Vol.2013-ARC-207 No.36
Vol.2013-HPC-142 No.36
2013/12/17

NAS Parallel Benchmarksによる Omni OpenACCコンパイラの評価

田渕 晶大[1,a)]　中尾 昌広[2]　佐藤 三久[1,2]

概要：GPGPU 等のアクセラレータによる高速化が注目されている．しかしそのプログラミングはアクセラレータごとに異なり複雑であるためプログラミングコストが高く，広いアプリケーションに対する適用を妨げてきた．OpenACC はコードの一部をアクセラレータにオフロードするための指示文ベースプログラミングモデルであり，容易に記述が可能である．本研究では，OpenACC 指示文が挿入された C 言語のコードを NVIDIA 社の GPU プログラミング環境 CUDA の API を含むコードに変換するコンパイラを設計・実装した．NAS Parallel Benchmarks に含まれる CG, EP, IS を用いて評価を行った結果，コードからコードへの変換により NVCC で最適化されたマシンコードを生成できることが分かった．それにより CG, EP では命令数が減少し商用コンパイラよりも高い性能を示した．一方でループを並列化する際のループ変数の計算や，リダクション用の配列確保のオーバーヘッドによる性能低下が見られ，いまだ改善の余地がある．

1. はじめに

ハイパフォーマンスコンピューティングの分野では計算性能を向上させる手段として演算加速器（アクセラレータ）が使用されている．アクセラレータには GPU や Intel Xeon Phi などがあり，数十から数千個の演算器を用いて演算を並列に行うことが可能である．アクセラレータ用のプログラミングモデルはいくつか存在し，例えば NVIDIA からは自社の GPU 製品向けに GPU 上で実行するカーネルを C 言語のように記述できる CUDA という開発環境が提供されている．また OpenCL は様々なアクセラレータを対象とした可搬性の高いプログラミングモデルを提供している．OpenCL や CUDA を用いてアクセラレータに計算コードの一部を実行させるには，並列ループをカーネル関数に書き換えたり，ホストメモリとデバイスメモリ間のデータ転送を記述したりする必要があるためプログラミングが煩雑になり生産性が低下する．また CUDA のように特定のデバイスでしか実行できないプログラミングモデルの場合は可搬性も低下する．

これらの問題の解決策として OpenACC[1] というプログラミングモデルが提案されている．OpenACC はアクセラレータ用の指示文ベースプログラミングモデルであり，プログラマはコードに指示文を書き加えるだけでコードの一部をアクセラレータにオフロードすることができる．

現在 OpenACC に対応しているコンパイラは PGI Compiler, Cray Compiler, HMPP である．またオープンソースのコンパイラとして accULL[5] がある．accULL はスペインのララグーナ大学で開発されているコンパイラで，Python ベースの YaCF というソース to ソースコンパイラフレームワークを用いている．C コードを CUDA または OpenCL に変換が可能である．しかしながらオープンソースの実装がまだ少ないというのが現状である．そこで Omni Compiler Infrastructure を用いて OpenACC 指示文のある C コードを C と CUDA のコードに変換する Omni OpenACC コンパイラを実装した．さらに NAS Parallel Benchmarks（NPB）を用い，実装したコンパイラと Cray Compiler や PGI Compiler との比較を行った．

本研究では OpenACC を CUDA レベルへ変換する手順を明かにした．さらにポータブルな実装にすることで OpenCL 等へ変換する際も利用できる．性能評価からは CUDA へ変換することで NVCC の最適化が行われ，命令数が減少するというメリットが明かになった．同時にループ並列化のためのループ変数計算やリダクションに用いる配列確保が性能低下につながったことも判明した．

本論文は本章を含め 6 章で構成される．2 章では OpenACC について説明し，コード例を示す．3 章では Omni

[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] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] tabuchi@hpcs.cs.tsukuba.ac.jp

OpenACC Compiler の実装をコード変換例と共に説明する．4 章では NPB の OpenACC 実装について述べ，5 章では NPB を用いて本実装の評価を行う．6 章では結論と今後の課題を述べる．

