

# HMPP (Hybrid Multicore Parallel Programming) Workbench



my recommendations on research & development tools.

高橋 徹<sup>†</sup>, 三浦 衛<sup>†</sup>

キーワード: HMPP Workbench, GPU, 並列処理, 高速化

## 1. ま え が き

コンピュータグラフィックス, 画像処理, コンピュータビジョンなどの視覚情報処理の分野や, 計算論的モデリング, 科学計算, 経済シミュレーションなど, 幅広い多様な分野において, 高速に動作するプログラムが常に求められている. この要求に対して, 近年では, GPU (Graphics Processing Unit) を用いた並列処理によるプログラムの高速化が注目を集めている. GPUは, 3Dモデリングなどのような高負荷処理を効率的に行うことを目的として開発されている. 数百もの演算コアを一つにしたチップ, 同時処理される数千ものスレッド, および広域なメモリーバンド幅によって, 超マルチスレッドを実現しており, GPUの演算性能は極めて高い. この特徴をグラフィックス処理だけでなく, 汎用計算に利用する「GPGPU」(General-Purpose Computation on Graphics Processing Units) と呼ばれる試みが注目を集めている. これまでに計算量が多い科学計算などを中心に, さまざまな実装が提案されている. しかし, GPUを汎用のプログラミングに利用するためには, OpenGLなどのグラフィックスAPI (Application Programming Interface) を用いて処理する必要がある. このため, グラフィックスの専門家以外がGPUを利用する場合, プログラミングに大きな困難を伴う.

これに対して, 近年では, GPUを用いた汎用プログラミングのための高水準な開発環境が登場している. その代表として, NVIDIA社のCUDA (Compute Unified Device Architecture)<sup>1)2)</sup>が挙げられる. CUDAは, C言語をベースとしたプログラミング環境であり, 従来のグラフィックスAPIを用いた場合と比べ, 非常にプログラムしやすく, 既存のコードに組込むことが可能となっている. しかし, CUDAの習得や利用は, GPUの独特なアーキテクチャを利用するための特殊な記法やプログラミングモデルを理解する必要があり, 時間的なコストがかかる. 特にGPGPUに不慣れな研究者や開発者にとって, GPGPUを利用する敷

居は依然として高い. そこで, 本稿では, より手軽にGPUを汎用計算に利用するためのツールである「HMPP (Hybrid Multicore Parallel Programming) Workbench」を紹介する. 画像処理のプログラムを例に, 簡便な記述および操作で, GPUを用いた高速化が可能であることを示す.

## 2. HMPP Workbenchの概要

HMPP Workbench (以下, HMPP) は, CAPS Enterprise社<sup>3)</sup>が開発したGPUなどのハードウェアアクセラレータを対象とした並列プログラミング支援ツール群である. CAPS Enterprise社は, 2002年に仏国立情報学自動制御研究所 (INRIA), レンヌ大学等に所属する7名の研究者によるコンパイラ開発のプロジェクトチームが発祥である. 当初はクロスプラットフォームコンパイラを開発していたが, 現在はGPUコードジェネレータ開発に特化しており, HMPPパッケージソフトの開発販売が事業ドメインとなっている.

日本では, JCC-Gimmick社<sup>4)</sup>が唯一の販売代理店となっており (2010年2月現在), JCC-Gimmick社を通して同パッケージソフトを購入することができる. また, パッケージソフトの購入のみならず, 導入時のコンサルティングを受けることも可能である (2010年1月現在 Ver. 2.2.0). HMPPでサポートされている動作環境は以下の通りである.

