Vol.2012-HPC-136 No.4
2012/10/3

Omniコンパイラによる OpenACCの試作

田渕 晶大[1,a)]　中尾 昌広[2]　佐藤 三久[2,3]

概要：GPGPU などのアクセラレータによる高速化が注目されているが，そのプログラミングについては複雑であり，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，容易に記述が可能である．Omni コンパイラは，C と Fortran95 をソース変換するコンパイラインフラストラクチャである．本研究では，Omni コンパイラを用いて，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ことで，GPU を利用したアプリケーションを開発できるコンパイラを設計・試作した．その結果，行列積では CPU のみの場合と比較し最大約 2.9 倍，N 体問題では最大約 18 倍の高速化の見込みを得た．同時に，スレッドの割り当て方等の最適化について課題があることも分かった．

1. はじめに

CPU と比較して高速かつ電力性能比の高いアクセラレータを用いてアプリケーションの性能向上を行う試みが数多く行われている．CPU とアクセラレータのヘテロジニアス環境での並列コンピューティングのためのフレームワークとして OpenCL[2] が存在する．また NVIDIA 社製 GPU のために GPU プログラミング環境 CUDA[3] がある．OpenCL や CUDA を用いてアクセラレータにオフロードするにはホストとデバイス間のデータ転送・デバイスカーネルの記述等が必要となり，コード量が CPU のみの場合に比べてかなり多くなる．

アクセラレータへオフロードする際の従来の煩雑なコーディングを解決するために，記述の容易な指示文（directive）ベースのプログラミング言語である OpenACC[1] が CAPS 社・CRAY 社・NVIDIA 社・PGI 社により策定された．OpenACC では C・C++・Fortran のコードに指示文を挿入するだけで容易にアクセラレータへのオフロードが可能であるため，高い生産性を持つ．またアクセラレータの種類に依存しないので，コードの可搬性も高い．OpenACC ではホストとアクセラレータ間のデータ転送やアクセラレータで実行するプログラムの生成はコンパイラにより自動で行われる．OpenACC 指示文に指示節（clause）を書き添えることで，並列処理やデータ転送に関してチューニングを行うことも可能である．OpenACC に対応したコンパイラは数少ないので，C と Fortran95 をソース変換するコンパイラインフラストラクチャである Omni コンパイラ [4] をベースに OpenACC コンパイラを試作した．

本稿は本章を含め 6 章で構成される．2 章では OpenACC を概説し，主な指示文とコード例を示す．3 章では OpenACC の実装とコード変換を変換例と共に説明する．4 章では行列積と N 体問題を用いて性能評価を行う．5 章では関連研究を紹介し，6 章ではまとめと今後の課題を述べる．

2. OpenACC の概要

OpenACC はユーザ指示アクセラレータプログラミングモデルである．プログラムの大部分はホストで実行し，計算が集中する領域をユーザがオフロードの対象として指定することで，その領域がアクセラレータにオフロードされる．コンパイラが自動でプログラム領域をオフロードすることはない．またホストからは 1 つのアクセラレータへのみプログラム領域をオフロードできる．

ホストのみのプログラムとホスト+アクセラレータのプログラムの大きな違いはホストメモリとデバイスメモリが独立していることである．ホストはデバイスメモリを直接読み書きすることはできないため，ホストメモリとデバイスメモリ間のデータ移動をホストからのランタイムライブラリコールによって実行する必要がある．OpenACC モデルではメモリ間のデータ移動は指示文にもとづきコンパイラによって管理される．しかしユーザは，高速なアプリ

1 筑波大学情報学群
School of Informatics, 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
3 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
a) tabuchi@hpcs.cs.tsukuba.ac.jp

ケーションを作成するにはデータ転送量に対する計算量が大きくなければならないことや，デバイスメモリサイズの制限により巨大なデータ上で計算するコード領域をオフロードできないことを知っておく必要がある．