2. OpenACC

OpenACC はホストコードの一部をアクセラレータにオフロードするための指示文（directive）ベースのプログラミングモデルである．CAPS 社・CRAY 社・NVIDIA 社・PGI 社により 2011 年 11 月に仕様の Version1.0，2013 年 6 月には Version2.0 が策定された．OpenACC では C・C++・Fortran のコードに指示文を挿入することにより，負荷の高い計算をアクセラレータで実行させることが可能である．また指示文はアクセラレータの種類に依存しない形式である．そのため OpenACC には高い生産性と可搬性がある．

2.1 実行モデル

OpenACC ではホストから GPU のようなホストに接続されたデバイスに処理を行わせることが可能である．コード中の負荷の高い部分を *parallel* または *kernels* 指示文で指定すると，その処理はアクセラレータでカーネルとして実行される．OpenACC では上位から gang, worker, vector の 3 段階の並列性が存在する．gang は粗粒度，worker は細粒度の並列性である．各 gang は 1 つ以上の worker を持ち，各 worker では vector を用いてベクトル演算を行うことができる．これら 3 つの並列性が実際のアクセラレータとどのように対応付けられるかはコンパイラに依存する．並列化可能なループに対してどの並列性を用いるかは *loop* 指示文で指定可能である．

2.2 メモリモデル

OpenACC のメモリモデルではホストメモリとデバイスメモリはそれぞれ独立している．したがって，アクセラレータにオフロードするにはホストメモリとデバイスメモリ間で計算に用いるデータや計算結果を転送する必要がある．OpenACC ではホストメモリとデバイスメモリ間のデータ転送は基本的にコンパイラにより暗黙に行われるが，ユーザが指示文により明示し不要なデータ転送を削減することも可能である．

2.3 プログラム例

OpenACC 指示文を追加した C 言語のコードを**図 1** に示す．4 行目では data 指示文を使用してデバイスメモリを確保している．配列 a の値はデバイスで参照されるが更新はされないため，*copyin* 指示節を用いて data 領域の最初にホストメモリからデバイスメモリへコピーするよう指定している．逆に配列 b の値はデバイスで更新されるが参照はされないため，*copyout* 指示節を用いて data 領域の最後にデバイスメモリからホストメモリへコピーするよう指定している．6 行目では parallel loop 指示文を用いて i に関するループをデバイスで実行するとともに並列化するよう指定している．以上の指示文からコンパイラはこのループをアクセラレータで実行するよう適切に変換を行う．

```
1  #define N 1024
2  int main(){
3    int i, a[N], b[N];
4  #pragma acc data copyin(a) copyout(b)
5    {
6  #pragma acc parallel loop
7      for(i = 0; i < N; i++){
8        b[i] = a[i] + 1;
9      }
10   }
11 }
```

図 1 Sample code (sample.c)

3. Omni OpenACC Compiler の実装

Omni OpenACC Compiler の対象とする言語は C 言語とする．またオフロードの対象とするアクセラレータは現在最も普及している NVIDIA GPU とする．また実装する OpenACC の Version は 1.0 である．

実装には C と Fortran95 をソースからソースへ変換するコンパイラ用のプログラムセットで，コードの解析や変換が可能な Omni Compiler Infrastructure[2] を用いる．この使用例として，分散メモリ環境用の並列プログラミング言語 XcalableMP[3] のコンパイラである Omni XcalableMP Compiler がある．本実装の一部は XcalableMP の GPU 拡張である XcalableMP-dev[4] の実装を利用している．また GPU プログラミングには NVIDIA GPU のプログラミング環境として一般的な CUDA を用いる．

Omni OpenACC Compiler のコンパイルの流れを**図 2** に示す．OpenACC 指示文が記された C コードである sample.c をコンパイルすると，まず Omni frontend で XML で書かれた中間コードである XcodeML 形式に変換される．それを OpenACC translator が読み込み，コードを変換して sample_tmp.c と sample.cu を出力する．sample_tmp.c はホストコードで一般的な C コンパイラでコンパイルする．sample.cu は CUDA コードで，NVIDIA CUDA compiler(NVCC) 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 OpenACC runtime library とリンクして実行可能バイナリを生成する．

