Vol.2011-HPC-130 No.18
2011/7/27

# プログラム自動生成技術に基づく GPU コンピューティングの性能評価

菅原 誠[†1] 佐藤功人[†1] 小松一彦[†2]
滝沢寛之[†1,†3] 小林広明[†2,†3]

近年，描画処理用プロセッサ (Graphics Processing Unit: GPU) をアクセラレータとして利用して高速化を実現する複合型計算システムが普及しつつある．しかし，GPU を利用するためには，既存のプログラムを GPU 向けのプログラムに移植する必要があり，移植コストが問題となっている．本論文では，既存のプログラムにディレクティブを追記することにより GPU 向けのプログラムを自動生成する技術に着目し，その実用性と実効性能を評価する．また，ディレクティブを用いることで実現できる最適化を示す．そして，単純な行列積のプログラムを用いて性能を評価し，自動生成されたプログラムが実用的な性能を実現できることを示す．

# Evaluation of GPU Computing Based on An Automatic Program Generation Technology

Makoto Sugawara,[†1] Katsuto Sato,[†1] Kazuhiko Komatsu ,[†2]
Hiroyuki Takizawa[†1] and Hiroaki Kobayashi [†2,†1]

Recently, heterogeneous computing systems that achieve high-performance computing by using Graphics Processing Units (GPUs) as accelarators draw much attention in the area of computation sciences. However, a problem in use of GPUs is that it is necessary to port an existing program to a program for GPUs. To relieve the porting effort, this paper focuses on the technology to automatically generate a GPU program by inserting directives into an existing sequential code and evaluates the sustained performance of the auto-generated program. In addition, we show the achievable code optimizations by using directives. A simple matrix multiplication program is used for the evaluation to demonstrate that the automatically generated code can achieve a high sustained performance.

## 1. はじめに

近年，描画処理用プロセッサ (Graphics Processing Unit: GPU) をアクセラレータとして利用して高速化を実現する複合型計算システムが普及しつつある．しかし，GPU を利用するためには，NVIDIA 社製の GPU を対象としたフレームワークである Compute Unified Device Architecture(CUDA)[1] や，複合型計算システム全般を対象とした Open Computing Language(OpenCL)[2] 等を用いてアプリケーションプログラムを実装する必要がある．さらに，GPU を効率的に利用するためには，そのアーキテクチャに関する詳しい知識が要求される．一方，科学技術計算のすべてのプログラマが GPU のアーキテクチャに精通しているとは限らない．このため，CUDA や OpenCL に加えて，容易に GPU の持つ演算性能を利用する方法が求められている．

このような背景から，GPU 向けのプログラムを自動生成するツールが注目されている．プログラム自動生成ツールはプログラマからの指示 (ディレクティブ）に基づいてプログラムを分析し，複合型計算システム向けのプログラムを自動生成する．このため，プログラマは元のプログラムを書き換えることなく，ディレクティブを追記するだけで，複合型計算システムの高い演算能力を利用できる．

本論文の目的は，現在利用可能なプログラム自動生成ツールの実用性とその限界を明らかにすることである．そのために，プログラム自動生成ツールである CAPS 社の HMPP[3] を対象とし，プログラム自動生成ツールにより生成されたプログラムと，専門知識を有するプログラマが移植・最適化したプログラムの実効性能を定量的に評価する．

## 2. 複合型計算システムのソフトウェア開発環境

### 2.1 OpenCL

OpenCL[2] は，Khronos によって策定されているベンダ非依存の複合型計算システム向け標準プログラミングフレームワークである．OpenCL は特定のベンダに依存しないプログラミング環境であるため，可搬性の高いプログラム（以下，OpenCL プログラム）を記述する

†1 東北大学大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
†2 東北大学サイバーサイエンスセンター
Cyberscience Center, Tohoku University
†3 科学技術振興機構 戦略的創造研究推進事業
Japan Science and Technology Agency, Core Research for Evolutional Science and Technology

