Vol.2016-HPC-154 No.3
2016/4/25

PEZY-SC向けOmni OpenACCコンパイラの設計・試作

田渕 晶大[1,a)]　木村 耕行[2]　鳥居 淳[2]　松古 栄夫[3]　石川 正[3]　朴 泰祐[1,4]　佐藤 三久[1,5]

概要：電力当たりの性能が重要視される中，低消費電力のアクセラレータとして PEZY-SC が注目されている．PEZY-SC のプログラミングには OpenCL をベースとした PZCL が提供されているが，その記述は煩雑で生産性が低い．そこでアクセラレータ向けの指示文ベースプログラミングモデルである OpenACC のコンパイラを PEZY-SC 向けに設計・試作する．Suiren Blue（青睡蓮）を用いた評価では，OpenACC コードは PZCL コードと比較して N-Body では 98%以上，NPB CG では最大 88%の性能が得られた．また OpenACC は指示文を用いた簡潔な記述により PZCL の半分以下のコード行数で実装できたことから，高い生産性と十分な性能を達成できた．

1. 序論

スーパーコンピュータにおける電力供給には制約があることから，電力効率を高めるためにアクセラレータを利用することが広く行われている．スーパーコンピュータの電力当たりの性能のランキングである Green500[1] では，2015 年 11 月時点で上位 10 システムのすべてがアクセラレータを搭載している．その 1 位であるのが理研の Shoubu(菖蒲) であり，そのシステムは日本の ExaScaler により開発されたものである．そのシステムではアクセラレータとして PEZY-SC プロセッサが用いられており，1024 個のコアで最大 8192 スレッドを MIMD で実行することができる．ExaScaler は PEZY-SC 用のプログラミング環境として PZCL を提供している．PZCL は OpenCL[2] をベースとしているが，カーネルコードの記述が独自のものとなっている． PZCL を用いて PEZY-SC 向けのアプリケーションを開発するには，データや計算のオフロードのために多数のコードを記述する必要があるためプログラミングのコストが高いという問題がある．

ここ数年，アクセラレータのプログラミング方法として OpenACC[3] が注目されている．OpenACC はアクセラレータ向けの指示文ベースのプログラミングモデルであり，逐次プログラムから簡易にデータと計算をアクセラレータにオフロードすることが可能である．いくつかの商用コンパイラやリサーチコンパイラが主に GPU を対象として OpenACC をサポートしている．PEZY-SC でも OpenACC をサポートすることにより，迅速なアプリケーション開発や既存の OpenACC コードの再利用が可能になり，ユーザが PEZY-SC を利用しやすくなると期待できる．

以上の背景に基づき，我々は PEZY-SC 向けの Omni OpenACC コンパイラ [4] を設計・試作する．Omni OpenACC コンパイラは著者らが開発した元々は NVIDIA GPU を対象に CUDA[5] へ source-to-source 変換を行う OpenACC コンパイラである．N-body と NAS Parallel Benchmarks CG (NPB CG)[6] を用いて試作したコンパイラの性能を評価すると共に，PZCL と OpenACC の記述性や生産性を評価する．

本稿は，次に示す構成となっている．2 章では関連研究を紹介する．3 章では PEZY-SC プロセッサと PZCL によるプログラミング方法を説明する．4 章では PEZY-SC 向け Omni OpenACC コンパイラの設計と実装を解説し，5 章でベンチマークによる評価を行う．6 章で結論と今後の課題を述べる．

2. 関連研究

PEZY-SC を用いた計算科学アプリケーションの性能評価が中里により行われている [7]．既存の OpenCL コードをベースに実装が行われており，条件コンパイルやマクロを利用することで OpenCL と PZCL のコードを共通化してい

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 株式会社 ExaScaler
ExaScaler Inc.
[3] 高エネルギー加速器研究機構 (KEK) 計算科学センター
Computing Research Center, High Energy Accelerator Research Organization (KEK)
[4] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[5] 国立研究開発法人理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science
[a)] tabuchi@hpcs.cs.tsukuba.ac.jp

