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

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: †miura@lsc.c.titech.ac.jp, ††{q.chang,miyazaki}@c.titech.ac.jp

あらまし 本研究では，ModernGPU で提供されているデータ並列プリミティブを使用し，GPU メモリサイズよりも大きなデータを対象としたデータ並列プリミティブの実装方法について提案する．提案手法では，ストリームを利用し，データ転送と計算をパイプライン化し，オーバーラップさせることによる効率化も行う．評価実験では，CPU 上で同じ動作をするプログラムを比較手法とし，実行時間を比較した．また，プロファイラを用い，提案手法の動作の様子を測定した．評価実験の結果，いくつか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に処理することが可能であることが判明した．

キーワード GPU, データ並列プリミティブ, 大規模データ処理

1 はじめに

本来，画像処理を行うことを目的に設計されたプロセッサである GPU(Graphics Processing Unit) を汎用計算に流用する GPGPU(General-purpose computing on Graphics Processing Units) が存在する．NVIDIA 社の CUDA や，Apple 社の OpenCL などの GPGPU のための汎用的な開発環境の登場により，GPU のプログラマビリティは向上しているが，インターフェースがベンダー依存であったり，GPU ハードウェアアーキテクチャに対する理解が必要であったりするといった障壁が存在する．

そこで，入力配列の transform や，reduction，prefix scan，sort などの汎用計算を GPU 上で実装したデータ並列プリミティブが研究されている．データ並列プリミティブを提供するライブラリとして，ModernGPU や CUDPP などがある．また，データ並列プリミティブを使用した研究として，リレーショナルデータベースにおける Join 演算を実装した研究 [7] や，GPU 上でのリレーショナルクエリ処理システムに関する研究 [6] などが知られている．しかし，既存のデータ並列プリミティブは GPU メモリ内で完結できる大きさのデータを対象としているため，それらを使用して作成したアプリケーションでは処理できるデータサイズが GPU メモリサイズに制限されてしまうという問題を抱えている．

本研究では，既存のデータ並列プリミティブを用い，入力データを GPU メモリサイズに収まるように分割し，適切な順序で処理を行うことで入力データ全体に演算を行う手法を提案する．提案手法では，CUDA Stream を利用し，GPU メモリとホストメモリ間のデータ転送と計算をパイプライン処理し，オーバーラップさせることによる効率化も行なった．

今回実装したデータ並列プリミティブについて CPU 上で実装した比較手法と実行時間を比較することで評価を行なった．評価実験の結果，transform，reduce，scan，sort プリミティブにおいて比較手法よりも高速に処理できることが分かった．

2 関連研究

2.1 GPGPU

GPU は本来グラフィックス用の並列処理に特化したプロセッサであったが，性能が向上し，より一般化された結果，単純な制御ロジックの処理を得意とする軽量なコアを数千個持ち，大量のデータを並列処理することを得意とするプロセッサとなった．一方，CPU は GPU とは補完的な性質を持ち，複雑な制御ロジックを実行するためのコアを数十個持ち，逐次処理を実行するプロセッサである．このような特徴を持つ GPU を汎用計算に使用する技術が GPGPU である．この際，GPU は PCI-Express バスをを通じて CPU と連動しなければならない．そのため，CPU をホスト，GPU をデバイスと呼ぶ．

GPGPU アプリケーションを開発するための汎用的なプラットフォームとして，Apple 社の OpenCL [2] や NVIDIA 社の CUDA [5] がある．CUDA では C/C++言語向けの拡張が提供されている．GPGPU プログラムは CPU で実行されるホストコードと，GPU で実行されるデバイスコードの２つの部分から構成されている．ホストコードはアプリケーションの初期化や，デバイスの設定，データの管理などを行う．デバイスコードは GPU の特性を活かし，データ並列性の高い部分の処理の高速化に使用される．

2.2 CUDA

2.2.1 CUDA プログラムのコンパイルと実行

NVIDIA の CUDA コンパイラ nvcc（NVIDIA CUDA Compiler）はコンパイル時にデバイスコードをホストコードから切り離す．ホストコードは C/C++コンパイラによって更にコンパイルされる．デバイスコードは nvcc によって PTX（Parallel

Thread Execution）という GPU アセンブリ言語に変換される．PTX コードは GPU ドライバにより，実行する GPU に適したネイティブコードに変換される．

2.2.2 CUDA プログラムの構成

CUDA プログラムの構造は，主に以下の 5 つのステップで構成されている．

（1） GPU メモリの確保

（2） CPU メモリから GPU メモリへのデータコピー

（3） CUDA カーネルを呼び出し，プログラムに必要な計算を実行

（4） GPU メモリから CPU メモリへのデータコピー

（5） GPU メモリの開放

2.2.3 CUDA ストリーム

CUDA のストリームとは CUDA の一連の非同期操作のことであり，それらの操作はホストコードで発行された順序に従いデバイスで実行される．デバイス上でその操作を実行するのにふさわしいタイミングは CUDA ランタイムによって判断される．ストリームを使用することで，ホストとデバイス間のデータ転送や，カーネル実行などの GPU での処理を並列して行うことができる．また，ストリームを使用しパイプライン処理を行うことで，データ転送とカーネル実行をオーバーラップさせ，データ転送時間を隠蔽することができ，実行効率の向上につながる．パイプライン処理の様子を図 1 に示す．

