Vol.2019-ARC-237 No.17
2019/7/25

# Unified Memoryを伴うGPUシステムにおける階層統合型粗粒度タスク並列処理

渡辺 智之[1]　吉田 明正[1,2,a)]

**概要**：Unified Memory を伴う GPU システムが近年普及している．このようなシステムでは，従来の GPU システムで必要としていたホスト・デバイス間のデータ転送コードを，Unified Memory の利用により削減することができる．本手法では，GPU とマルチコアを計算資源として利用し，階層統合型実行制御による粗粒度タスク並列処理を実現する．GPU とマルチコアの共有データは Unified Memory に配置されており，粗粒度タスクは実行時にコアに割り当てられた際，ユーザ指定を伴って自コアあるいは GPU により実行される．性能評価では，ARMv8.2 CPU と Volta GPU を搭載した NVIDA Jetson AGX Xavier 上で，ヤコビ法，及び粒子法アプリケーションプログラムに提案手法を適用し，Xavier 上での性能評価の結果から提案手法の有効性を確認した．

## 1. はじめに

マルチコアシステムにおける並列処理手法として，ループ並列性に加えて，粗粒度タスク並列性 [1][2] を最大限に利用する階層統合型粗粒度並列手法 [1][3][4] が提案されている．階層統合型粗粒度並列処理の並列コードは，現在までに Java Thread や Java Fork/Join Framework による実装 [3][4] が提案されている．また，近年 GPU を伴うマルチコアシステムの普及に伴い，CPU と GPU からなるヘテロジニアスな並列システムにおいてアプリケーションを高速化する研究が進められている [5][6][7][8].

本研究では，OpenMP による独自スケジューラ [9] を実装をすることで階層統合型実行制御により粗粒度タスク間の並列性を利用しつつ，処理時間の大きい粗粒度タスクを GPU に割り当てる実行手法が提案されている．本稿では，GPU とマルチコアを計算資源として利用する際に，従来の GPU システムで必要としていたホスト・デバイス間のデータ転送コードを，Unified Memory の利用により削減しつつ，階層統合型実行制御による粗粒度タスク並列処理を実現する．GPU とマルチコアの共有データは Unified Memory に配置されており，粗粒度タスクは各コアで実行されているスケジューラによってユーザ指定を伴って自コアあるいは GPU により実行される．性能評価では，ARMv8.2 CPU と Volta GPU を搭載した NVIDA Jetson AGX Xavier 上で，ヤコビ法，及び粒子法アプリケーションプログラムに提案手法を適用し，Xavier 上での性能評価を行う．

本稿の構成は以下のとおりとする．2 章では，階層統合型粗粒度並列処理について述べる．3 章では，Unified Memory を考慮した階層統合型粗粒度並列処理コードについて述べる．4 章では，Unified Memory を考慮した階層統合型粗粒度並列処理の性能評価を述べる．5 章では，まとめを述べる．

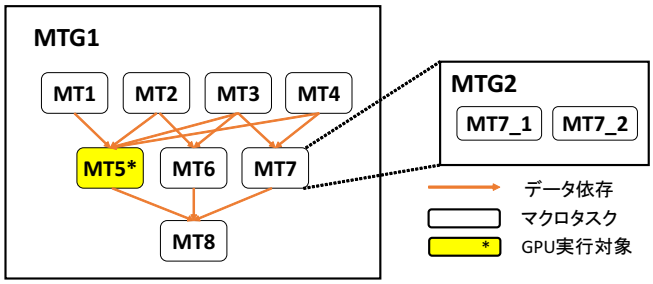


**図 1**　階層型マクロタスクグラフ（MTG）.

## 2. 階層統合型粗粒度並列処理

本手法の基盤として用いる階層統合型粗粒度並列処理について述べる．

### 2.1 階層統合粗粒度並列処理の概要

階層統合型粗粒度並列処理では，階層型マクロタスクグラフ（MTG）[1] を生成し，マクロタスク（MT）を階層的に定義する．その後，最早実行可能条件 [1] を満たした全階層のマクロタスクを，ダイナミックスケジューラが統一的にコアに割り当てて実行する．例えば，図 1 のような階層型マクロタスクグラフで表されるプログラムを 4 コア＋1GPU で Unified Memory を用いずに実行したイメージは図 2 のようになる．この場合，複数階層のマクロタスク間の並列性が最大限に利用されていることが分かる．また，本稿で提案する Unified Memory を用いた実行では，図 3 に