る．そのため，OpenCL から PZCL へのコード移植はさほど困難ではないと考えられる．本研究では，ベースとなる OpenCL コードがない場合でも逐次コードから OpenACC で簡易に開発可能にし，さらに既存の OpenACC コードをそのまま利用できるようにすることで，生産性の向上を目指す．

accULL[8]，OpenUH-OpenACC[9]，OpenARC[10]，RoseACC[11] 等のオープンソースの OpenACC コンパイラが開発されている．また GCC でも OpenACC への対応が進められている [12]．accULL は Python ベースの YaCF というソース・プログラム変換を行うコンパイラフレームワークを用いてコードを最適化し CUDA または OpenCL コードに変換する．OpenUH-OpenACC はコンパイラフレームワーク OpenUH を用いた OpenACC コンパイラで CUDA や OpenCL コードへの変換を行う．OpenARC はアクセラレータ向けコンパイラフレームワークであり，オープンソースでは初めて OpenACC1.0 の全機能をサポートした．Cetus compiler infrastructure を元に作られており，様々なコード解析や変換の機能が備わっている．RoseACC はオープンソースコンパイラである Rose Compiler ベースの OpenACC コンパイラで，OpenCL コードへの変換を行う．これらのコンパイラは主に GPU を対象としている．また OpenCL に変換できるものは CPU や MIC や FPGA など様々なデバイスでの動作が可能であると考えられる．しかしながら，PEZY-SC で用いる PZCL は OpenCL とは異なるため，これらのコンパイラは利用できない．本研究では Omni OpenACC コンパイラにおいてコードを PZCL へ変換する実装を行い，PEZY-SC にも対応する．

3. PEZY-SC

本章では PEZY-SC のプロセッサとそのプログラミング環境である PZCL について説明する．

3.1 プロセッサ

PEZY-SC プロセッサは PEZY 社が開発したメニーコアプロセッサである．プロセッサの構成を図 **1** に示す．1 プロセッサ内に 1024 個の Processing Element (PE) があり，それらが MIMD で動作する．また各 PE では 8 スレッドが SMT (Simultaneous MultiThreading) で動作するため，全体では 8192 スレッドが MIMD で動作することになる．プロセッサの要素は小さい方から順に PE，Village，City，Prefecture によって階層的に構成されている．PE は 8 スレッド分のレジスタ，16KB のローカルメモリ，2 つの ALU と FPU 等から構成される．Village は 4 つの PE と 2 つの PE 毎の 2KB の L1 キャッシュで構成される．City は 4 つの Village と特殊関数 Unit (SFU) と 64KB の L2 キャッシュからなる．Prefecture は 16 個の City と 2MB の L3 キャッシュからなる．1 プロセッサにはこの Prefecture が 4 つあり，PE は全部で 1024 個となる．

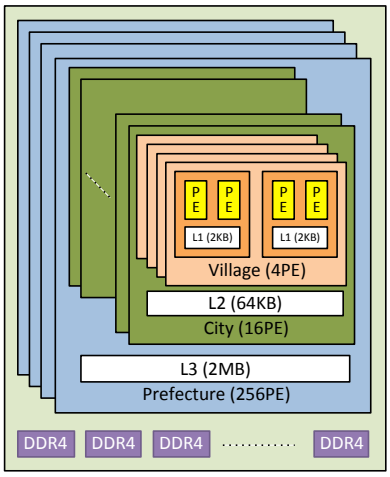

図 1　PEZY-SC プロセッサの構成

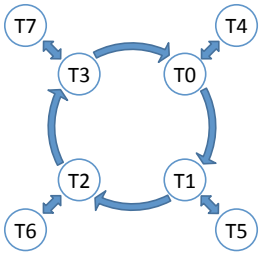

図 2　スレッドの表裏

PE 内の 8 スレッドは実際には図 **2** のように 4 つのスレッドの表と裏で構成されている．通常は片側の 4 スレッドのみがクロック単位でラウンドロビンで実行される．同期や明示的に表裏の切り替えが行われた際には，反対側が実行されるようになる．

