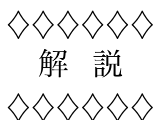

九大研究用計算機システムでのGPGPU活用法
- GPGPU，Intel MPI並列，OpenCVの併用による画像処理高速化 -

林　豊洋[1]

1　概要

情報科学センターでは，大規模演算やクラウドコンピューティングに対する学内の要求に応えるため，研究用計算機システムの利用支援を行っています．2014年度は大規模演算環境として，九州大学情報基盤研究開発センターに導入されている高性能演算サーバシステム (構成1:富士通 PRIMERGY CX400 クラスタ) およびスーパーコンピュータシステム (構成2:富士通 PRIMEHPC FX10) の利用補助を行っています．

CX400クラスタは，Intel64アーキテクチャを採用したPCサーバをクラスタ化した環境であり，多くの方が所有しているPCと同様のアーキテクチャとなります．従って，Intel64アーキテクチャ用に用意された様々なツール・ライブラリが利用可能です．加えて，CX400クラスタにはNVidia製のGPGPUボードが搭載されており，CUDAを用いた大規模演算が可能です．

本稿では，CX400クラスタの活用例として，以下の特徴を持つ画像処理アプリケーションを作成します．

- OpenCV (CX400クラスタに未導入) をIntel Compilerを用いてセルフビルドを行い，プログラム中で利用する
- CUDAを用いて演算の一部をGPGPU実装する
- MPIによりプログラムを複数のノードに分散化・並列動作させる

上記プログラムをCX400クラスタでMPI動作させ，最大16基のGPGPUボードを用いた大規模な画像処理アプリケーションの性能を示します．なお，実行手順や必要な設定を記事中に掲載しています．順次参照することにより，本環境を試していただくことが可能です．

2　高性能演算サーバシステム (CX400クラスタ) の概要

高性能演算サーバシステム (CX400クラスタ) は，九州大学情報基盤研究開発センターの研究用計算機システムのサービスシステムとして，2012年10月より運用を開始しました．ハードウェア諸元およびソフトウェア構成については，以下の表1，2の通りです (九大Webサイト [1] より一部引用，加筆しています)．

ハードウェア諸元に示す通り，CX400クラスタはIntel64アーキテクチャの計算機によって構成されています．アーキテクチャが一般的なPCと同様であるため，研究室等で作成したプログラムを最低限

[1]情報科学センター　助教　toyohiro@isc.kyutech.ac.jp

表 1: CX400 クラスタハードウェア諸元

CPU	Intel Xeon E5-2680 2.70GHz (Intel64)
演算性能	345.6GFLOPS
主記憶容量	128GB
GPGPU	NVIDIA Tesla K20m (Kepler, 主記憶容量 5GB)
GPGPU 演算性能	1.17TFLOPS(倍精度)
総ノード数，性能	1476 ノード，966.2TFLOPS

表 2: CX400 ソフトウェア構成

OS	RHEL6 (64bit)
コンパイラ	Fortran, C, C++ (富士通，Intel), PGI Fortran
数値計算ライブラリ	SSL II, BLAS, ScaLAPACK 等
アプリケーション	Gaussian, ANSYS, MATLAB, AMBER 等
並列プログラム環境	MPI, OpenMP
GPGPU ツールキット・ミドルウェア	NVIDIA CUDA(ver. 5.0, 5,5)

の移植作業により実行できます (再コンパイルのみで可能な場合もあります)．また，従来から九大に導入されているシステムと同様に，本システムは富士通系のプログラミング環境が整備されています．富士通系のコンパイラや数値計算ライブラリ (SSL II 等) に依存したプログラムも，これまで通り利用できます．なお，本学は CX400 クラスタの包括利用契約を締結しています．利用形態によっては，最大 16 ノードを用いたバッチ処理が可能です．

CX400 クラスタで注目すべき点は，従来の富士通系の環境に特化した構成から，他の環境が利用できる形態に変更された点です．まず第一に，コンパイラとして Intel 製のコンパイラ (Fotran, C, C++，MPI 対応) およびライブラリ (MKL：数値演算ライブラリ，IPP：マルチメディアライブラリ,TBB：並列処理ライブラリ等) が追加され，利用可能になりました [2]．Intel 製のコンパイラは，富士通系のコンパイラと比較した場合，実行性能はやや劣る可能性がありますが，C++の標準実装に近いことが利点です．したがって，GNU コンパイラ向けに作成されたプログラムやライブラリのソースコードがそのままコンパイル可能で，かつ高い実行性能を得ることができます．