現在の GPU のように一部のアクセラレータは weak memory model で，メモリの一貫性を保証していない．そのため 2 つの処理が同じ領域に値を書き込もうとするなら，その結果は常に同じであることが保証されない．

一部のアクセラレータはソフトウェアキャッシュやハードウェアキャッシュを持っている．OpenACC モデルでは指示文を頼りにコンパイラがキャッシュを管理する．

2.1 主な指示文の解説

ここでは主に使われる指示文の解説を行う．すべての指示文は OpenACC の仕様書を見ていただきたい．C・C++ での OpenACC 指示文の文法は

#pragma acc *directive [clause [[,] clause]...]*

である．各々の指示文は**#pragma acc** から始まる．指示文ごとに指定できる指示節が決められている．指示文は直後の文や構造化ブロックやループに適用される．

2.1.1 parallel 構文

アクセラレータデバイス上で並列実行を開始する．async 節がなければ同期実行され，並列領域が終わるまでホストプログラムは待機する．領域内で参照されるデータのうち，指示節で指定されないデータや parallel 構文を囲む data 構文で指定されないデータは自動的に present_or_copy 節中で指定されているとみなされる．

#pragma acc parallel *[clause [[,] clause]...]*
structured-block

clause ::= if(cond) | async(exp) | num_gangs(exp) | num_workers(exp) | reduction(op:list) | ...

以下は clause の説明である．

if(condition)
条件が非ゼロならば並列領域がアクセラレータで実行され，そうでなければホストで実行される．

async(exp)
並列領域をホストと非同期に実行する．

num_gangs(exp)
並列ギャング数を指定する．

num_workers(exp)
ギャング内のワーカー数を指定する．

reduction(operator:list)
list 内の変数を領域の最後に operator で縮約する．

2.1.2 loop 構文

直後のループに対しループを実行する際の並列性を記述する．

#pragma acc loop *[clause [[,] clause] ...]*
for loop

clause ::= collapse(n) | seq | reduction(op:list) | ...

以下は clause の説明である．

collapse(n)
n 個のネストしたループに指示文を適用する．

seq
アクセラレータ上でループを逐次実行する．

reduction(operator:list)
parallel 構文の reduction と同じ．

2.1.3 data 構文

デバイスメモリに確保されるスカラー・配列・部分配列が，構造化ブロックの始まりでホストからデバイスにコピーされるか，構造化ブロックの終わりにデバイスからホストにコピーされるかを定義する．

#pragma acc data *[clause [[,] clause] ...]*
structured-block

clause ::= copy(list) | copyin(list) | copyout(list) | create(list) | present(list) | present_or_copy(list) | ...

以下は clause の説明である．

copy(list)
list 内のデータをデバイスで確保し，領域の開始時にホストからデバイスにコピー，領域の終了時にデバイスからホストにコピーする．

copyin(list)
list 内のデータをデバイスで確保し，領域の開始時にホストからデバイスにコピーする．

copyout(list)
list 内のデータをデバイスに確保し，領域の終了時にデバイスからホストにコピーする．

create(list)
list 内のデータをデバイスに確保する．

present(list)
list 内のデータは既にデバイス上に存在することを示す．もし存在しなければプログラムはエラー終了する．

present_or_copy(list)
list 内のデータが既にデバイス上に存在するならそのデータを使い，存在しないなら copy 節と同じ動作をする．

2.1.4 parallel loop 構文

並列領域とループの指定を同時に行う．parallel・loop 指示文で使用できる指示節は parallel loop 指示文でも使用できる．

#pragma acc parallel loop *[clause [[,] clause] ...]*
for loop

```
#define N 1024
int main(){
int i;
double a[N], b[N], c[N];
#pragma data copyin(a,b) copyout(c)
{
#pragma acc parallel loop
  for(i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }
}
}
```