3.1 parallel 構文のコード変換

parallel 指示文で指定されたコード領域を GPU で実行するにはカーネル関数を作成しそれをスレッドブロックのサイズと数を指定して起動する必要がある．OpenACC には

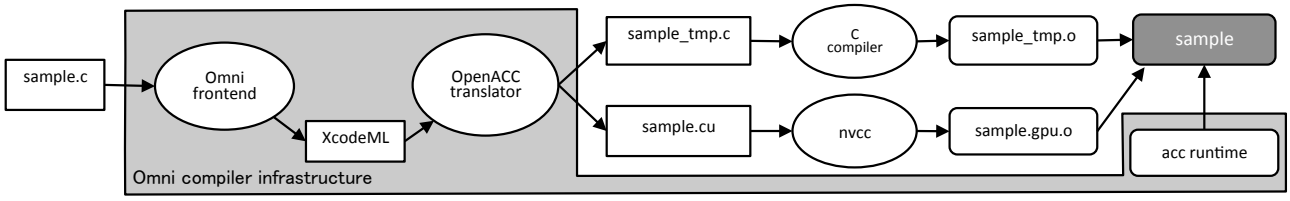

図 2　コンパイルの流れ

```
#pragma acc parallel num_gangs(4) vector_length(128)
{
  /* codes in parallel region */
}
```

(a) ACC parallel

```
__global__ static
void _ACC_GPU_FUNC_0_DEVICE(...)
{
  /* codes in parallel region */
}
extern "C"
void _ACC_GPU_FUNC_0(...)
{
  int block_x=(4), block_y=(1), block_z=(1);
  int thread_x=(128), thread_y=(1), thread_z=(1);
  dim3 DIM3_block(block_x, block_y, block_z);
  dim3 DIM3_thread(thread_x, thread_y, thread_z);

  _ACC_GPU_FUNC_0_DEVICE<<<DIM3_block,DIM3_thread>>>(...);
  _ACC_GPU_M_BARRIER_KERNEL();
}
```

(b) translated code

図 3　parallel 構文のコード変換

gang, worker, vector の 3 レベルの並列性があるが，CUDA ではブロックとスレッドの 2 種類のみ提供されている．そこで本実装では gang をブロックに vector をスレッドに割り当て，各 gang は 1 つの worker を持つこととする．カーネル関数を起動する際のブロックのサイズはデフォルトで 256 とする．またブロック数はブロックサイズと parallel 領域内のループの反復数から決定する．

図 3 に parallel 構文とその変換例を示す．この例では指示節により gang 数は 4，vector 長は 128 に指定している．関数_ACC_GPU_FUNC0 では kernel 関数を 4 ブロック，128 スレッドで起動し，その後ホストと同期を取っている．

3.2　loop 構文のコード変換

loop 指示文が書かれた for ループは GPU 上で並列に実行するように変換される．各ループ反復をスレッドに割り当てるために，まず 0〜(反復数-1) の仮想インデックスを用意する．各スレッドではブロックの ID とスレッドの ID から担当する仮想インデックスの範囲を求める．実際に反復の一部を実行する際には仮想インデックスから元のインデックスを求めて，ループ本体を実行する．

loop 指示文に reduction 節が書き添えられている場合，

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

(a) ACC loop

```
/* inside gpu kernel function */
int i, _idx, _init, _cond, _step;
_ACC_gpu_init_thread_x_iter(&_init, &_cond, &_step,0,N,1);

for(_idx=_init;_idx<_cond;_idx+=_step){
  _ACC_gpu_calc_idx(_idx,&i,0,N,1);
  a[i]++;
}
```

(b) translated code

図 4　loop 構文のコード変換

その節で指定されている変数に対して縮約を行うようにする．スレッド間で縮約を行う場合は値をスレッド間で共有するために長さ 64 の配列をシェアードメモリを用意し，そこに 64 スレッドごとに値を縮約する．さらに最後にその 64 個を 1 ワープが 1 つの値に縮約することで完了する．