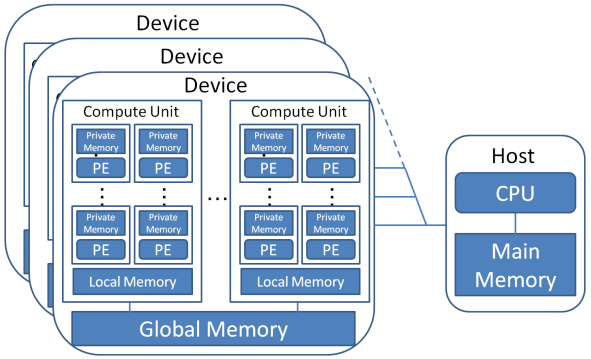


**図 1** OpenCL における複合型計算機のアーキテクチャ

ことができる．

OpenCL における複合型計算システムのアーキテクチャを図 1 に示す．OpenCL における複合型計算システムは，主に制御を行うホストと，演算のみを行う GPU 等の演算アクセラレータ (以下デバイス) から構成されている． 図 1 に示すとおり，OpenCL ではデバイスは複数の演算ユニット（Compute Units）から構成され，演算ユニットは複数の演算要素（Processing Elements）から構成されている．

OpenCL プログラムは，ホスト側で実行されるプログラムであるホストコードと，デバイス側で実行されるプログラムであるデバイスコードから成る．デバイスコードは複数のカーネルから構成されている．OpenCL では複数の演算ユニットが一つのデバイスコードを並列に実行する SPMD 方式 (Single-Program, Multiple-Data) である．SPMD 方式の並列処理を行うために，OpenCL では演算ユニットで実行される処理を work-item（以下，ワークアイテム）として定義している．ワークアイテムは階層的にまとめられて管理されている．複数のワークアイテムをまとめたものを work-group（以下，ワークグループ）と呼び，すべてのワークグループを合わせて NDRange と呼ぶ．

また，SPMD 方式の並列処理を記述するために，各ワークグループおよび各ワークアイテムを示す識別子（インデックス）が用いられる．ワークグループのインデックスをワークグループ ID，ワークアイテムのインデックスをワークアイテム ID と呼ぶ．それぞれのワークアイテムがワークアイテム ID に基づいて計算対象のデータを決めることにより，SPMD 方式の並列処理を実現する．ワークアイテム ID やワークグループ ID は，カーネルがホストコードから起動された際に自動的に決定される．また，カーネル起動時に生成されるワークアイテム数やワークグループサイズは，ホストコードで明示的に指示される．

図 1 で示すように，ホストとデバイスはそれぞれ別のメモリ空間を持っており，ホストコード上でデバイスメモリの確保やホストとデバイス間のデータ転送など，カーネルを実行するために必要な処理を記述する．さらに，OpenCL のデバイスは特徴の異なる複数のメモリ空間を利用することが可能であり，それらを適材適所で使い分けることが，高い実効性能を達成するためには必要である．

プライベートメモリ（Private Memory）は，デバイス上で演算を行う過程で値を一時的に保持するために使われる．そのため，プログラマが明示的に使用することはできない．ローカルメモリ（Local Memory）は，容量が小さく高速なオンチップメモリであり，プログラマが明示的に確保することで使用することができる．再利用性のあるデータをローカルメモリに格納することで，オフチップメモリへのアクセス回数を削減できるため，ソフトウェアマネージドキャッシュ（Software-managed Cache）とも呼ばれる．オフチップメモリであるグローバルメモリ（Global Memory）は，デバイス内で唯一ホストからのデータの読み書きが可能なメモリ空間であり，容量は大きいが低速である．ホストとデバイス間でデータ転送を行うためには，グローバルメモリにデータを一度格納しなければならない．