図 1　OpenACC のコード例（test.c）

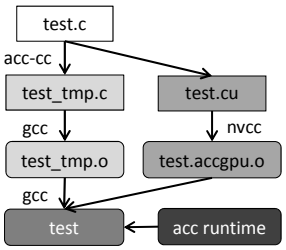

図 2　コンパイルの流れ

2.2　OpenACC のコード例

OpenACC のコード例として OpenACC 指示文を用いてベクトルの加算をするコード test.c を図 **1** に示す．まず data 構文を用いてデバイスメモリを確保する．a,b の値はデバイスでの計算に必要となるがデバイスで値は更新されない．したがって copyin(a,b) として data 領域の始めにホストメモリからデバイスメモリへ転送するよう指示する．c の値はデバイスで参照されないがデバイスで値が更新される．そのため copyout(c) として data 領域の最後にデバイスメモリからホストメモリに転送するよう指示する．それから parallel loop 構文により並列実行領域および i に関するループの並列化を指定する．

3.　OpenACC の実装

3.1　コンパイラ実装

Omni コンパイラは指示文を解析する機能があり，OpenMP をコンパイルするコンパイラとして広く用いられている．我々の OpenACC コンパイラは Omni XcalableMP Compiler[5] を拡張した XcalableMP-dev[6] をベースにしている．コンパイルの流れを図 **2** に示す．C 言語と OpenACC で書かれた test.c をコンパイルすると中間コードである test_tmp.c と test.cu が生成される．test_tmp.c は gcc でコンパイルし，test.cu は nvcc でコンパイルする．コンパイルされたオブジェクトファイル test_tmp.o と test.accgpu.o をランタイムライブラリとリンクすることで実行可能ファイルが生成される．今回は実装が未完成なため，ハンドコンパイルのコードで評価した．

表 1　ノード構成

CPU	AMD Opteron Processor 6272 (16 コア 2.1GHz)
Memory	DDR3-1600 4GBx4 (16GB)
GPU	NVIDIA X2090 (GDDR5 6GB)
OS	CNL (Compute Node Linux)
Compiler	CCE8.1.0.143, GCC4.6.2, CUDA4.0.17a

3.2　コード変換

test.c を用いて parallel loop 構文の変換例を示す．test.c を変換したコードを図 **3**，図 **4** に示す．test_tmp.c では，まず_ACC_gpu_init_data によりベクトル a,b,c のためのデバイスメモリを確保する．そして_ACC_gpu_sync により a,b はデータ領域の開始時にホストメモリからデバイスメモリに転送される．_ACC_GPU_FUNC_0 により GPU で c=a+b のベクトル加算が並列実行される．_ACC_gpu_sync により c はデータ領域の終了時にデバイスメモリからホストメモリに転送される．最後に_ACC_gpu_finalize_data によりデバイスメモリを解放する．

test.cu には GPU カーネルを呼び出す関数と GPU カーネルが含まれる．_ACC_GPU_FUNC_0 関数はループのインデックス i から GPU カーネルを実行する際のスレッドブロック数やスレッド数を計算し，GPU カーネルを起動する．その後，_ACC_GPU_M_BARRIER_KERNEL により GPU カーネルが終了するまで待機する．_ACC_GPU_FUNC_0_DEVICE 関数は GPU カーネルで，各 GPU スレッドでは_ACC_GPU_THREAD_ID からそのスレッドにおける i を求め，c[i] を計算する．

4.　性能評価

OpenACC コンパイラの性能評価のために行列積と N 体問題を用いた．性能評価には CrayXK6m-200 の 1 ノードを使用した．ノードの構成を表 **1** に示す．

4.1　行列積

図 **5** は OpenACC 指示文を用いた行列積のコードである．test.c と同様に，a,b はデータ領域の開始時にホストからデバイスに転送するため data copyin を指定し，c はデータ領域の終了時にデバイスからホストへ転送するため data copyout を指定する．並列領域と i に関するループの指定は parallel loop 構文で，j に関するループは loop 構文により指定する．