次に，グラフィックボードを活用した並列プログラムである GPGPU に対応しました．GPGPU として，科学技術計算用に特化した GPGPU ボードである NVIDIA Tesla K20m (Kepler 世代) を搭載しています．GPGPU ミドルウェアには，NVIDIA CUDA ver.5 系列が採用されています [3][5]．この組み合わせにより，Computing capability 3.5 までの CUDA プログラムの実装・実行が可能です[2]．

3 CX400 クラスタによる大規模演算例：画像処理の高速化

本稿では，CX400 クラスタで新たに提供された Intel コンパイラおよび GPGPU を活用した演算を行います．今回は,「画像処理の高速化」をターゲットとして解説を行います．

3.1 プログラムの設計概要

画像処理アルゴリズムの多くは，処理対象が画素，部分領域画像，データ列等であり，それぞれが他の対象に依存せず処理可能 (独立して処理可能) です．例えば，基本的な画像処理アルゴリズムである色

[2]初期の CUDA と比較して，定義可能なスレッド数が増えたため，コーディングが容易になった等のメリットがあります．

空間の変換やエッジ抽出等のフィルタ処理は，各画素毎に演算できるため，独立しています．本稿では，入力画像 (RGB 画像) を輝度変換し，輝度画像を出力する画像処理プログラムを作成します (図 1).

このような対象の演算を行う場合，計算機アーキテクチャの並列性を活用することにより，高速化を行うことができます．CX400 クラスタは，演算ノード単体の性能については，一般的な PC サーバと同等です．従って，多数の演算ノードにプログラムおよびデータを展開し，並列処理する SPMD(Single Program to Multiple Data) による大規模化が必要となります．今回は，Intel Compiler を用いてプログラムのコンパイルを行うため，**Intel MPI による並列処理**を行います (プログラムの記法は，通常の MPI[6] と同様です).

また，各演算ノードに入力されたデータ (部分領域画像) について，搭載された **GPGPU ボードを用いて並列処理**を行います．今回は，GPGPU ミドルウェアである NVIDIA CUDA を用いて，画像処理アルゴリズム (輝度変換) を GPU によって実行します．

加えて現在では，画像処理プログラムを開発する際，その実装を容易にすることや実装のエラーを低減するため，提供されたライブラリを用いることが一般的です．よく利用されるライブラリとして，OpenCV[4] が代表的です．

残念ながら，CX400 クラスタには OpenCV が導入されていません．ただし，OpenCV はソースコードを入手することにより，利用者自身がビルドすることが可能です．ビルドする際には，いくつかのライブラリが導入されている必要がありますが，必須ライブラリについては CX400 クラスタに導入済みであることを確認しています．本稿では，**Intel Compiler を用いた OpenCV のセルフビルド**を行い，高性能な OpenCV ライブラリを構築し，画像処理プログラムで利用します．