1　明治大学大学院先端数理科学研究科
2　明治大学総合数理学部
a)　akimasay@meiji.ac.jp

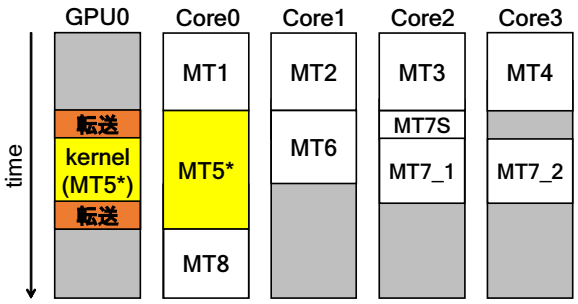


**図 2** 4 コア＋ 1GPU 上での階層統合型粗粒度並列処理の実行イメージ.

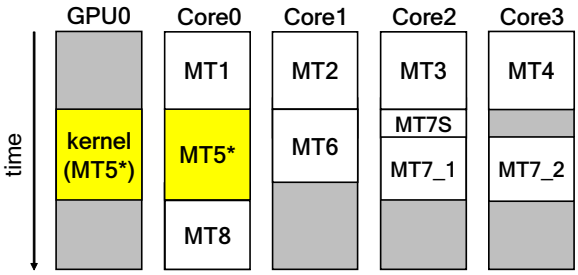


**図 3** Unified Memory を伴う 4 コア＋ 1GPU 上での階層統合型粗粒度並列処理の実行イメージ.

示すように GPU0 において，ホスト・デバイス間のデータ転送が除去されており，実行時間が短縮されている．ここで，図 1 の MT8 の最早実行可能条件は MT5∧MT6∧MT7 と求めることができる．これは，MT5 と MT6 と MT7 の実行が終了した後に，MT8 の実行が可能になるということを表している．表 1 に示す各マクロタスクの最早実行可能条件は，図 1 の階層型マクロタスクグラフに対応している．

### 2.2 マクロタスクの階層的定義

粗粒度タスク並列処理では，まず，与えられたプログラム（全体を第 0 階層マクロタスクとする）を第 1 階層マクロタスク（MT）に分割する．マクロタスクは，基本ブロック，繰り返しブロック（for 文等のループ），サブルーチンブロック（メソッド呼び出し）の 3 種類から構成される [1]．次に，第 1 階層マクロタスク内部に複数のサブマクロタスクを含んでいる場合は，それらのサブマクロタスクを第 2 階層マクロタスクとして定義する．同様に，第 $L$ 階層マクロタスク内部において，第 $(L+1)$ 階層マクロタスクを定義する．

階層統合型実行制御 [1] を適用する場合，全階層のマクロタスクを統一的に取り扱うため，階層開始マクロタスクを導入する．第 $L$ 階層マクロタスクをサブマクロタスクとして内部に持つ上位の第 $(L-1)$ 階層マクロタスクを，第 $L$ 階層用の階層開始マクロタスクとして取り扱う．この階層開始マクロタスクは，内部の第 $L$ 階層マクロタスクの実行を開始するために使用される．階層開始マクロタスクの導入により，当該階層のマクロタスクの実行が可能になったことが保証され，全階層のマクロタスクを同時に取り扱うことが可能となる．

本手法では，処理時間の大きいマクロタスクを事前に GPU 実行対象として決定しておき，それらのマクロタスクを GPU 上で実行するための GPU（CUDA）コードを用意しておく．

**表 1** 階層統合型実行制御の最早実行可能条件.

| MTG 番号 | MT 番号 | 最早実行可能条件 | 終了通知 |
|---|---|---|---|
| 1 | 1 | true | 1 |
| | 2 | true | 2 |
| | 3 | true | 3 |
| | 4 | true | 4 |
| | 5 | 1∧2∧3∧4 | 5 |
| | 6 | 2∧3 | 6 |
| | 7† | 3∧4 | 7S |
| | 8 | 5∧6∧7 | 8 |
| | 9(EndMT) | 8 | 9 |
| 2 | 7_1 | 7S | 7_1 |
| | 7_2 | 7S | 7_2 |
| | 7_3(ExitMT) | 7_1∧7_2 | 7_3, 上位 MT(7) |

†：メソッド内部の第 2 階層 MTG の階層開始 MT

### 2.3 GPU と CPU を対象としたマクロタスクスケジューリング

マクロタスクの生成後，各階層におけるマクロタスク間の制御依存とデータ依存を解析し，階層型マクロタスクグラフ [1] を生成する．その後，マクロタスク間の並列性を最大限に引き出すため，各マクロタスクの最早実行可能条件 [1] を解析する．これらは，図 1 のような階層型マクロタスクグラフとして表現できる．

