Vol.2015-HPC-148 No.24
2015/3/3

Extreme SIMD アーキテクチャのプログラミングモデル拡張Cによる性能評価

宇川 斉志[1,a)]　佐藤 三久[1,2]　朴 泰祐[1,2]　児玉 祐悦[1,2]　山口 佳樹[1]　山本 淳二[3]

概要：エクサスケールの計算機の実現にあたって文部科学省の委託研究により大量の演算器を 1 つのマスタープロセッサで制御する Extreme SIMD アーキテクチャが提案された．本稿では Extreme SIMD アーキテクチャ上で実行できるバイナリを生成するためのプログラミングモデルの一つとして Extreme SIMD 向け拡張 C について述べ，それを用いて Extreme SIMD アーキテクチャのシミュレータによる評価を行う．Extreme SIMD 向け拡張 C は PE で実行される文を SIMD 制御文で囲むだけで簡単に指定することが可能で，メモリ操作などはライブラリ関数で提供されるという特徴を持つ．2 次元のラプラス方程式の差分法による計算を実行したところローカルメモリに入りきる 4096 × 4096 格子の問題サイズではピーク性能に対して最大 74%と高い実効効率を示すが，入りきらない問題サイズでは実効効率が 1 割未満になるという急激な性能低下を起こすことを確認した．

1. はじめに

近年，スーパーコンピュータの消費電力の増大が問題になってきている．2011 年に Top500 で 1 位を獲得した京は約 12MW [1] もの消費電力を必要とする．電力効率で言うと約 0.8GFlops/W，2014 年の 11 月にスーパーコンピュータの消費電力の指標を示す Green500^{TM} で 1 位を獲得したドイツ GSI ヘルムホルムセンターのスーパーコンピュータ L-CSC でも 5.2GFlops/W [2] にとどまっている．そんな中，近年のスーパーコンピュータの性能向上率が今後も続くと仮定すると 2018 から 2020 年にはその演算性能が 1ExaFlops まで到達すると期待されている．運用時の予算的な問題により一般的なスパコンセンターに供給できる電力は 20MW から 30MW が限界と言われているので，1ExaFlops を実現するには 33〜50GFlops/W の電力効率が必要ということになる．しかし，現在一般的に使われている汎用プロセッサのアーキテクチャはプロセスルールの進歩による消費電力の減少だけでは約 4 倍程度までしか電力効率が伸びないといわれている [3]．そこでより電力効率の高い計算機を検討するために文部科学省の委託研究「演算加速機構を持つ将来の HPCI システムに関する調査研究の研究開発」により Extreme SIMD アーキテクチャ [4] が提案された．本研究では Extreme SIMD アーキテクチャのプログラミングモデルとして拡張 C によるモデルにより，シミュレータを用いて Extreme SIMD アーキテクチャの定量的な評価を行うことを目的とする．

本稿は本章を含めて 8 章で構成される．2 章では Extreme SIMD アーキテクチャの実装例として PACS-G を取り上げ，その概要について述べる．3 章では PACS-G の関連技術について説明する．4 章では Extreme SIMD アーキテクチャの実行モデルに関して説明を行う．5 章では Extreme SIMD アーキテクチャ向けのプログラミングモデルの拡張 C についての説明と，2 次元のラプラス方程式の差分法による計算を例にあげてプログラムの説明を行う．6 章では評価環境として PACS-G のシミュレータと，それを改良したシミュレータに関して言及する．7 章では 6 章で言及したシミュレータを用いた 2 次元のラプラス方程式の差分法による計算の性能評価を行う．なお，現状では Extreme SIMD 向け拡張 C のプログラムからバイナリを生成するコンパイラを開発中であるため，評価は出力されるであろうアセンブリを直接記述して評価した．8 章では結論と今後の課題を述べる．

[1] 筑波大学大学院 システム情報工学研究科
Graduate school of Systems and Infomation Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] 筑波大学 計算科学研究センター
Center of Computational Science, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[3] （株）日立製作所 中央研究所
Hitachi, Ltd., Central Research Laboratory, Yokohama, Kanagawa, 244–0817, Japan
[a)] ukawa@hpcs.cs.tsukuba.ac.jp

ⓒ 2015 Information Processing Society of Japan

2. Extreme SIMD アーキテクチャ PACS-G の概要

Extreme SIMD アーキテクチャとは演算器とローカルメモリ (以下 LM) からなる多数の Processor Element(以下 PE) を Master Processor(以下 MP) から発行される SIMD 命令で制御するアーキテクチャである．本章ではその実装例としてアクセラレータ型演算加速装置の 1 つである PACS-G の概要について説明する．図 1 に PACS-G の 1 チップの概要を示す．単純な PE と呼ばれる演算器を大量に並べ，演算器をコントロールする MP からの SIMD 命令で全ての PE が一斉に同一の命令の実行を行う．この SIMD 命令は MP からの 1 つの命令 (single instruction) を，各 PE 上の LM 上のデータ (multiple data) に対して実行するという意味で，Intel 社の SSE 及び AVX 命令のようなコア内で複数のデータに対する演算を 1 つにまとめるものとは異なる．MP はキャッシュをもつ汎用プロセッサを用いる．各 PE は MP によってデコードされたマイクロコードを実行するので汎用プロセッサでいうフロントエンド部分を持たない．そのためより多くの回路を演算部分に費やすことができる．PE は GRAPE-DR [5] の PE をベースにコアを拡張したものであり，昨年発表された予測では 14nm プロセスで 2048 から 4096 コア程度まで搭載できる見込みとなっている [4]．各 PE はキャッシュを持たず，PE 当たり容量 128KB の，アドレスで参照を行う LM が搭載される．PE 内部には 128bit の浮動小数点数の乗算器及び加算器，整数演算器の 3 つの演算器を持つ．浮動小数点数の加算のどちらかと整数演算と、浮動小数点数の乗算は内部的な制約を満たせばそれぞれは並列に処理を行うことが可能である．したがって PE の動作周波数が 1GHz と仮定すると 1PE の演算性能は $1\text{GHz} \times 2\text{Flop} \times 2 = 4\text{GFlops}$ となる．これに対して，LM のメモリバンド幅は 128bit/cycle が想定されており，1GHz では 16GB/s なので演算性能に対するメモリバンド幅は 4Byte/Flop となり，バランスのとれた構成になっている．各 PE は隣接同士が 4 次元のメッシュによって接続される．隣接 PE 間の通信速度は片方向通信で 64bit/cycle，バンド幅は 8GB/s となる．2048PE での演算性能は $2048 \times 4\text{GFlops} = 8\text{TFlops}$ である．またこの隣接間の接続以外にもリダクションネットワークとして TREE 状のネットワークの搭載も検討されている．

