Vol.2013-HPC-140 No.43
2013/8/2

並列プログラムをメモリ階層利用可能とするランタイム

遠藤 敏夫[1,2,a)]

概要：将来の HPC システムではメモリウォール問題のためにメモリバンド幅と容量の双方を必要とする高性能・大規模計算がより困難となる。その問題の緩和のためには、高速だが小容量のメモリと、大容量だが低速のメモリからなる異種メモリ階層アーキテクチャと、それを効率的に利用するソフトウェア技術が必要である。しかしメモリ階層間のデータ移動をソフトウェアで明示的に記述するのは手間が大きい。本論文では、既存の並列プログラムへの変更を少ないままで、上記の問題を緩和するランタイムライブラリ HHRT/MC の設計およびプロトタイプ実装について述べる。本ランタイムにより GPU アクセラレータを搭載したシステム上で、MPI および CUDA を用いて記述された並列アプリケーションが、大容量のホストデバイスを利用可能とする。テンポラルブロッキングを適用したステンシル計算を例題にとり、記述方法と性能について論じる。

1. はじめに

ポストペタスケール時代・エクサスケール時代における高性能計算システムの構築に向けた重要な課題として、電力性能比の向上の困難や故障頻度の上昇などと並んで挙げられるのが、メモリウォール問題である。これはプロセッサの演算速度の向上よりもメモリのバンド幅および容量の向上が遅いという問題であり、今後のスパコン上において気象・医療・防災などの重要なシミュレーションをさらに大規模化・精緻化する上での障害となると考えられている。

単一のメモリ階層では容量と性能の両立は今後ますます困難となり、メモリ階層の効率的な利用が必要である。古典的にはキャッシュ・メモリ・ハードディスクから成る記憶階層が知られているが、近年のメモリアーキテクチャの進展によって、GDDR メモリ・不揮発性メモリ・DRAM 積層技術など、現在または近い将来に利用可能なメモリの種類は増加し、メモリウォール問題の緩和が期待される。しかしそのためにはアプリケーションソフトウェア開発者への負担の軽減を行うための研究開発が求められる。

異種メモリからなる階層を利用する機会は、現在のスパコンにおいても、GPU や Xeon Phi などのアクセラレータの利用の普及とともに増加している。現在のアクセラレータアーキテクチャの多くは、メニーコアプロセッサから直接アクセスできるデバイスメモリ (GDDR5) を 2〜8GB 程度持ち、そのバンド幅は 100〜200GB/s というものである。ホスト側のメモリ (通常は DDR 系 DRAM) の容量が数十 GB あったとしても、アクセラレータ側からは 8GB/s 程度の PCI-Express バスを介してしか利用できない。

このような状況下で、アクセラレータを用いたいくつかのアプリケーションのメモリ利用法について以下で議論する。ここで述べるのは、NVIDIA 社 Tesla GPU を約 4200 枚搭載した、東京工業大学 TSUBAME2.0 スーパーコンピュータ上で大規模実行されたソフトウェアである：

(a) デバイスメモリ上のステンシル計算: 金属結晶凝固シミュレーション [1] や都市気流シミュレーション [2] では、シミュレーション対象空間を規則的格子で表現したステンシル計算が主要カーネルである。計算対象の格子は、各 GPU のデバイスメモリに分散され、その格子点数はデバイスメモリの容量 (GPU あたり 3GB 弱) と利用 GPU 数の積で抑えられてしまう。

(b) ホストメモリ容量を利用する密行列計算: 半正定値計画問題ソルバーである SDPARA GPU 版 [3] や、異種プロセッサ対応 Linpack[4] では、大規模問題に対応するために、演算対象の密行列をホストメモリに確保しておき、一部ずつデバイスメモリに転送してから GPU で計算している。転送のためのコードは明示的に記述されている。この手法で高性能が得られる理由は、カーネル演算である密行列の Cholesky や LU 分解の局所性が高く、演算コストで転送コストを十分隠ぺいできるためである。

ステンシル計算において安直にホストメモリの大容量を利用しようとすると、局所性の悪さのために、PCI-Express 通信のコストが非常に大きくなってしまう。ステンシル計

[1] 東京工業大学学術国際情報センター
[2] JST, CREST
[a)] endo@is.titech.ac.jp

算の局所性を向上させるために、後述するテンポラルブロッキングと呼ばれる手法が知られている [5], [6], [7], [8]。しかしそのコーディングは複雑になる傾向がある。また、局所性が良好で、ホストメモリの容量を効率的に利用可能な密行列計算においても、階層間のデータ移動のためにコードが複雑となる。