L1 から L3 まであるキャッシュにはコヒーレンシがない．そのため，ある PE が他の PE が書き込んだ値を読むためにはキャッシュをフラッシュする必要がある．

3.2 PZCL

PZCL は PEZY 社が提供している PEZY-SC プロセッサ用のプログラミング環境である．OpenCL をベースとしたものであるが，特にデバイスカーネルの記述が OpenCL のものとは異なる．

3.2.1 ホストコード

ホストコードに用いる API は OpenCL 1.1 のサブセットである．コンテキスト，コマンドキュー，メモリオブジェクトなど，一般的な OpenCL コードが用いるであろう API の大部分は使用可能である．大きな違いとしては，カーネルのビルドと起動方法が挙げられる．一般的な OpenCL の実装では実行時にカーネルをコードからコンパイルするオンラインコンパイルを用いることが多いが，PZCL では事前にカーネルコードをコンパイルし，実行時にカーネルバ

イナリをロードするオフラインコンパイルにのみ対応している．またカーネルの起動時にはスレッドの起動が City 単位 (128 スレッド) であるためグローバルワークサイズが 128 の倍数でなければならず，さらに各 PE のスレッド数は 8 で固定なのでローカルワークサイズは指定できないという制約がある．さらに OpenCL ではワークアイテムの次元は少なくとも 3 次元まで指定できるが，PZCL では 1 次元のみしか指定できない．なおグローバルワークサイズが 8192 を超える時はカーネルが複数回起動されることになるため，8192 以下にすることが望ましい．

3.2.2 デバイスコード

デバイスコードは C/C++で記述する．OpenCL と同様にホストから起動するカーネルは関数として記述する．ただし OpenCL では関数に`__kernel` 修飾子を付けたり，引数に`__global`，`__local` を付けたりするが，PZCL では関数名のプレフィックスを “pzc_” とするだけで，引数には何も付けなくて良い．その代わりに PZCL では PE 内のスレッドで共有するメモリ（OpenCL でのローカルメモリ相当）を定義することができない．関数内では PE やスレッドの ID を用いて処理を分担させることで並列化する．プロセスやスレッドの ID はそれぞれ get_pid()，get_tid() で求められる．これらは OpenCL における get_group_id(0)，get_local_id(0) に相当する．また PE 数やスレッドの数はそれぞれ get_maxpid()，get_maxtid() で求められる．これらは OpenCL における get_num_groups(0)，get_local_size(0) に相当する．

PZCL の独自の機能としては，chgthread()，sync()，flush() が挙げられる．chgthread() はスレッドの表裏を切り替えることができる．sync() はプロセッサ全体のスレッドで同期を行う．また sync_L1() では Village，sync_L2() では City，sync_L3() では Prefecture 単位で同期をとることが出来る．また flush() は全体同期と全てのキャッシュされたデータの掃き出しを行う．また flush_L1() では Village レベルの同期と L1 キャッシュのフラッシュ，flush_L2() では City レベルの同期と L1,L2 キャッシュのフラッシュを行う．これらの命令を用いることで，細かな性能チューニングを行うことが可能である．

4. PEZY-SC 向け Omni OpenACC コンパイラ

本章では PEZY-SC 向けに実装した Omni OpenACC コンパイラの設計と実装の解説を行う．

4.1 設計

OpenACC は C/C++/Fortran を対象としているが，本実装では最も実装の容易な C を対象とする．また最新の仕様は OpenACC 2.5 であるが，本実装では OpenACC 1.0 相当の機能の実装にとどめる．コンパイラでは OpenACC

