalueHashTable は，与えられた key 配列から重複する key を除いたユニークな key 配列を生成し，その配列のインデックスを value としてハッシュテーブルを作成する．格納

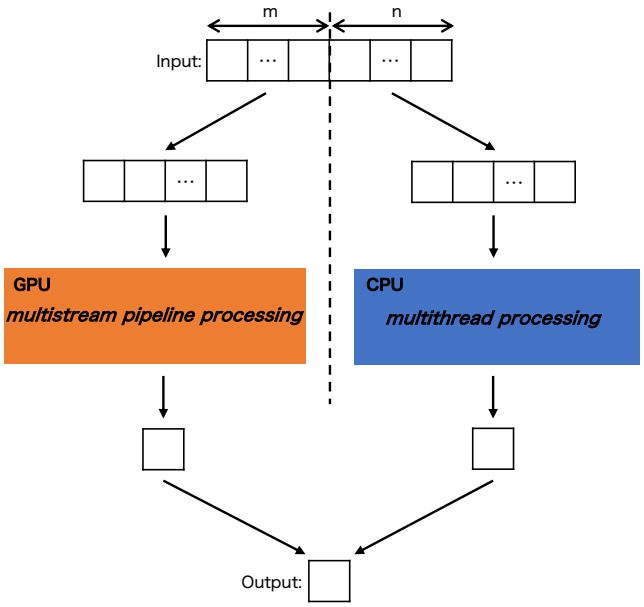

図 2 reduce の動作例

される値は uint2 型で，上位 32bit に key の値，下位 32bit に value の値を持つ．一方で，MultivalueHashTable は，与えられた value 配列を key の順にソートして保持する．その取得には cudppMultivalueHashGetAllValues() 関数を用い，cudppHashRetrieve() 関数を使って key ごとのオフセットを取得することができる．また，cudppHashRetrieve() 関数は，key に対して求めたハッシュ値でテーブルを探索する．探索して見つけたエントリに格納されている key とクエリとして用いた key を照合し，一致する場合には，その value を取得してオフセット情報の取得に利用する．

各ストリームは，どちらも変更を加えた cudppMultivalueHashGetAllValues() 関数と cudppHashRetrieve() 関数を呼び出す．後者の関数によって得られる key ごとのオフセット配列を用いて，前者の関数によって得られる key の順にソートされた value 配列を，C++の std::unordered_map を用いて実装したハッシュテーブルに格納する．ハッシュテーブルへの格納は，POSIX スレッドに key を割り当て，各スレッドがその key に対応する部分を value 配列から取り出し，ハッシュテーブルへ格納する方法で行う．この CPU 上での処理は，GPU の処理と並行して行う．hashtbable の動作例を図 3 に示す．

4.3 segmented sort

セグメントの大きさに従って処理方法を変えることで，効率的に入力配列内の全てのセグメントをソートする．GPU の使用可能なメモリサイズを M，1 つの CUDA ストリームに割り当てたメモリサイズを M' とする．大きさが M' 以下のセグメントは，それらを 1 つ以上集めてストリームに割り当て，ModernGPU の segmented_sort を適用する．M' より大きく M 以下のセグメントは，ストリームを使用したオーバーラップは行わず，1 つずつ順にデータ転送し ModernGPU の