本論文では、上記の問題の解決に向けて、既存の並列プログラムへの変更を少なく抑えたままで、メモリ階層の利用を可能とするランタイムライブラリの提案と、プロトタイプ実装・予備実験結果について述べる。現状では、NVIDIA GPU を搭載したスパコンアーキテクチャにおいて、GPU のデバイスメモリとホストメモリの階層に注目し、対象プログラムとしては MPI と CUDA で記述されたものを前提とする。本論文で述べるランタイムライブラリを **Hybrid Hierarchical Runtime for MPI+CUDA (HHRT/MC)** と呼ぶ。このランタイムでは、複数のユーザプロセスが単一 GPU を共有し、それぞれがデバイスメモリからホストメモリへスワップアウト、また逆にスワップインしながら動作する。このスワップ機能によって、ユーザプログラムへの変更を少なく抑えつつ、大容量のホストメモリを利用可能とする。

提案ランタイムライブラリの検証を、テンポラルブロッキングを用いたステンシル計算プログラムを用い、TSUBAME2.0 スパコン上で行う。

2. HHRT/MC ランタイムライブラリの設計

2.1 概要

本稿で対象とするアプリケーションは、MPI および NVIDIA CUDA (ランタイム API) で記述され、複数 GPU アクセラレータを用いて動作するものとする。つまり、複数計算ノードまたは複数プロセス間の連携に MPI によるメッセージ通信を用い、各プロセスが GPU を制御するために CUDA を用いるものとする。さらに、1 節に述べた (a) のように、主要データ構造がデバイスメモリの容量に収まるものを主な対象とする [*1]。この場合の実行モデルの概略図を図 1 の上図に示す。ここでは計算ノードあたり 1GPU を搭載し、ノードあたり 1 プロセス起動した場合を示す。各プロセスが持つデータ構造はデバイスメモリの容量以内であり、ホストメモリの容量を活用できていないことが分かる。

このようなアプリケーションを、HHRT/MC 上で動作させた場合の実行モデルは図 1 の下図となる。アプリケーションから見える動作の主体は**仮想プロセス (VP)** と呼ばれる。上図との違いは、複数の VP が 1 つの GPU を共有していることである。本モデルの特徴は、VP 単位で高位メモリ (デバイスメモリ) から低位メモリ (ホストメモリ)

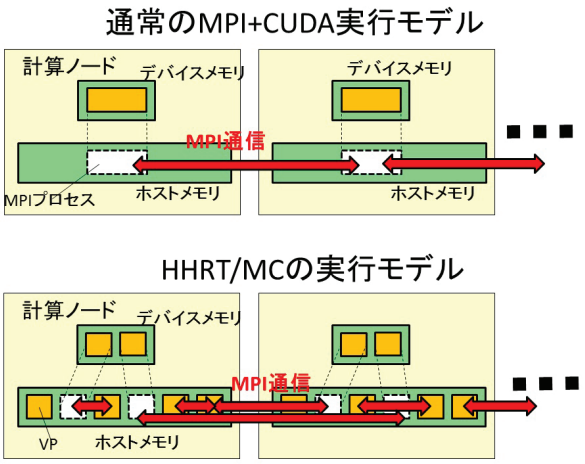

図 1 (上図) 対象とする並列プログラムと、(下図)HHRT/MC の実行モデル

へのスワップアウトを行う点である。各 VP は通常モードと、スワップアウトモードのいずれかを取る。たとえば図の一番左の VP はスワップアウトモードにあり、デバイスメモリを利用していない。また、VP 一つあたりが持つデータ構造のサイズはデバイスメモリ容量以内であるとする。なお、複数 VP が同時に 1GPU のデバイスメモリを共有する (図では 2 つの VP により共有) ことはあるが、GPU のプロセッサコアを用いて計算を行うのは、同時には原則 1VP である。つまり、VP 達は GPU プロセッサコアを時分割で共有する [*2]。

このような前提のもと、十分多数の VP 達が、限られた資源であるデバイスメモリ容量を獲得しあいながら動作を続けるモデルである。各 GPU 上で同時に通常モードになりうる VP の個数はデバイスメモリ容量によって制限され、それ以外の VP はスワップアウトモードとなる。そして全 VP の合計利用メモリ量はデバイスメモリの容量を超え、ホストメモリ側の容量を活用することをねらいとする。

2.2 想定するアプリケーション

上述したように、HHRT/MC 上で動作するアプリケーションは、MPI と CUDA で記述されたものとする。アプリケーションは大別して以下のような動作を行う。

- ホスト上の処理。`malloc/free` によるホストメモリの確保・解放も含まれるが、HHRT/MC は原則的に感知しない。
- MPI によるプロセス間通信。HHRT/MC 上で動作する場合には VP 間通信となる。
- `cudaMalloc/cudaFree` によるデバイスメモリの確保・解放。他 VP によってすでにデバイスメモリが占有されているケースも考慮する必要がある。
- `cudaMemcpy` によるホスト・デバイス間通信

[*1] (b) のようにハンドコーディングによりすでにホストメモリの容量を活用しているものも動作可能であるが、主な対象ではない

[*2] 最近の世代の GPU が対応する concurrent kernel 機能を用いればその限りではないが、本論文では対象外とする

- GPU 上で動作するカーネル関数の呼び出し

2.3 HHRT/MC の詳細設計

HHRT/MC ランタイムライブラリは、前節のようなアプリケーションとリンクされ動作するために、原則的には MPI 互換 API と CUDA 互換 API を持つ。さらに、後述する理由のために追加の独自 API を持つ。