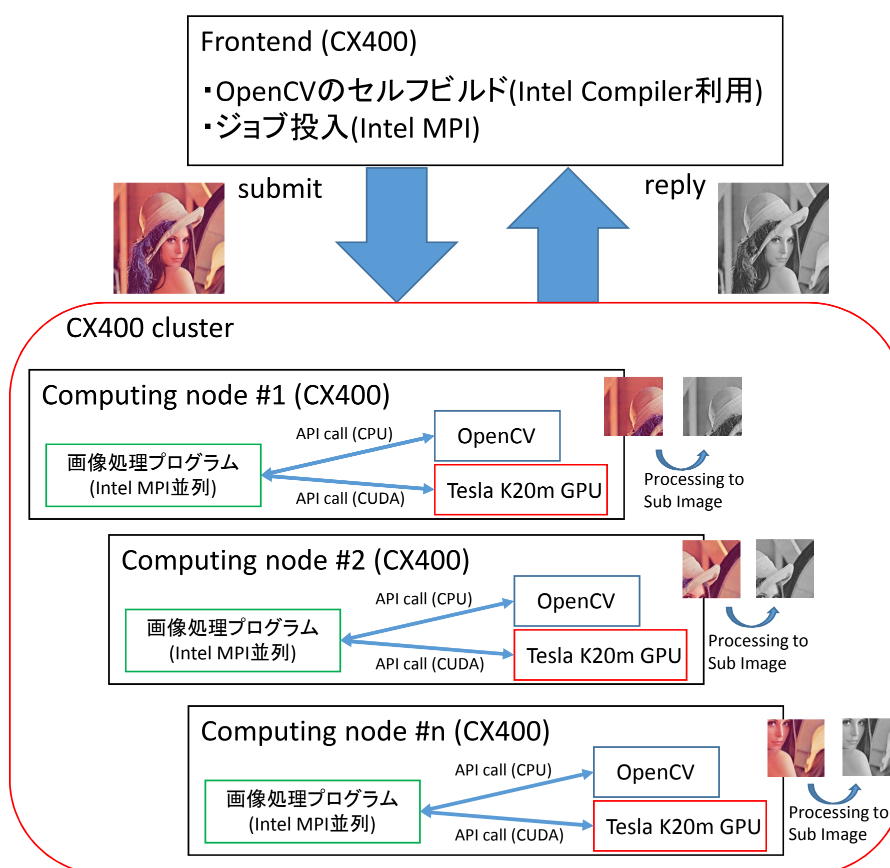

図 1: 画像処理プログラムの設計概要

プログラム構築の下準備　本稿では，Intel コンパイラおよび CUDA を利用します．また，OpenCV は各利用者のホームディレクトリ以下の "cvwork" サブディレクトリに展開し，OpenCV のバージョンは執筆当時最新であった 2.4.9 を利用します．これらのライブラリを実行時に呼び出すために，.bashrc に以下 (リスト 1) の設定の追加が必要となります．

リスト 1: 本環境向けの .bashrc

```
# 末尾に以下を追加
source /home/etc/intel2013.sh
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:~/cvwork/opencv-2.4.9/build/lib:
    $LD_LIBRARY_PATH
```

3.2 Intel Compiler を用いた OpenCV のセフルビルド

まずは，CX400 クラスタに導入されていない OpenCV について，セルフビルドを行い利用できるよう準備を行います．OpenCV のソースコードやサンプルデータは，zip 形式で sourceforge.net 上で配布されています．zip ファイルの展開後，cmake により環境に合わせた Makefile を作成・ビルドを行うことにより，各種バイナリファイルやライブラリ (.so) が作成されます．今回は，Intel Compiler を用いることにより，GNU Compiler を用いた場合よりも高速に動作する OpenCV 環境を構築します．Intel Compiler をビルドツールとして用いる場合，cmake のオプション中の C コンパイラ指定 (`CMAKE_C_COMPILER`) と C++ コンパイラ指定 (`CMAKE_CXX_COMPILER`) を明示する必要があります．これら CX400 上での OpenCV のビルド手順を以下のリスト 2 に示します[3]．

リスト 2: Intel Compiler を用いた OpenCV のビルド

```
$ cd && mkdir cvwork && cd cvwork
$ wget wget http://sourceforge.net/projects/opencvlibrary/files/opencv-unix
    /2.4.9/opencv-2.4.9.zip
$ unzip opencv-2.4.9.zip
$ cd opencv-2.4.9
$ cmake -DCMAKE_C_COMPILER=/usr/local/intel2013/composer_xe_2013_sp1.2.144/bin/
    intel64/icc -DCMAKE_CXX_COMPILER=/usr/local/intel2013/composer_xe_2013_sp1
    .2.144/bin/intel64/icpc -DBUILD_EXAMPLES=ON -DCMAKE_BUILD_TYPE=RELEASE -
    DINSTALL_C_EXAMPLES=ON -DWITH_OPENCL=OFF -DWITH_UNICAP=ON -DWITH_OPENMP=ON -
    DCMAKE_INSTALL_PREFIX=./build .
$ make && make install
```