ストリームの作成は cudaStreamCreate() 関数，ストリームの削除は cudaStreamDestroy() 関数を使用して行うことができる．CUDA Runtime API にはデータ転送を非同期に行う cudaMemcypAsync() 関数がある．この関数は引数にデータの入出力先，データサイズ，データ転送の方向，ストリームを取る．また，データ転送を非同期で実行する際には，ページング不可ホストメモリ（ピンメモリ）を使用しなければならない．ピンメモリを使用せずにデータ転送を非同期で実行した場合，CUDA ランタイムが転送している最中の配列をオペレーティングシステムが物理的に移動してしまい，正常に動作しない恐れがある．ピンメモリを確保するには cudaMallocHost() 関数を使用する．また，CUDA を使用した並列プログラミングのためのライブラリである Thrust では，C++の std::vector でピンメモリを確保するためのクラスとして pinned_allocator が提供されている．結果を取得するには，CUDA API を使って非同期操作が完了していることを確認しなければならない．cudaStreamSynchronize() 関数は引数にストリームを取り，指定されたストリームの操作が全て完了するまで，ホストを強制的にブロックする．cudaStreamQuery() 関数は指定されたストリームの操作が全て完了しているか確認する．

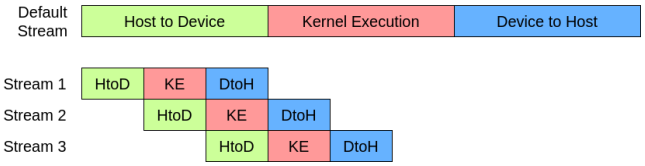

図 1　CUDA ストリームを使用したパイプライン処理

2.3　データ並列プリミティブ

データ並列プリミティブとは，transform や reduce，scan [10] [9]，sort [8] といったアルゴリズムにおける基本的で汎用的な構成要素を効率よく並列処理できるようにしたものである．GPU 上でのデータ並列プリミティブを実装したライブラリとして，ModernGPU [4] や CUDPP [1] などが知られている．これらではデータ並列プリミティブは最適化されており，高速に処理できるようになっている．本研究で使用または作成したデータ並列プリミティブについて以下に述べる．

transform

このデータ並列プリミティブは要素数 n の配列 $[a_0, a_1, \cdots, a_{n-1}]$ と，各要素を変換する関数 f を受け取り，f を適用した配列 $[f(a_0), f(a_1), \cdots, f(a_{n-1})]$ を出力する．

reduce

このデータ並列プリミティブは要素数 n の配列 $[a_0, a_1, \ldots, a_{n-1}]$ と，結合則を満たす二項演算子 $\oplus$ を受け取り，$a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$ を出力する．

scan

このデータ並列プリミティブは要素数 n の配列 $[a_0, a_1, \ldots, a_{n-1}]$ と，結合則を満たす二項演算子 $\oplus$ を受け取る．scan には inclusive scan と exclusive scan の 2 つが存在する．inclusive scan は配列の先頭から現在の値までを reduce プリミティブを使って計算した結果の配列 $[a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})]$ を出力する．exclusive scan では現在の値を含まない配列を出力する．

scatter

scatter プリミティブの動作を図 2 に示す．このデータ並列プリミティブは要素数 n の配列 A_{in} と，配列の並べ方を指定する要素数 n の配列 B を受け取り，配列 A_{in} を並べ替えた配列 A_{out} を出力する．このとき，$A_{out}[B[i]] = A_{in}[i], i = 0, \cdots, n-1$ となる．

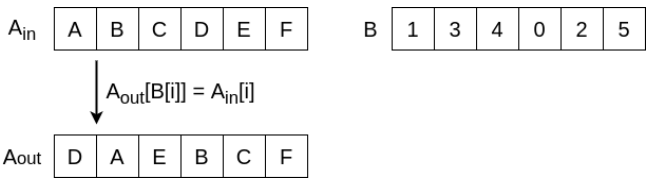

図 2　scatter プリミティブの動作例

sortedserach

このデータ並列プリミティブはソート済みの要素数 n の配列 A と，ソート済みの要素数 m の配列 B を受け取り，A の各要素について B を分割する境界を二分探索によって求めた値を格納した要素数 n の配列 C を出力する．図 3 に指定された要素以上の値が現れる最初の位置で分割する例を示す．例えば A の 3 番目の要素 $A[3] = 8$ について，$B[3] = 7 < A[3] = 8 \leq B[4] = 11$ であるので，$C[3] = 4$ となる．

mergesort

このデータ並列プリミティブは要素数 n の配列 A と，A の要素を比較する関数 f を受け取り，マージソートを行う．

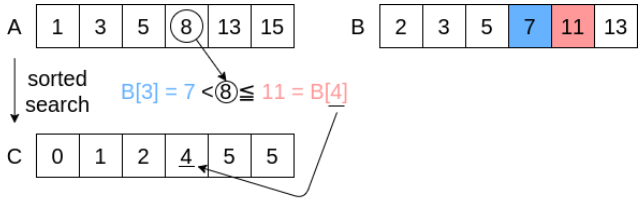

図 3　sortedsearch プリミティブの動作例

3　提 案 手 法

本節では，今回作成した GPU メモリサイズよりも大きなデータを対象としたデータ並列プリミティブについて，その具体的な実装方法について述べる．今回作成したデータ並列プリミティブは transform，reduce，scan，scatter，sort の 5 個である．