HHRT/MC の主要機能は VP 単位のスワップアウト/スワップインである。現在の設計においては、スワップアウトモードになった VP は、アプリケーションの観点からは一切進行しない、スリープした状態となる。ある VP(a とする) がスワップアウトモードになる可能性のあるのは、VP a が以下の動作を行った時点である。

- MPI 関数呼び出しによりブロックし、かつ下記の条件 (1) が満たされたとき
- `HH_yield`(スワップアウトのタイミングを指定する独自 API) を明示的に呼び出し、かつ条件 (1) が満たされたとき
- `cudaMalloc` によりデバイスメモリを確保しようとし、かつデバイスメモリ不足が検知されたとき

ここで条件 (1) とは、VP a と同一 GPU を共有している VP のうち、スワップアウトモードにあり、かつ走行可能 (MPI 通信待ちでブロックされていない) なものが存在することをいう。

このとき、VP a がそれまでに `cudaMalloc` 等によって確保したデバイスメモリ上の全領域をホストメモリへ退避させ、VP a はスワップアウトモードとなる。デバイスメモリ上にあった領域は解放される。スワップアウトモードとなった VP は、ランタイムライブラリの中でブロックし、アプリケーションの処理は進行できないものとする。

ひとたびスワップアウトモードとなり動作がブロックされた VP は、その後デバイスメモリの空き容量が増え、自らが動作するのに必要な空き容量 (スワップ時に退避した量) ができたときにスワップインされる。

スワップインの際には、その VP のスワップアウト時に退避された全領域をデバイスメモリ上に再確保し、その内容の復元を行う。その後 VP はアプリケーションの処理の進行が可能となる。

このとき、一度スワップアウトされその後スワップインされたデバイスメモリ上の領域のために、以前と同じアドレスが確保できるとは限らない。アドレスが HHRT/MC によって変更されてしまうとアプリケーションの実行に支障をきたすため、現在の設計ではアプリケーション側の協力を得て、以下のような対処を取る。

アプリケーションは `cudaMalloc` などによって確保された領域のアドレスと、整数であらわされ、ユーザが決める領域 ID の紐づけを、`HH_registerDevmem(int id, void *ptr)` という API を呼び出すことにより行う。その後領域のアドレスがスワップアウト/インにより変更されてしまうケースが起こる。アプリケーションはデバイスメモリ上の領域を利用する前に、`void *HH_findDevmem(int id)` を呼び出すことにより、領域 ID から最新のアドレスを取得するものとする。これを可能とするため、HHRT/MC 内部では領域 ID とアドレスの表を管理し、スワップ時に更新する。スワップが起こりうるタイミングは上記のように MPI 関数、`cudaMalloc`、`HH_yield` に限られているため、ユーザが `HH_findDevmem` を呼び出す必要があるのは、これらの呼び出しから、デバイスメモリの利用の間の区間となる。

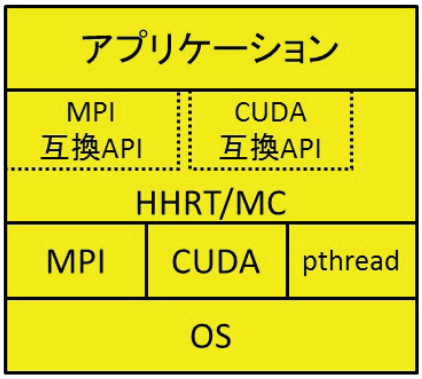

図 2 現在のプロトタイプ実装におけるソフトウェアスタック

2.4 設計についての議論

メモリ領域のスワップアウト/スワップインを行うには、通常の OS がメインメモリと二次記憶間で行っているように、ページ単位で行うのが一般的である。これに対して VP 単位のスワップという方針を取ったのは以下の理由である。

現在の NVIDIA CUDA においては、仮想メモリと物理メモリのマッピングを自由に変更できる API や、Linux 等における segmentation fault 時の挙動を変更可能とするシグナルハンドラに相当する機能が提供されていない。このため、単一プロセスが高位のデバイスメモリ上に確保されたメモリ領域とスワップアウトされた領域の両方を持つという状況を作りにくいという制限があり、ページ単位の代わりにプロセス (VP) 単位のスワップとした。

3. HHRT/MC プロトタイプ実装

3.1 実装の方針

前節で述べた HHRT/MC ライブラリのプロトタイプ実装を行った。現在の実装では、以下のような方針を取っている：

(1) MPI 関数や CUDA 関数にフックした処理を行うために、HHRT/MC ライブラリはそれらの互換 API(現在は一部に鍵有れる) を含み、そこから既存の MPI・