### (1) OS

- ・ Debian GNU/Linux 4.0以降
- ・ RedHat Enterprise Linux 4.5以降
- ・ RedHat Enterprise Linux 5.1以降

### (2) ハードウェアアクセラレータ

- ・ CUDA 2.2以上をサポートしているNVIDIA製GPU
- ・ Brook+ 1.0およびCAL 1.0をサポートしているAMD製GPU
- ・ SSE2命令を扱えるIntel製もしくはAMD製CPU

### (3) コンパイラ

- ・ GCC 4.1以上
- ・ Intel C++ Compiler 9.1以上
- ・ Intel Fortran Compiler 9.1以上

HMPPの概要を図1に示す. ユーザが既存のコードに対

<sup>†</sup> 東北大学 大学院情報科学研究科

"HMPP (Hybrid Multicore Parallel Programming) Workbench" by Toru Takahashi and Mamoru Miura (Graduate School of Information Sciences, Tohoku University, Miyagi)

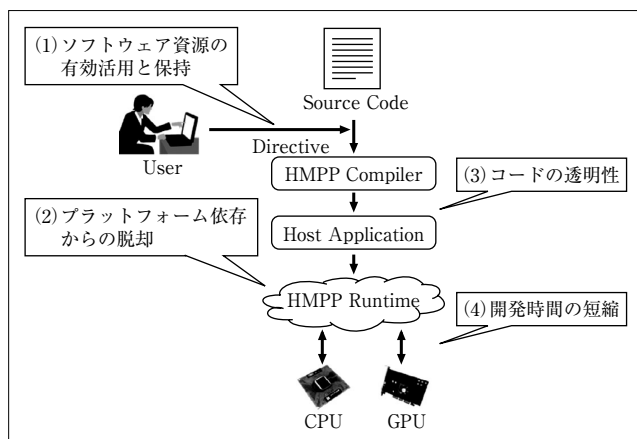


図1 HMPP Workbenchの概要

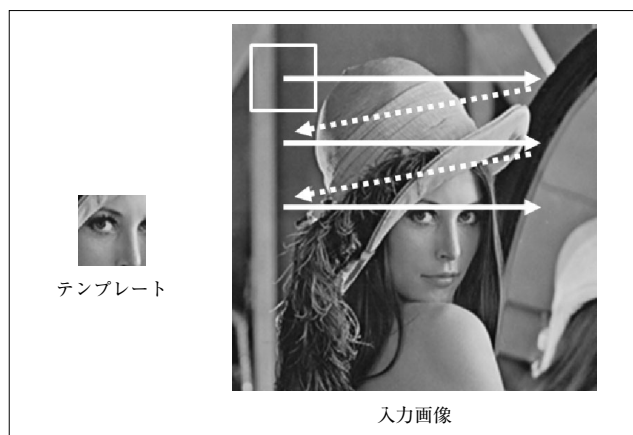


図2 ラスタスキャンによるテンプレートマッチング

してディレクティブと呼ばれる制御コードを挿入し、並列化を行う処理を指示することで、ハードウェアアクセラレータを用いた並列処理を実現する。HMPPの持つ4つの特徴を以下に示す。

#### (1) ソフトウェア資源の有効活用と保持

HMPPでは、ディレクティブを用いてGPU用のコードを生成する対象を指定する。このため、他のプログラミングツールと同時に使用することが可能である。また、プリプロセッサ等によりディレクティブを除去することが可能であるため、既存のコードを保持することが容易である。これらの特徴により、既存のソフトウェア資源を有効に活用および保持することができる。

#### (2) プラットフォーム依存からの脱却

GPUを用いた計算を行う場合、使用するマシンの動作環境を認識し、これに基づきプログラミングを行う必要がある。利用可能なデバイスの種類や数、利用可能なGPU向けのコードを把握しなければならない。HMPPでは、HMPP Runtimeを利用することでこの問題を解決している。HMPP Runtimeがプログラムとデバイスとの間に位置することで、実行時に環境を識別することができ、複数台のGPUを用いた環境への移行も容易となる。

#### (3) コードの透明性

HMPPを利用する際には、対応するコマンドを用いることで、ホスト向け・デバイス向けそれぞれのプログラムのコンパイルが行われる。このとき、HMPPは、それぞれに対応するコードを生成し、コンパイラを用いて実行コードを生成するため、すべてのソースが修正・再構成可能である。このため、生成されたコードの中身を、ユーザ側で確認することができるだけでなく、さらに効率の良いコードへと修正することが可能となっている。