3.1　transform

transform プリミティブでは，入力配列を分割し，各分割を順にストリームに割り当て並列に ModernGPU の transform プリミティブを適用する．ここで，使用するストリームの個数を N，使用できる GPU メモリサイズを M とする．また，入力配列の型を T とする．このとき，各ストリームが使用できる GPU メモリサイズは $M' = \frac{M}{N}$ であり，配列を $D = \frac{M'}{\mathrm{sizeof}(T)}$ 個ずつ分割する．また，i 番目の分割は $i\%N$ 番目のストリームに割り当てる．$D = 4$ の場合の transform プリミティブの動作を図 4 に示す．

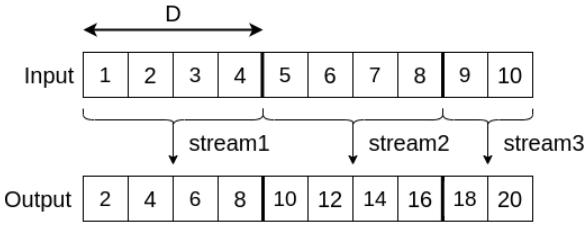

図 4　transform プリミティブの動作例

3.2　reduce

reduce プリミティブでは，transform プリミティブと同様に入力配列を分割し，各分割をストリームに割り当て並列に ModernGPU の reduce プリミティブを適用し，分割ごとの結果を求める．この結果を別の配列に格納し，再帰的に reduce プリミティブを適用することで，最初の入力配列全体の結果を求めることができる．ここで，入力配列の長さを L とし，D 個ずつ分割するとすると，reduce は $\lceil \log_D L \rceil + 1$ 回呼び出される．reduce プリミティブの動作例を図 5 に示す．

3.3　scan

今回，2 つの scan プリミティブの実装を作成した．1 つは比較的小さな配列を逐次的に処理を行うもので，もう 1 つは大きな配列を並列に処理を行うものである．後者の処理手順の途中

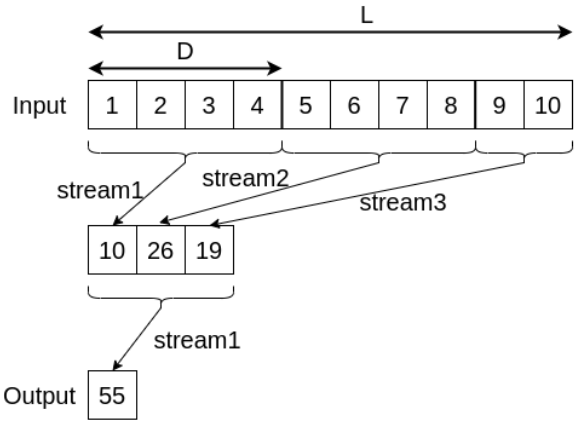

図 5　reduce プリミティブの動作例

に出てくる小さな配列に対し前者を使用する．

1 つ目の scan プリミティブの実装について述べる．transform プリミティブと同様に入力配列を分割するが，最初の分割から逐次的に MordernGPU の scan プリミティブを適用する．ModernGPU の scan プリミティブは scan の結果と同時に reduce の結果を取得できるため，これら 2 つを記録しておく．最初の分割以外では前の分割の reduce の結果を含めて scan を行うことにより，正しい結果を求めることができる．

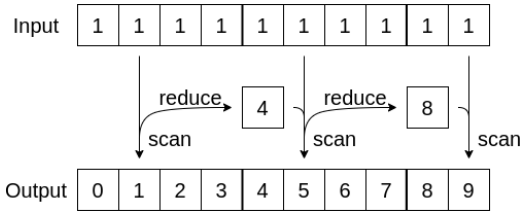

図 6　小さな配列のための scan プリミティブの動作例

2 つ目の scan プリミティブの実装について述べる．今回の実装では 3 ステップから構成されている．動作の様子を図 7 に示す．

Step(1)．transfrom プリミティブと同様に入力配列を分割し，各分割をストリームに割り当て並列に ModernGPU の scan プリミティブを適用し，scan の結果を出力配列に記録し，reduce の結果を配列 R に記録する．

Step(2)．配列 R に対し，1 つ目の scan プリミティブを適用する．

Step(3)．出力配列を分割し，最初以外の分割に対して ModernGPU の transform プリミティブを適用する．具体的には，分割幅を D とし，出力配列の $k(1 \leqq k < D)$ 番目の分割 $[a_{kD}, a_{kD+1}, \cdots, a_{kD+D-1}]$ に対し，$[a_{kD} \oplus R[k], a_{kD+1} \oplus R[k], \cdots, a_{kD+D-1} \oplus R[k]]$ とする．図 7 では出力配列の赤 (青) 色に塗った分割に対し，R の赤 (青) 色に塗った値を足している．