表 1 のような最早実行可能条件は，マクロタスクの実行制御に用いられる．ここで，MT7 は図 1 に示すように MT7_1 と MT7_2 で構成されたメソッドの呼び出しに対応する．それゆえ，MT7 は階層開始マクロタスクとして動作しており，階層開始マクロタスクの処理を終了した時に 7S の終了通知を発行する．一方，MT7 のメソッド呼び出しの終了通知は，メソッド内の MT7_3 が終了通知 7 を発行する．ダイナミックスケジューリングの際には，ステート管理テーブルに保存された各マクロタスクの終了通知，分岐通知，最早実行可能条件を調べることにより，新たに実行可能なマクロタスクを検出することが可能となる [1]．階層統合型実行制御によるマクロタスクスケジューリングでは，各マクロタスクは最早実行可能条件を満たした後，レディキューに投入される．その後，レディキューから順に取り出されてコアに割り当てられ実行される．

階層統合型粗粒度並列処理を GPU を伴うマルチコアシステムで実現するためには，マクロタスクスケジューリングにおいて，GPU 実行対象のマクロタスクを管理するレディキューを用意する必要がある．本手法では，CPU 実行対象マクロタスクが投入される CPU キュー，GPU 実行対象マクロタスクが投入される GPU キューを導入する [9]．

ここで，GPU キューは共通 GPU キュー（任意 GPU 割当て）と指定 GPU キュー（指定 GPU 割当て）からなる 2 種類を用意する．以下に任意 GPU 割当て，及び指定 GPU 割当てを伴うマクロタスクスケジューリングの実行手順について述べる．

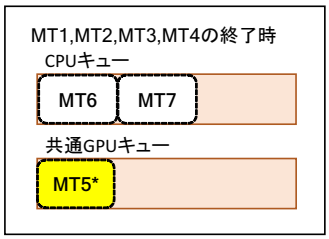


**図 4** 階層統合型粗粒度並列処理のレディキューの例.

**表 2** Unified Memory のメモリ確保.

| CUDA API | アクセス速度 | キャッシュ |
|---|---|---|
| cudaHostAlloc() | 遅い | 無効 |
| cudaMallocManaged()（本手法で利用） | 速い | 有効 |

**表 4** 対象プログラムのデータ配置方法.

| メモリ割当て | 配列の特徴 |
|---|---|
| malloc() | CPU のみで利用する場合 |
| cudaMalloc() | GPU のみで利用する場合 |
| cudaMallocManaged() | CPU/GPU で利用する場合 |

#### 2.3.1 任意 GPU 割当てを伴うマクロタスクスケジューリング

提案手法では GPU の利用率を高めるために, CPU キューと共通 GPU キューの 2 種類を使用した任意 GPU 割当てを伴うマクロタスクスケジューリングを採用している．例えば，図 1 のマクロタスクグラフを 4 コア＋ 1GPU のシステムで実行する場合の MT5 を取り上げる．MT1，MT2，MT3，MT4 の実行が終了した段階で，レディキューは図 4 のようになっている．この時，図 2 の Core0 で動作しているスケジューラは，MT5 を共通 GPU キューから取り出してその GPU（CUDA）コードを GPU0 で実行する．但し，Core0 は GPU0 の終了まで待機している．その後，Core1，Core2 は CPU キューから MT6，MT7 をそれぞれ取り出して，各コアで実行する．このように 2 つのレディキュー（CPU 用，GPU 用）にマクロタスクが投入されている場合，GPU キュー即ち共通 GPU キューを優先する．

#### 2.3.2 指定 GPU 割当てを伴うマクロタスクスケジューリング

同様に，図 1 のマクロタスクグラフを 4 コア＋ 1GPU のシステムで実行する場合を取り上げる．MT1，MT2，MT3，MT4 の実行が終了した段階で，MT5 は指定された GPU 番号の指定 GPU キューに投入される．次に CPU キューに MT6，MT7 が投入される．その後，Core0 のスケジューラが，MT5 を指定 GPU キューから取り出して GPU0 で実行する．同様に Core1，Core2 のスケジューラは MT6，MT7 を CPU キューから取り出し，実行する．指定 GPU 割当てを伴うマクロタスクスケジューリングでは，各 GPU のデバイスメモリにデータ保持が可能であり，データの局所性を考慮した実行が可能である．