#### (4) 開発時間の短縮

HMPPを利用すると、既存のコードにディレクティブを挿入し、コンパイルするだけでよい。GPUに対応し

たコードを自動で生成・実行することが可能となる。そのため、GPUのアーキテクチャへの理解が充分でなくとも、手軽にGPUを用いた高速化の利用が可能である。GPUに関する詳細な知識を必要としないため、開発時間の大幅な短縮を見込むことができる。

以上の特徴から、HMPPを用いることで、CやFortranで書かれた既存のコードを利用し、GPUを用いた高速化が可能になる。ディレクティブによって、対象の指定のみならず、より細かい制御を行うことも可能である。特に、ホスト・デバイス間のデータ転送のタイミング制御や入出力情報を指定することで、より効率的なプログラムを作成することも可能となっている。

### 3. HMPP Workbenchを用いたプログラミング

本章では、データ並列処理に適した画像処理を例に、HMPPの基本的なプログラミングの特徴を示す。画像処理の多くは、画像の各画素に対して一連の処理が行われるため、GPUを用いた並列処理に適している。本稿では、SSD (Sum of Squared Difference) を利用したテンプレートマッチング<sup>5)</sup>のプログラムを例に、HMPPを用いた高速化について示す。

テンプレートマッチングは、図2に示すように、あらかじめ定めた標準パターン(テンプレート)を用意し、入力画像とのマッチングを行う処理である。テンプレートマッチングを行うことで、用意したテンプレートと同じパターンが画像中に存在するかどうか、また存在するとしたらどの位置にあるのか調べることができる。テンプレートの大きさを $M \times N$ ピクセル、テンプレートの位置 $(i, j)$ における画素値を $T(i, j)$ 、テンプレートと重ね合わせた入力画像の画素値を $I(i, j)$ とすると、SSDによる相違度 $R$ の計算は、以下の式で定義される。

```

1  #pragma hmpp tm_ssd codelet, target=CUDA, args[out].io=out
2  void templatematch_ssd(int widthof_block, int heightof_block, int widthof_image, int heightof_image,
3                          int T[heightof_block][widthof_block],
4                          int G[heightof_image][widthof_image],
5                          int out[heightof_image][widthof_image])
6  {
7      int i;
8      for(i=0; i<heightof_image; i++) {
9          int j;
10         for(j=0; j<widthof_image; j++) {
11             int dm = 0;
12             int tmp = 0;
13             int validpixel = 0;
14             int y;
15             for(y=0; y<heightof_block; y++) {
16                 int x;
17                 for(x=0; x<widthof_block; x++) {
18                     int im_x = j - widthof_block/2 + x;
19                     int im_y = i - heightof_block/2 + y;
20                     if(0 < im_x && im_x < widthof_image && 0 < im_y && im_y < heightof_image) {
21                         tmp = G[im_y][im_x];
22                         dm += (tmp - T[y][x])*(tmp - T[y][x]);
23                         validpixel++;
24                     }
25                 }
26             }
27             out[i][j] = dm/validpixel;
28         }
29     }
30 }

```

図3 サンプルプログラム (SSDによるテンプレートマッチング)

$$R = \sum_{j=0}^{N-1} \sum_{i=0}^{M-1} (I(i,j) - T(i,j))^2$$

相違度 $R$ が最も小さくなる位置が、入力画像中においてテンプレートと同じパターンを持つと考えられる。

本稿で例に示すサンプルプログラムでは、ラスタスキャンによる全探索を行い、探索範囲の相違度を計算し、相違度マップを生成する。ラスタスキャンによる全探索において、各画素位置における相違度の計算は独立しているため、並列化による高速化が期待できる。そこで、サンプルプログラムでは、画像全体から相違度を計算する部分にディレクティブを挿入し、各画素位置における相違度の計算をGPUで処理させることで高速化を図る。なお、サンプルプログラムでは、画像の入出力の処理を省き、乱数によって生成された2次元データ配列を入力画像と見なし、その配列の一部分を切り出し、テンプレート画像としている。