### 2.2 Hybrid Multicore Parallel Programming workbench(HMPP)

既存のプログラムを複合型計算システムへ移植する作業を補助するアプローチの一つとして，C や Fortran で記述されたプログラムにディレクティブを追記することにより複合型計算システム向けのプログラムを自動生成するアプローチが主流になりつつある[4)5)6)]．本論文では CAPS 社の Hybrid Multicore Parallel Programming workbench(HMPP)[3)] を用いて，プログラム自動生成の実用性を評価する．

HMPP はディレクティブに基づいて複合型計算システム向けのプログラムを自動生成するツールであり，HMPP コンパイラと HMPP ランタイムから構成される．HMPP では，演算アクセラレータ上で実行されるデータ転送，メモリ割り当て，カーネル実行などを含んだコードをコードレット（codelet）と定義している．ディレクティブを追加したソースコードから，HMPP コンパイラによってコードレットが生成される．コードレット以外のプログラムは CPU 用のコンパイラによってコンパイルされる．アプリケーションの実行時には，HMPP ランタイムがコードレットとデータ転送を管理することにより，計算機構成にあわせて利用可能なアクセラレータ上での実行が保証される．また，HMPP コンパイラによって生成された OpenCL プログラムを手動でさらに最適化することも可能であり，手動最適化

された OpenCL プログラムから，コードレットを生成することもできる．

HMPP で OpenCL プログラムを自動生成するためには，プログラム上でアクセラレータへ割り当てたい部分にディレクティブを挿入する．HMPP のディレクティブは，OpenMP[7] のディレクティブと同様に，HMPP を利用できない環境下では無視される．このため，ディレクティブを追記したプログラムは，ディレクティブを追記する前のプログラムと同様に CPU で実行することができ，プログラムの可搬性を保つことができる．

OpenCL プログラムへの手動移植作業では，アクセラレータ上のデバイスメモリの確保，ホストとデバイス間のデータ転送を明示的に記述する必要があり，アクセラレータで並列処理を行いたい部分のカーネルへの変換を行う必要がある．このため，一般に OpenCL プログラムの行数は，元のプログラムと比較して増加する．一方，プログラム自動生成ツールである HMPP を用いる場合，わずか一行のディレクティブを追記するだけでも GPU を利用できるため，移植作業に要する時間は大幅に短縮される．

さらに，HMPP には OpenCL の仕様に対応するディレクティブが提供されている．たとえば，HMPP により自動生成される OpenCL プログラムのワークグループサイズは標準で 32×4 に設定されているが，ディレクティブを追加することでワークグループサイズを指定できる．CPU と GPU 間のデータ転送に関するものや，ローカルメモリ等の OpenCL のメモリ階層を使用することを明示的に指定することが可能なディレクティブも用意されている．これらのディレクティブを既存のプログラムに適切に追加することで，自動生成される OpenCL プログラムに最適化を施すことができる．

## 3. HMPP および OpenCL を用いた一般行列積の実装

### 3.1 OpenCL プログラムへの変換

C や Fortran で記述されたプログラムを変換するためには，アクセラレータで実行したい部分をカーネルとして書き直す必要がある．代表的なアクセラレータである GPU は，大量のワークアイテムを並列実行する処理方式であるため，高い性能を達成するにはデータ並列性を利用することが必須である．一般に，プログラム中のループ処理は実行時間の大部分を占めるとともに，ループの繰り返し方向でデータ並列性が存在することが多いため，ループ処理をカーネルとして書き直すことで高速化を図ることができる．ループ処理を並列化する場合，ループ変数のいくつかをワークアイテム ID に置き換えることで実現できる．