3.4　scatter

scatter プリミティブの動作例を図 8 に示す．scatter プリミティブでは入力配列を出力配列へコピーする際，デバイスメモリ上の中間配列 C を使用する．入力配列 A_{in} の長さを L，C

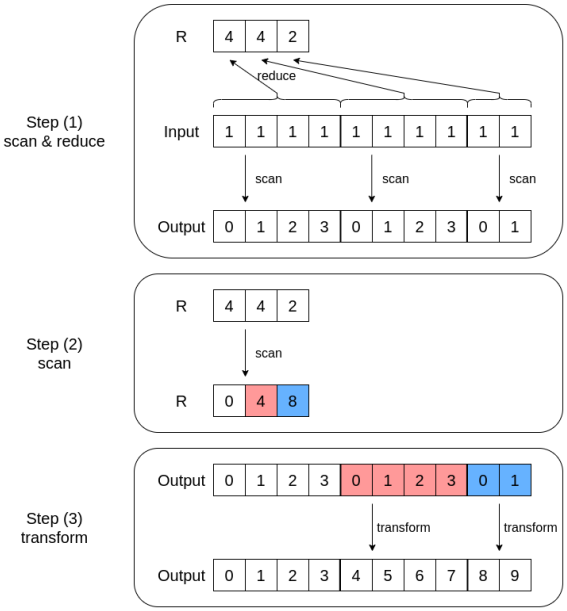

図 7　大きな配列のための scan プリミティブの動作例

の長さを L_C とし，入力配列を D 個ずつ分割するすると，以下に述べるコピーを $\lceil \frac{L}{L_C} \rceil$ 回繰り返す．i 回目のコピーでは，各分割について，$L_C * i \leqq B[idx] < L_C * (i+1)$ をみたす idx について，A_{in} から C へコピーしたのち，C を $A_{out}[L_C * i]$ から $A_{out}[L_C * (i+1) - 1]$ にコピーする．図 8 では 1 回目は赤いマスのコピーを行い，2 回目は青いマスのコピーを行なっている．この操作を擬似コードで表すと Algorithm1 のようになる．

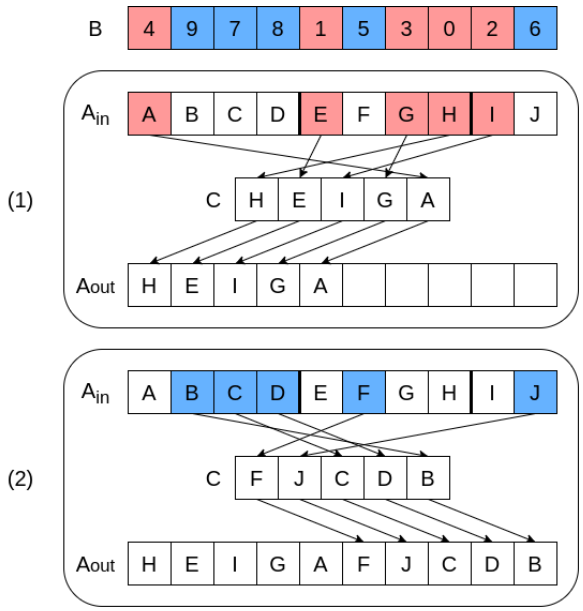

図 8　scatter プリミティブの動作例

3.5　sort

sort プリミティブの動作例を図 9 に示す．sort プリミティブは 8 ステップから構成されている．各ステップについて説明する．

Step(1)．　入力配列の長さを L とする．L に ModernGPU の mergesort が適用できる場合，そのまま適用して終了する．そうでないとき，Step(2). へ進む．

Algorithm 1 scatter プリミティブの疑似コード

```
1: for i ← 0 to ⌈L/L_C⌉ do
2:   // A_in → C
3:   for j ← 0 to ⌈L/D⌉ do in parallel
4:     for k ← 0 to D do in parallel
5:       idx ← D * j + k;
6:       //インデックスが範囲内かチェック
7:       if idx < L then
8:         if L_C * i ≦ B[idx] < L_C * (i + 1) then
9:           C[B[idx] − L_C * i] = A_in[idx];
10:        end if
11:      end if
12:    end for
13:  end for
14:  // C → A_in
15:  for j ← 0 to ⌈L/D⌉ do in parallel
16:    for k ← 0 to D do in parallel
17:      idx ← D * j + k;
18:      //インデックスが範囲内かチェック
19:      if idx < L then
20:        A_out[idx] = C[k];
21:      end if
22:    end for
23:  end for
24: end for
```

Step(2)．　入力配列に格納されている値を P 個に分割する値を決定する．図 9 では 3 個に分割すると仮定し Pivots $= [4, 8, 12]$ としている．このとき，入力配列の赤 (緑，青) のマスは Pivots の赤 (緑，青) のマス未満であることを示している．

Step(3)．　入力配列を D 個ずつに分割し，各分割と Step(2). で求めた Pivots を入力とした ModernGPU の sortedsearch をストリームを使用して並列に適用する．結果を配列 Index に記録する．