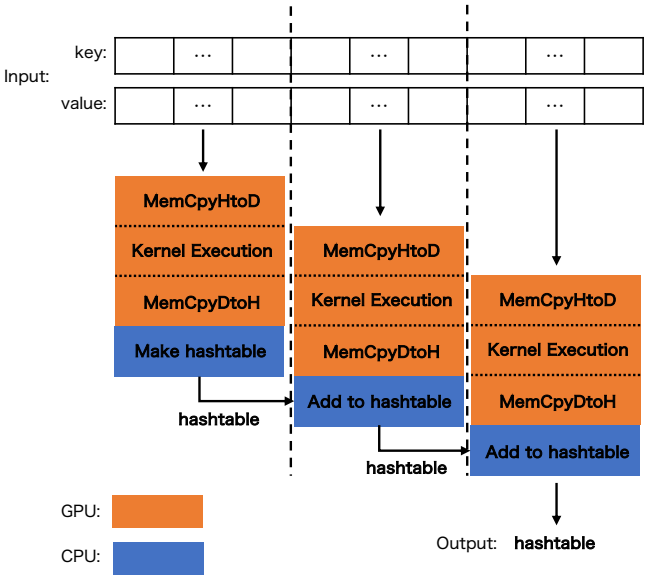

図 3　hashtable の動作例

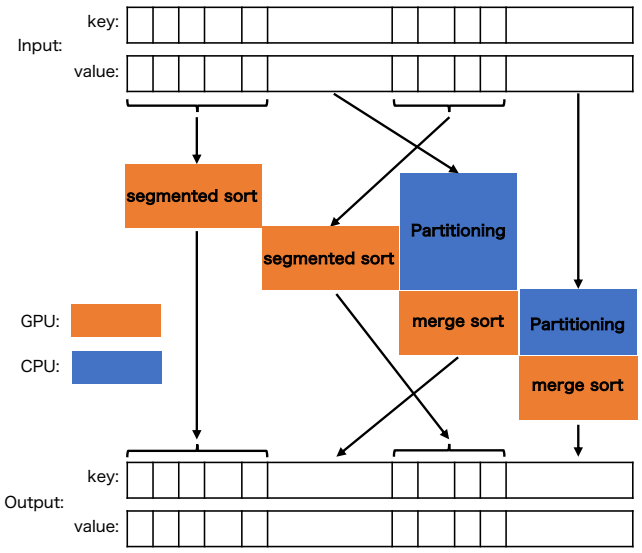

図 4　segmented sort の動作例

mergesort を適用する．M よりも大きなセグメントは，CPU 上でクイックソートの分割アルゴリズムにより M 以下に分割し，ModernGPU の mergesort を適用する．

この分割では，ストリームを使用したオーバーラップによって短縮される時間よりも，分割にかかる時間の方が大きいため，M' 以下への分割は行わない．分割アルゴリズム中の分割の基準となるピボットの選択は，分割する配列の長さを n としてインデックスが 0，$\frac{n}{2}-1$，$n-1$ の要素の中から中央値を求めてピボットとする方法を用いる．分割が終了しないことがないように，ひとつ前の分割で使用したピボットと同じ値がピボットに選択された場合，ピボット選択に使用する要素のインデックスを 1，$\frac{n}{2}$，$n-2$ のようにずらせていく．また，CPU 上での分割は GPU 上の処理と並行して行うことで，全体の処理時間を短縮する．

GPU 上でソートしたセグメントをホストメモリにある入力配列の適切な位置にコピーすることで，全てのストリームの処理が完了すると，入力配列はセグメントごとに key についてソートされたことになる．segmented sort の動作例を図 4 に示す．

5　評 価 実 験

本節では，データ並列プリミティブと同様の処理を CPU 上で実装した手法，および POSIX スレッドを用いて CPU 上で実装した手法を比較手法として，提案手法と比較し評価を行う．

5.1　比 較 手 法

比較手法として用いる，CPU 上でデータ並列プリミティブと同様の処理を実装した方法，および POSIX スレッドを用いて CPU 上で実装した方法について述べる．比較手法の実装には C++を用い，POSIX スレッドを使った実装ではスレッド数を 16 とした．

CPU 上で実装した reduce は，入力配列の先頭から 2 つずつ要素を取り出して計算を行い，全ての要素について計算し長さが半分になった配列に対し，同様の処理を再帰的に行う．POSIX スレッドを使った実装では，入力配列をスレッド数で分割してスレッドに割り当て，最後に全てのスレッドの計算結果に対して，シングルスレッドで reduce を行う。POSIX スレッドを使った比較手法の実装を Algorithm1 に示す．