図 2 に OpenCL を用いた場合の一般行列積のカーネルを示す．図 2 の 7 行目と 8 行目でワークアイテム ID を取得し，ループ変数とすることでそれぞれのワークアイテムが並列処理を行う．このほかに，カーネルの実行に必要なデータはホスト側のメインメモリに保持されているため，メインメモリ上のデータを GPU のグローバルメモリへと転送するコードも記述する必要がある．

```
1  ////////////////////////////////////////////////////////////////
2  // MatrixMul : C = alpha * A * B + beta * C
3  // m is A's width , n is A's height and k is B's height
4  ////////////////////////////////////////////////////////////////
5  __kernel void MatrixMul( int m, int n, __global float*A, __global float* B, __global float* C )
6  {
7     int i = get_global_id(0); // work-item ID
8     int j = get_global_id(1); // work-item ID
9     int l; // Induction variables
10    float AB = 0.0f; // Temporary result
11    for( l = 0; l < n ; ++l){
12       AB += A[i * m +l] * B[ l * n + i];
13    }
14    C[ j* m + i] = alpha * AB + beta * C[ j* m + i];
15 }
```

**図 2** 一般行列積を行う OpenCL のデバイス用のカーネル

HMPP を用いる場合には，図 3 の 5 行目に示すように，デバイスで実行したい処理の部分にディレクティブを 1 行追記し，GPU 上で書き込みがある配列の指定をすることで HMPP コンパイラがカーネルとデータ転送を行う OpenCL プログラムを自動生成することができる．また，関数呼び出し部分にもディレクティブを 1 行追加する必要があるため，2 つのディレクティブを追記する．

### 3.2 GPU 向けの最適化

アクセラレータの演算性能を効率的に利用するためには，アクセラレータのアーキテクチャを考慮して OpenCL プログラムを最適化する必要がある．例えば，アクセラレータの一つである GPU は，高い浮動小数点演算性能に対してグローバルメモリのバンド幅は相対的に低いという特徴がある．そのため，GPU をアクセラレータとして利用して高い実効性能を達成するためには，グローバルメモリアクセス回数を削減することにより，限られたバンド幅を有効に利用する必要がある．

グローバルメモリアクセス回数を削減する方法の一つとして，高速なオンチップメモリであるローカルメモリを使用する方法が考えられる．ローカルメモリは，同一ワークグループ内のワークアイテム間で共有される．このため，同一ワークグループ内のワークアイテムが

```
1  ////////////////////////////////////////////////////////////////////////////////
2  // MatrixMul : C = alpha * A * B + beta * C
3  // m is A's width , n is A's height and k is B's height
4  ////////////////////////////////////////////////////////////////////////////////
5  #pragma hmpp MatrixMul codelet, target=OPENCL, args[C].io=inout
6  void MatrixMul( int m, int n, int k, float* A, float* B, float* C, float alpha, float beta)
7  {
8     int i,j,l; // Induction variables
9     float AB; // Temporary result
10    for( int j = 0 ; j < m ; j++ ) {
11       for( int i = 0 ; i < k ; i++ ) {
12          AB = 0.0f;
13          for( int l = 0 ; l < n ; l++ ){
14             AB += A[j * m + l ] * B[ l * n + i] ;
15          }
16          C[ j * m + i] = alpha * AB + beta * C[ j * m + i] ;
17       }
18    }
19 }
```

**図 3** HMPP ディレクティブを追加した一般行列積を行う C プログラム

同じデータに複数回アクセスする場合，そのデータをローカルメモリに保持することによって，グローバルメモリアクセス回数を削減することができる．

図 4 に示すように，行列 C の行の計算には，行列 B の同じ行の行要素が繰り返し利用される．同様に，行列 C の列の計算には，行列 A の同じ列が繰り返し利用される．このため，これら再利用性のある要素をローカルメモリに格納し，同一ワークグループ内のワークアイテムで共有することで，グローバルメモリアクセス回数を大幅に削減することが可能である．

しかし，ローカルメモリの容量は小さいため，行列 A と行列 B の値をすべて格納することはできない．ローカルメモリを用いる最適化手法のひとつである，ブロッキング[8] の概略図を図 5 に示す．ブロッキングでは，行列 C の計算をいくつかのブロックに分割し，必要なデータのみをローカルメモリに格納することで，ローカルメモリ上でデータを共有しローカルメモリを効率的に使用する．