ブロック間でリダクションを行う場合は 2 通りある．int 型や float 型で一部 atomic 演算に対応する縮約では atomic 演算を用いて縮約を行う．atomic 演算に対応していない縮約ではグローバルメモリに値を一時的に保存するための配列を確保し，ブロックごとに縮約結果を配列に書き込む．そして最後に配列に書き込んだブロックが配列の値を縮約して完了する．一時配列の確保にはホストとの同期が必要な cudaMalloc ではなく同期が必要ない GPU カーネル内の malloc を使用している．

図 4 に loop 構文とその変換例を示す．関数_ACC_gpu_init_thread_x_iter で，そのスレッドが担当する仮想インデックスの範囲を求める．その後ループ内では関数_ACC_gpu_calc_idx により実際のインデックスを求め，ループ本体を実行する．

3.3　data 構文のコード変換

データ指示文が指定された領域では指示節に基づいて，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が必要に応じて挿入されるランタイム関

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

(a) ACC data

```
int a[100], b;
{
  void *DEV_ADDR_a,*HOST_DESC_a,*DEV_ADDR_b,*HOST_DESC_b;
  _ACC_gpu_init_data(&(HOST_DESC_a),&(DEV_ADDR_a),a,
                     (100)*sizeof(int));
  _ACC_gpu_init_data(&(HOST_DESC_b),&(DEV_ADDR_b),&(b),
                     sizeof(int));
  _ACC_gpu_copy_data(HOST_DESC_a, 400); /* host to device */
  {
    /* some codes using a and b */
  }
  _ACC_gpu_copy_data(HOST_DESC_a, 401); /* device to host */
  _ACC_gpu_copy_data(HOST_DESC_b, 401);
  _ACC_gpu_finalize_data(HOST_DESC_a);
  _ACC_gpu_finalize_data(HOST_DESC_b);
}
```

(b) translated code

図 5 data 構文のコード変換

数_ACC_gpu_init_data によりホストの変数や配列のためのデバイスメモリを確保する．_ACC_DEVICE_ADDR_*name* はホストの *name* に対応するデバイスメモリのポインタである．_ACC_HOST_DESC_*name* は *name* のホストアドレス，デバイスアドレス，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ンタイム関数_ACC_gpu_copy_data により行われる．転送の方向は第 2 引数で与える．領域の最後ではランタイム関数_ACC_gpu_finallize_data によりデバイスメモリを解放する．

図 5 に data 構文とその変換例を示す．この data 構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され，copyout 節内で指定されている変数 *b* は領域の最後で転送される．

4. NPB の OpenACC による実装

NPB に含まれるカーネルベンチマークのうち CG, EP, IS を OpenACC で実装した．元の C のシリアルコードはソウル大学校が公開している SNU NPB Suite[6] に含まれる NPB3.3-SER である．また実装の際には OpenACC Home[1] でサンプルコードとして公開されている NPB2.3-OpenACC-C を参考にした．

4.1 CG

CG(Conjugate Gradient) 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．*parallel* 指示文によりオフロードする部分は NPB2.3-OpenACC-C と同じである．CG には *parallel* 指示文でオフロードする領域が連続することが多いという特徴がある．*parallel* 領域はデフォルトではアクセラレータ側の実行が終わるまでホストが待機するよう同期が行われる．*parallel* 領域が連続する場合には各領域の終わりに同期を行う必要はない．そこで *parallel* 指示文に *async* 節を付けてホストと非同期に実行するようにし，ホストとの同期が必要になった場所でのみ *wait* 指示文により同期を行うことで，不要な同期を削減した．

4.2 EP

EP（Embarrassingly Parallel）は乗算合同法による一様乱数，正規乱数の生成を行うベンチマークである．乱数を生成する一番外側のループに *parallel* 指示文を追加し乱数生成部分全体をオフロードするようにした．NPB2.3-OpenACC-C の実装と同様に各スレッドの書き込みが衝突する可能性のあるカウント配列 *q* のために二次元配列 *qq* を用意し各スレッドの書き込み先が異なるようにし，最後に reduction で配列 *q* にまとめるようにした．本実装ではさらに SNU NPB Suite に含まれる OpenCL 実装 [7] を参考に 2*NK 個の一様乱数を配列 *x* に一度格納してからその値を 2 つずつ使用して NK 個の正規乱数を生成する部分を，2 個の一様乱数を変数 *x_1*, *x_2* に格納して 1 つの正規乱数を生成するのを NK 回繰り返すようにし，一時配列 *x* をなくしてレジスタへのアクセスのみになるようにした．