## 3. Unified Memory を考慮した階層統合型粗粒度並列処理コード

本章では，GPU を伴うマルチコアシステムを対象として Unified Memory を利用した階層統合型粗粒度並列処理コードの構成について述べる．

```
1  #define N 100 /*データサイズ*/
2  #define THREAD 1024 /*GPU のスレッド数*/
3  #define BLOCK N/THREAD /*GPU のブロック数*/
4  スケジューラ用変数の宣言;
5  MT 間共有変数の宣言;
6
7  /*Unified Memory 用変数の宣言*/
8  double *data;
9
10 /*kernel 関数*/
11 __global__ void kernel(double *d_data){
12   GPU コード;
13 }
14 /*マクロタスクのコード（CPU)*/
15 void MT1(){
16   /*CPU で data を扱う*/
17   for(int i=0; i<N; i++)
18     data[i] = ...;
19 }
20 ...
21 /*マクロタスクのコード（GPU)*/
22 void MT5(){
23   int dev = ...;
24   cudaSetDevice(dev); /*デバイス番号を dev に設定*/
25   kernel<<<BLOCK, THREAD>>>(data); /*GPU で実行*/
26   cudaDeviceSynchronize(); /*同期*/
27 }
28 ...
29 /*最早実行可能条件*/
30 void EEC(int mt){
31   mt の最早実行可能条件をチェック;
32 }
33 /*ダイナミックスケジューラ*/
34 void SCHEDULER(int num){
35   while(終了条件を満たすまで){
36     if(GPU がアイドル状態&&GPU キューの投入 MT 数>=1)
37       GPU キューから 1MT を取り出す;
38     else if(CPU キューの投入 MT 数>=1)
39       CPU キューから 1MT を取り出す;
40     if(取り出したMT の属性==GPU){
41       GPU で MT を実行する;
42       MT の終了・分岐通知;
43     }else{
44       CPU で MT を実行する;
45       MT の終了・分岐通知;
46     }
47     EEC()を満たしたMT をキューに投入;
48   }
49 }
50 /*main 関数*/
51 int main(){
52   /*Unified Memory にデータ確保*/
53   cudaMallocManaged((void **)&data, sizeof(double)*N);
54
55   データの初期化;
56   #pragma omp parallel
57   {
58     SCHEDULER(omp_get_thread_num());
59   }
60   return 0;
61 }
```

**図 5** Unified Memory を用いた階層統合型粗粒度並列処理コード.

### 3.1 Unified Memory の概要

Unified Memory は NVIDIA が提供する CUDA[10] においてホストメモリとデバイスメモリを統一的に扱う概念である．表 2 に示すように cudaHostAlloc() と cudaMallocManaged() があるが，本稿ではキャッシュが有効で cudaHostAlloc() よりメモリアクセスの速い cudaMallocManaged() を利用して GPU と CPU の両方で扱うデータ

**表 3** 同時メモリアクセスの制約.

| 並列実行 MT のメモリアクセス | 同時アクセス |
|---|---|
| パターン 1：(CPU)malloc() ＋ (GPU)cudaMalloc() | ○ |
| パターン 2：(CPU)malloc() ＋ (GPU)cudaMallocManaged() | ○ |
| パターン 3：(CPU)cudaMallocManaged() ＋ (CPU)cudaMallocManaged() | ○ |
| パターン 4：(CPU)cudaMallocManaged() ＋ (GPU)cudaMalloc() | × |
| パターン 5：(CPU)cudaMallocManaged() ＋ (GPU)cudaMallocManaged() | × |

```
1  #define N 100 /*データサイズ*/
2  #define THREAD 1024 /*GPU のスレッド数*/
3  #define BLOCK N/THREAD /*GPU のブロック数*/
4  スケジューラ用変数の宣言;
5  MT 間共有変数の宣言;
6
7  /*ホスト用変数の宣言*/
8  double *data;
9  /*デバイス用変数の宣言*/
10 double *d_data;
11 ...
12 /*マクロタスクのコード（GPU)*/
13 void MT5(){
14   int dev = ...;
15   cudaSetDevice(dev); /*デバイス番号を dev に設定*/
16   cudaMemcpy(d_data, data, sizeof(double)*N,
17     cudaMemcpyHostToDevice);
18   kernel<<<BLOCK, THREAD>>>(d_data); /*GPU で実行*/
19   cudaMemcpy(data, d_data, sizeof(double)*N,
20     cudaMemcpyDeviceToHost);
21   cudaDeviceSynchronize(); /*同期*/
22 }
23 ...
24 /*main 関数*/
25 int main(){
26   /*ホストにデータ確保*/
27   data = (double *)malloc(sizeof(double)*N);
28   /*デバイスにデータ確保*/
29   cudaSetDevice(dev)
30   cudaMalloc((void **)&d_data, sizeof(double)*N);
31   データの初期化;
32   #pragma omp parallel
33   {
34     SCHEDULER(omp_get_thread_num());
35   }
36   return 0;
37 }
```