CUDA ライブラリを改めて呼ぶとする。そのためソフトウェアスタックは図 2 のようになる。

(2) 一つの仮想プロセス (VP) は一つの pthread として実装される。(OS) プロセスは一 GPU に対応し、その中に複数の VP が存在する。

現状の実装では、MPI 互換 API や CUDA 互換 API の各関数には HHMPI_Send や HHcudaMemcpy のようにプレフィックスがつけられており、アプリケーションはそれらを呼ぶ。

3.2 MPI 互換 API

アプリケーションの観点からは、各 VP に通し番号のランクが割り振られており、アプリケーションはそのランクを通信相手としてメッセージ通信関数を呼び出す。しかし方針 (2) に示した通り、VP は実際にはスレッドであり、複数の VP が OS プロセスに含まれる。この差異を吸収するために、HHRT/MC 内部で VP のランクと、下層の MPI に見せている OS プロセスのランクの変換を行っている。OS プロセス内の VP どうしの通信であれば下層 MPI を利用せずに共有メモリ上で通信を行う。逆に、OS プロセスをまたぐ通信の際に下層 MPI を利用する。MPI メッセージがあて先の OS プロセスに届いた後、その中の正しい VP にメッセージを届けるのも HHRT/MC の役割である。

また、2.3 節に示した通り、MPI 通信のためにブロックしている間、VP がスワップアウトされうるので、その点の実装も必要である。

3.3 CUDA 互換 API

複数スレッドが GPU を共有しているとき、GPU プロセッサコアや PCI-Express 通信路の調停は、既存の CUDA ライブラリが対応している。そのため、HHRT/MC でフックすべき CUDA 関数は多くない。ただし cudaMalloc によるデバイスメモリ確保時に VP がスワップアウトしうるため、その点の実装が必要である。さらに、VP 単位のスワップアウトを実現するために、各 VP がデバイスメモリ上に確保した全領域を管理する領域テーブルを管理する。

他に、0 番ストリームの扱いが挙げられる。通常の CUDA プログラムがメモリ転送や GPU カーネル関数呼び出しを (明示的にストリームを指定せずに) 行うと、その処理は 0 番ストリームと呼ばれる特殊なストリームに割り当てられる。HHRT/MC 上で、複数 VP が GPU 上の 0 番ストリームを共有してしまうと、本来オーバラップできる処理が逐次化されてしまうケースが生じるため、各 VP に別ストリームを用意することとした。

3.4 スワップ処理

VP のスワップアウトを行う際には、前節に述べた領域テーブルを調べ、含まれている全デバイスメモリ領域を cudaMemcpy を用いてホストメモリ側へ退避する。その後デバイスメモリ側の領域を cudaFree で解放し、空き領域を作る。

各 VP がスワップアウト状態にある間は、継続的にデバイスメモリの空き容量を確認する。自らが動作するのに必要な空き容量ができたときにスワップインを行うが、この際には複数の VP が無駄にスワップインを試みないよう、排他制御が行われる。スワップイン時には、領域テーブルを基に、cudaMalloc により新たにデバイスメモリを確保後、cudaMemcpy で内容を復帰する。この際に以前と違うアドレスが確保されうる問題とその対処については 2.3 節で述べた。

3.5 アプリケーションコードへの変更点

HHRT/MC のねらいは、既存アプリケーションへの変更を最小限にしつつ、メモリ階層を利用可能することであるが、現在の実装においては以下のような変更が必要である。これらはソースコード上の局所的な修正ですみ、構造の大幅な変更は不要と考えられる。

- i) MPI 関数、CUDA 関数にプレフィックス HH をつける (HHMPI_Send や HHcudaMemcpy など)。
- ii) main 関数の名前を HHmain と変更する。
- iii) 大域変数の利用を除去する。gcc コンパイラを利用するのであれば、大域変数の宣言に__thread を追加しスレッド毎の変数とすることで対処できる。
- iv) ストリームを明示せずに GPU カーネル呼び出しをしている箇所で、HH_STREAM0 というストリームを指定する。
- v) デバイスメモリを確保した後に HH_registerDevmem を、デバイスメモリの利用前に HH_findDevmem を呼び出す。

上記の i)∼iv) については、コード変換器などの実装によりユーザから隠ぺいすることができると考えられる。また、そのうち ii)∼iv) については、VP をスレッドとして実装していることに起因しており、VP をプロセスで実装するコストが小さいことが確認できれば、プロトタイプ実装を修正することを検討している。v) については、CUDA 上においてメモリの再確保時に同じアドレスが取得できれば解決できるが、この点については引き続き検討する。

4. ステンシル計算によるケーススタディ

実装した HHRT/MC のプロトタイプを用いたアプリケーションの記述について、三次元ステンシル計算を取り上げて議論する。1 節で議論したように、ステンシル計算を単にスワップしながら動作させるのでは性能が得られないため、局所性向上のための技法として知られる**テンポラルブロッキング**を前提とする。この技法は、計算対象のステンシル領域のうち、その一部について複数回 (段数、以

```
{
  領域全体をホストメモリからデバイスメモリへ転送
  for (jt = 0; jt < nt; jt++) [ // 時間ループ
    領域に含まれる全点を更新
  }
  領域全体をデバイスメモリからホストメモリへ転送
}
```