Step(4)．　長さ $L \cdot P$ の配列 Intermidiate を用意し，0 で初期化する．配列 Index の i 番目の値 $v =$ Index$[i]$ について，配列 Intermidiate の $v \cdot L + i$ 番目の値を 1 とする．

Step(5)．　Step(4). で求めた配列 Intermidiate に対し，inclusive scan を行う．この際，$L, 2L, 3L, \cdots$ 番目の値 (図 9 の赤枠で囲んだ値) はそれぞれ Pivots[0], Pivots[1], Pivots[2], $\cdots$ 未満の値の個数を表しているため，配列 Count に記録する．

Step(6)．　transform プリミティブを用い，配列 Index の i 番目の値 $v =$ Index$[i]$ を配列 Intermidiate の $v \cdot L + i$ 番目の値から 1 を引いたもので置き換える．すなわち，Index$[i] =$ Intermidiate[Index$[i] * L + i] - 1$ とする．

Step(7)．　入力配列 Input と Step(6). で求めた配列 Index を用い，scatter プリミティブを適用する．今回，GPU を使用した scatter を使用せず，CPU による scatter を行なった．

Step(8)．　入力配列について，半開区間 $[0, Count[0]), [Count[0], Count[1]), [Count[1], Count[2]), \cdots$

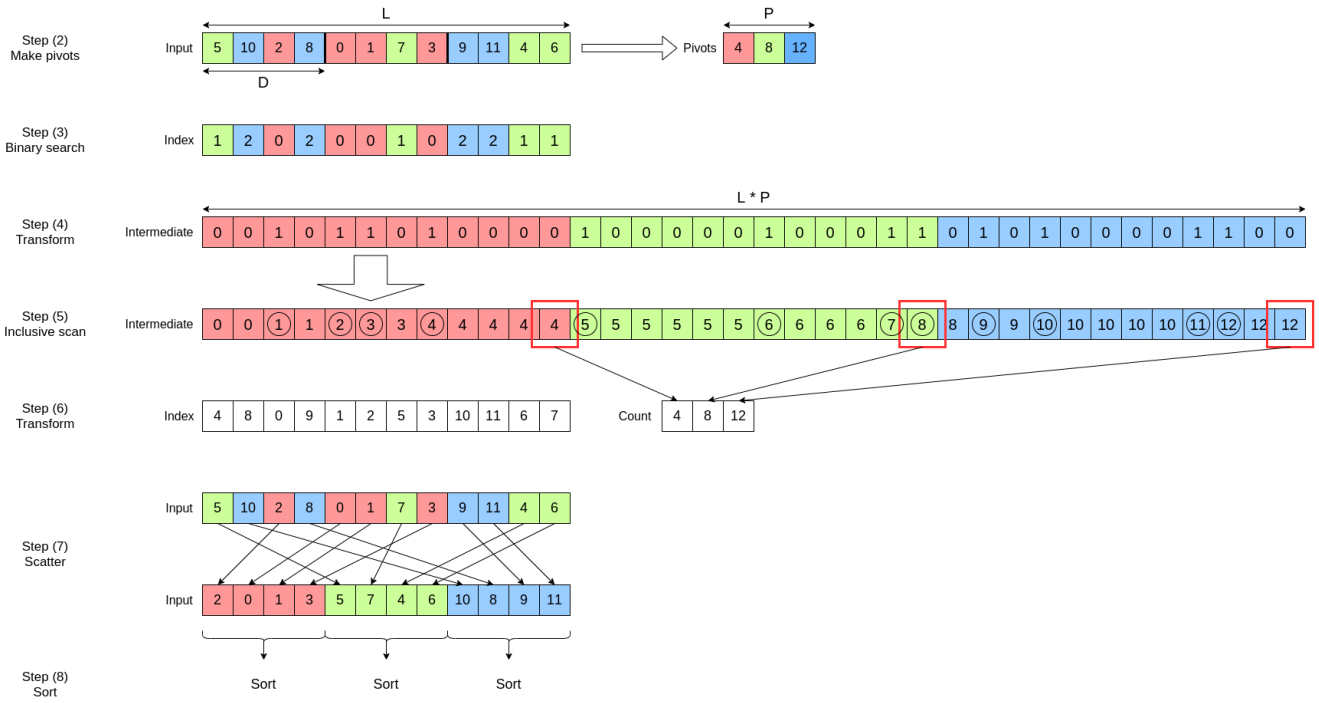

図 9　sort プリミティブの動作例

について再帰的にアルゴリズムを適用する．

4　評価実験

本節では，提案手法と CPU 上で実装した同様の動作をする比較手法を比較し評価を行う．

4.1　比較手法

比較手法として用いる CPU 上で同様の動作をするプログラムの実装方法について述べる．比較手法の実装には C++を用いた．関数のソースコードを Listing1 に示す．

4.2　実験環境

評価実験に用いたマシンの構成を表 1 に示す．また，デバイスとホスト間でのデータの転送速度を CUDA Bandwidth Test を使って測定した．測定結果を表 2 に示す．