**図 6** Unified Memory を用いない階層統合型粗粒度並列処理コード.

のメモリ確保を行う．本稿の性能評価に使用したプログラムのマクロタスクグラフはこれらの制約を考慮して作成した．cudaMallocManaged() で確保した領域は，CPU と GPU の両方からアクセスすることができる．この機能によりホスト・デバイス間のデータ転送コードを削減することができる．本稿では，CPU と GPU の両方で実行されるデータに cudaMallocManaged() を適用する．

cudaMallocManaged() の特性として，同時メモリアクセスの制約を表 3 に示す．まず，パターン 1 に示されるように cudaMalloc() で確保したデータの GPU アクセスと malloc() で確保したデータの CPU アクセスは同時実行が可能である．次に，パターン 2 では，cudaMallocManaged() で確保したデータの GPU アクセスと malloc() で確保したデータの CPU アクセスは同時実行可能である．パターン 3 では，cudaMallocManaged() で確保したデータの CPU アクセスは同時実行可能である．一方パターン 4 では，cudaMalloc() で確保したデータの GPU アクセスと cudaMallocManaged() で確保したデータの CPU アクセスは同時アクセスができない．最後に，パターン 5 では cudaMallocManaged() で確保したデータの GPU アクセスと cudaMallocManaged() で確保したデータの CPU アクセスは同時実行することはできない．

### 3.2 Unified Memory を用いた粗粒度並列コード

Unified Memory を利用した粗粒度並列コードの構成を図 5 に示す．

#### 3.2.1 データの配置

階層統合型粗粒度並列処理におけるデータ配置方法を表 4 に示す．マクロタスクで扱うデータにおいて CPU のみで利用する配列は malloc() でメモリ割り当てを行い，GPU のみで利用する配列は cudaMalloc() でメモリ割り当てを行う．GPU と CPU の両方で利用する配列は cudaMallocManaged() で割り当てを行う．

#### 3.2.2 OpenMP によるマクロタスクスケジューリングコード

図 5 の main() 関数では，まず，OpenMP の指示文によりコア数分のスレッドを生成し，SCHEDULER() 関数を実行する．この関数ではマクロタスクの最早実行可能条件を EEC() 関数（図 5 の 30 行目から 32 行目）で確認した後に，47 行目で最早実行可能条件を満たすマクロタスクをレディキューへ投入する．ここで，36 行目と 37 行目で GPU 実行対象のマクロタスクを GPU キューから取り出す．また，38 行目と 39 行目で CPU 実行対象のマクロタスクを取り出している．最後に，40 行目から 46 行目でレディキューから取り出したマクロタスクに対応する関数（マクロタスクコード）を実行する．これらの一連の手順によって，終了条件を満たすまでマクロタスクが実行される．

#### 3.2.3 Unified Memory を用いたマクロタスクコード

Unified Memory を使用する場合，図 5 の 53 行目で cudaMallocManaged() によってデータ領域を確保する．cudaMallocManaged() を使用して確保したデータは CPU と GPU の両方からアクセスが可能となる．GPU 実行対象のマクロタスクは 22 行目から 27 行目の MT5() のように記述される．この関数では，24 行目で cudaSetDevice() により GPU デバイスの指定を行い，25 行目で GPU のブロックとスレッド数を指定して 11 行目から 13 行目の kernel 関数を実行する．最後に，26 行目で cudaDeviceSynchronize() で同期を行う．CPU 実行対象のマクロタスクで cudaMallocManaged() で確保したデータを扱う際は 15 行目から 19 行目のように記述される．Unified Memory を用いた場合の実行イメージは図 3 になる．

#### 3.2.4 Unified Memory を用いないマクロタスクコード

従来のように Unified Memory を使用しない場合，図 6

表 5　NVIDA Jetson AGX Xavier の構成.

| | |
|---|---|
| CPU | 8-Core ARM v8.2 64-Bit CPU, 8MB L2 + 4MB L3 |
| メモリ | 16GB |
| GPU | 512-Core Volta GPU with 64 Tensor Cores |
| OS | Ubuntu 18.04 |
| 処理系 | GCC 7.4.0，CUDA Toolkit 10.0 |
| 電源モード | 30W，CPU1200MHz，GPU900MHz |

表 6　NVIDA Jetson AGX Xavier の電源モード.