また，PACS-G は LM 以外にもチップ内に Broadcast Memory(以下 BM)，チップ外に Global Memory(以下 GM) を持つ．BM は 2 次元にマッピングされた各行に 1 つ配置されるメモリで，行上にある全ての PE に対してデータをブロードキャストすることができる．GM はこの BM に接続されるチップ外のメモリで，現在広く利用されている DDR ではなく，より高速なメモリ転送を目指して開発されている新規格である Hybrid Memory Cube(HMC) または High Bandwidth Memory(HBM) を想定している．PE が GM にアクセスするには BM を介する必要がある．

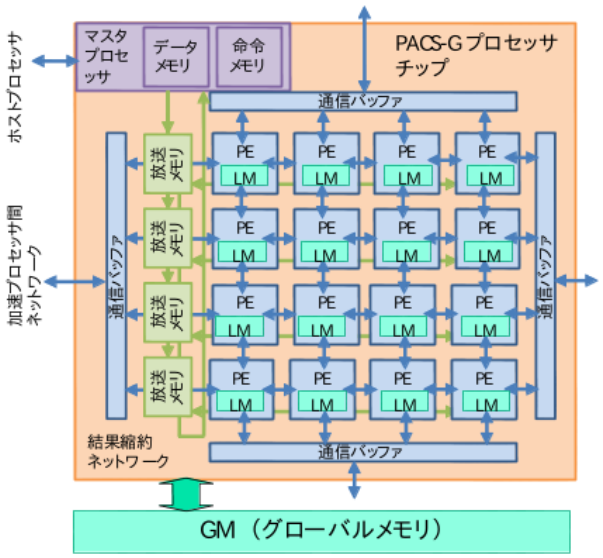

図 1 PACS-G チップの概要

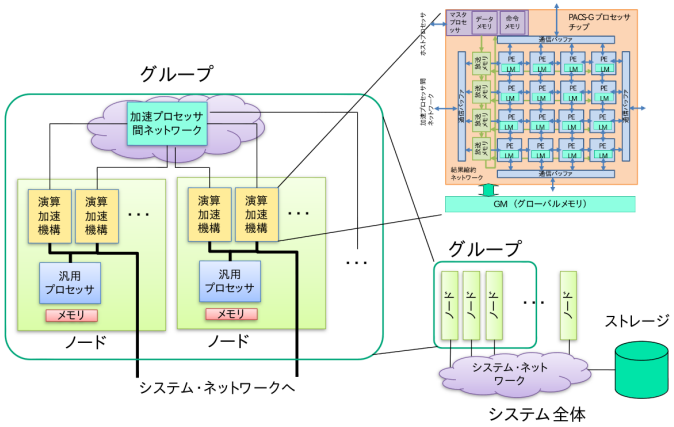

図 2 PACS-G を用いたシステムの構成

次に PACS-G を用いたシステムの構成を図 2 に示す．各ノードは汎用プロセッサと複数の PACS-G を用いた演算加速機構から構成される．また，演算加速機構同士は 1 リンク 20GB/s，16 方向にリンクを持ち全体で 320GB/s 程度のバンド幅を持つ加速プロセッサ間ネットワークによって 4 次元トーラスで接続される．演算加速機構間の通信には 2 次元にマッピングされた PE の各辺に存在する通信バッファを用いて行う．加速プロセッサ間ネットワークは 2048 から 4096 演算加速機構ごとに構成され，加速プロセッサ間ネットワークを用いた演算加速機構間での高速な直接通信を実現することでストロングスケーリングにも対応可能にする．

3. 関連技術

3.1 GPGPU

ハイパフォーマンスコンピューティングの分野でしばし

ⓒ 2015 Information Processing Society of Japan

ば演算を高速化するために使用される装置として Graphic Processing Unit(以下 GPU) があげられる．GPU は元々画像表示のための処理を行う専用チップとして発達してきたが，その高い演算性能を科学技術計算に使用する General Purpose computing on GPU が広まってきている．GPU の一つである NVIDIA 社の Geforce®GTX™TITAN black は SMX と呼ばれるマルチプロセッサを 15 個持ち，各 SMX は 32thread を 1 つにまとめた warp と言う単位で実行される．メモリアクセスなどの高いレイテンシの処理は warp を切り替えることで隠蔽する．

3.2 メニーコアプロセッサ

演算を加速させる目的で開発されたメニーコアプロセッサとして Intel 社の Xeon Phi™ があげられる．Xeon Phi™ は PCI Express スロットに挿して使用するアクセラレータ型の演算器で，内部に 60 個程度のコアを有する．Xeon Phi™ は映像出力端子などが存在せず PACS-G と同様の純粋な演算装置となっている．各コアは x86 アーキテクチャのインテル®Pentium®に SIMD 演算器を追加した構成となっているため既存の x86CPU 用に作成されたプログラムをほぼそのまま実行することができる．また GPU のようにオフロードさせて使用することも出来るが，Xeon Phi™ 上で Linux を動かしその上でプログラムを実行することも可能である．GPU や PACS-G と大きく異なる点は各コアが独立した命令を実行する点である．