HMPP では，ローカルメモリを使用するディレクティブが用意されている．よって，元のプログラム上でブロッキングを行い，ローカルメモリを使用するディレクティブを用いることで間接的に GPU 向けの最適化を施すことができる．

他の最適化手法として，NVIDIA 社製の GPU の場合 16 ワークアイテム毎に一括してグ

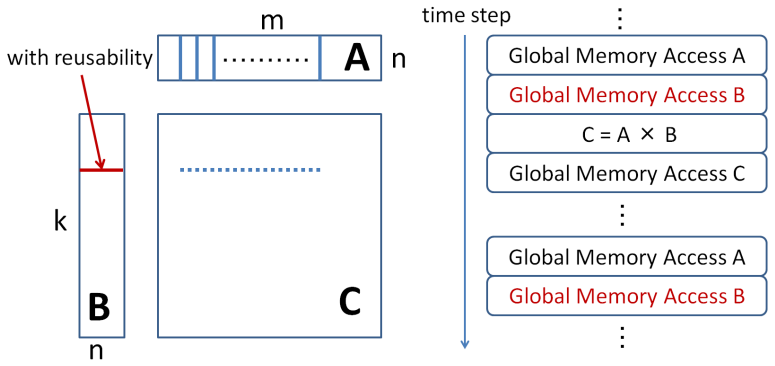


**図 4** 一般行列積の概略図

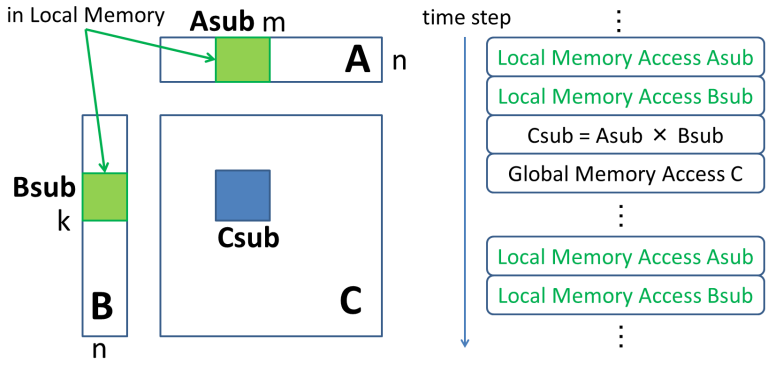


**図 5** ブロッキングの概略図

ローバルメモリにアクセスする[9] ため，ワークグループサイズが実効性能に影響を与える．多くの場合，ワークグループサイズの第 1 次元の大きさを 16 の倍数としたとき高い性能が得られる[8]．HMPP では，最適なワークグループサイズを自動的に設定する機能はないが，ディレクティブを追加することによりワークグループサイズを設定することができる．

## 4. 性能評価

自動生成されたプログラムの評価を行うために，OpenCL を用いて手動で一般行列積を実装したプログラムとの間で性能比較を行う．評価環境には，CPU に Intel Core i7 920 を，アク