| Mode ID | power budget | CPU | GPU |
|---|---|---|---|
| 0 | n/a | 2265.6MHz(8 コア) | 1377MHz |
| 1 | 10W | 1200MHz(2 コア) | 520MHz |
| 2 | 15W | 1200MHz(4 コア) | 670MHz |
| 3 | 30W | 1200MHz(8 コア) | 900MHz |
| 4 | 30W | 1450MHz(6 コア) | 900MHz |
| 5 | 30W | 1780MHz(4 コア) | 900MHz |
| 6 | 30W | 2100MHz(2 コア) | 900MHz |

の main() 関数 27 行目で malloc() によりホスト側にデータを確保する．そして，29 行目で GPU 番号を指定した後に，30 行目で GPU での演算に必要なデータは CPU での演算に必要なデータとは別に cudaMalloc() を使用して確保する．また，kernel 関数の実行前（16 行目と 17 行目）に GPU での演算に必要なデータを CPU から GPU へ cudaMemcpy() を用いて転送し，実行後（19 行目と 20 行目）に CPU での演算に必要なデータを GPU から CPU に転送する．なお，Unified Memory を用いない場合の実行イメージは図 2 になる．

## 4. NVIDIA Jetson AGX Xavier 上での階層統合型粗粒度並列処理の性能評価

本章では，NVIDA Jetson AGX Xavier において連立 1 次方程式反復解法のヤコビ法プログラム，粒子ベースの流体解析を行う粒子法プログラム [12] を用いて性能評価を行う．

### 4.1 NVIDA Jetson AGX Xavier の構成

本性能評価では表 5 に示す NVIDIA Jetson AGX Xavier を使用する．ARM v8.2 (8 コア)，Volta GPU (512CUDA コア) を 1 台，メモリ 16GB を搭載している．OS は Ubuntu 18.04，処理系は GCC 7.4.0，CUDA Toolkit 10.0 である．また，Xavier には表 6 のとおり 7 つの電源モードが導入されており，本性能評価では Mode ID を 3 に設定した．その場合，power budget を 30W として各コアは 1200MHz，GPU は 900MHz となる．

### 4.2 ヤコビ法プログラムを用いた性能評価

性能評価プログラムとして連立 1 次方程式反復解法のヤコビ法を用いる．ヤコビ法の反復計算は，収束条件を満たすまで繰り返され，行列サイズは 4,000 × 4,000 とした．ヤコビ法プログラムは図 7 のように 2 階層のマクロタスクグラフで表現される．ここでは，23 個のマクロタスクから構成されている．また，GPU 実行対象とするマクロタスクは

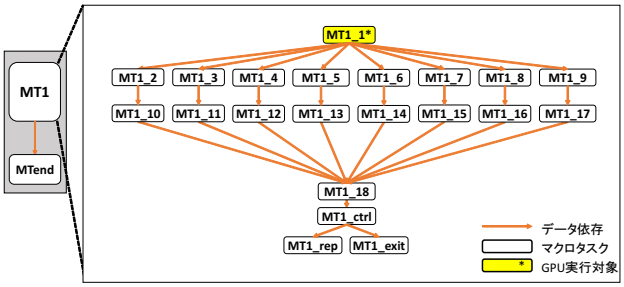


図 7　ヤコビ法プログラムのマクロタスクグラフ.

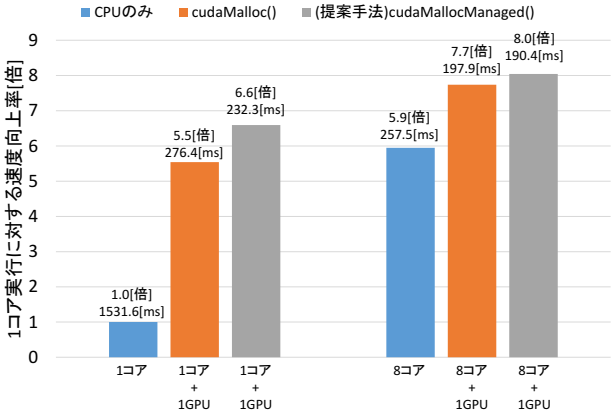


図 8　Xavier 上でのヤコビ法プログラムによる性能評価.