図 3 単純なステンシル計算

```
{
  領域全体を nd 個の部分領域に分割
  for (jt0 = 0; jt0 < nt; jt0 += k) [ // 時間ループ (外)
    for (jd = 0; jd < nd; jd++) { // 部分領域ループ
      jd 番目の部分領域をホストメモリからデバイスメモリへ転送
      (冗長な袖部分の転送を含む)
      for (jt=jt0; jt < jt0+k; jt++) [ // 時間ループ (内)
        jd 番目の部分領域に含まれる全点を更新
        (冗長な袖部分の計算を含む)
      }
      jd 番目の部分領域をデバイスメモリからホストメモリへ転送
    }}
}
```

図 4 テンポラルブロッキングを適用したステンシル計算：ハンドコーディング版

下 k とする) の時間発展を計算するというものである。既存研究としては、キャッシュやレジスタの再利用性向上を行ったもの [5], [6] や、本稿と同様に GPU デバイスメモリとホストメモリを効率利用を目的としたもの [7], [8] が挙げられる。

以下ではまず、著者らが以前発表した、ハンドコーディングによるテンポラルブロッキング手法 [8] について述べる。そして、それとほぼ同等なアルゴリズムを HHRT/MC 上で記述した場合について説明する。

4.1 ハンドコーディングの場合

単一 GPU 上の単純なステンシル計算の流れについて、図 3 に示す。ここでは計算対象の領域はデバイスメモリ内に収まることを仮定し、ホストメモリとデバイスメモリの間の転送は時間ループの外にあり、性能への影響は少ない。領域内の全点の更新処理のためには、GPU 上で多数スレッドを起動して行うのが効率的であり、以降のアルゴリズムでも計算部分については同様の方法を取る。また本来はダブルバッファリングに伴う処理も必要であるが図では割愛している (以下の説明も同様)。

図 4 には、計算対象領域がデバイスメモリ容量を超えた場合のアルゴリズムを示す。これも単一 GPU 上のものである。ここでは計算対象領域を部分領域に分割し、それぞれの部分領域はデバイスメモリに収まるようにする。それを順次ホストメモリからデバイスメモリに転送して計算する。この転送処理は時間ループの内側にあるため、コスト削減が必要である。そのために、k 段のテンポラルブロッキングを行い、一度転送したら GPU 内で k 回の時間発展

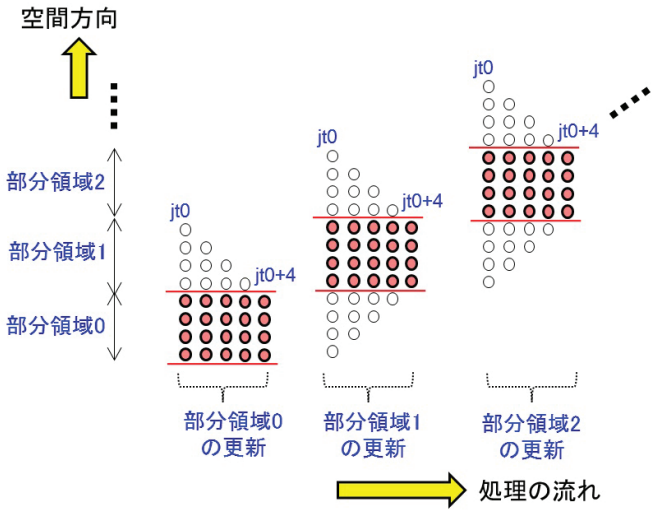

図 5 テンポラルブロッキングを適用したステンシル計算の処理の流れ ($k = 4$ の場合)

を進めてしまう (疑似コードの「時間ループ (内)」の部分)。

ここで問題となるのが、ステンシル計算においては、各点の更新のために、前の時間ステップの領域上の隣接点も必要であることである [*3]。その様子を模式的に図 5 に示す ($k = 4$ の場合)。依存関係のため、ある部分領域を局所的に計算するためには、部分領域からの距離が k 以下であるような全点のデータを必要とする。このために保持する隣接部分を一般的に袖領域と呼ぶが、k が大きくなるほどこの領域は大きくなり、冗長な計算量・転送量が増えてしまう。図における白い部分が、テンポラルブロッキングによって増大した計算量に対応する。

この袖領域のための計算量・通信量を削減する最適化技法も提案されている。紙面の都合上これらは割愛するが、詳細は論文 [8] を参照されたい。

図 4 ではブロッキングのためのコードとデータ転送のコードがいずれも明示的に表れていることが分かる。これに加え、最適化技法を適用するとさらに複雑となり、また本図のコードは単独 GPU 向けであるため、複数 GPU を用いる際には MPI による通信がさらに必要となる。