4.3 IS

IS（Integer Sort）は Bucket ソートにより大規模整数ソートを行うベンチマークである．アルゴリズムはヒストグラムを計算し，要素の個数からソート後の各要素の先頭アドレスを prefix sum で求めるという単純なものである．ヒストグラムを並列に求めるためには EP と同様に各スレッド用に配列を用意し最後に reduction するか，もしくは atomic 演算を用いる必要がある．IS で用いられるヒストグラムの配列は最も小さな問題サイズでも 2048 であり，数百個のスレッドを実行するにはメモリが不足する．atomic 演算は OpenACC2.0 から導入されており，Omni OpenACC Compiler ではまだサポートしていない．そのためヒストグラム計算はホストで実行するようにした．prefix sum は並列計算できるようにシリアルコードを書き換えてオフロードを行った．

5. 性能評価

評価に用いた計算機の構成を**表 1**，**表 2** に示す．ベンチマークには NPB に含まれる CG, EP, IS ベンチマークを用い，問題サイズは小さい方からクラス S,W,A,B,C の 5 種類とした．これらを Omni OpenACC Compiler でコンパイルして実行し，結果を 1 秒当たりに何百万回の演算を行っ

表 1　K20-AMD の構成

CPU	AMD Opteron 6272 2.1GHz (16Core)
Memory	DDR3-1600 32GB
GPU	Tesla K20 (GDDR5 5GB, ECC on)
OS	CNL (Compute Node Linux)
Compiler	CCE8.2.0, CUDA5.5

表 2　M2050-Xeon の構成

CPU	Intel Xeon E5630 2.53GHz (4Core x2)
Memory	DDR3 24GB
GPU	Tesla M2050 (GDDR5 3GB, ECC on)
OS	CentOS 6.3
Compiler	GCC4.4.6, CUDA5.0, PGI Compiler13.6

たかを表す mop/s（Mega Operations Per Second）で取得した．また比較のために K20-AMD では Cray Compiler 8.2.0，M2050-Xeon では PGI Compiler13.6 を用いてコンパイルして実行した場合の性能も測定した．このとき GPU カーネルのブロックサイズを Omni OpenACC コンパイラのデフォルトである 256 スレッド/ブロックとして指定し，ブロックサイズによる性能の違いが出ないようにした．

5.1　CG

CG ベンチマークの評価結果を図 **6**，図 **7** に示す．K20-AMD においてはすべてのクラスにおいて Cray Compiler より性能が劣っている．特にクラス S では Cray Compiler でコンパイルした場合の 41%ほどしか性能が出ていない．この原因は各スレッドがループのどの範囲を担当するかの計算が多く，またリダクションがあるループでは一時配列を確保するための時間がかかっているからである．

M2050-Xeon においてはクラス S-A では PGI Compiler に劣るものの，クラス B, C では勝る性能となった．特にクラス B では PGI Compiler でコンパイルした場合より 14%性能が高い．カーネル実行時間を調べると疎行列ベクトル積以外のカーネル実行時間は PGI Compiler でコンパイルしたものよりも長かった．これは K20-AMD と同様に大きなオーバーヘッドが原因である．一方で疎行列ベクトル積のカーネル実行時間は PGI Compiler でコンパイルした場合より短かった．調べてみるとグローバルメモリの Load Efficiency がより高い事が判明した．さらに Load Efficiency が高かった原因を調べるため，両者の PTX コードを比べてみた．すると，PGI Compiler でコンパイルした方ではループ内で rowstr[j] の値をグローバルメモリからロードしていた．rowstr[j] はループで不変のため実際は毎回ロードする必要はない．このロードが Load Efficiency が低下していた原因である．しかしながら Omni OpenACC コンパイラが出力した CUDA コードの段階では，同様にループ内で rowstr[j] を参照している．このことから，Omni OpenACC コンパイラでは PGI Compiler よりも NVCC による最適化がされやすいコードを生成していると言える．