CPU 上で実装した hashtable は、std::unordered_map をハッシュテーブルとして用い，格納される value は std::vector を用いて実装した．std::unordered_map は木構造で実装される std::map とは異なりハッシュ表で実装されており，ハッシュテーブルへの格納は，key と value の同じ長さの入力配列に対して同じインデックスの値をペアとして，先頭から 1 ペアずつハッシュテーブルに追加する。POSIX スレッドを使った実装では，スレッド数を N として以下の手順で処理を行う．

（1）　入力を M 個に分割する

（2）　i 番目の分割をさらに N 個に分割し N 個のハッシュテーブルを作成する

（3）　ハッシュテーブルを 2 つずつ結合する

（4）　ハッシュテーブルが 2 つになるまで（3）の処理を繰り返す

（5）　$i=M$ であれば，$M\times 2$ 個のハッシュテーブルからベースとするものを 1 つ選び，残りの $M\times 2-1$ 個を順にベースのハッシュテーブルへ結合する

（6）　$i=i+1$ として（2）に戻る

手順（1）と（5）は，実行時間への影響が少ない範囲でメモリ使用量を抑えるために行う．POSIX スレッドを使った比較手法の実装を Algorithm2 に示す．

CPU 上で実装した segmented sort は，入力配列のセグメントに対して順に std::sort を適用し，全てのセグメントをソートする．POSIX スレッドを使った実装では，はじめに要素数が n 以上のセグメントを C++の Boost ライブラリの

Algorithm 1 比較手法の reduce の擬似コード

```
1: function TH_REDUCE
2:     while num ≧ 1 do
3:         for i ← 0 to num/2 do
4:             input[i] ← input[2 * i] + input[2 * i + 1]
5:         end for
6:         num ← num/2
7:     end while
8:     result[tid] ← input[0]
9: end function
10:
11: function REDUCE
12:     num ← inputSize/numThreads
13:     for i ← 0 to numThreads do
14:      \\multi thread
15:         TH_REDUCE()
16:     end for
17:      \\single thread
18:     TH_REDUCE()
19: end function
```

Algorithm 2 比較手法の hashtable の擬似コード

```
1: function TH_MAKETABLE
2:     for i ← 0 to num do
3:         insert pair(key[i], values[i]) to table
4:     end for
5: end function
6:
7: function TH_MERGETABLE
8:     for i ← 0 to numKeys do
9:         move table_src[i] to table_dest[i]
10:    end for
11: end function
12:
13: function HASHTABLE
14:    for i ← 0 to partition do
15:        num ← inputSize/(partition*numThreads)
16:        for j ← 0 to numThreads do
17:          \\multi thread
18:            TH_MAKETABLE()
19:        end for
20:        while numTable ≧ 2 do
21:            for i ← 0 to numTable/2 do
22:              \\multi thread
23:                TH_MERGETABLE()
24:            end for
25:            numTable ← numTable/2
26:        end while
27:    end for
28:    for i ← 0 to pertition * 2 − 1 do
29:        \\multi thread
30:        TH_MERGETABLE()
31:    end for
32: end function
```

block_indirect_sort を用いてソートする．次に行う要素数が n 以下のセグメントのソートは，1 つのスレッドに 1 つのセグメントを割り当て，std::sort を適用する方法でソートを行う．boost::block_indirect_sort の性能を調べる実験を事前に行い，本研究では n を 1×10^7 とした．boost::block_indirect_sort は最悪計算時間と平均計算時間が $n \log n$ のソートアルゴリズムである．POSIX スレッドを使った比較手法の実装を Algorithm3 に示す.

5.2 実験準備

評価実験に用いたマシンの構成を表 1 に示す.