4. Extreme SIMD アーキテクチャの実行モデル

Extreme SIMD アーキテクチャの命令列は MP で実行される逐次部と PE で実行される並列部に分けられる．これらはともに GM 上に存在し，MP によって制御される．図 3 に実行の流れを示す．プログラムが開始されると一般的な汎用プロセッサの様に逐次部の命令列が MP にフェッチされ実行が始まる．MP と PE の間にはシーケンサーと呼ばれるモジュールが存在し，このシーケンサーにより PE で並列部の命令列のデコード及び実行の終了判定などの制御が行われる．MP が並列部の開始を指示するとシーケンサーに並列部の命令列が読み込まれデコードが開始される．デコードされたマイクロコードは全 PE に配布され並列部の実行が始まる．PE が実行する命令は SIMD なので基本的に全ての PE は同一の命令を実行するが，PE が解釈できる命令は prefix がついており，prefix で指定されるマスクが示す条件次第でその命令を実行するか否かを決定することができる．PE の命令には分岐命令は存在しないので各 PE での命令実行の制御はこのマスクを用いて行われる．MP とシーケンサーは非同期に動作するので MP は開始した並列部の終了を待たずに次の並列部の実行を指示することができる．

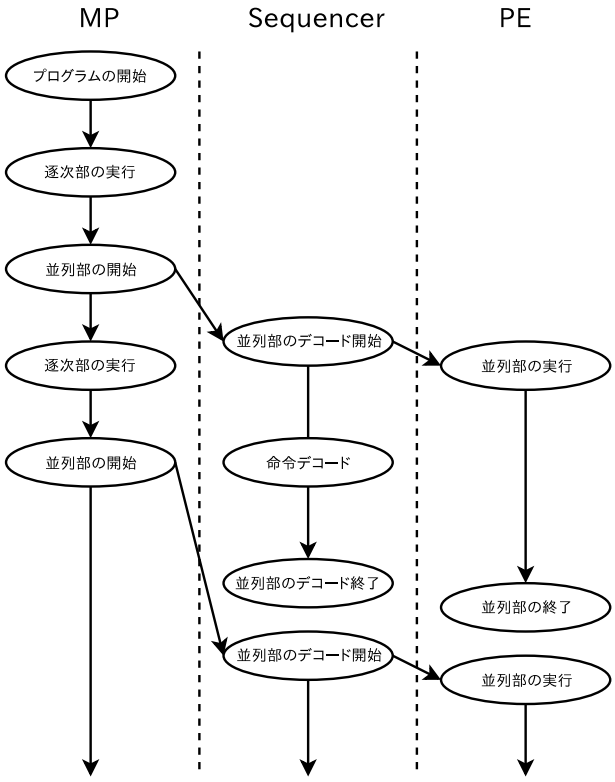

図 3 Extreme SIMD アーキテクチャでのプログラムの実行の流れ

MP から参照できるモジュールは BM までであり，PE 側のレジスタ及び LM は参照できない．GM と PE の間でデータをやりとりするには MP が GM と BM の間での DMA 転送を起動し，データが BM に到着したら PE が BM から LM にデータを転送する必要がある．BM から LM へデータの転送を行うとその BM が所属する行の全 PE に同一のデータがブロードキャストされるため受け取ったデータを LM に書き込むか否かは前述した prefix のマスクで指定する．逆に LM から BM にデータを転送する場合も同様である．

5. Extreme SIMD アーキテクチャのプログラミングモデル拡張 C

Extreme SIMD アーキテクチャ上で実行できるプログラミングモデルとして拡張 C を提案する．拡張 C は従来の C 言語に SIMD の制御を行うための構文やデータ転送に関する組み込み関数及びライブラリ関数を追加したものである．

5.1 関数の実行と変数

拡張 C では関数に修飾子を付加することで PE で実行される関数を指示することができる．以下に修飾子を示す．

__all__ 修飾子

関数全体が PE 上で実行される．

__global__ 修飾子

関数内に PE で実行される部分が存在する．

修飾子なし

MP 上で実行される．

__all__修飾子が付加された関数では局所変数の領域は PE 上に確保される．__global__修飾子が付加された関数では MP，PE 上で局所変数が確保され，引数は MP から値をコピーする．大域変数については MP と PE の同じアドレスに割り付けられる．したがって大域変数の実体は各 PE 及び MP 上に重複して存在することになる．

5.2 SIMD 実行制御文

PE 上で実行される SIMD の実行を制御する構文として__do_all__構文が存在する．__do_all__構文は次のように記述される．

```
__do_all__ { STATEMENT }
```

__do_all__構文ではブロック内の STATEMENT で記述された文はすべての PE で実行することを意味する．__do_all__構文が使われる関数はすべて__global__修飾子を付加しなければならない．また，__all__修飾子は関数全体に__do_all__構文を用いることと同義である．なお，__do_all__構文内でいかなる変数を変更してもその実体はその PE にしかないため他の PE が持つデータに影響を与えることはない．

5.3 template 及び template ループ文

template とは PE に割り当てられる仮想的なインデックス空間である．与えられたサイズで空間を分割し，PE に割り当てる．template は 1 次元から 4 次元まで存在し，それぞれライブラリ関数で作成される．template ループ文では作成した template に対して範囲を指定し，指定された範囲のインデックスに対して各 PE でループを実行する構文である．template ループ文は__for_all__構文で記述される。2 次元の template に対する__for_all__構文は次のように記述される．

```
__for_all__(template; xmin:xmax;
            ymin:ymax) { STATEMENT }
```