図3は、SSDの相違度マップを算出する関数を記述したサンプルプログラムである。1行目がHMPPを利用するために挿入したディレクティブであり、「tm\_ssd」がラベル、

「target=CUDA」で、ハードウェアアクセラレータのターゲットとして「CUDA」を指定している。

図4は、main関数を記述したプログラムであり、40行目でSSDによる相違度を算出するプログラムを呼び出している。この関数呼び出しの直前である39行目にディレクティブを挿入する。ここでは、前述のディレクティブで指定したラベル「tm\_ssd」を記述するだけである。HMPPを利用するために行ったコードの改変は、図3のプログラムの1行目と図4のプログラムの39行目のディレクティブの文だけであり、計2行の追加にとどまっている。このサンプルプログラムをHMPPによりコンパイルすることで、自動的に実行ファイルおよびディレクティブで指定した処理部分のCUDA用のコード「tm\_ssd.cu」とCUDAのランタイムライブラリー「tm\_ssd.so」が生成される。

HMPPでサンプルプログラムをコンパイルし、実行した結果を表1に示す。入力画像としてVGA(640×480ピクセル)とXGA(1,024×768ピクセル)を想定し、それぞれについて、テンプレート画像のサイズを32×32ピクセルおよび64×64ピクセルとした場合について、実行時間



```

1  double main(void)
2  {
3      int i;
4
5      int X[HEIGHTOF_IMAGE][WIDTHOF_IMAGE];
6      int Y[HEIGHTOF_IMAGE][WIDTHOF_IMAGE];
7      int dm[HEIGHTOF_IMAGE][WIDTHOF_IMAGE];
8
9      int j;
10     srand((unsigned int)time(NULL));
11     for(i=0; i<HEIGHTOF_IMAGE; i++) {
12         for(j=0; j<WIDTHOF_IMAGE; j++) {
13             X[i][j] = rand()%256;
14             Y[i][j] = rand()%256;
15         }
16     }
17
18     int x = 100;
19     int y = 100;
20     int im_x, im_y;
21
22     int T[HEIGHTOF_BLOCK][WIDTHOF_BLOCK];
23
24     for(i=0; i<HEIGHTOF_BLOCK; i++) {
25         for(j=0; j<WIDTHOF_BLOCK; j++) {
26             im_x = x - WIDTHOF_BLOCK/2 + j;
27             im_y = y - HEIGHTOF_BLOCK/2 + i;
28             if(0 < im_x && im_x < WIDTHOF_IMAGE && 0 < im_y && im_y < HEIGHTOF_IMAGE) {
29                 T[i][j] = X[im_y][im_x];
30             } else {
31                 T[i][j] = 0;
32             }
33         }
34     }
35
36     double t1, t2;
37     t1 = gettimeofday_sec();
38
39     #pragma hmpp tm_ssd callsite
40     templatematch_ssd(WIDTHOF_BLOCK, HEIGHTOF_BLOCK, WIDTHOF_IMAGE, HEIGHTOF_IMAGE, T, Y, dm);
41
42     t2 = gettimeofday_sec();
43
44     return (t2-t1);
45 }

```

図4 サンプルプログラム (main関数)

を測定した。HMPPを用いた効果を確認するために、元のプログラムをCPUのみで処理した実行時間と比較した。それぞれの場合について100回実行し、その平均時間を比較した。実行環境は以下の通りである。

- ・ CPU: Intel Xeon E5450 3.00GHz
- ・ Memory: 32GB
- ・ GPU: Tesla C1060

・ コンパイラ: GCC 4.1.2, HMPP 2.2.0  
 実行時間を表1に示す。HMPPを利用することで、CPUのみで処理した場合の実行時間に比べ、大幅に高速処理できていることがわかる。入力画像およびテンプレート画像のサイズが大きくなるほど大幅な処理速度向上が達成されており、GPUを用いたことによる高速化の効果を確認できる。