表 1　実験に用いたマシンの構成

OS		Ubuntu 18.04 LTS
CPU		Intel Core i7-7800X (3.5GHz, 6 コア)
GPU		NVIDIA GTX 1060 (Pascal)
	CUDA コア	1152
	ベースクロック	1506MHz
	メモリ量	3GB GDDR5
RAM		128GB
CUDA		CUDA 11.5

表 2　バンド幅のベンチマーク結果

転送方向	Bandwidth(GB/s)
Host to Device	12.4
Device to Host	13.2

4.3　実験内容

本実験では，比較手法，提案手法ともに実行時間の測定とホストメモリへのアクセスの詳細の測定を行い，提案手法についてプロファイルの取得を行った．メモリアクセスの詳細の測定には Intel VTune Amplifier を使用し，プロファイルの取得には NVIDIA Visual Profiler を使用し，実行時間の計測には C++の std::chrono 名前空間で定義されている関数やクラスを使用した．また，提案手法の実行時間計測ではホストメモリからデバイスメモリへ入力を転送する前から，デバイスメモリからホストメモリへ結果を転送し終えるまでを計測した．したがって，ホストとデバイス間のデータ転送時間も実行時間に含まれる．

transform プリミティブでは与えられた配列の要素に定数を掛けるタスクと，$f(x) = \sin^2 x + \cos^2 x$ を適用するタスクを行なった．reduce プリミティブでは与えられた配列の総和を求めるタスクを行なった．scan プリミティブでは与えられた配列の累積和を求めるタスクを行なった．scatter プリミティブでは $[1, 2, \cdots]$ をシャッフルした配列を並べ方を指定する配列とし，入力配列をランダムにシャッフルするタスクを行なった．sort プリミティブでは一様分布に従う乱数で初期化した配列を昇順にソートするタスクを行なった．

transform，redce，scan，sort プリミティブの入力には 64bit 浮動小数点数型の配列を用いた．また，scatter プリミティブの入力には 64bit 浮動小数点数型の配列，並べ方の指定には 64bit 整数型の配列を用いた．transform，reduce，scan プリミティブでは入力配列の長さを 0 から 10^{10}(約 74.5GB) まで変化させた．scatter プリミティブでは入力配列と，並べ方を指定する配列の長さを 0 から 5×10^9(それぞれ約 37.3GB) まで変化させた．sort プリミティブでは入力配列の長さを 0 から 10^9(約

Listing 1　比較手法のソースコード

```
template <typename T> using vec = std::vector<
    T,
    thrust::system::cuda::experimental::
        pinned_allocator<T>
>;
template<typename T, typename F>
void transform(vec<T> &input, vec<T> &output, F f)
{
    for (size_t i = 0; i < input.size(); ++i)
        output[i] = f(input[i]);
}
template<typename T, typename OP>
void reduce(vec<T> &input, T &output, OP op) {
    output = T();
    for (auto v: input) output = op(output, v);
}
template<scan_type_t scan_type, typename T,
     typename OP>
void scan(vec<T> &input, vec<T> &output, OP op) {
    output[0] = scan_type == scan_type_inclusive ?
        input[0] : T();
    for (size_t i = 1; i < input.size(); ++i)
        output[i] = output[i - 1] + input[i];
}
template<typename T>
void scatter(vec<T> &input, vec<T> &index, vec<T> &
     output) {
     for (size_t i = 0; i < input.size(); ++i)
        output[index[i]] = input[i];
}
template<typename T>
void sort(vec<T> &input) {
    std::sort(input.begin(), input.end());
}
```

7.45GB) まで変化させた．

4.4　実験結果

提案手法と比較手法の実行時間の測定結果の比較を図 10 から図 15 に示す．図の黒い縦線は，GPU メモリサイズである 3GB に相当するデータ要素数の位置を示している．transform プリミティブでは，定数倍するタスクでは比較手法に比べて 0.89–1.02 倍の性能，三角関数のタスクでは 28.6–35.2 倍の性能，reduce プリミティブでは比較手法に比べて 3.24–3.34 倍の性能，scan プリミティブでは比較手法に比べて 1.41–1.42 倍の性能，scatter プリミティブでは比較手法に比べて 0.06–0.84 倍の性能，sort プリミティブでは比較手法に比べて 1.87–3.98 倍の性能となった．

transform プリミティブのプロファイルを図 16，図 17 に示す．定数倍のような時間計算量の小さなタスクではデータの転送とカーネルの実行がオーバーラップしている部分 (図 16 で赤枠で囲んだ部分) の割合が全体の実行時間に比べ小さいことがわかる．また，データの転送はホストからデバイス，デバイスからホストの両方で切れ間なく行われている．三角関数の計算のような時間計算量の大きなタスクでは，カーネルの実行時間が増加し，オーバーラップしている割合が増加している．また，カーネル実行時間がメモリ転送時間よりも長いため，データ転送に切れ目が生じている．また，reduce プリミティブのプロファイルを図 18，scan プリミティブの Step(1) のプロファイルに示す．これらプリミティブではカーネルの実行時間は短