__for_all__の次の括弧の template は使用する template を，xmin 及び xmax は x 次元の範囲を，ymin 及び ymax は y 次元の範囲を表す．範囲の指定は template の次元にあわせて行う．STATEMENT には指定された範囲のインデックスに対して実行される文が記述される．

図 4 に template の作成及び template ループ文の実行の例を示す．図 4 の上部のソースコードではライブラリ関数 pg_template2D() で 1024×1024 の要素に対して PE に割り当てられる 2 次元の仮想インデックスを作成し，__for_all__構文で指定された範囲のインデックスに対して $a = b + c$ の実行を行っている．__xi__は PE 内の x 方向の仮想インデックス，__yi__は PE 内の y 方向の仮想インデックスを示す．実行される範囲を図で表すと下部のようになる．黒枠の四角形が物理的な PE を示していて，割り当てられた 1024×1024 の要素の内，赤枠の四角形の部分の計算が実行される．template はこのようにデータ領域全体に対して各 PE に割り当てられるインデックスを自動で計算するものである．

5.4 template を用いたデータ転送

template を用いることで GM と LM の間でのデータ転送や，PE 間の袖領域の更新を簡単に記述することが出来る．袖領域とは，偏微分方程式を差分法で解くときになどに隣接した PE 間でデータ転送が必要になる領域で，次節で詳しい説明を行う．

2 次元の template を用いた GM から LM へのデータ転送を行うライブラリ関数 pg_memCpyG2P_W_2D を図 5 に示す．g_a は GM 上の転送したいデータの開始アドレスを，g_ldim は GM を 2 次元に認識した時の x 方向の幅を，dx 及び dy はそのうち転送したい x 方向の幅及び y 方向の幅を示す．tid で使用する template を指定し，px 及び py で転送先の開始インデックスを指定する．転送先のインデックスの x 方向の幅及び y 方向の幅は dx，dy で指定される．l_a は LM 上の転送先の開始アドレスを，l_ldim は LM を 2 次元に認識した時の x 方向の幅を示す．pg_memCpyG2P_W_2d を実行すると GM 上の赤い四角で囲まれた領域のデータが PE の赤い四角で囲まれたインデックスで指定される PE の LM にコピーされる．GM から各 PE へ，PE ごとに template に基づいた異なるデータを転送したい場合はこのライブラリ関数を用いて行う．

次に 2 次元の template を用いた袖領域の更新を行うライブラリ関数 pg_doReflect_W_2D を図 6 に示す．GM か

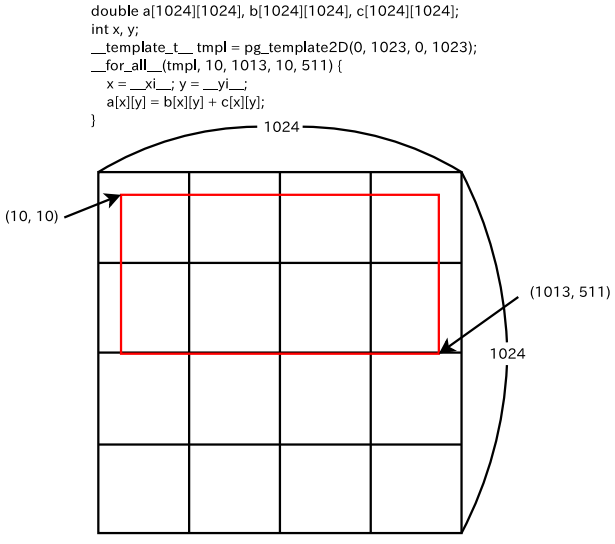

図 4 template の作成及び template ループ文の実行例

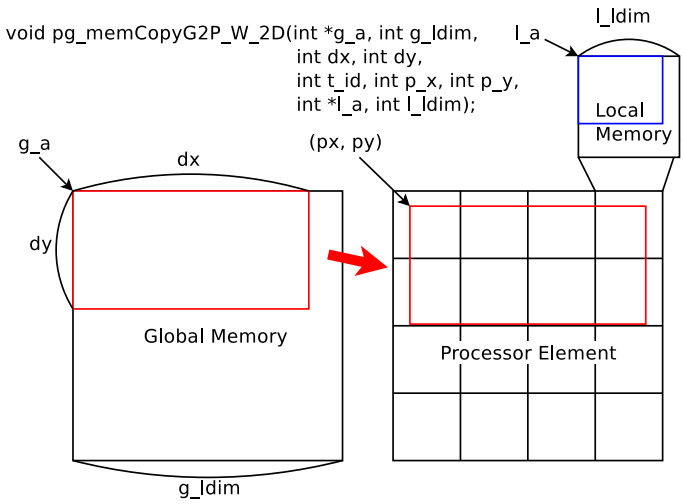

図 5 template を用いたデータ転送

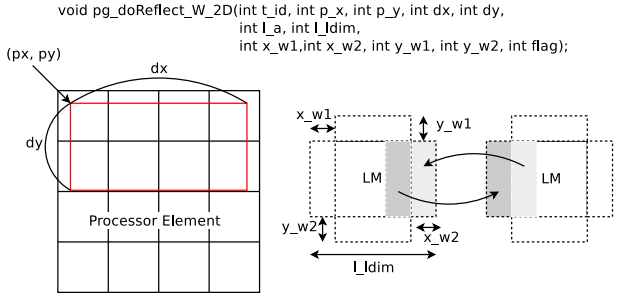

図 6 template を用いた袖領域の更新

ら LM へのデータ転送と同様に，tid は使用する template を，px 及び py は転送先の開始インデックスを，dx 及び dy は転送先のインデックスの x 方向の幅及び y 方向の幅を示す．l_a は袖領域の更新を行いたいデータが存在する LM の開始アドレスを，l_ldim は袖領域を含めた LM の x 方向の幅を示す．袖領域自体は x_w1，x_w2，y_w1，y_w2 によって指定する．pg_doReflect_W_2D を実行すると，図の様に赤い四角形のインデックスの範囲内で，右の PE の薄い灰色の袖領域が左の PE の薄い灰色の領域に，左の PE の濃い灰色の袖領域が右の PE の濃い灰色の領域にそれぞれ PE の隣接間通信を用いてコピーされる．