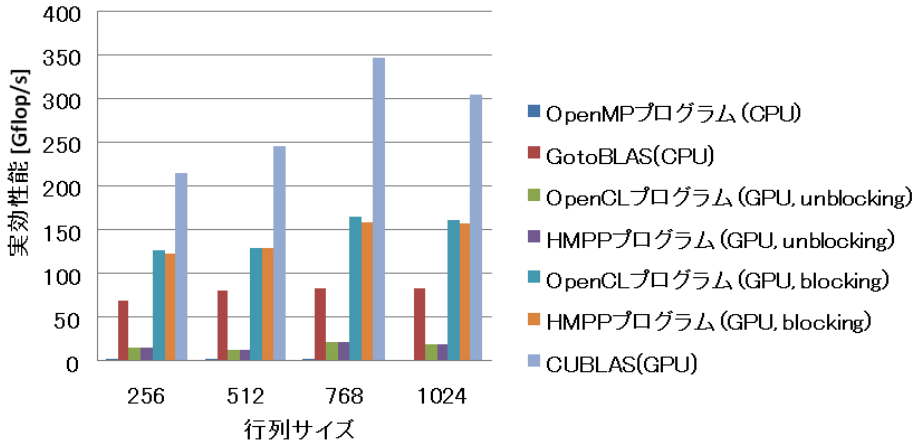


**図 6** Core i7 と Tesla C1060 における性能の比較

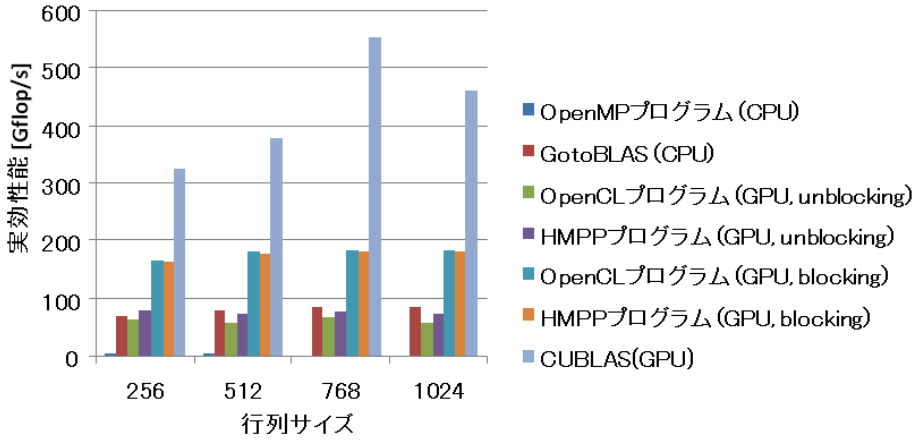


**図 7** Core i7 と Tesla C2070 における性能の比較

セラレータに NVIDIA Tesla C1060 と NVIDIA Tesla C2070 を用いる．OS は Cent5.5(Linux 2.6.18)，コンパイラは GCC4.1.2 と HMPP version2.4.0 を使用する．また，GPU を利用した場合の性能は全てカーネルの実行時間を評価の対象とし，CPU と GPU 間のデータ転送時間は考慮しない．本評価の中では，OpenCL を用いて手作業で実装したプログラムを OpenCL プログラム，HMPP を用いて自動生成したプログラムを HMPP プログラム，OpenMP を用いてマルチコア CPU 向けに実装したプログラムを OpenMP プログラムと表記する．このほかに，アーキテクチャに最適化されている数値演算ライブラリとして，GotoBLAS[10] および CUBLAS[11] を用いて性能比較を行う．なお，ワークグループサイズは OpenCL プログラムおよび HMPP プログラムの両方とも 16 × 16 に設定し，OpenMP プログラムにおける並列スレッド数は 8 とする．本評価で用いる一般行列積のルーチンは，3 重ループで単精度の行列の積を計算するプログラムであり，正方行列サイズを 256，512，768，1024 と変化させて性能を 10 回測定し，平均値を求める．

単精度一般行列積プログラムを用いて性能を比較した結果を，キャッシュメモリを搭載しないアクセラレータである Tesla C1060 と，搭載する Tesla C2070 毎に図 6，図 7 にそれぞれ示す．図 6 と図 7 では，OpenMP によって CPU で並列処理を行った場合の性能 (OpenMP プログラム)，CPU 向けに高度に最適化されたライブラリである GotoBLAS を用いた場合の性能 (GotoBLAS)，GPU 向けに高度に最適化されたライブラリである CUBLAS を元の C のプログラム上で用いた場合の性能 (CUBLAS)，および OpenCL プログラムの性能を比較している．手動で移植を行った場合の性能 (図中の OpenCL プログラム) と HMPP によって自動生成されたコードの性能 (図中の HMPP プログラム) は，それぞれブロッキングを行った場合 (図中の blocking) と行わない場合 (図中の unblocking) に分けて示されている．