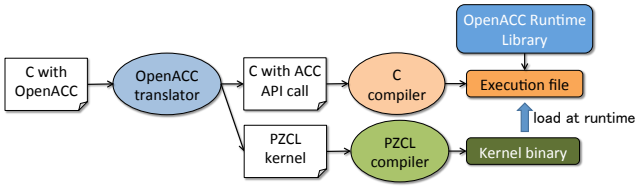

図 3 Omni OpenACC Compiler のコンパイルの流れ

コードから PZCL コードにソースプログラム変換を行い，それを PZCL コンパイラでコンパイルする．この手法には PEZY-SC プロセッサ用のコード生成を PZCL コンパイラに任せられるという利点がある．OpenACC コードからはホスト CPU で実行するコード（ホストコード）とアクセラレータで実行するコード（デバイスコード）を生成する．ホストコードでは OpenACC 指示文を主に実行時ライブラリ呼び出しに変換する．それによりコード変換が容易になる上，GPU 向けと PEZY-SC 向けで処理を共通化できる．デバイスコードでは指示文でオフロードが指定されたコードを並列化した PZCL コードに変換する．こちらの変換でもループ範囲の計算やリダクション等に関数を用いることでコード変換を簡易化・共通化する．

4.2 実装

このコード変換を実現するために，本実装では C や Fortran95 のコードの解析や変形が可能である Omni Compiler Infrastructure[13] を用いる．コンパイルの流れを**図 3** に示す．OpenACC 指示文のある C コードを OpenACC translator で読み込んでコードを変換し，C のホストコードと PZCL のデバイスコードを出力する．ホストコードは一般的な C コンパイラでコンパイルし，デバイスコードは PZCL コンパイラでコンパイルする．最後にコンパイルされた 2 つのオブジェクトファイルを Omni OpenACC runtime library とリンクして実行可能バイナリを生成する．

4.2.1 data 指示文

`data` 指示文は直後のコード領域におけるアクセラレータ上のデータを宣言するための指示文である．指示節にもとづいて，データ領域の最初にデバイスメモリの確保とホストメモリからデバイスメモリへの転送，データ領域の最後にデバイスメモリからホストメモリへの転送とデバイスメモリの解放を行う．これらの操作はすべて実行時ライブラリ呼び出しに変換する．**図 4** に data 指示文とその変換例を示す．この data 指示文では領域内で配列 a と変数 b のためのデバイスメモリを確保する．また copy 節内で指定されている配列 a は領域の最初と最後で転送し，copyout 節内で指定されている変数 b は領域の最後でのみ転送する．デバイスメモリの確保にはライブラリ関数_ACC_init_data を用いる．この際，配列の場合には各次元の lower と length を指定する．DEV_ADDR_*name* はホスト上の *name* に対応するデバイスメモリのポインタであ

```
int a[100], b;
#pragma acc data copy(a) copyout(b)
{
  /* some codes using a and b */
}
```

(a) 変換前のコード

```
int a[100], b;
{
  void *DESC_a,*DEV_ADDR_a,*DESC_b,*DEV_ADDR_b;
  unsigned long long _lower[] = {0};
  unsigned long long _length[] = {100};
  _ACC_init_data(&(DESC_a),&(DEV_ADDR_a),a,sizeof(int),1
                                          ,_lower,length);
  _ACC_init_data(&(DESC_b),&(DEV_ADDR_b),&(b),sizeof(int),0
                                            ,NULL, NULL);
  _ACC_copy_data(DESC_a,_ACC_HOST_TO_DEVICE,_ACC_ASYNC_SYNC);
  {
    /* some codes using a and b */
  }
  _ACC_copy_data(DESC_a,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
  _ACC_copy_data(DESC_b,_ACC_DEVICE_TO_HOST,_ACC_ASYNC_SYNC);
  _ACC_finalize_data(DESC_a);
  _ACC_finalize_data(DESC_b);
}
```