Intel Compiler による性能向上を確認するため，OpenCV 付属の性能測定プログラム (`opencv_perf_core`) を実行し，演算完了に要した時間を比較します．GNU Compiler(OpenMP 利用) により作成したプログラムが要した時間が 875(sec) であるのに対し (リスト 3)，Intel Compiler(OpenMP 利用) により作成したプログラムは 539(sec) で演算を完了しており (リスト 4)，40% 程度の大幅な性能向上を示していることが確認できます[4]．

リスト 3: OpenCV 性能測定 (gcc+OpenMP 8core)

```
$  ~/cvwork/opencv-2.4.9/bin/opencv_perf_core
Time compensation is 37
```

[3] この手順では，利用者フロントエンド上でビルドが実行されます．ジョブ投入による CX400 の演算ノードを用いたバッチ処理も可能です．長時間・大規模なビルド作業を行う場合に有効です．

[4] なお，CX400 には富士通製の C++コンパイラが導入されていますが，C++の言語準拠チェックが通過できず，ビルドが行えませんでした．

```
OpenCV version: 2.4.9
OpenCV VCS version: unknown
Build type: release
Parallel framework: openmp
CPU features: sse sse2 sse3
[==========] Running 1682 tests from 47 test cases.
[----------] Global test environment set-up.
[----------] 12 tests from Size_MatType_sum
[ RUN      ] Size_MatType_sum.sum/0

...

[ VALUE    ]    (127x61, 32SC1)
[       OK ] Size_MatType_abs.abs/14 (1 ms)
[ RUN      ] Size_MatType_abs.abs/15
[ VALUE    ]    (127x61, 32FC1)
[       OK ] Size_MatType_abs.abs/15 (0 ms)
[----------] 16 tests from Size_MatType_abs (2802 ms total)

[----------] Global test environment tear-down
[==========] 1682 tests from 47 test cases ran. (875458 ms total)
[  PASSED  ] 1682 tests.
```

リスト 4: OpenCV 性能測定 (intel compiler+OpenMP 8core)

```
$  ~/cvwork/opencv-2.4.9/bin/opencv_perf_core
Time compensation is 54
OpenCV version: 2.4.9
OpenCV VCS version: unknown
Build type: release
Parallel framework: openmp
CPU features: sse sse2
[==========] Running 1682 tests from 47 test cases.
[----------] Global test environment set-up.
[----------] 12 tests from Size_MatType_sum
[ RUN      ] Size_MatType_sum.sum/0

...

[ VALUE    ]    (127x61, 32SC1)
[       OK ] Size_MatType_abs.abs/14 (0 ms)
[ RUN      ] Size_MatType_abs.abs/15
[ VALUE    ]    (127x61, 32FC1)
[       OK ] Size_MatType_abs.abs/15 (1 ms)
[----------] 16 tests from Size_MatType_abs (3029 ms total)

[----------] Global test environment tear-down
[==========] 1682 tests from 47 test cases ran. (539258 ms total)
[  PASSED  ] 1682 tests.
```

3.3 CUDA による演算の GPGPU 実装

本稿では，入力画像 (RGB) 画像の輝度変換を行う画像処理プログラムを作成します．プログラムの一部は GPGPU によって記述し，並列処理を行います．

GPGPU を用いた並列化を行う場合,「どのような粒度で GPU による並列化を行うべきか」という設計が重要となります．多くの場合,「数式 1 つに相当する程度の小規模なレベル」を抽出し，GPU 上で動作させることが望ましいと言えます．これは，大規模なプログラムが,「GPU 上での実装が難しく並列度が向上しない」ことや,「最悪の場合，演算の独立性がなくなり並列性が失われる」ことが理由となります.「小規模なレベル」を意識して，GPGPU による処理粒度にこの問題を適用すると,「画像中の各画素に対して，GPU によって RGB→ 輝度変換を行う」という処理になります．

CUDA による本アルゴリズムの実装 (CUDA プログラム) について，リスト 5 に示します．本稿では，リスト 5 を"d.cu"という名称のファイルとして扱います．