このコードをハンドコンパイルしたコードの実行時間を計測した．また比較のために CPU のシングルスレッドの場合，OpenACC に対応した Cray コンパイラでコンパイルした場合，shared メモリを使わずに直接 CUDA コードを書いた場合の実行時間も計測した．行列積の実行時間を図 **6** に示す．Omni コンパイラは CPU のみの場合に比べて N=1K では約 2.9 倍，N=2K〜8K では約 2.3 倍速い結

```
int main()
{
  int i;
  double a[1024], b[1024], c[1024];
  //...

  void * _ACC_GPU_HOST_DESC_a; void * _ACC_GPU_DEVICE_ADDR_a;
  void * _ACC_GPU_HOST_DESC_b; void * _ACC_GPU_DEVICE_ADDR_b;
  void * _ACC_GPU_HOST_DESC_c; void * _ACC_GPU_DEVICE_ADDR_c;
  _ACC_gpu_init_data(&(_ACC_GPU_HOST_DESC_a), &(_ACC_GPU_DEVICE_ADDR_a), a, (0x00000040011)*(sizeof(double)));
  _ACC_gpu_init_data(&(_ACC_GPU_HOST_DESC_b), &(_ACC_GPU_DEVICE_ADDR_b), b, (0x00000040011)*(sizeof(double)));
  _ACC_gpu_init_data(&(_ACC_GPU_HOST_DESC_c), &(_ACC_GPU_DEVICE_ADDR_c), c, (0x00000040011)*(sizeof(double)));
  {
    _ACC_gpu_sync(_ACC_GPU_HOST_DESC_a, 600);
    _ACC_gpu_sync(_ACC_GPU_HOST_DESC_b, 600);
    {
      int _ACC_loop_init_i; int _ACC_loop_cond_i; int _ACC_loop_step_i;
      _ACC_loop_init_i=(0); _ACC_loop_cond_i=(1024); _ACC_loop_step_i=(1);
      _ACC_GPU_FUNC_0(_ACC_GPU_DEVICE_ADDR_a, _ACC_GPU_DEVICE_ADDR_b,_ACC_GPU_DEVICE_ADDR_c,
                      _ACC_loop_init_i, _ACC_loop_cond_i, _ACC_loop_step_i);
    }
    _ACC_gpu_sync(_ACC_GPU_HOST_DESC_c, 601);
  }
  _ACC_gpu_finalize_data(_ACC_GPU_HOST_DESC_a);
  _ACC_gpu_finalize_data(_ACC_GPU_HOST_DESC_b);
  _ACC_gpu_finalize_data(_ACC_GPU_HOST_DESC_c);
}
```