あらかじめユーザが指定する．本性能評価では収束ループ内において，1 コア実行で処理時間が大きいマクロタスク (MT1_1) を GPU 実行対象とした．GPU では 512CUDA コアが利用可能であり，kernel() 関数の実行には 4 ブロック× 1000 スレッドを指定している．また，CPU のみを使用する際には MT1_1 を 8 分割した合計で 30 個のマクロタスクからなるヤコビ法プログラムで性能評価を行っている．

実行結果は図 8 に示すとおりである．まず，階層統合型粗粒度並列処理を実装した CPU の 1 コアの実行時間は 1531.6[ms]，cudaMalloc() を使用した 1 コア＋ 1GPU で 276.4[ms]，cudaMallocManaged() を使用した 1 コア＋ 1GPU で 232.3[ms] となり，1 コア実行に対する速度向上率は 5.5 倍，6.6 倍が確認できる．

次に，GPU を用いない 8 コアの実行時間は 257.5[ms]，cudaMalloc() を使用した 8 コア＋ 1GPU で 197.9[ms]，cudaMallocManaged() を使用した 8 コア＋ 1GPU で 190.4[ms] となり，1 コア実行比で 5.9 倍，7.7 倍，8.0 倍の速度向上が得られた．また，これらの結果から 1 コア＋ 1GPU の実行においては，Unified Memory の適用により 16%，8 コア＋ 1GPU の実行においては，4%実行時間が短縮された．

### 4.3 粒子法プログラムを用いた性能評価

次に，粒子法プログラム [12] を用いて性能評価を行う．粒子法は計算対象の流体を複数の粒子の集まりとして表し，数値的に解くための離散化手法の一つである．粒子法の代表的なものとして SPH 法と MPS 法があり，本性能評価では MPS 法を対象としている．シミュレーションには水柱崩壊を利用した．実行結果を図 9 に示す．図 9(a)，(b)，(c)，

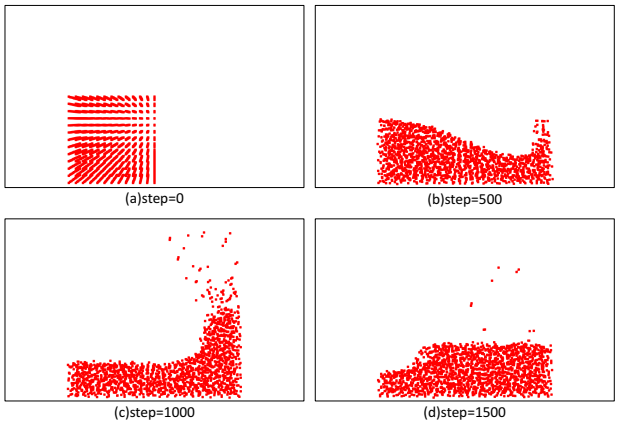


**図 9** 水中崩壊のシミュレーション結果.

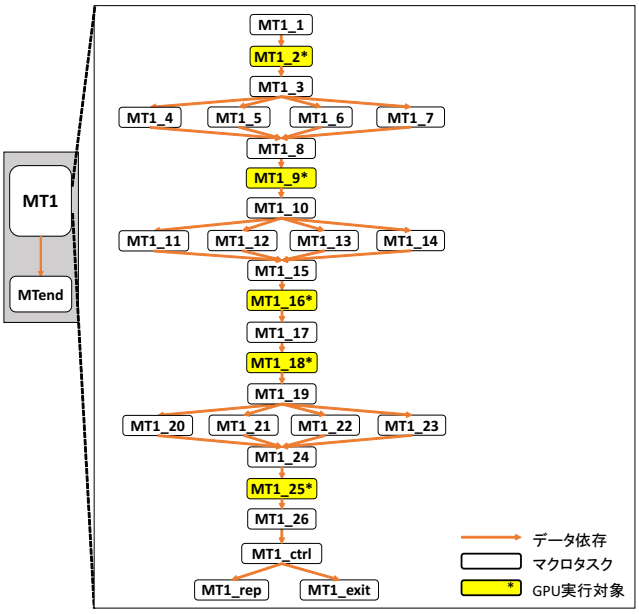


**図 10** 粒子法プログラムのマクロタスクグラフ.

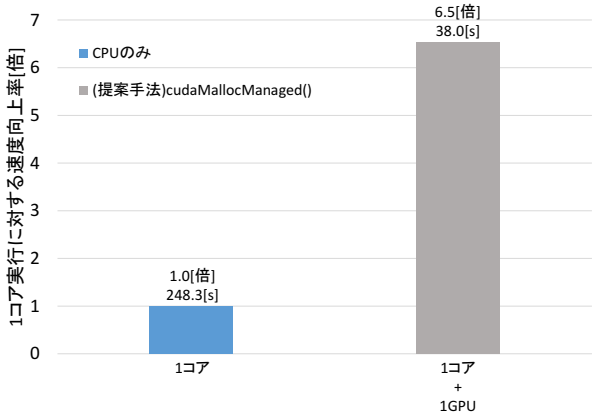


**図 11** Xavier 上での粒子法プログラムによる性能評価.

(d) は各タイムステップのシミュレーション結果である.