表 1　実験に用いたマシンの構成

CPU		Intel Core i7-10700 (2.90GHz，8 コア)
RAM		24GB
OS		Ubuntu 20.04.5 LTS
GPU		NVIDIA GTX 1650 (Turing)
	CUDA コア	896
	ベースクロック	1485MHz
	メモリ量	4GB GDDR5
CUDA		CUDA 11.7

5.3 実験内容

本実験では，比較手法と提案手法の実行時間の計測，および NVIDIA Nsight Systems を用いて提案手法のプロファイルを行った．提案手法の実行時間の計測は，ホストメモリからデバイスメモリへのデータ転送開始前から，デバイスメモリからホストメモリへの計算結果の転送と，CPU 上での必要な処理の両方が完了するまでを実行時間として計測した.

reduce は，二項演算子に + 用いて，配列の全ての要素の和を求める計算を行なった．hashtable は，一様分布に従う [0, 9] の範囲の乱数で初期化した key 配列，要素数を n として [0, $n-1$] の範囲の一意の整数で初期化した value 配列を入力として受け取り，std::unordered_map で実装した CPU 上のハッシュテーブルに全ての key と value のペアを格納するタスクを行なった．segmented sort は，一様分布に従う [1, 20] の範囲の乱数で初期化した key 配列，hashtable と同様の value 配列，および zipf 分布に従う [1, $n-1$] の乱数で初期化した要素数 99 のオフセット配列を入力として受け取り，セグメントごとに key 配列と value 配列を key についてソートするタスクを行なった．zipf 分布のパラメータ s は，1.5 とした.

reduce の入力には 32bit 浮動小数点型配列を用い，hashtable と segmented sort の key，value の入力配列には 32bit 整数型配列を用いた．reduce では入力配列の長さを 0 から 3×10^9(約 11.2GB) まで変化させ，hashtable と segmented sort では入力配列の長さを 0 から 1.6×10^9(約 5.96GB) まで変化させた.

Algorithm 3 比較手法の segmented sort の擬似コード

```
1: function TH_SORT
2:     sort(array)
3: end function
4:
5: function TH_MULTISORT
6:     multisort(array)
7: end function
8:
9: function SEGMENTED_SORT
10:    for i ← 0 to numSegments do
11:        if segmentSize ≧ n then
12:            TH_MULTISORT()
13:        end if
14:    end for
15:    for i ← 0 to numSegments do
16:        if segmentSize < n then
17:            TH_SORT()
18:        end if
19:    end for
20: end function
```

5.4 実験結果

比較手法と提案手法の実行時間の測定結果を，図 5 から図 7 に示す。reduce では，提案手法が，比較手法のうち CPU 実装に比べて約 6.09-6.84 倍，POSIX スレッドを使った CPU 実装に比べて約 1.80-1.96 倍の性能，hashtable では，比較手法のうち CPU 実装に比べて約 21.0-26.3 倍，POSIX スレッドを使った CPU 実装に比べて約 2.76-3.84 倍の性能となった．segmented sort では，比較手法のうち CPU 実装に比べて約 46.7-160.8 倍，POSIX スレッドを使った CPU 実装に比べて約 8.62-32.8 倍の性能となった．

提案手法のプロファイル結果を，図 8 から図 10 に示す。図 10 は，segmented sort の提案手法において，CUDA ストリームに割り当てたメモリサイズ以下のセグメントに対して，GPU 上で行なった処理のプロファイル結果である．実験では 3 つまたは 4 つの CUDA ストリームを使い，図 8 から図 10 では，上部の黒枠で囲んだ部分は全てのストリームを合わせた処理の流れ，その下は各ストリームの処理の流れを表示している．図の緑の部分はホストメモリからデバイスメモリへのデータ転送，青の部分はカーネル実行，赤の部分はデバイスメモリからホストメモリへのデータ転送の処理を表している．reduce では，ホストメモリからデバイスメモリへのデータコピーが実行時間の大半を占め，CUDA ストリームを利用した処理のパイプライン化とオーバーラップにより，カーネル実行とデバイスメモリからホストメモリへのデータコピーが隠蔽されたことがわかる．hashtable では，カーネル実行が最も時間がかかる処理で，処理全体ではホストメモリからデバイスメモリへのデータコピーが他の 2 つの処理に隠蔽されたことがわかる．segmented sort でも，カーネル実行が最も時間がかかる処理で，hashtable と同様な処理のオーバーラップが行われたことがわかる．