4.2 HHRT/MC 上でのコーディング

テンポラルブロッキングを適用したステンシル計算を、HHRT/MC 上で記述した疑似コードを図 6 に示す。単一 GPU のみを使う場合であっても、プログラムの実行は複数 VP が協調して行い、疑似コードは一つの VP の挙動を示すものである。計算対象領域は VP 間で分割され、各 VP の担当する領域はデバイスメモリ内に収まるサイズとする。つまり、前節における部分領域が、各 VP に相当することになる。

各 VP は担当領域の更新を行う前に、隣接領域を担当す

[*3] 本論文では、一つ隣りまでの点に依存する計算を仮定するが、それ以外のステンシル計算についても同様の議論ができる

```
{
  自分の担当領域をホストメモリからデバイスメモリへ転送
  for (jt0 = 0; jt0 < nt; jt0 += k) [ // 時間ループ (外)
    // 通信部分
    k 段分の袖領域をデバイスメモリからホストメモリへ転送し、
    隣接プロセスへ MPI で送信
    k 段分の境界領域を隣接プロセスから MPI で受信し、
    ホストメモリからデバイスメモリへ転送
    HH_findDevmem により、領域配列のポインタを復帰 (*)
    // 計算部分
    for (jt=jt0; jt < jt0+k; jt++) [ // 時間ループ (内)
      自分の担当領域に含まれる全点を更新
      (冗長な袖部分の計算を含む)
    }}
  自分の担当領域をデバイスメモリからホストメモリへ転送
}
```

図 6 テンポラルブロッキングを適用したステンシル計算：HHRT/MC 上のコード

る VP と境界部分の交換を MPI(厳密には MPI 互換 API) を用いて行う。本コードには k 段のテンポラルブロッキングが適用されており、k 時間ステップ毎に一回のみ、MPI 通信が発生する。その際には、以降の k ステップの計算を局所的に行うことを可能とするため、k 段分の境界領域の交換を一度に行っておく。

本疑似コードは、(*) のついた行を除去すれば、通常の MPI と CUDA ライブラリ上でも動作するものである。HHRT/MC 上で動作させた場合の特徴は、VP 単位のスワップアウト/インを行いながら処理を進行させる点であり、これにより全 VP で計算可能なデータ量の合計を、デバイスメモリ容量より大規模にすることができる。

また、本疑似コード内ではスワップアウトの起こりうる箇所は MPI 通信の箇所のみであり、「時間ループ (内)」に含まれる計算の最中にはスワップが起こらないことが保証される。

本コードを図 4 のハンドコーディング版と比べると、時間ブロッキングによるループ構造の変更の記述については、どちらでもユーザが行うという共通点がある。一方、ハンドコーディング版で記述が必要だった部分領域のデータ転送については、HHRT/MC 上では記述は不要である。この転送については、スワップアウトという形でシステムが行うためである。

さらに、図 6 のコードにはすでに MPI 通信コードが含まれており、プログラム起動時に GPU と VP の対応を変更するだけで、同一のコードで複数 GPU に対応可能である。一方、図 3, 4 のコードを複数 GPU に対応させるためには、MPI 通信の追加が必要であるという差異がある。

5. ステンシル計算による予備性能評価

5.1 評価環境および条件

ハンドコーディングされたステンシル計算および HHRT/MC 上で記述されたステンシル計算の性能の予備評価を、東京工業大学 TSUBAME2.0 スーパーコンピュータの単独 GPU 上で行った。

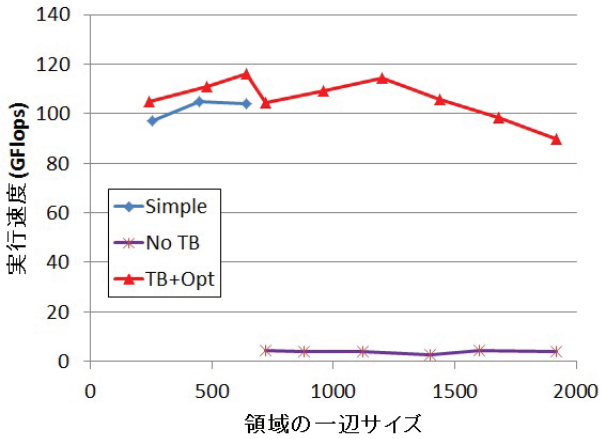

図 7 ハンドコーディング版ステンシル計算の性能

TSUBAME2.0 に搭載されている NVIDIA Tesla M2050 GPU は、理論性能値 515GFlops/1030GFlops(倍精度/単精度) であり、デバイスメモリの容量は 3GB である。ただし ECC 機能の影響などにより実際に利用可能な容量は約 2.6GB である。TSUBAME2.0 の計算ノードである HP Proliant SL390s には 54GB のホストメモリが搭載されているので、ホストメモリを有効活用できれば 20 倍近くの格子点数の計算ができることになる。