(b) 変換後のコード

図 4 data 指示文のコード変換例

る．DESC_*name* は *name* のホストアドレス，デバイスアドレス，配列形状，要素サイズ等が格納された構造体へのポインタである．ホストメモリとデバイスメモリ間の転送はライブラリ関数_ACC_copy_data で行う．ライブラリ関数_ACC_finalize_data により最後にデバイスメモリの解放を行う．

4.3 parallel 指示文

parallel 指示文はコード領域をデバイスで並列に実行するための指示文である．オフロードするコード領域をカーネル関数にし，ホストコードからそのカーネルを呼び出すように変換する．

4.3.1 デバイスコード

まずデバイスコードの生成について説明する．OpenACC には gang, worker, vector という 3 種類の並列性があるが，PZCL では PE とスレッドを主に用いる．試作段階では PE を gang に，スレッドを vector に対応付けることにするが，この割り当ては今後変更する可能性がある．

デバイスでの処理はカーネル関数とする必要があるため，オフロード対象のコードから関数を生成する．コード内でアクセスされる変数は，カーネル関数の引数としてポインタで渡す．ただし，firstprivate 節によって初期値をホストから渡す必要があるものに関してはポインタではなく値を引数としてとる．また，private 節によって PE・スレッドごとに持つ必要のある変数は関数内の宣言で定義する．

4.3.2 ホストコード

ホストコードでは生成したカーネル関数を起動するように変換する．関数の引数にはポインタの場合は対応するデバイスメモリオブジェクトを，値の場合はホストでの値を渡す．

```
#pragma acc parallel present(a) num_gangs(16)
{
  /* codes in parallel region */
}
```

(a) 変換前のコード

```
/* host code */
{
  int _ACC_ngangs = 16;
  int _ACC_nworkers = 1;
  int _ACC_veclen = 8;
  int _ACC_conf[] = {_ACC_ngangs, _ACC_nworkers, _ACC_veclen};

  void* _ACC_args[] = {&DEV_ADDR_a};
  size_t _ACC_argsizes[] = {sizeof(void*)};
  _ACC_launch(_ACC_program, 0, _ACC_conf, ACC_ASYNC_SYNC,
            1, args, arg_sizes);
}

/* kernel function in device code */
void pzc__ACC_kernel_0(int *a)
{
  /* codes in parallel region */
}
```

(b) 変換後のコード

図 5 parallel 指示文のコード変換例

PE 数は num_gangs 節で指定可能である．また指定しなければコード中のループの並列数から PE 数を自動で決定する．**図 5** に parallel 指示文とその変換例を示す．関数 pzc__ACC_kernel_0 がカーネル関数である．gang 数が 16 なので 16PE とし，スレッド数は固定で 8 スレッドとする．_ACC_args はカーネル関数の引数で_ACC_argsizes は各引数のサイズである．_ACC_launch がカーネル関数を起動するライブラリ関数である．第 1 引数の_ACC_program は cl_program および複数の cl_kernel オブジェクトが格納された構造体へのポインタである．カーネルバイナリはプログラムの開始時にロードされこの構造体に格納されている．第 2 引数がカーネル番号である．_ACC_launch 内では clEnqueueNDRangeKernel により実際にカーネルを起動する．その際，PEZY-SC に合わせて全スレッド数が 128 の倍数でかつ 8192 以下となるよう調整する．この例では全スレッド数が 128 なので調整は必要ない．

4.4 loop 指示文

loop 指示文はオフロード領域の中で for ループに関して主に並列化を指定するための指示文である．並列化の際は並列性（gang または vector）が指定されていればそれを用い，指定がない場合はコンパイラが自動で使用する並列性を決定する．ループは PE・スレッドともに cyclic で分割される．さらに loop 指示文に reduction 節が付記されている場合は指定された変数用にプライベート変数を用意し，

```
/* inside parallel region */
#pragma acc loop vector reduction(+:sum)
for(i = 0; i < N; i++){
  a[i]++;
  sum += a[i];
}
```