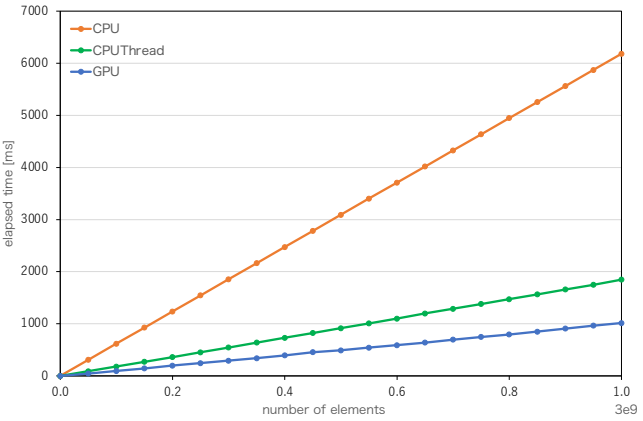

図 5　reduce の実行結果

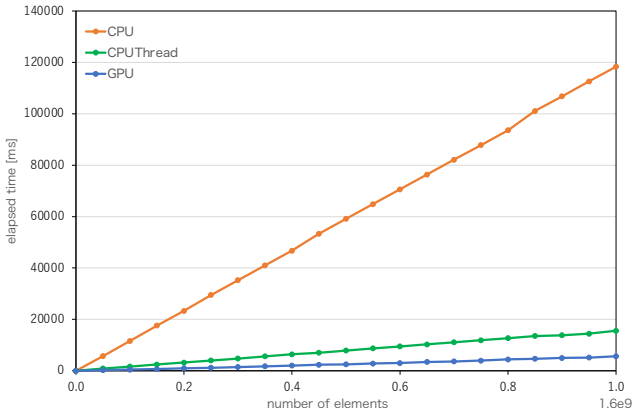

図 6　hashtable の実行結果

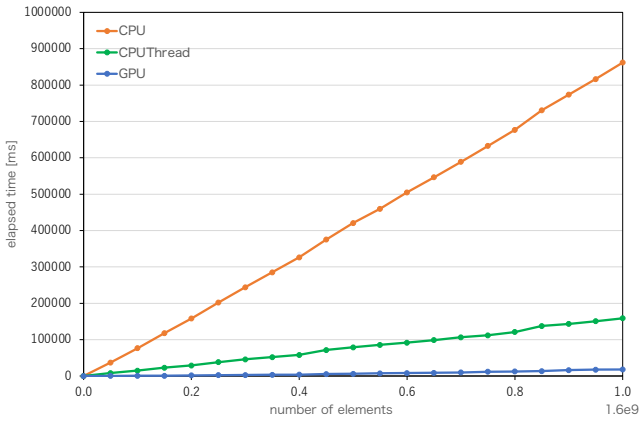

図 7　segmented sort の実行結果

図 8　reduce のプロファイル結果

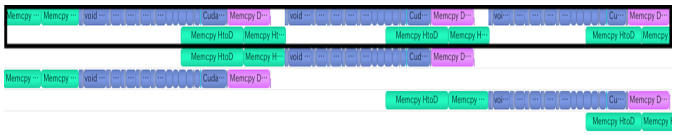

図 9　hashtable のプロファイル結果

図 10　segmented sort のプロファイル結果

5.5 考　　察

reduce では，提案手法において，入力配列を分割して CPU と GPU で同時に処理を行なったことに加えて，GPU 上での処理のオーバーラップによってカーネル実行がデータ転送に隠蔽されたことで，実行時間が短縮されたと考えられる．その結果，提案手法は POSIX スレッドを使用して効率化を行なった比較手法よりも実行時間が短くなったと考えられる．