CUDA プログラムは，C++言語の拡張として実装されており，いくつかの予約語以外の記法は C++言語と同様です．プログラム中の関数 `kernel1` 内に記述のある各変数 `blockIdx`, `blockDim`, `threadIdx` は，CUDA が自動的に算出したデータの保存された位置およびオフセットを示しています．これらの変数を用いてアクセスすべきデータ位置 (`index`) を計算し，画像処理に相当する数式を適用します．このような CUDA プログラムを記述することにより，関数 `kernel1` が GPU の備えるハードウェア性能に応じて並列に実行され，高速な処理が実現できます．

GPGPU に関するより詳細な解説については，本センター広報記事 [7] をご参照ください．

リスト 5: RGB→ 輝度変換 GPGPU コード (d.cu)

```
#include <stdio.h>

__global__ void kernel1(const unsigned char *src,unsigned char *dst)
{
        long index = blockIdx.x * blockDim.x + threadIdx.x;
        dst[index] = (unsigned char)(
                    (double)src[index*3+0]*0.3+
                    (double)src[index*3+1]*0.6+
                    (double)src[index*3+2]*0.1);
}

void dconvi_test(const unsigned char *src,unsigned char *dst,long k,long t)
{
        kernel1<<<k,t>>>(src,dst);
}
```

3.4 MPI による並列プログラム

画像処理プログラム全体を統括するメインプログラムの作成を行います．メインプログラムでは，OpenCV による画像の読み込み，保存に加え，前節で作成した GPGPU プログラムを呼び出し，画像処理を実行します．従って，プログラムの記法 (メモリ管理等) が，OpenCV 由来と CUDA 由来の併用となります．

留意すべき点は，3.1 節で述べた通り，CX400 はクラスターを構成しているため，SPMD による並列化を行うことによって大規模化・性能向上が可能となります．従って，メインプログラムは MPI による並列化を考慮して記述する必要があります．なお，CX400 の各演算ノードには，GPGPU ボードが 1 基搭載されています．メインプログラムは 1 基の GPU を扱うことのみを考慮すれば，GPGPU ボードを扱うことが可能です[5]．

[5]演算ノードに複数の GPGPU ボードが搭載されている場合，複数の GPGPU ボードにデータを振り分けることを考慮する必要が生じるため，プログラムが複雑なものとなります

画像処理のメインプログラムについて，リスト 6 に示します．本稿では，リスト 6 を”main.cc”という名称のファイルとして扱います．

リスト 6: メインプログラム:MPI，OpenCV 併用

```
#include <mpi.h>

#include <stdio.h>
#include <time.h>

#include <cuda.h>
#include <cuda_runtime.h>

#include <fstream>
#include <string>
#include <sstream>

#include <opencv2/opencv.hpp>

void dconvi_test(const unsigned char *src,unsigned char *dst,long k,long t);

int main(int argc,char **argv)
{
  timespec tp;

  int my_rank, my_size;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&my_size);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  printf ("%d␣:␣size␣(numprocs)␣=␣%d,␣my␣rank␣(id)␣=␣%d\n", my_rank, my_size,
      my_rank);

  //all rank read img
  cv::Mat src;
  src = cv::imread("./lena.jpg");

  //on all rank gen subimg
  long sibsize = src.cols * src.rows *src.elemSize() / my_size;
  long  siboff = sibsize * my_rank;

  long sirbsize = src.cols * src.rows / my_size;

  unsigned char *subimg = new unsigned char[sibsize];
  ::memcpy(subimg,&src.data[siboff],sizeof(unsigned char)*sibsize);


  unsigned char *subimgres = new unsigned char[sirbsize];

  //on all rank do cuda
  int devCount;
  cudaGetDeviceCount(&devCount);
  printf("%d␣:␣CUDA␣Device␣Query...\n",my_rank);
  printf("%d␣:␣There␣are␣%d␣CUDA␣devices.\n", my_rank, devCount);

  clock_gettime(CLOCK_MONOTONIC,&tp);
```

```
printf("%d␣:␣tick␣at␣0␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

unsigned char *devmem_subimg;
unsigned char *devmem_subimgres;

cudaMalloc((void**)&devmem_subimg,sibsize);
cudaMalloc((void**)&devmem_subimgres,sirbsize);

cudaMemcpy(devmem_subimg,subimg,sibsize,cudaMemcpyHostToDevice);
//kernel
dconvi_test(devmem_subimg,devmem_subimgres,sirbsize/512,512);

cudaMemcpy(subimgres,devmem_subimgres,sirbsize,cudaMemcpyDeviceToHost);

cudaFree(devmem_subimg);
cudaFree(devmem_subimgres);

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣1␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

//wait for all kernel
MPI_Barrier(MPI_COMM_WORLD);

cv::Mat *result;

//if rank0, gen cv::mat
if(my_rank == 0)
  {
    result = new cv::Mat(src.size(),CV_8UC1);
  }

//if rank0 -> copy subimgres to result (local), else copy to rank0 node by MPI
if(my_rank == 0)
  {
    memcpy(&result->data[0],subimgres,sizeof(unsigned char)*sirbsize);
  }
else
  {
    MPI_Send(subimgres, sirbsize, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD);
  }

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣2␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

//if rank0 -> recv subimggres from other rank
if(my_rank == 0)
  {
    for(int i=1;i<my_size;i++)
      {
        MPI_Status status;
        MPI_Recv(&result->data[sirbsize*i],sirbsize,MPI_UNSIGNED_CHAR,i,0,
            MPI_COMM_WORLD,&status);
      }
  }

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣3␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);
```

```
  if(my_rank == 0)
    {
      cv::imwrite("./dest.jpg",*result);
    }

  clock_gettime(CLOCK_MONOTONIC,&tp);
  printf("%d␣:␣tick␣at␣4␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

  delete result;

  delete [] subimg;
  delete [] subimgres;

  MPI_Finalize();

  return 0;
}
```