粒子法プログラムのマクロタスクグラフは図 10 の通りであり，31 個のマクロタスクから構成される．粒子数は 7606 に設定されている．また，GPU 実行対象とするマクロタスクはあらかじめユーザが指定する．本性能評価では 1 コア実行で処理時間が大きい 5 個のマクロタスクを GPU 実行対象とした．GPU 実行対象マクロタスクの kernel() 関数には 119 ブロック× 64 スレッドを指定している．

実行結果は図 11 に示すとおりであり，GPU を用いない 1 コアの実行時間は 248.3[s]，cudaMallocManaged() を使用した 1 コア＋ 1GPU で 38.0[s] という実行結果となり，これらの結果から，1 コア実行比で 6.5 倍の速度向上が得られている．

## 5. おわりに

本稿では，Unified Memory を伴う GPU システムにおける階層統合型粗粒度タスク並列処理のための並列コードの生成手法を提案した．本並列コードは OpenMP 及び CUDA を用いて実装されている．

ヤコビ法プログラムにおける性能評価では，ARM v8.2 の 8 コアと Volta GPU からなる NVIDIA Jetson AGX Xavier 上で実行したところ，最大で 1 コア比 8.0 倍の速度向上が得られた．また，粒子法プログラムでは，1 コア＋ 1GPU で実行したところ，1 コア比で 6.5 倍の速度向上が得られ，本手法の有効性が確認された．

今後の課題としては，マルチコア/GPU 環境に対応した粗粒度並列処理コードを自動生成する並列化コンパイラの開発が挙げられる．

## 参考文献

[1] 吉田明正: 粗粒度タスク並列処理のための階層統合型実行制御手法，情報処理学会論文誌，Vol.45，No.12 pp.2732-2740，2004.
[2] 笠原博徳，小幡元樹，石坂一久: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理，情報処理学会論文誌，Vol.42，No.4，pp.910-920，2001.
[3] Yoshida, A., Ochi, Y., Yamanouchi, N.: Parallel Java Code Generation for Layer-unified Coarse Grain Task Parallel Processing, IPSJ Transactions on Advanced Computing Systems, Vol.7, No.4 pp.56-66, 2014.
[4] Yoshida, A., Kamiyama, A., Oka, H.: A Task-Driven Parallel Code Generation Scheme for Coarse Grain Parallelization on Android Platform, IPSJ Transactions on Advanced Computing Systems, Vol.10, No.1 pp.1-12, 2017.
[5] 林明宏，和田康孝，渡辺岳志，関口威，間瀬正啓，白子準，木村啓二，笠原博徳: ヘテロジニアスマルチコア向けソフトウェア開発フレームワークおよび API，情報処理学会論文誌，Vol.5，No.1 pp.68-79，2012.
[6] 福田圭祐，丸山直也，松岡聡: 動的タスクスケジューリングによる CPU/GPU ヘテロジニアス環境での FMM の最適化，情報処理学会研究報告，Vol.2011-ARC-197，No.28 pp.1-9，2011.
[7] 小田嶋哲哉，李珍泌，朴泰祐，佐藤三久，塙敏博，児玉祐悦，RaymondNamyst，SamuelThibault，OlivierAumage: GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 協調計算，情報処理学会研究報告，Vol.2013-HPC-136，No.25 pp.1-9，2013.
[8] 後藤 公太，笠原 浩太，大上 雅史，中村 春木，秋山泰: SHAKE 法に着目した分子動力学ソフトウェア myPresto/Omegagene の CPU・GPU 混在環境における高速化，情報処理学会研究報告，Vol.2016-MPS-108，No.17 pp.1-7，2016.
[9] 渡辺智之，吉田明正: マルチコア/GPU 環境における階層統合型粗粒度タスク並列処理，情報処理学会研究報告，Vol.2018-ARC-232，No.25 pp.1-7，2018.
[10] NVIDIA: CUDA C PROGRAMMING GUIDE,

https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf，2019.
[11] NVIDIA: NVIDIA Jestson AGX Xavier, https://www.nvidia.com/ja-jp/autonomous-machines/embedded-systems/jetson-agx-xavier/，2019.
[12] 越塚誠一，柴田和也，室谷浩平: 粒子法入門，丸善出版株式会社，2014.