hashtable では，提案手法において，ホストメモリからデバイスメモリへのデータ転送が，カーネル実行と逆方向のデータ転送に隠蔽されたことで，GPU 上での処理が効率的に行われた．さらに，GPU 上での計算結果を使用してハッシュテーブルに要素を追加する CPU 上での処理を，GPU に関する処理とオーバーラップさせることで，POSIX スレッドを使用して効率化を行なった比較手法よりも高速に，ハッシュテーブルを作成することができたと考えられる．

segmented sort では，提案手法において，CUDA ストリームに割り当てたメモリサイズ以下のセグメントに対して GPU 上で処理を行う部分で，hashtable と同様の効率化が行われた．また，GPU のメモリサイズより大きなセグメントの数が比較的少ない場合には，CPU 上で行う分割処理の大部分が，GPU 上で行うその他のセグメントの処理にオーバーラップする．これらにより，POSIX スレッドを使用して効率化を行なった比較手法に比べて，提案手法の性能が上回ったと考えられる．

6 終 わ り に

本研究では，データ並列プリミティブである reduce, hashtable, segmented sort を，CPU と GPU を併用して GPU のメモリサイズを超えるデータの処理が可能となるように拡張する手法を提案した．

評価実験では，比較手法として，データ並列プリミティブと同様の処理を CPU 上で実装した手法と，POSIX スレッドを用いて CPU 上で実装した手法を用意した．評価実験は，各データ並列プリミティブに対して，入力配列の大きさを変えながら実行時間を測定した．評価実験の結果から，CPU 上の実装と比べて約 6.09-160.8 倍，POSIX スレッドを使った CPU 上での実装と比べて約 1.80-32.8 倍の性能向上を実現した．

今後の課題として，提案手法に最適な実装方法を検討し，さらなる高速化を実現すること，CPU と GPU をより効果的に併用する手法の考案，本研究で対象としなかったデータ並列プリミティブの拡張などが挙げられる．また，本研究で対象としたデータ並列プリミティブを用いて，大原ら [9] が提案した，GPU 上での SQL/RPR の処理手法のような既存のアプリケーションや処理手法を，メモリサイズの制約がないものに拡張することも 1 つの課題と考えられる．さらに，併用するプロセッサや計算機の性能差や通信手段からデータの分割比率を求める自動最適化手法の考案も，本研究に関連する課題として挙げられる．

謝　　辞

本研究は、JST CREST JPMJCR22M2 の支援を受けたものである。

文　　献

[1] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient Parallel Scan Algorithms for Many-core GPUs. eScholarship, University of California, 2011.

[2] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–10, 2009

[3] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 511–524, June 2008.

[4] CUDPP, https://github.com/cudpp/cudpp (2023 年 1 月 9 日アクセス)

[5] Sean Baxter: moderngpu 2.0, https://github.com/moderngpu/moderngpu (2023 年 1 月 9 日アクセス)

[6] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Almenta. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, pp. 39–53. Elsevier, 2012

[7] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[8] 三浦 真矢, 常 穹, 宮崎 純. GPU メモリサイズより大きなデータを対象としたデータ並列プリミティブの開発. 第 13 回データ工学と情報マネジメントに関するフォーラム (DEIM2022), B24-2(day2 p43), 2022.

[9] Tsubasa Ohara, Qiong Chang, and Jun Miyazaki. Fast SQL/Row Pattern Recognition Query Processing Using Parallel Primitives on GPUs. In *Proceedings of the 32nd International Conference on Database and Expert Systems Applications (DEXA2021), LNCS 12923*, pp. 22-34, September. 2021.