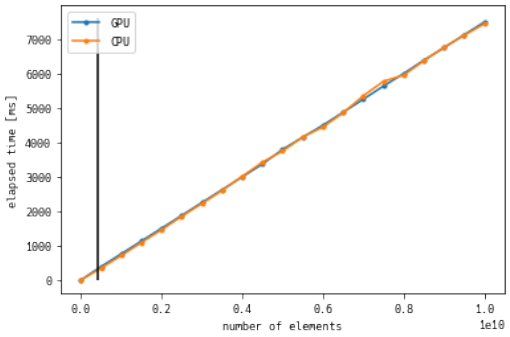

図 10　transform プリミティブの実行結果 (定数倍するタスク)

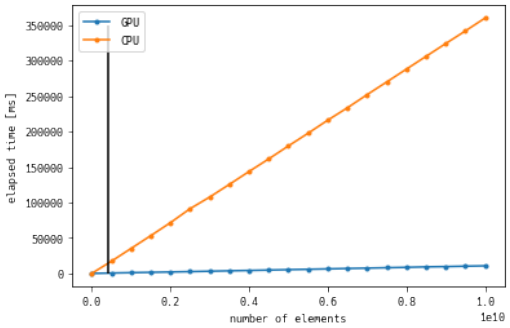

図 11　transform プリミティブの実行結果 (三角関数のタスク)

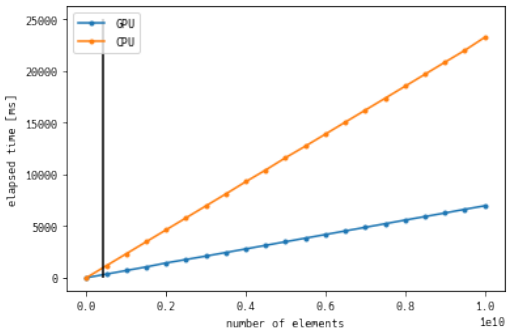

図 12　reduce プリミティブの実行結果

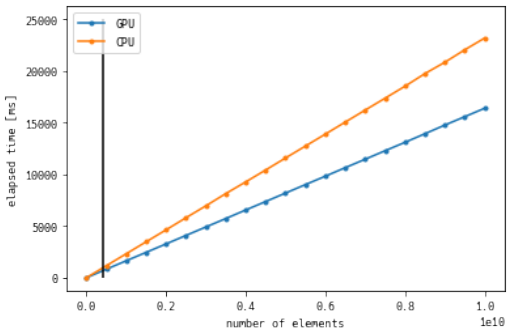

図 13　scan プリミティブの実行結果

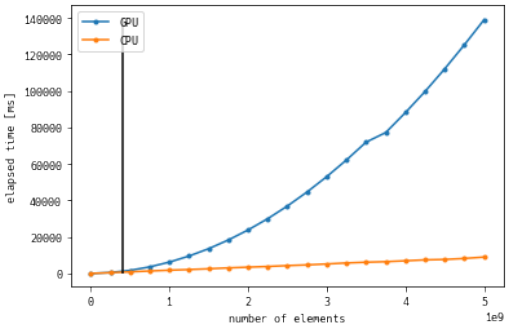

図 14　scatter プリミティブの実行結果

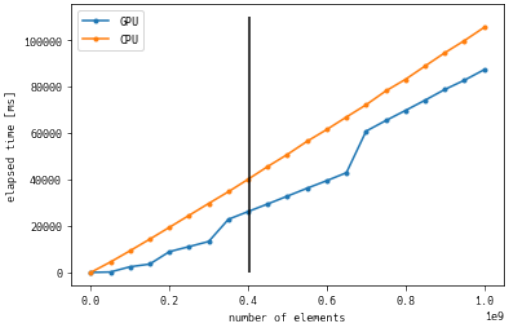

図 15　sort プリミティブの実行結果

いが，データの転送とカーネルの実行がオーバーラップできていない．

図 16　transform プリミティブのプロファイル (定数倍するタスク)

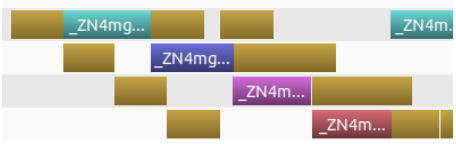

図 17　transform プリミティブのプロファイル (三角関数のタスク)

図 18　reduce プリミティブのプロファイル

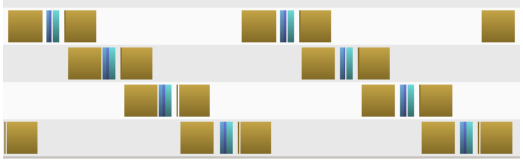

図 19　scan プリミティブのプロファイル

DTLB の測定結果を表 3 に示す．DTLB は，TLB ミスにより費やされる CPU クロックの割合を表している．また，TLB(Translation Lookaside Buffer) とは，仮想アドレスと物理アドレスの対応を格納するためのキャッシュであり，CPU がメモリアクセスを行う際，TLB 上に仮想アドレスが存在しない場合 TLB ミスとなり，メモリアクセスに時間を費やすことになる．したがって値が小さいほど効率的にメモリアクセスが行えていることを表す．

表 3　DTLB の測定結果

プリミティブ	DTLB	
	比較手法	提案手法
transform(定数倍)	3.90%	16.20%
transform(三角関数)	0.40%	23.80%
reduce	0.10%	18.40%
scan	0.10%	33.90%
scatter	0.20%	54.90%
sort	0.30%	51.80%