図 3　test_tmp.c

```
__global__ static
void _ACC_GPU_FUNC_0_DEVICE(double a[1024], double b[1024], double c[1024], int _ACC_loop_init_i,
                            int _ACC_loop_cond_i, int _ACC_loop_step_i, unsigned long long _ACC_GPU_TOTAL_ITER)
{
  int i;
  unsigned long long _ACC_GPU_THREAD_ID;
  _ACC_gpu_calc_thread_id(&_ACC_GPU_THREAD_ID);
  _ACC_gpu_calc_iter(_ACC_GPU_THREAD_ID, _ACC_loop_init_i,_ACC_loop_cond_i,_ACC_loop_step_i,&i);
  if((_ACC_GPU_THREAD_ID)<(_ACC_GPU_TOTAL_ITER))
    {
      (c[i]) = ((a[i]) + (b[i]));
    }
}
extern "C"
void _ACC_GPU_FUNC_0(double a[1024], double b[1024], double c[1024],
                     int _ACC_loop_init_i, int _ACC_loop_cond_i, int _ACC_loop_step_i)
{
  int _ACC_GPU_DIM3_block_x; int _ACC_GPU_DIM3_block_y; int _ACC_GPU_DIM3_block_z;
  int _ACC_GPU_DIM3_thread_x; int _ACC_GPU_DIM3_thread_y; int _ACC_GPU_DIM3_thread_z;
  unsigned long long _ACC_GPU_TOTAL_ITER;
  _ACC_gpu_calc_config_params(&_ACC_GPU_TOTAL_ITER,&_ACC_GPU_DIM3_block_x,&_ACC_GPU_DIM3_block_y,
                              &_ACC_GPU_DIM3_block_z,&_ACC_GPU_DIM3_thread_x,&_ACC_GPU_DIM3_thread_y,
                              &_ACC_GPU_DIM3_thread_z,_ACC_loop_init_i,_ACC_loop_cond_i,_ACC_loop_step_i);
  {
    dim3 _ACC_GPU_DIM3_block(_ACC_GPU_DIM3_block_x, _ACC_GPU_DIM3_block_y, _ACC_GPU_DIM3_block_z);
    dim3 _ACC_GPU_DIM3_thread(_ACC_GPU_DIM3_thread_x, _ACC_GPU_DIM3_thread_y, _ACC_GPU_DIM3_thread_z);
    _ACC_GPU_FUNC_0_DEVICE<<<_ACC_GPU_DIM3_block, _ACC_GPU_DIM3_thread>>>(a,b,c,
                                    _ACC_loop_init_i,_ACC_loop_cond_i,_ACC_loop_step_i,_ACC_GPU_TOTAL_ITER);
    _ACC_GPU_M_BARRIER_KERNEL();
  }
}
```

図 4　test.cu

```
int main(){
int i,j;
double a[N][N], b[N][N], c[N][N];
#pragma data copyin(a,b) copyout(c)
{
#pragma acc parallel loop
  for(i = 0; i < N; i++){
#pragma acc loop
    for(j = 0; j < N; j++){
      double sum = 0.0;
      for(int k = 0; k < N; k++) sum += a[i][k] * b[k][j];
      c[i][j] = sum;
    }
  }
}
}
```

図 5　行列積のコード

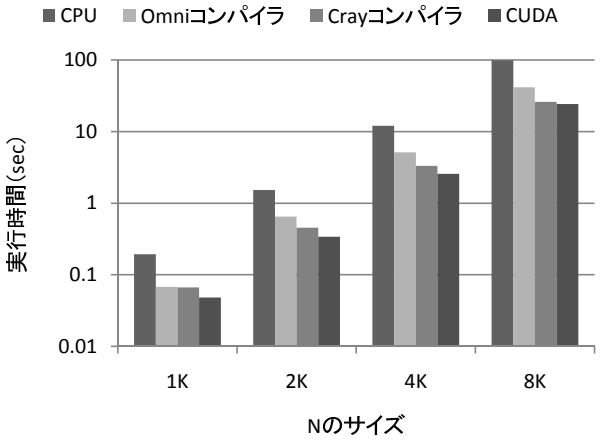

図 6　行列積の実行時間

果となった．しかし，Cray コンパイラと比較すると遅い結果となっている．最も速かったのは直接書いた CUDA コードであった．

Omni コンパイラと Cray コンパイラと直接書いた CUDA の実行時間の違いは i,j の二重ループをどのようにスレッドブロックやスレッドに割り当てたかの違いに起因すると考えられる．例として N=1K の時，Omni コンパイラでは i と j に関するループを 1 次元化し（idx=i*N+j），それを 256 スレッドのスレッドブロック 4096 個に順次割り当てている．それに対し Cray コンパイラでは i に関するループを 1024 個のスレッドブロックで分割し，j のループをスレッドブロック内の 128 個のスレッドに割り当てていた．直接書いた CUDA のコードでは 2 次元上で 1024x1024 個の (i,j) を 16x16 のタイルに分割し，それぞれを 16x16 スレッドのスレッドブロックに割り当てた．このスレッドの割り当て方の違いによりそれぞれ GPU の global メモリへの読み書きのパターンが異なる．Omni コンパイラに比べ，Cray コンパイラのほうがキャッシュの効きやすい global メモリアクセスを行う割り当て方をしていたと考えられる．