3.5 プログラムのビルド・バッチシステムへの投入・実行

作成したプログラムのビルドを行い，CX400 クラスタにジョブを投入することにより，画像処理プログラムが実行されます．

プログラムのビルド CUDA プログラム (d.cu) とメインプログラム (main.cc) のコンパイルおよび，各オブジェクトファイルのリンク・実行ファイルの作成手順をリスト 7 に示します．

リスト 7: プログラムのビルド

```
$ cd ~/code
$ nvcc -ccbin icc -Xcompiler "-O3" -gencode=arch=compute_20,code=sm_20 -gencode=
    arch=compute_30,code=sm_30 -gencode=arch=compute_35,code=sm_35 -gencode=arch
    =compute_35,code=compute_35 -c d.cu
$ mpiicc -I`echo $HOME`/cvwork/opencv-2.4.9/build/include -L`echo $HOME`/cvwork/
    opencv-2.4.9/build/lib -I/usr/local/cuda/include -L/usr/local/cuda/lib64 -c
    main.cc
$ mpiicc -L/usr/local/cuda/lib64 -L/home/usr3/c74013a/cvwork/opencv-2.4.9/build/
    lib main.o d.o -lcuda -lcudart -lopencv_core -lopencv_highgui
```

上記手順では，第一に CUDA プログラムのコンパイルを行います．CUDA プログラムは C++言語の拡張記法であるため，通常の C++コンパイラでは解釈できません．従って，CUDA 専用のコンパイラである `nvcc` を用いてコンパイルを行います．本稿では，Intel Compiler 環境を用いますので，`nvcc` の用いる C++コンパイラを指定するオプションである `-ccbin` に対して，Intel C++ コンパイラの実行コマンドである `icc` を記述しています．

次に，メインプログラムのコンパイルを行います．MPI を用いる場合，その前処理を行う専用のコンパイルコマンドが用意されています．Intel C++コンパイラを用いて MPI プログラムをコンパイルする場合，`mpiicc` コマンドを用います．オプションとして，OpenCV および CUDA 環境のインクルードパス，ライブラリパスを指定しています．

最後に，オブジェクトファイルのリンクを行い，実行ファイルを作成します．メインプログラムと同様に，`mpiicc` コマンドを用いてリンクを行います．正しくビルドが完了した場合，実行ファイル a.out が生成されます．

プログラムの実行　上記の手順で作成したプログラムを MPI 並列で実行するためには，バッチスクリプトを作成と，バッチシステムへの投入が必要となります．

本稿では，Intel コンパイラを用いた MPI プログラムを作成しているため，CX400 で通常用いられる富士通系向けのバッチスクリプトと記法が異なります．また，CUDA および OpenCV を実行時に用いるため，バッチスクリプトに追加の記述が必要となります．16 台の演算ノードを用いる場合のバッチスクリプトおよびバッチシステムへの投入手順について，リスト 8 に示します[6]．ここでは，バッチスクリプトを"run_intelmpi.sh"という名称で保存しているとして扱います．