図 7 より，ブロッキングを行わない場合，HMPP プログラムを Tesla C2070 で実行した性能は，OpenMP を用いて CPU で実行した性能に対して最大で約 73 倍の実効性能を達成している．また，図 6 では，HMPP プログラムは OpenCL プログラムとほぼ同等の性能を実現している．手動移植の際に追加したプログラム行数は 55 行であり，HMPP を用いた場合ではディレクティブを 3 行追加するだけであった．この結果から，HMPP は少ない労力で GPU の演算性能を利用可能であり，同じ処理を行う CPU のプログラムと比較して非常に高い性能を達成でき，手作業で移植を行った場合に比べてもほぼ同じ性能を達成可能であることが明らかとなった．また，Tesla C2070 のようにキャッシュメモリを搭載しているアクセラレータでは，ブロッキングを行わない場合でも HMPP プログラムの性能は GotoBLAS に匹敵する性能が得られている．これより，アクセラレータによっては簡単なディレクティ

ブを追加するだけで GPU の性能を引き出すことが可能であることが示された．

一方，Tesla C1060 で実行した OpenCL プログラムの性能は GotoBLAS を用いた場合の約 20%程度となっている．CPU と比較して GPU の方が理論演算性能が高いにもかかわらずこのような結果となる理由は，GotoBLAS が CPU の性能を最大限に引き出しているのに対して，OpenCL プログラムは図 2 に示されるように 3 重ループによる行列積を素直にカーネル化しただけの単純なプログラムであり，GPU の性能を十分に引き出せないためである．よって，高い実効性能を得るためには個々の GPU 向けの最適化を行わなければならない．

次に，代表的な最適化であるブロッキングを行なった場合の HMPP プログラムの性能と GotoBLAS の性能を比較する．ブロッキングを行うことでグローバルメモリアクセス回数が削減され，GotoBLAS を用いた場合の CPU 実行時に対して最大で約 2 倍の実効性能を達成できている．この結果から，プログラム自動生成技術とブロッキングのような一般的なコード最適化技法を組み合わせることで，GPU の演算性能を引き出すことが可能であり，CPU だけでは実現困難な高い実効性能を達成できることがわかる．

さらに，ブロッキングを行った場合の HMPP プログラムの性能と CUBLAS の性能を比較する．ブロッキングを行うことにより HMPP プログラムは，CUBLAS の半分程度の実効性能を達成している．実アプリケーションの開発において，すべてのカーネルが CUBLAS のように徹底的に最適化されることは稀である．このため，CUBLAS のような高度に最適化されたライブラリが存在しない場合であっても，HMPP と一般的な最適化技法を組み合わせることで十分高い性能を達成することができる．

以上より，少ないコンパイラ指示行で大幅な性能向上を容易に達成できる HMPP は，アクセラレータを利用するアプリケーションの開発において有用なツールであると言える．

ブロッキングを行っていない場合，Tesla C2070 では HMPP プログラムのほうが OpenCL プログラムよりも高い性能を示す現象が見られる．両者をアセンブリコードレベルで比較してもわずかな違いしかなく，性能差の原因解析は今後の課題である．また，ブロッキングを行なった場合では Tesla C1060 と Tesla C2070 の両方において，HMPP プログラムの方が OpenCL プログラムに比べてわずかに性能が低い．生成された OpenCL プログラムを比較すると for 文の変換において if 文が追加されている．これは，HMPP コンパイラが変数の大小関係を仮定できないために生じているもので，ほかの部分に処理内容の大きな違いは見られないことから，この部分が性能差を生じている原因と考えられる．プログラムを記述する際の前提条件として用いる情報量の違いに起因している．