また全実行時間に占める疎行列ベクトル積の割合が大きいため，問題サイズが大きな B や C ではオーバーヘッドによる性能低下よりも疎行列ベクトル積の性能向上の方が大きくなり全体として PGI Compiler よりも速くなった．

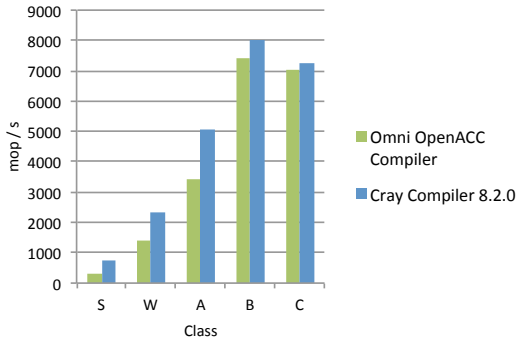

図 6　CG の評価結果 (K20-AMD)

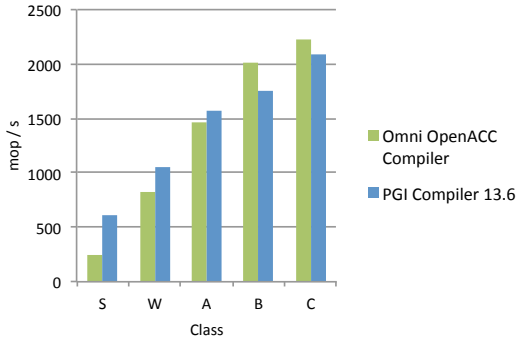

図 7　CG の評価結果 (M2050-Xeon)

5.2　EP

EP ベンチマークの評価結果を図 **8**，図 **9** に示す．K20-AMD ではすべてのクラスにおいて 1.3〜2.2 倍ほど Cray Compiler より良い結果が得られた．調べると Omni Compiler によるコードの方が GPU での命令発行回数が大幅に少ないことが分かった．この差はプログラムの変換方法に起因する．我々の実装では CUDA コードを出力し，それを NVCC でコンパイルする．この時 NVCC 内部ではコードを PTX と呼ばれる疑似アセンブリ言語に変換し，それを PTXAS(PTX optimizing assembler) で最終的に機械語に変換する．一方 Cray Compiler は直接 PTX コードを出力し，それを PTXAS でコンパイルする．EP はメモリアクセスが少なく演算が多いため，演算回数が性能を左右する．NVCC の方がより効率のよい PTX コードを生成できたため，Cray Compiler より大幅に性能が向上した．

M2050-Xeon では PGI Compiler より若干性能が劣っている．これは前述のオーバーヘッドによるものである．

5.3　IS

IS ベンチマークの評価結果を図 **10**，図 **11** に示す．K20-AMD ではクラス S, W を除き，Cray Compiler と大きな差は見られなかった．これは GPU での計算部分が prefix sum の部分だけであり，90%以上の時間が CPU での計算

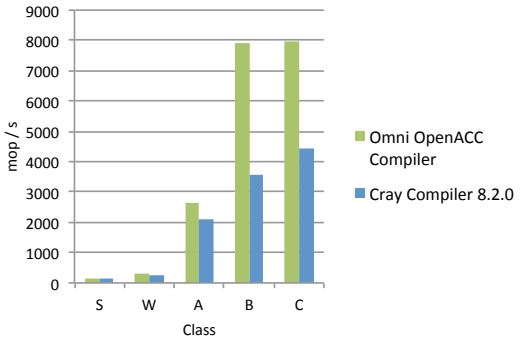

図 8　EP の評価結果 (K20-AMD)

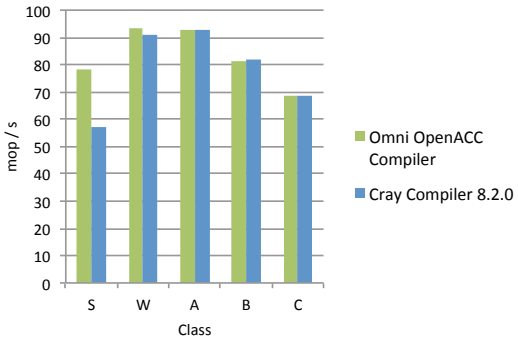

図 10　IS の評価結果 (K20-AMD)