表1 サンプルプログラムの実行速度

入力画像サイズ[ピクセル]	640×480 (VGA)		1024×768 (XGA)	
テンプレート画像サイズ[ピクセル]	32×32	64×64	32×32	64×64
CPUのみの処理時間[s] (GCCでコンパイル)	4.1416	16.4708	10.7711	42.6671
CPU+GPUを用いた処理時間[s] (HMPP+GCCでコンパイル)	0.1842	0.3834	0.2919	0.8045
処理時間比(CPUのみ/ CPU+GPU)	22.5	43.0	36.9	53.0

CPU: Intel Xeon E5450 3.00GHz GPU: Tesla C1060  
GCC 4.1.2 HMPP 2.2.0

本稿で示したサンプルプログラムでは、NVIDIA製のGPUを用いていたが、他のハードウェアアクセラレータを用いた場合においても、同様の記述で並列処理が可能である。具体的には、挿入するディレクティブにおいて、「target=CUDA」の部分を使用するハードウェアアクセラレータに応じて書き直せばよい。例えば、ATI製のGPUを用いる場合、「target=CAL」とコードを書換え、HMPPを用いてコンパイルすれば、ATI製のGPU向けのコードを自動的に生成する。このように、異なるハードウェアアクセラレータを用いた場合についても簡便な記述と操作で対応可能であり、既存のコードを保持した上でその資源を有効に利用することができる。

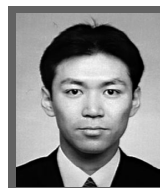
#### 4. む す び

本稿では、HMPP Workbenchを紹介した。HMPPの使用例として、画像処理のテンプレートマッチングを実装し、GPUを用いて高速に処理が可能であることを示した。HMPPは、CやFortranで開発した既存のコードにディレクティブを挿入するだけで高速化を実現可能である。また、コンパイラにGPUコードジェネレータを内蔵しており、専門的なGPUの知識が不要である。さらに、本稿ではとりあげていないが、OpenMP等の利用により、マルチGPU環境に対応することも可能である。

HMPPは、最小の開発時間で非常に高いパフォーマンスが得られる可能性を有しており、その利用の幅は大きいと考えられる。現在のところ、HMPPの動作環境は64ビットのLinuxに限られてはいるが、近日中にWindows版もリリースする予定である。また、サポートするハードウェアアクセラレータも拡充され、OpenCL等のサポートをすることで、より利便性の高いものになる予定である。現在までにNVIDIAやATIから提供されているハイエンドのGPUは、1TFlopsを超える性能を持つと言われており、今後、GPUを汎用計算に積極的に利用する動きは加速すると思われる。HMPPのさらに詳しい情報については、CAPS-Enterprise社<sup>2)</sup>やJCC-Gimmick社<sup>4)</sup>のホームページを参照していただきたい。  
(2010年3月8日受付)

#### 〔文 献〕

- 1) CUDA Zone, [http://www.nvidia.co.jp/object/cuda\\_home\\_new\\_jp.html](http://www.nvidia.co.jp/object/cuda_home_new_jp.html)
- 2) L. Hamid, 高橋裕樹: "CUDA (Computer Unified Device Architecture), 映情学誌, 63, 4, pp.465-470 (Apr. 2009)
- 3) CAPS-Enterprise, <http://www.caps-entreprise.com/index.php>
- 4) JCC-Gimmick, <http://www.jcc-gimmick.com/>
- 5) 奥富正敏 (編), デジタル画像処理, CG-ARTS協会 (2004)



たかはし 高橋 徹 2006年、東北大学工学部情報卒業。2008、同大学大学院情報科学研究科修士課程修了。現在、同大学院博士課程在学中。画像・映像処理に関する研究に従事。



みつうら 三浦 衛 2010、東北大学工学部情報卒業。現在、同大学大学院修士課程在学中。画像・映像処理に関する研究に従事。