(a) 変換前のコード

```
/* inside kernel function */
int _niter_i, _idx, _init, _cond, _step, _red_sum;
_ACC_init_reduction_var(&_red_sum,0);
_ACC_calc_niter(&_niter_i, 0, N, 1);
_ACC_init_thread_iter(&_init,&_cond,&_step,_niter_i);
for(_idx = _init; _idx < _cond; _idx += _step){
  int i;
  _ACC_calc_idx(_idx, &i, 0, N, 1);
  a[i]++;
  _red_sum += a[i];
}
_ACC_reduction_thread(sum,_red_sum, 0);
```

(b) 変換後のコード

図 6 loop 指示文のコード変換例

ループの終わりにリダクションして元の変数に書き込むように変換する．

図 6 に loop 指示文とその変換例を示す．関数_ACC_calc_niter でループ長を求め，関数_ACC_init_thread_iter で，そのスレッドが担当するイテレーションの初期値・最大値・ステップを求める．その後ループ内では関数_ACC_calc_idx により実際のループ変数の値を求め，ループ本体を実行する．またリダクション部分に関しては，始めに関数_ACC_init_reduction_var によりスレッドローカル変数を初期化し，ループ実行後に関数_ACC_reduction_thread によりスレッド間でローカル変数のリダクションを行う．

5. 評価

本章では，実装した OpenACC コンパイラの性能の評価および PZCL と OpenACC の生産性の評価を行う．ベンチマークとして，重力多体問題 (N-Body) と NPB CG を用いる．N-Body は多数の粒子間の相互作用を計算してその動きをシミュレーションするもので，今回はナイーブに全粒子間の相互作用を計算するアルゴリズムを用いる．NPB CG は正値対称な大規模疎行列の最小固有値を共役勾配法によって解くベンチマークである．

5.1 性能

評価環境として KEK の Suiren Blue（青睡蓮）を用いる．その環境は**表 1** に示すとおりである．まず N-Body の実行時間を**図 7** に示す．凡例の “PZCL” は OpenACC と同数のカーネルを用いたもので，“PZCL (merged kernel)” はメインの 2 つのカーネルを 1 カーネルにマージしたものである．“PZCL (merged kernel, chgthread)” はさらにカーネルに chgthread() を追加したものである．PZCL コードに比べて OpenACC コードは 97.8〜100.0%とほぼ同じ性能であった．また，カーネルのマージや chgthread() の追加による性能の変化は非常に小さかった．N-Body では非常に演算時間の割合が大きいために，それらによる影響が小さかったと言える．

表 1 評価環境 (Suiren Blue)

CPU	Intel Xeon-E5 2618Lv3 2.3GHz
Memory	DDR4 1866MHz, 64GB
Accelerator	PEZY-SC (DDR4 1866MHz 16GB)
Compiler	ICC 14.0.2, PZSDK 2.1, Omni OpenACC compiler for PEZY-SC

次に NPB CG の性能を**図 8** に示す．mop/s（Mega Operations Per Second）は 1 秒当たりに何百万回の演算を行ったかを表す単位である．凡例の “PZCL” は OpenACC と同数のカーネルを用いたもので，“PZCL (merged kernel)” はそのうちの主関数 conj_grad 内の 7 個のカーネルを 1 カーネルにマージしたものである．“PZCL (merged kernel, chgthread)” はさらにカーネルに chgthread() を追加したものである．PZCL コードと OpenACC で同数のカーネルを使用した場合は，91.9〜100.3%の性能が出ている．OpenACC 版で遅くなる要因としては，コンパイラのコード変換によるオーバーヘッドや余分なホスト・デバイス間転送が挙げられる．CG ではすべてのリダクション変数の初期値は 0 であるため，それを計算前にホストからデバイスにコピーする必要はないが，OpenACC 版ではそれを行っているため，コンパイラの改善が必要である．また ClassB において OpenACC の方が性能が上回ってしまった原因については調査中である．