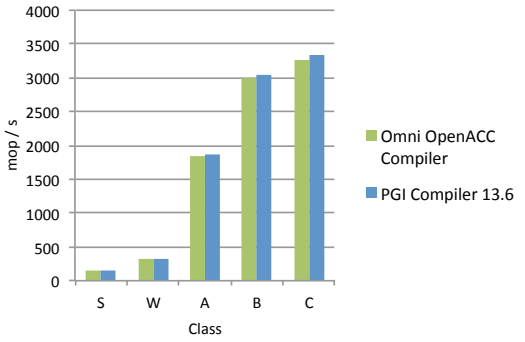

図 9　EP の評価結果 (M2050-Xeon)

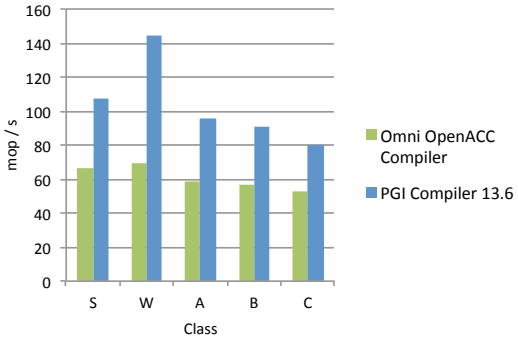

図 11　IS の評価結果 (M2050-Xeon)

時間だからである．GPU カーネルの実行時間を調べると我々の実装によるコードの方が少し長かった．しかし Cray Compiler でコンパイルしたものでは GPU カーネルの終了を待機する wait 指示文で余計に時間がかかっていた．この原因については現在調査中である．クラス S や W ではこの wait の時間により我々の実装の方が性能が良くなっている．

M2050-Xeon ではすべてのクラスで PGI Compiler でコンパイルした場合の 48〜66%ほどで，かなり性能が劣っている．これは C コードのコンパイラの違いから生じていた．PGI Compiler は C コードを自身のコンパイラでコンパイルするのに対して我々の実装では gcc でコンパイルしており CPU での実行時間が長くなっていた．我々の実装で変換した C コードを PGI Compiler でコンパイルしようとしたが，エラーが出てしまうためできなかった．GPU カーネルの実行時間を調べると前述のオーバーヘッドにより PGI Compiler でコンパイルしたものより長くなっていたため，CPU での実行時間が同じになったとしても PGI Compiler より性能は劣ると予想される．

6. 結論

OpenACC コードを CUDA コードに変換する OpenACC Compiler を設計・実装し，OpenACC Version 1.0 の大部分を実装した．性能評価のために NAS Parallel Benchmarks のうち CG, EP, IS を OpenACC で実装し，それ用いてコンパイラの評価を行った．Cray Compiler や PGI Compiler と比較すると，実行するループ範囲の計算やリダクション用の一時配列確保等のオーバーヘッドが大きく，未だ改善の余地がある．一方で CUDA コードに変換する手法を取ることで，より性能の高いプログラムを生成できることが判明した．

今後はカーネルのオーバーヘッドを削減するようコード変換の見直しを行い，また OpenACC の仕様を満たすように実装を進める予定である．さらに accULL など他のコンパイラとの性能比較も行う必要がある．

参考文献

[1] “OpenACC Home” `http://www.openacc-standard.org/`
[2] “Omni Compiler Project” `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/`
[3] “XcalableMP” `http://www.xcalablemp.org`
[4] J. Lee, M. T. Tran, T. Odajima, T. Boku, and M. Sato. An Extension of XcalableMP PGAS Language for Multi-node GPU Clusters. *Euro-Par'11 Proceedings of the 2011 International Conference on Parallel Processing*, pp. 429-439, 2011.
[5] R. Reyes, I. López-Rodríguez, J. J. Fumero, F. de Sande. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, Lecture Notes in Computer Science, pp. 871-882, 2012.
[6] SNU NPB Suite `http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU_NPB_Suite.html`
[7] S. Seo, G. Jo, J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 137-148, 2011.