5.5 プログラムの例

拡張 C を用いたプログラムの例として，図 7 に 2 次元のラプラス方程式の差分法での解法，及びその実行フローを示す．差分法での解法では 2 次元にまたがる各格子点に対して，上下左右の隣接点の値から中心の格子点の値を更新するという計算を行う．2 次元の格子データを 2 次元で分割して各 PE に割り当てたとき，各 PE 内の辺部分の格子点の値を計算するために隣接 PE の値が必要となる．このような隣接 PE の格子点の値を格納する領域を袖領域と呼び予め確保しておく．差分法のプログラムでは 1 度の格子点の値の計算ごとにこの袖領域を隣接 PE の計算結果で更新する必要がある．

main() 関数では格子点の値の初期化や GM 及び LM のメモリのアロケート，GM と BM 間のデータ転送など，主に MP で実行されるプログラムが記述されている．74 行目から 79 行目で境界を含めたデータ領域に対して template を作成し，template から各 PE に割り当てられるデータサイズを計算しメモリの確保を実行する．84，85 行目で template を用いて GM に存在する各格子点の初期値を 2 次元に分割して各 PE の LM へ転送し，87 行目で計算を実行するサブルーチンを呼び出す．PE の実行が終了すると 89 行目で結果を 84，85 行目とは逆に LM から GM に転送する．

lap_iter() 関数では PE での計算の実行が記述されている．16 から 21 行目の__for_all__構文で境界領域を除いた全格子点でデータの更新を行い，23 行目で PE 間の隣接通信を用いて袖領域の更新を行う．25 から 30 行目の__for_all__構文で 16 から 21 行目と同様に境界領域以外の全格子点の値を隣接している 4 格子点のデータから計算する．lap_iter() 関数は修飾子が__global__なので全てが PE 上で実行される訳ではなく，for 文の制御は MP で行われる．lap_iter() 関数内で MP が実行するべき処理が全て終わると，MP は PE の実行が終了するまで待機する．PE の実行が終了すると main() 関数に戻り，前述した処理を行った後プログラムを終了する．

6. 評価環境

性能を評価する環境として PACS-G のシミュレータを拡張したものを用いる．これ以降 PACS-G のシミュレータを PACSG_sim，拡張したシミュレータを PACSG-MP_sim と呼ぶ．PACSG_sim はサイクルレベルで Extreme SIMD アーキテクチャのシミュレーションを実行するものだが，MP に相当するモジュールがなかったため PACSG_sim と GDB の ARM™ プロセッサを結合することでより詳細なシミュレーションを可能にしたものが PACSG-MP_sim である．PACSG-MP_sim のブロック図を図 8 に示す．破線で囲まれた範囲は PACSG_sim を示す．GDB を用いた MP は PACSG_sim に対して PE にプログラム実行開始の指示と，GM，BM 間の DMA の起動のみを行う．また PACSG-MP_sim にはシーケンサーに当たる部分は存在せず，PE 部の起動を行うと GDB とは別に PE 部が実行バイナリを読み込み指定されたアドレスから実行を開始する．現状では拡張 C で書かれたプログラムを PACSG-MP_sim で実行できるバイナリに変換できるコンパイラを開発中のため，出力されるであろう命令列を直接アセンブラで書いたプログラムでの評価を行う．

7. 評価

PACSG-MP_sim 上で 2 次元のステンシル計算であるラ

```
10 __global__ void lap_iter(double *_u, double *_uu, int niter,
11                          int wx, int wy, __template_t__ tmpl) {
12
13   int i, j, k;
14   int x, y;
15
16   for(i = 0; i < niter; i++) {
17     __for_all__(tmpl; 1:XSIZE; 1:YSIZE) {
18       x = __xi__;
19       y = __yi__;
20       _uu[(y+1)*wx + x+1] = _u[(y+1)*wx + x+1];
21     }
22
23     pg_doReflect_D_2D(tmpl, 1, 1, XSIZE, YSIZE, _uu+wx+1, wx, 1, 1, 1, 1);
24
25     __for_all__(tmpl; 1:XSIZE; 1:YSIZE) {
26       x = __xi__;
27       y = __yi__;
28       _u[(y+1)*wx + x+1] = (_uu[y*wx + x+1] + _uu[(y+1)*wx + x] +
29                             _uu[(y+1)*wx + x+2] + _uu[(y+2)*wx + x+1]) / 4.0;
30     }
31
32     __do_all__{ sum = 0.0;}
33     __for_all__(tmpl; 1:XSIZE; 1:YSIZE) {
34       x = __xi__;
35       y = __yi__;
36       sum += _uu[(y+1)*wx + x+1] - _u[(y+1)*wx + x+1];
37     }
38     pg_reduce_sum_double(&sum);
39   }
40 }
41
42 int main() {
42
43   __template_t__ tmpl;
44   int g_ldim, l_ldim;
45   int x_blk_size, y_blk_size;
46   int blk_size;

   ～～～～ 中略 ～～～～

74   tmpl = pg_template2D(0, XSIZE+1, 0 ,YSIZE+1);
75   x_blk_size = pg_templateBlockSize(tmpl, 0);
76   y_blk_size = pg_templateBlockSize(tmpl, 1);
77   blk_size = (x_blk_size + 2) * (y_blk_size + 2);
78   u = (double*)pg_pe_malloc(blk_size*sizeof(double));  // PE内のローカルメモリの確保
79   uu = (double*)pg_pe_malloc(blk_size*sizeof(double));
80
81   g_ldim = XSIZE + 2;
82   l_ldim = x_blk_size + 2;
83   /* transport GM data to PE */
84   pg_memCopyG2P_D_2D(ug, g_ldim, XSIZE+2, YSIZE+2, tmpl, 0, 0, u+l_ldim+1, l_ldim);
85   pg_memCopyG2P_D_2D(ug, g_ldim, XSIZE+2, YSIZE+2, tmpl, 0, 0, uu+l_ldim+1, l_ldim);
86   /* calculation */
87   lap_iter(u, uu, niter, x_blk_size+2, y_blk_size+2, tmpl);
88   /* transport PE data to GM */
89   pg_memCopyP2G_D_2D(uug, g_ldim, XSIZE+2, YSIZE+2, tmpl, 0, 0, u+l_ldim+1, l_ldim);
90   pg_pe_free(uu);
91   pg_pe_free(u);


   ～～～～ 後略 ～～～～
```