本評価では，HMPP のディレクティブに基づく自動変換によって，手動で最適化された OpenCL プログラムと同等の性能を達成できるコードを生成できることが明らかになった．また，手動で OpenCL プログラムを書く場合と同様に，ブロッキングのような一般的な最適化技法も性能向上に非常に有効であることが示された．ただし，ディレクティブを用いた最適化を行う場合においても，OpenCL や GPU のメモリ階層に関する知識が必須となる．このようなアクセラレータ依存の最適化を自動化することは，プログラムの自動生成ツールにおいて重要な研究課題である．

## 5. おわりに

本論文では，GPU 向けプログラム自動生成技術の実用性を明らかにするために，HMPP により自動生成されたプログラムの性能を単精度一般行列積を用いて評価した．既存のプログラムにディレクティブを追記することで自動生成された OpenCL プログラムは，OpenMP を用いてマルチコア CPU 上で実行した場合よりも高い性能を示すとともに，手作業で移植を行った OpenCL プログラムとほぼ同等の性能が得られることが明らかとなった．また，ブロッキング等の一般的な最適化も，適切なディレクティブを用いることで既存のプログラム上で適用可能であることを示し，GPU 向けに高度に最適化された CUBLAS の半分ほどの性能まで達成できることを明らかにした．一方で，アクセラレータにキャッシュメモリが存在する場合には，ブロッキングのような最適化を施さなくても，GotoBLAS に匹敵するほどの性能を得られることを示し，アクセラレータのアーキテクチャによってはディレクティブを追加するだけでも高い性能が得られることを示した．

しかし，アクセラレータのアーキテクチャに合わせて最適化のためのディレクティブを自動的に追加することは自動化されておらず，プログラマに知識と経験を要求する．よって，このような最適化を支援あるいは自動化していくことが今後の大きな課題である．

今後は行列積のような単純なプログラムだけではなく，実アプリケーションにおいても同様の評価を行っていく予定である．

## 謝　辞

本論文の執筆にあたり，JCC ギミック社 (CAPS 社日本代理店) のスタッフの方々には大変有用なご助言をいただきました．本研究の一部は，文部科学省科研費若手研究 (B)(23700028) と科学技術振興機構 (JST) 戦略的創造研究推進事業 (CREST) 研究領域「ディペンダブル VLSI システムの基盤技術」研究課題「自己修復機能を有する 3 次元 VLSI システムの創製の助成を受けている．

## 参 考 文 献

1) NVIDIA Corporation. *NVIDIA CUDA Programming Guide 3.0*, 2010.
2) Khronos OpenCLWorking Group. The OpenCL Specification version 1.1.
3) R.Dolbeau et al. HMPP: A Hybrid Multicore Parallel Programming Environment. *Workshop on GPGPU 2007*, 2007.
4) The Portland Group. PGI Accelerator Programming Model for Fortran & C. http://www.softek.co.jp/SPG/Pgi/Accel/, 2010.
5) Seyong Lee and R.Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. pp. 1 –11, nov. 2010.
6) T.D. Han and T.S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *Parallel and Distributed Systems, IEEE Transactions on*, Vol.22, No.1, pp. 78 –90, jan. 2011.
7) OpenMP.org. OpenMP Application Program Interface. http://openmp.org/wp/, 2008.
8) NVIDIA Corporation. *NVIDIA OpenCL Best Practice Guide 2.3*, 2009.
9) Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, Vol.28, pp. 39–55, 2008.
10) 後藤和茂. Texas Advanced Computing Center. http://www.tacc.utexas.edu/.
11) NVIDIA Corporation. CUDA Toolkit 4.0 CUBLAS Library. http://developer.nvidia.com/nvidia-gpu-computing-documentation, 2011.