カーネルをマージした PZCL と比較すると，OpenACC は 68.5〜98.8%の性能であった．特に問題サイズが小さい時にカーネルの起動オーバーヘッドの削減の効果が大きく，相対的な性能の低下が著しい．OpenACC においてカーネル数を減らせなかったのはオフロードに `parallel` 指示文を用いたからである．OpenACC にはオフロードのために `parallel` と `kernels` の 2 つの指示文が用意されており，今回用いた `parallel` は指定されたコードが 1 つのカーネルで実行される，明示的な指示文である．仕様上 `parallel` 領域の途中では全体の同期ができないため，全体同期が必要な部分では `parallel` 領域を分割する必要があり，カーネル数を減らすことができなかった．一方 `kernels` は指定されたコード領域をコンパイラが適宜分割して 1 つ以上のカーネルで実行して良い指示文である．PZCL では実行中でも sync() により全体同期を行うことができるため，`kernels` 領域は 1 つのカーネルに変換が可能である．現在のコンパイラの実装では `kernels` を用いた場合でも GPU 用と同じようにカーネルが分割されてしまうため，PZCL

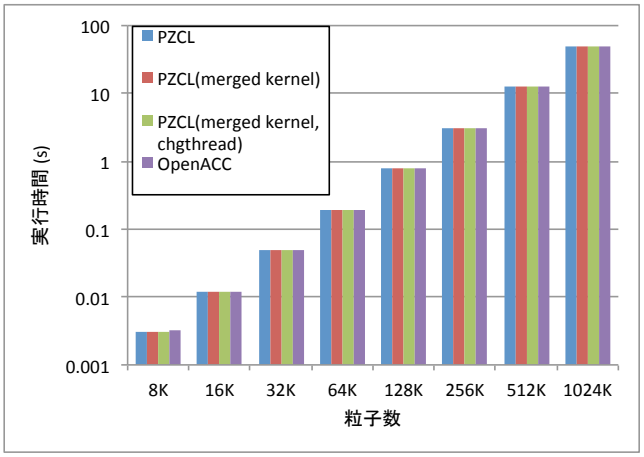

図 7　N-Body の計算時間

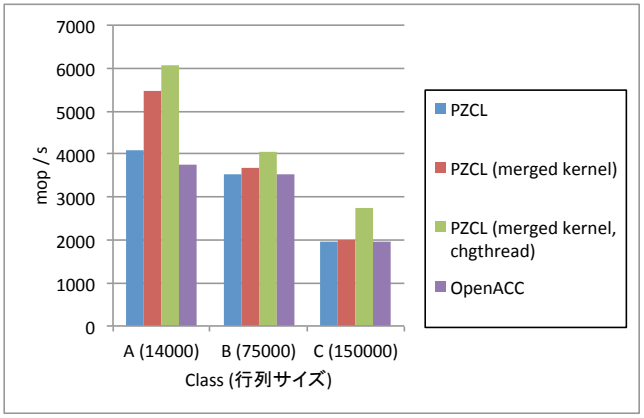

図 8　NPB CG の性能

用に修正が必要である．

chgthread() を使用した PZCL コードでは用いない PZCL コードと比較して，11〜38%高速化された．CG ではメモリアクセスが多いため，chgthread() によって表裏の反対側のスレッドがキャッシュを有効利用できたため性能が向上したと考えられる．この PZCL 版と比較すると OpenACC 版は 61.6〜87.5%の性能であった．PZCL 版の性能に近づけるために，OpenACC コンパイラにおいても `kernels` 指示文でのカーネルのマージや chgthread() の利用による最適化が必要である．