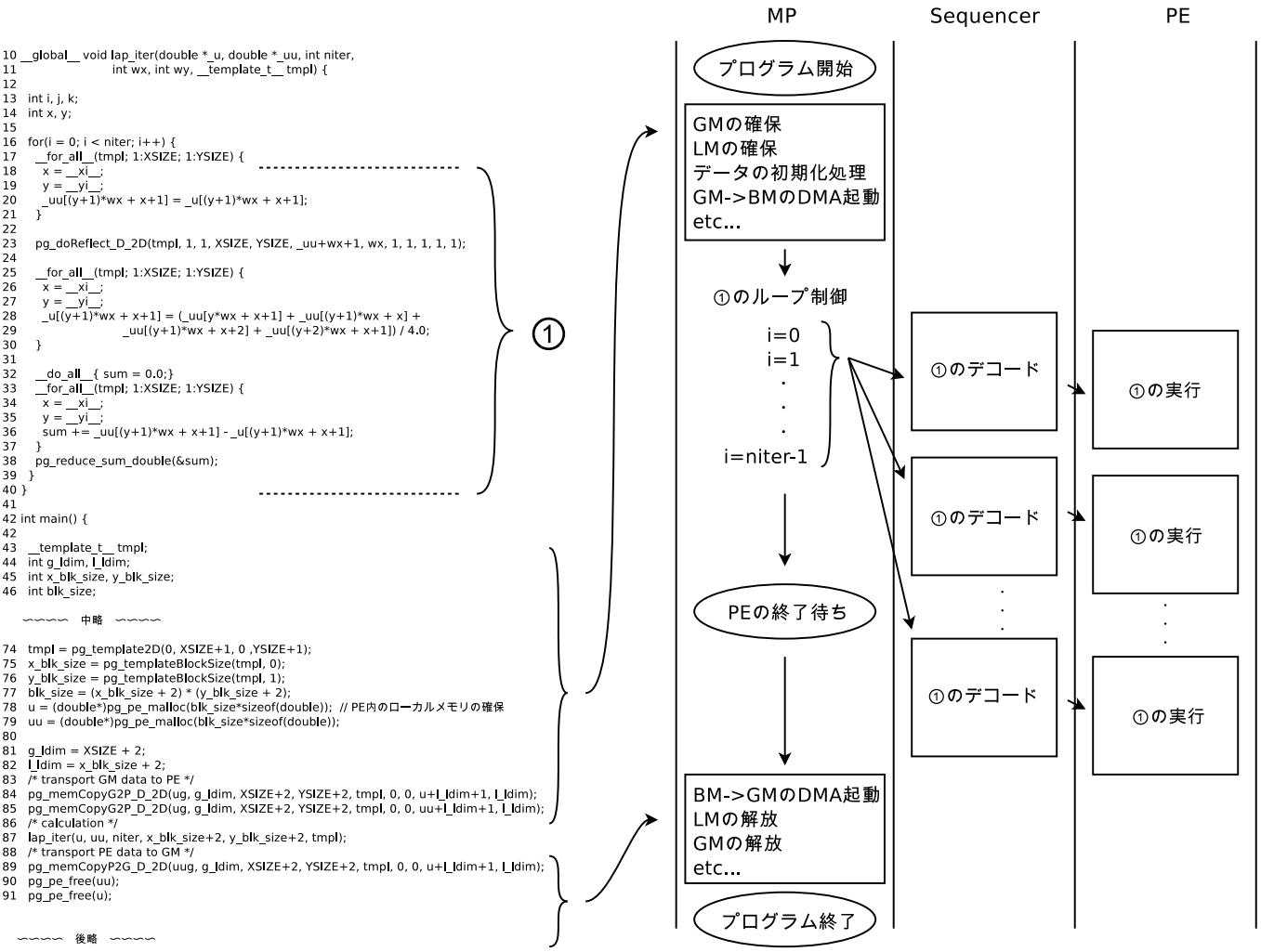

図 7 ラプラス方程式の解法及びその実行フロー

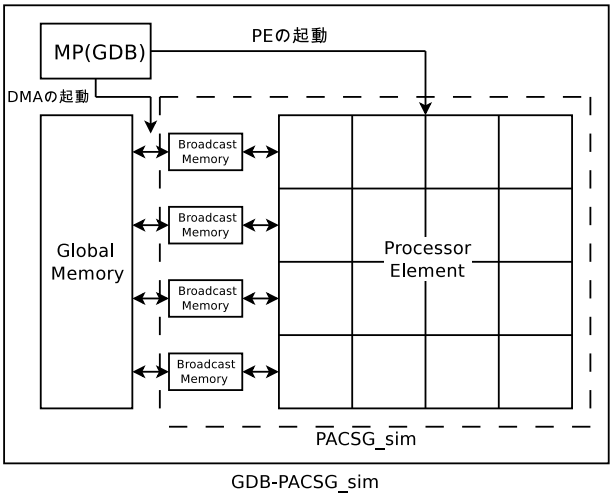

図 8 PACSG-MP_sim のブロック図

プラス方程式の差分法での解法を実行し，評価を行う．

7.1 シミュレーション環境の構成

表 1 に評価を行った Extreme SIMD アーキテクチャの構成を示す．type A はチップ当たりの PE 数が多い代わりに PE 当たりの LM サイズが小さい．PE 数が多い分周波数を低くして電力の削減を図る．type B はこれに対して PE 数が少ない代わりに PE 当たりの LM サイズが大きい．PE 数が少ない分周波数を高くして性能の向上を図る．チップ当たりの LM サイズは type A，type B ともに同一である．