今回評価したステンシル計算は立方体の三次元領域を対象とし、各点は定数係数の 7 点ステンシルにより計算される。データ型は単精度浮動小数である。

5.2 ハンドコーディング版の性能

ハンドコーディング版の性能を図 7 のグラフに示す。グラフ中の'Simple' は図 3 に対応し、容量の問題により一辺約 700 を超えると動作しない。'No TB' は、部分領域に分割するがテンポラルブロッキングを利用しない場合である。'Simple' に比べ 1/20 ～ 1/30 に性能低下してしまっており、これはデータ転送頻度が高いためである。'TB+Opt' は、図 4 のようにテンポラルブロッキングを適用し、さらに冗長計算削減などの種々の最適化技法が施されたものである。なお、ブロッキング段数 k については、それぞれの問題サイズにおいてパラメータサーチを行い、最良の性能を示すものが選択されている。最良の k は、問題サイズに依存するが、24 ～ 240 の範囲であった。グラフから、この'TB+Opt' によりデバイスメモリ容量を超える場合であっても、小さい問題の場合とほぼ同等な性能を実現できていることが分かる。

5.3 HHRT/MC プロトタイプ上の性能

図 6 の実装を HHRT/MC プロトタイプ上で実行したところ、デバイスメモリ容量を超えるような問題サイズでも

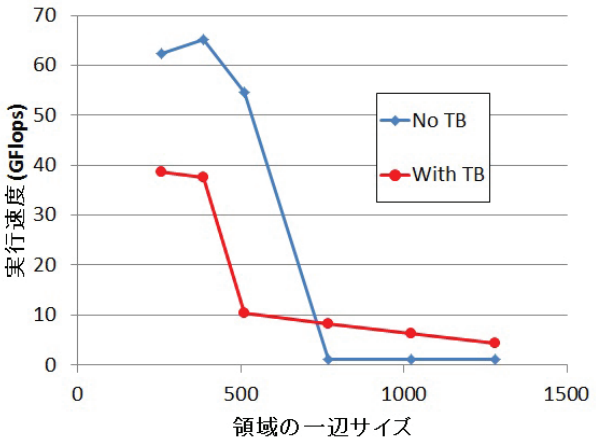

図 8　HHRT/MC 版ステンシル計算の性能

実行可能であり、HHRT/MC のスワップ機能が動作することを確認した。TSUBAME2.0 の単一 GPU 上で実行した場合の性能を図 8 に示す。グラフ中の'No TB' はテンポラルブロッキングを行わない場合であり、'With TB' は行う場合である。後者についてはブロッキング段数 $k = 32$ とした。なお起動 VP 数については、VP あたりのデバイスメモリ利用量が約 1GB となるように調整したところ、4 ～ 16 の範囲となった。

スワップが発生する、一辺サイズが 768 以上の場合について'No TB' と'With TB' を比較すると、テンポラルブロッキングを適用した後者のほうが 4 ～ 6 倍程度高速となっている。しかしながら、前節のハンドコーディングの性能と比べると 1/10 以下の性能となっており、改良の余地が大きい状況にある。

この問題には、現在の HHRT/MC プロトタイプの実装の問題と、アプリケーション側の問題の双方が影響していると推測される。より詳細な調査を行ったところ、一度の MPI 通信で特定の VP が複数回スワップアウト・インを繰り返してしまうケースも見受けられ、ランタイムの実装の改善を早急に行う予定である。一方アプリケーション側の改善については、ハンドコーディング版に適用されていた、冗長計算除去などの最適化が、HHRT/MC 上の簡潔な記述と両立可能であるかの検討を行う予定である。

6. 関連研究

メモリウォール問題の部分的解決として、high bandwidth memory や hybrid memory cube [9] のような DRAM チップの三次元積層技術に基づいたデバイスが 2015 年頃に利用可能と期待されている。これらは DDR 系 DRAM に比較して高いバンド幅 (チップあたり数百 GB/s 程度) を持つ一方で、容量は不利と考えられている。NVIDIA 社が発表している Echelon プロジェクト [10] では、2018 年頃にエクサスケールのシステムの実現を目標とし、積層 DRAM が利用される予定である。さらにメモリ容量の問題を緩和するために、不揮発性メモリも併用するとされているが、そのような異種メモリをアプリケーションからどのように用いるべきかは明らかではない。

GPU などのアクセラレータを持つ異種混合アーキテクチャにおいてアプリケーションを効率的に動作させるシステムとして、StarPU[11] や DAGuE[12] などの細粒度タスクスケジューラが挙げられる。これらのアプローチの主眼は異種プロセッサの効率的利用であり、本研究で取り組んでいるようなメモリ階層の活用ではない。また、アプリケーションを細粒度タスクとその間の依存関係を示す DAG という形で記述する必要があるのに対し、本研究では、すでに広まっている MPI と CUDA で記述されたアプリケーションを少ない変更で動作可能とする。