DRAM のメモリバンド幅の測定結果を表 4 と表 5 に示す．transform(定数倍) において，メモリバンド幅がどちらもほぼ同じ値となっていることがわかる．その他のプリミティブでは比較手法に比べ提案手法はより速いメモリバンド幅となっていることがわかる．

表 4　比較手法のメモリバンド幅の測定結果

プリミティブ	Bandwidth(GB/s)	
	Read	Write
transform(定数倍)	10.438	10.378
transform(三角関数)	0.248	0.224
reduce	3.512	0.011
scan	3.580	3.492
scatter	13.616	4.529
sort	2.091	2.048

表 5　提案手法のメモリバンド幅の測定結果

プリミティブ		Bandwidth(GB/s)	
		Read	Write
transform(定数倍)		11.967	10.761
transform(三角関数)		9.540	6.677
reduce		11.505	0.002
scan	step(1)	9.026	8.856
	step(3)	11.982	10.769
scatter	$A_{in} \rightarrow C$	12.412	0.011
	$C \rightarrow A_{out}$	0.577	13.900
sort	step(3) (ソート)	1.128	0.009
	step(3) (二分探索)	2.044	4.062
	step(4)	11.877	10.702
	step(5) (step(1))	9.131	9.045
	step(5) (step(3))	11.839	10.649
	step(6)	8.789	2.963
	step(7)	14.842	4.940

4.5 考　　察

transform プリミティブでは，カーネルの実行とデータ転送がオーバーラップしており，定数倍を行うタスクでは提案手法と比較手法のメモリアクセスの速度がほぼ同一であったため，メモリアクセスがボトルネックとなり性能が向上しなかったと考えられる．また，三角関数を行うタスクでは計算時間が長いためデータ転送を隠蔽することができることに加えメモリバンド幅が速いことで性能が向上したと考えられる．reduce プリミティブではオーバーラップによる高速化はできていないが，メモリアクセスが高速に行えたことで性能が向上していると考えられる．scan プリミティブでは，reduce プリミティブと同様にメモリアクセスが高速に行えているが，step(1) と step(3) の 2 回長い配列に対して処理を行う必要があるため，reduce ほど性能が向上していないと考えられる．scatter プリミティブでは一度のループで中間配列の長さのデータしか移動できないため，計算量が増加している．そのため並列処理による高速化よりも計算量の増加の影響が大きく，提案手法より処理速度が低下したと考えられる．さらに，DTLB の結果から，効率的なメモリアクセスが行えていないことも性能が向上しない原因となっている．sort では中間配列の中から値を取り出すなどの操作があり，DTLB の値が大きくなっていることからメモリアクセスの効率化はできていないが，メモリバンド幅の測定結果からメモリに対するデータの読み書きは高速にできているため，比較手法に比べ性能が向上していると考えられる．

5 おわりに

本研究では GPU メモリサイズよりも大きなデータを対象とした transform，reduce，scan，scatter といった基本的なデータ並列プリミティブと，それらを組み合わせて作成できる sort プリミティブについての実装方法を提案した．

評価実験では，CPU 上で同じ動作をする比較手法を実装し，実際に GPU メモリサイズよりも大きな入力データを与え処理する時間を計測し比較した．評価実験の結果から，scatter プリミティブ以外のデータ並列プリミティブに関して比較手法よりも 0.89–35.2 倍の性能向上をすることができた．一方今回の scatter プリミティブの実装ではメモリの転送回数の増加が並列処理による高速化を上回ることとメモリアクセスが効率的に行えていないことにより比較手法よりも性能が低下することが判明した．

今後の課題として次のようなものが挙げられる．まず，現在の実装方法の最適化を行い，より高速に動作するように改良することが求められる．また，segmented sort や hash table [3] などまだ実装を行なっていないデータ並列プリミティブや，実際のアプリケーションを実装する際に必要となるプリミティブに対する実装方法を開発し，より多くのアプリケーションに対応する必要がある．さらに，複数のプリミティブを組み合わせた際に高速化する方法について考える必要がある．

謝　　辞

本研究の一部は，JSPS 科研費 (18H03242，18H03342，19H01138) の助成を受けたものである．

文　　献

[1] CUDPP. `https://cudpp.github.io/`.
[2] OpenCL. `https://www.khronos.org/api/opencl`.
[3] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*, pp. 39–53. Elsevier, 2012.
[4] Sean Baxter. moderngpu 2.0. `https://github.com/moderngpu/moderngpu/wiki`, 2016.
[5] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, Vol. 28, No. 4, pp. 13–27, 2008.
[6] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, Vol. 34, No. 4, dec 2009.
[7] 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*, SIGMOD '08, p. 511–524, New York, NY, USA, 2008. Association for Computing Machinery.
[8] 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.
[9] Shubhabrata Sengupta, Mark J Harris, Michael Garland, and John D Owens. *Efficient parallel scan algorithms for many-core gpus*. eScholarship, University of California, 2011.
[10] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In Mark Segal and Timo Aila, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. The Eurographics Association, 2007.