表 1 Extreme SIMD アーキテクチャの構成

	type A	type B
MP アーキテクチャ	ARM^{TM}32bit	ARM^{TM}32bit
PE 数	4096	2048
PE 周波数	0.75GHz	1.0GHz
LM サイズ/PE	64KB	128KB
LM サイズ/chip	256MB	256MB
BM サイズ	128KB	128KB
BM LM 間バンド幅	8GB/s	8GB/s
GM BM 間バンド幅	1024GB/s	512GB/s
ピーク性能/chip	12TFlops	8TFlops

7.2 計算速度及び実効効率

これらの構成に対してステンシル計算のプログラムを

実行した際の初期値の転送を省いた計算速度とピーク性能に対する計算速度を図 9 に示す．棒グラフが計算速度を，線グラフが実効効率を示す．type A，type B ともに 2048 × 2048 格子までは順調に計算速度が伸びているのがわかる．2048 × 2048 格子までで type A と type B を比較すると type A が 2048 × 2048 格子で最大約 8.5TFlops と，計算速度が速いのに対して，type B は 2048 × 2048 格子で実効効率約 74%とピーク性能に対する計算速度が高くなっている．type A が計算速度が速いのは PE 数の多さからみると当然の結果と言える．type B が実効効率が高いのは，type B は type A に比べて LM サイズが大きく各 PE が計算する格子数が 2 倍となり計算量は 2 倍になるが，2 次元に配置された格子のうち袖領域の更新が必要な辺にあたる格子の数は 1.5 倍にしかならない．そのため type A に比べて計算時間が全体に占める割合が高くなるためである．4096 × 4096 格子以上の問題サイズになると急激に計算速度が低下し，実効効率が 1 割以下になっていることがわかる．これは 4096 × 4096 格子以上になると PE 内の LM にデータが入りきらなくなり，一ステップ計算する毎に GM と LM の間でデータのやり取りをしなければならなくなるためである．

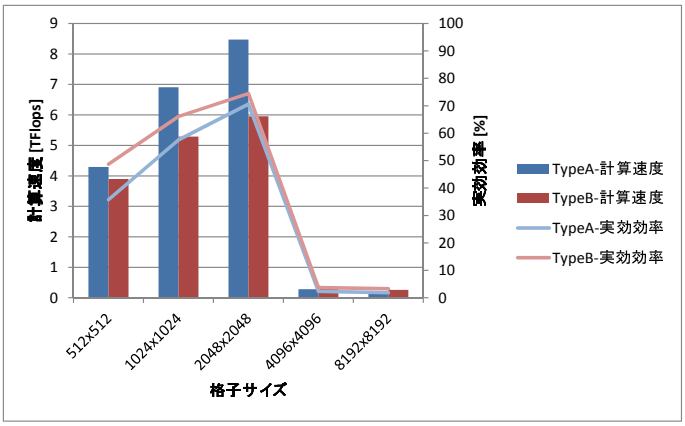

図 9 ラプラス方程式の解法における type A と type B の計算速度及びその効率

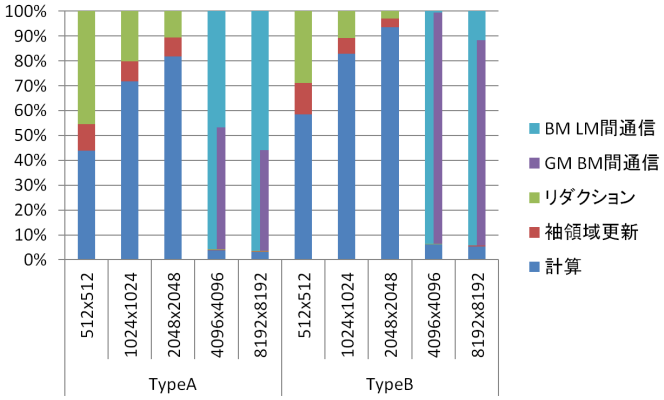

図 10 ラプラス方程式の解法における type A と type B の実行時間の内訳

7.3 実行時間の内訳

次に，各問題サイズでの実行時間の内訳を図 10 に示す．この図は一ステップの計算の実行時間のうち計算を実行している時間，隣接間通信を行っている時間，リダクションを行っている時間，GM と BM 間で通信を行っている時間，BM と LM 間で通信を行っている時間を割合で示したものである．2048 × 2048 格子までの問題サイズではステップごとに GM と通信する必要がないため全体に対して計算が占める割合が非常に高くなっている．type A と type B を比較すると type B の方が全体に占める計算時間が長い．これは前述した通り，type A より type B の方が必要な計算量に対して袖領域の更新に必要な通信量が少ないためである．

一方で 4096 × 4096 格子以上では GM-LM 間通信，BM-LM 間通信が 9 割以上を占める．これは前述した通り一ステップの計算ごとに GM と LM の間でデータのやりとりをしなければならないためである．type A では一度の計算は LM に入る 2048 × 4096 格子で行うために，計算ごとに 2048 × 4096 × 2 × 8 = 128MB の転送が必要となる．BM-LM 間のバンド幅は全体で 512GB/s なので転送に必要なクロック数は約 20 万クロック，一度の計算にかかるクロックは約 5 千クロックなので，約 40 倍計算に比べて転送に時間がかかることになる．同様に type B では約 25 倍計算に比べて転送に時間がかかる．そのため急激に計算速度が下がっている．これ以降，格子数を増やしても一度に PE で計算出来る格子数及びそのデータを転送するためにかかる時間は変化しない．そのため 8192 × 8192 格子以上の問題サイズでは性能が一定になると予測される．