本研究のランタイムライブラリにより近いモデルを持つシステムとして Adaptive MPI[13] が挙げられる。これは並列オブジェクト指向プログラミングシステムである Charm++[14] に基づいて実装されており、MPI 互換 API を持つ。Adaptive MPI 上では MPI で記述されたアプリケーションが動作するが、計算ノード上に比較的細粒度なプロセスが多数起動される。この点は本研究の HHRT/MC と共通する一方で、Adaptive MPI そのものはアクセラレータに対応するものではなく、またメモリ階層の活用についても触れられていない。一方、Adaptive MPI の利点である、動的負荷分散への対応や、通信と計算の自動的なオーバラップについては、HHRT/MC にも適用可能であると考えられる。

7. おわりに

本研究の目標は、メモリウォール問題の緩和に向けて、高速だが小容量の高位メモリと、大容量だが低速な低位メモリからなるメモリ階層を効率的に活用することである。その目標に向けて、GPU アクセラレータを搭載したシステムにおいて、デバイスメモリとホストメモリを有効活用するランタイムライブラリである HHRT/MC の設計とプロトタイプ実装について述べた。

ランタイムライブラリの効果を実証するために、ステンシル計算を例にとった。メモリ階層間でスワップアウト/インしながら動作する場合にはアルゴリズムの局所性の向上が必須であり、テンポラルブロッキング手法を用いた。提案するランタイムライブラリ上で、そのような手法を取り入れたアプリケーションも簡便に記述可能であることを示した。

現状のプロトタイプ実装は、ハンドコーディングされたステンシル実装に比べるとまだ性能は不十分であることが分かったため、近い将来の予定としては、詳細な解析と実装の改良、およびハンドコーディングの場合に適用した種々の最適化技法がランタイム上で記述可能かの検証を行う。

また、現在のスワップアウト先はホストメモリとしてい

るが、高速 Flash メモリや別計算ノードの遠隔メモリの利用により、更に計算可能な問題サイズを大規模化する予定である。更には、現在 MPI+CUDA に限られているアプリケーションの範囲を拡大する。候補としては、MPI+OpenACC で記述されたアプリケーションや、複数 Xeon Phi 上のアプリケーションなどが挙げられる。

謝辞

本研究は JST-CREST の研究課題「ポストペタスケール時代のメモリ階層の深化に対応するソフトウェア技術」の支援を受けております。本稿で比較対象としたソフトウェアの実装・評価は東京工業大学松岡聡研究室の金光浩さんによります。

参考文献

[1] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Akinori Yamanaka, Akira Nukada, Toshio Endo, Naoya Maruyama, Satoshi Matsuoka: Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In Proceedings of IEEE/ACM Supercomputing (SC11), 11pages (2011).

[2] 小野寺直幸, 青木尊之, 下川辺隆史, 小林宏充: 格子ボルツマン法による 1m 格子を用いた都市部 10km 四方の大規模 LES 気流シミュレーション. 情報処理学会ハイパフォーマンスコンピューティングと計算科学シンポジウム (HPCS2013), pp. 123–131 (2013).

[3] Katsuki Fujisawa, Toshio Endo, Hitoshi Sato, Makoto Yamashita, Satoshi Matsuoka, Maho Nakata: High-Performance General Solver for Extremely Large-scale Semidefinite Programming Problems. In Proceedings of IEEE/ACM Supercomputing (SC12), 11pages (2012).

[4] 遠藤 敏夫, 額田 彰, 松岡 聡: スーパーコンピュータ TSUBAME 2.0 における Linpack 性能 1 ペタフロップス超の達成. 情報処理学会論文誌コンピューティングシステム, Vol. 4, No.4 (ACS 35), pp.169–179 (2011).

[5] M. Wittmann, G. Hager, and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages (2010).

[6] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In Proceedings of IEEE/IEEE Supercomputing (SC10), 13pages (2010).

[7] Leonardo Mattes and Sergio Kofuji: Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. In Proceedings of International Conference on Microwave and Millimeter Wave Technology (ICMMT), pp.1536–1539 (2010).

[8] Guanghao Jin, Toshio Endo and Satoshi Matsuoka: A Multi-level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), in conjuncition with IEEE IPDPS2013, 8pages (2013).

[9] Hybrid Memory Cube Consortium: `http://www.hybridmemorycube.org/`

[10] Bill Dally: Technical Challenges on the Road to Exascale. NVIDIA Booth Talk, IEEE/ACM Supercomputing (SC12) (2012).

[11] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation. Practice and Experience, Special Issue: Euro-Par 2009, **23** pp. 187–198 (2011).

[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra: DAGuE: A generic distributed DAG Engine for High Performance Computing. Parallel Computing, T. Hoefler eds. Elsevier, **38**(1-2), pp. 27–51 (2012).

[13] Chao Huang, Orion Lawlor, L. V. Kale: Adaptive MPI. In proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958, pp.306–322 (2003).

[14] L. V. Kale, S. Krishnan: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In proceedings of OOPSLA'93, pp. 91–108 (1993).