5.2　生産性

PZCL ではカーネルの作成や API によってデバイスメモリの管理・カーネルの起動等を行う必要があるのに対して，OpenACC では逐次コードに指示文を追加するだけでデータと処理を容易にオフロード可能である．また OpenACC は標準規格であるため，既に OpenACC で記述されたコードを実行したり，開発したコードを他の環境で実行可能である．

また生産性を定量的に測るために，コード行数 (SLOC) を数えた．N-Body，NPB CG のそれぞれのコード行数を**表 2** に示す．PZCL と比較して，OpenACC では N-Body

表 2　N-Body と NPB CG のコード行数．内数は指示文の行数

	N-Body	NPB CG
逐次	109	418
PZCL	245	1064
PZCL(merged kernel)	238	989
PZCL(merged kernel, chgthread)	240	1001
OpenACC	114 (5)	447 (25)

が 48%，NPB CG が 45%と非常に少ない行数で実装できており，ほとんど逐次コードと変わらないことが分かる．以上のことから OpenACC を用いることで PZCL より簡易にコード開発が可能であると言える．

6.　結論と今後の課題

本稿では PEZY-SC におけるアプリケーション開発の生産性向上を目的とし，指示文による簡易な記述が可能な OpenACC のコンパイラを PEZY-SC 向けに設計・試作した．実装には著者らが開発した NVIDIA GPU を対象に CUDA へコード変換を行う Omni OpenACC コンパイラをベースに用い，OpenACC 指示文の書かれた C コードから PEZY-SC のプログラミング環境である PZCL へコード変換するコンパイラを実装した．評価では，N-Body では OpenACC コードで PZCL コードの 98%以上の性能が得られ，NPB CG では最大で PZCL コードの 88%の性能が得られた．生産性の点では，OpenACC は逐次コードに指示文を追加するのみでオフロードが可能であり，コード行数は PZCL と比較して N-Body で 48%，NPB CG で 45%と非常に少なく済むことから，高い生産性を実現できたと言える．

今後の課題は，性能向上のためのコンパイラの改善である．まず `kernels` 指示文でのカーネルのマージやリダクション変数の初期値の転送の削減といった改善を行う予定である．また chgthread() によるスレッドの切り替えでキャッシュを有効利用する最適化も検討する．

参考文献

[1] The green500. `http://www.green500.org`.
[2] Khronos Group, `https://www.khronos.org/opencl/`. *OpenCL - The open standard for parallel programming of heterogeneous systems.*
[3] OpenACC-Standard.org, `http://www.openacc.org`. *OpenACC Home.*
[4] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source openacc compiler for cuda. In *Euro-Par Workshops*, pp. 178–187, 2013.
[5] NVIDIA, `http://www.nvidia.com/object/cuda_home_new.html`. *Parallel Programming and Computing Platform — CUDA.*
[6] NASA Advanced Supercomputing Division, `http://www.nas.nasa.gov/publications/npb.html`. *NAS Parallel Benchmarks.*
[7] 中里直人. Suiren（睡蓮）による計算科学アプリケーショ

ンの性能評価. 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], No. 11, dec 2015.

[8] Ruymán Reyes, Iván López-Rodríguez, JuanJ. Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. In *Euro-Par 2012 Parallel Processing*, Vol. 7484 of *Lecture Notes in Computer Science*, pp. 871–882. Springer Berlin Heidelberg, 2012.

[9] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling a high-level directive-based programming model for gpgpus. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 105–120. Springer International Publishing, 2014.

[10] Seyong Lee and Jeffrey S. Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pp. 115–120, New York, NY, USA, 2014. ACM.

[11] University of Delaware and LLNL, `http://roseacc.org/`. *RoseACC*.

[12] GCC, `https://gcc.gnu.org/wiki/OpenACC`. *OpenACC - GCC Wiki*.

[13] RIKEN AICS and University of Tsukuba, `http://omni-compiler.org`. *Omni Compiler Project*.