また，GM-BM 間の通信時間のグラフが BM-LM 間の通信時間のグラフに重ねて表示しているのは，一度の転送量を BM のサイズの半分以下にしてダブルバッファリングすることで GM-BM 間と BM-LM 間の通信時間をオーバーラップしているためである．今回の構成では GM-BM 間のバンド幅が BM-LM 間のバンド幅より広いか，同等なので GM-BM 間の通信時間をほぼ全て BM-LM 間の通信で隠蔽している．4096 × 4096 格子では，type A では GM-BM 間のバンド幅が BM-LM 間のバンド幅に比べて 2 倍あるので BM-LM 間に比べて GM-BM 間の通信時間は約半分程度になる．type B では GM-BM 間のバンド幅が BM-LM 間のバンド幅と同等なのでほぼ同じ通信時間となる．type A，type B ともに 4096 × 4096 格子にくらべて 8192 × 8192 格子では GM-LM 間の通信時間の比率が減少しているのは，8192 × 8192 格子では袖領域を一時的に BM に格納しているのでその転送のために BM-LM 間の通信時間が伸びるためである．

ⓒ 2015 Information Processing Society of Japan

8. 結論と今後の課題

2 次元のステンシル計算を対象として Extreme SIMD アーキテクチャのシミュレータによる評価を行った．結果として LM に入りきる 2048 × 2048 格子までの問題サイズでは Extreme SIMD アーキテクチャは高い実効効率を示すことが分かった．このような問題サイズで type A と type B を比較すると type A が計算速度が速いことが確認できたが，実効効率の観点から言うと type B が優れているということが分かった．

一方で 4096 × 4096 格子以上の問題サイズでは急激な性能低下が起こった．これは一ステップごとに GM と LM の間で通信が発生するためである．GM-BM 間の通信は BM-LM 間の通信でオーバーラップ可能だが，元々の通信時間が長いため，メモリ律速になることは変わらない．BM-LM 間の通信時間が長い一因として 1 つの BM を複数の PE で共有して使うことがあげられる．そのため BM-LM 間のバンド幅が接続されている PE の数だけ分割されてしまい，速度の低下を起こす．

一つの解決策としては BM の数を増やして 1 つの BM に接続される PE の数を減らす方法がある．特に type A では GM-BM 間のバンド幅が BM-LM 間のバンド幅の 2 倍あるので，BM 数を増やすことでこの差を調整することが出来る．ただし type B では GM-BM 間のバンド幅と BM-LM 間のバンド幅が同等なので，BM を増やすと GM-BM 間のバンド幅が BM-LM 間のバンド幅より少なくなってしまい GM-BM 間の通信時間がボトルネックになるだろう．最適な解は GM として実装される HBM もしくは HMC のバンド幅と，BM-LM 間のバンド幅が同等程度になる BM 数に調整することで，それぞれの通信時間が同程度になるようにすることである．

今後の課題としては，シミュレーション精度を上げるために MP に使用するシミュレータをサイクルレベルでのシミュレーションを行えるものに変更することがあげられる．本稿では GDB の ARM^{TM} シミュレータを MP として扱ったが，GDB の ARM^{TM} シミュレータはサイクルレベルでのシミュレーションは行っていない．サイクルレベルでシミュレーションできるものに gem5 [6] と呼ばれる汎用プロセッサのシミュレータが存在するので，これを MP として使用することを考えている．また，今回評価した Extreme SIMD アーキテクチャの比較対象として，キャッシュを持つ様な汎用マルチコアプロセッサ及びメニーコアプロセッサに対してシミュレーションも現在行っているため，その結果との比較を行い Extreme SIMD アーキテクチャの利点，欠点を明らかにすることもあげられる．

謝辞 本研究の一部は，平成 24-25 年度 文部科学省ＨＰＣＩ構築事業将来のＨＰＣＩシステムのあり方の調査研究「演算加速機構を持つ将来の HPCI システムに関する調査研究の研究開発」による．

参考文献

[1] 宮崎博行, 草野義博, 新庄直樹, 庄司文由, 横川三津夫, 渡邊貞, “スーパーコンピュータ「京」の概要”, FUJITSU, Vol.63, No.3, pp.237-246, May 2012
[2] The Green 500 List, “The Green 500 List - November 2014”, http://www.green500.org/news/green500-list-november-2014
[3] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, Doug Burger, “Dark silicon and the end of multicore scaling”, ACM SIGARCH Comp. Arch. News, Vol. 39, Issue 3, pp.365-376, June 2011
[4] 児玉 祐悦, 山口 佳樹, 中里 直人, 牧野 淳一郎, 朴 泰祐, 佐藤 三久, “大規模 SIMD 型アクセラレータの検討”, SWoPP 2013, August 2013
[5] J.Makino, K.Hiraki, M.Inaba, “GRAPE-DR: 2-Pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing”, SC’07, Proc. 2007 ACM/IEEE Conf., pp.1-11, Nov. 2007
[6] N.Binkert, B.Beckmann, G.Black, S.K.Reinhardt, A.Saidi, A.Basu, J.Hestness, D.R.Hower, T.Krishna, S.Sardashti, R.Sen, K.Sewell, M.Shoaib, N.Vaish, M.D.Hill, D.A.Wood, “The gem5 simulator”, ACM SIGARCH Comp. Arch. News, Vol.39, Issue 2, pp.1-7, May 2011

ⓒ 2015 Information Processing Society of Japan