4.2　N 体問題

図 **7** は N 体問題の C 言語のコードに OpenACC 指示文を挿入したコードである．p_x,p_y,p_z は粒子の位置，v_x,v_y,v_z は粒子の速度，m は粒子の質量である．1 つ目のループで粒子同士の引力による粒子の速度変化を計算し，2 つ目のループで粒子の位置を次の時間の位置に更新する．2 つのループに分けているのはすべての粒子において働く引力を計算するまでは，粒子の位置を更新できないためである．m,v_x,v_y,v_z はデータ領域の開始時のみホストメモリからデバイスメモリへ転送するため data copyin を指定し，p_x,p_y,p_z はデータ領域の開始時にホストからデバイスへ，終了時にデバイスからホストへ転送するために data copy を指定している．

このコードをハンドコンパイルしたコードの実行時間を計測した．行列積と同様に比較として CPU シングルスレッドの場合，Crya コンパイラを使用した場合，直接 CUDA コードを書いた場合の実行時間も計測した．N 体問題の実行時間を図 **8** に示す．Omni コンパイラは CPU のみの場合に比べて N=1K では約 3 倍，N=32K では 18 倍速い結果となった．Cray コンパイラは N=1K では約 5 倍，N=32K では 27 倍速くなり，Omni コンパイラは Cray コンパイラと比較すると遅い結果となっている．

N 体問題における実行時間の差は計算数の違いが主な原因と考えられる．粒子の加速度 acc_x〜acc_z を計算する部分では r での除算が 3 回ある．この部分は a/r を 1 度計算して，その結果を使うことで除算の回数を 1 回に減らすことができる．直接書いた CUDA のコードではこの最適化を行っている．Cray コンパイラと直接書いた CUDA の実行時間はほぼ同じであるから，Cray コンパイラもこの最適化が行われていると考えられる．一方 Omni コンパイラではこの最適化がされなかったため，Cray コンパイラよりも遅かったと考えられる．

4.3　課題

行列積の結果から，ループをスレッドへ割り当てる方法を改善する必要がある．よりキャッシュの効きやすい global メモリアクセスとなるように割り当て方を工夫しなければならない．さらに global メモリへのアクセス数を削減するために shared メモリをキャッシュとして使うことも考えられる．

N 体問題の結果から，実行時に不要な計算が行われないようにするために共通部分式の除去を行う方法を検討する必要がある．

5.　関連研究

OMPCUDA[7] や OpenMPC[8] は OpenMP の GPGPU 拡張である．OMPCUDA は OpenMP 指示文のみである

```
int main(){
int i,t;
double p_x[N], p_y[N], p_z[N], m[N];
double v_x[N], v_y[N], v_z[N];
#pragma acc data copyin(m, v_x, v_y, v_z) copy(p_x, p_y, p_z)
for (t = 0; t < TIME_STEP; t++) {
#pragma acc parallel loop
  for (i = 0; i < N; i++) {
    double x_i, y_i, z_i, x_j, y_j, z_j;
    double dx, dy, dz, r2, r, a;
    double acc_x = 0, acc_y = 0, acc_z = 0;

    x_i = p_x[i]; y_i = p_y[i]; z_i = p_z[i];
    for (int j = 0; j < N; j++) {
      if (i != j) {
        x_j = p_x[j]; y_j = p_y[j]; z_j = p_z[j];
        dx = x_j - x_i; dy = y_j - y_i; dz = z_j - z_i;
        r2 = (dx * dx) + (dy * dy) + (dz * dz) + EPSILON;
        r = sqrt(r2); a = G * m[j] / r2;
        acc_x += a * (x_j - x_i) / r;
        acc_y += a * (y_j - y_i) / r;
        acc_z += a * (z_j - z_i) / r;
      }
    }
    v_x[i] += acc_x * DT; v_y[i] += acc_y * DT;
    v_z[i] += acc_z * DT;
  } //for (i = 0; i < N; i++)

#pragma acc parallel loop
  for(i = 0; i < N; i++) {
    p_x[i] += v_x[i] * DT; p_y[i] += v_y[i] * DT;
    p_z[i] += v_z[i] * DT;
  }
} //for (t = 0; t < TIME_STEP; t++)
}
```

図 7　N 体問題のコード

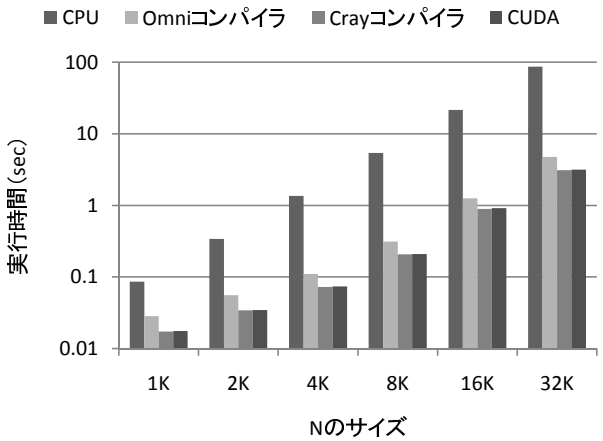

図 8　N 体問題の実行時間

が，OpenMPC は OpenMP 指示文に加えて独自の指示文を使ったチューニングが可能である．

HMPP(Hybrid Multicore Parallel Programming) Workbench[9] は独自の指示文により GPU とホスト間のデータ転送や GPU カーネルの起動を指示することができる．HMPP ではバックエンドとして CUDA と OpenCL を使用しているため，NVIDIA だけでなく AMD（ATI）の GPU も使用可能である．

accULL[10] はバックエンドに CUDA と OpenCL を用いた OpenACC コンパイラである．OpenCL を用いているため，OpenCL に対応するアクセラレータを使用できるうえ，アクセラレータを使わずにマルチ CPU を使った高速化も可能である．

6. まとめ

本研究では Omni コンパイラを用いて OpenACC 指示文が挿入された C 言語のコードを CUDA の API を含むコードに変換することで，GPU を利用したアプリケーションを開発できる OpenACC コンパイラを設計・試作した．行列積と N 体問題のコードに対して OpenACC コンパイラが出力する予定のコードを作成し性能評価を行った．行列積では CPU のみと比較し最大約 2.9 倍，N 体問題では CPU のみと比較し最大約 18 倍の高速化の見込みを得た．

今後の課題はまず OpenACC コンパイラの実装を完成させることである．次に行列積で見られたようなスレッドの割り当て方法，および shared メモリを使用した最適化の手法を検討したい．

謝辞　本研究の一部は文部科学省委託研究「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」による．

参考文献

[1] OpenACC.org : http://www.openacc-standard.org/.
[2] OpenCL : http://www.khronos.org/opencl/.
[3] NVIDIA CUDA: http://www.nvidia.com/object/cuda_home_new.html.
[4] Omni Compiler Project: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/.
[5] XcalableMP Official Website: http://www.xcalablemp.org.
[6] 李珍泌，Tran Minh Tuan，小田嶋哲哉，朴泰祐，佐藤三久: PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作. 情報処理学会論文誌 Vol.51 No.10 123-1252 (Oct.2010)
[7] Satoshi Ohshima, Shoichi Hirasawa, and Hiroki Honda: OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, Lecture Notes in Computer Science, pages 161-173. 2010.
[8] Seyong Lee and Rudolf Eigenmann: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'10, pages 1-11, 2010.
[9] HMPP Workbench: http://www.caps-entreprise.com/technology/hmpp/.
[10] Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero, Francisco de Sande: accULL: An OpenACC Implementation with CUDA and OpenCL Support. In *Euro-Par 2012 Parallel Processing*, Lecture Notes in Computer Science, pages 871-882. 2012.