リスト 8: バッチスクリプト (16 ノード)・ジョブ投入

```
$ cd ~/src
$ cat run_intelmpi.sh
#!/bin/bash
#PJM -L "rscgrp=xxxxxx"
#PJM -L "vnode=16"
#PJM -L "vnode-core=1"
#PJM -P "vn-policy=abs-unpack"
#PJM -X

source /home/etc/intel.sh

NUM_NODES=${PJM_VNODES}
NUM_CORES=1
NUM_PROCS=16

export I_MPI_PERHOST=$NUM_CORES
export I_MPI_FABRICS=shm:ofa

export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=`echo $HOME`/cvwork/opencv-2.4.9/build/lib:/usr/local/
    cuda/lib64:/usr/local/intel2013/lib/intel64:$LD_LIBRARY_PATH

mpdboot -n $NUM_NODES -f ${PJM_O_NODEINF} -r /bin/pjrsh

#debug mode
#mpiexec -check_mpi -n $NUM_PROCS ./a.out

#release mode
mpiexec -n $NUM_PROCS ./a.out
mpdallexit

$ pjsub ./run_intelmpi.sh
```

バッチスクリプト中の留意すべき点について示します．各演算ノードには，1 基の GPGPU ボードが搭載されています．MPI プログラムは，それぞれが 1 基の GPGPU ボードを扱う動作を行うため，**MPI プログラムは各演算ノードで 1 つ起動する**よう制御する必要があります．このための記述が，バッチスクリプト中の `vnode-core=1` に相当します．このパラメータを 1 に設定することにより，各演算ノードで 1 つの MPI プログラムが起動するよう抑制できます．MPI の並列度を制御する記述は，`vnode=16` に相当します．このパラメータ数に対応した MPI プログラムが起動します (この例では，16MPI 並列となります)．

スクリプト中盤に記述されている "export PATH"，"export LD_LIBRARY_PATH"は，MPI プログラムが CUDA および OpenCV のライブラリを参照するために必要となります．

[6]ジョブクラスを指定するパラメータ `rscgrp` については，正式な名称を伏せて記述しています．

スクリプトの後半では，MPI プログラムの実行を行います．富士通向けの手順と異なり，Intel コンパイラを用いた MPI プログラムでは，`mpdboot` コマンドを事前に実行し，プログラムの実行後に `mpdallexit` コマンドを実行する必要があります．記述がない場合，MPI プログラムが動作しないため，留意が必要です．

このようなバッチスクリプトを作成し，`pjsub` コマンドを用いてバッチシステムにジョブを投入することにより，CX400 クラスタ上で作成したプログラムが動作します．

3.6 演算性能の比較

これまで解説を行った「CX400 クラスタ上での GPGPU，Intel MPI 並列，OpenCV の併用」について，演算性能の評価を行います．評価は，他の演算手法および計算機環境で，同様の画像処理プログラムを動作させ，その演算時間を計測することによって行います．演算を実行する 4 つの環境について，以下の表 3 に示します．

表 3: 演算環境

名称	計算機	コンパイラ	演算 HW	クロック 周波数	演算 HW 主記憶容量	演算性能 (FLOPS, 倍精度)
CX400 (CPU)	九大 CX400	Intel C++	Xeon E5-2680	2.7GHz	128GB	172.8G(1CPU)
CX400 (K20m)	九大 CX400	Intel C++	Tesla K20m (2496 core)	706MHz	5GB	1.17T
PC(CPU)	筆者所有 PC	MSVC++	Core i7 -4770K	3.5GHz	32GB	70〜100G(1CPU)
PC (GTX780)	筆者所有 PC	MSVC++	GeForce GTX780 (2394 core)	863MHz	3GB	1.3T

入力データには，解像度の異なる 4 種類の画像データ (5120×5120, 10240×10240, 20480×20480, 40960×40960, いずれも OpenCV 付属の lena.jpg の拡大処理により生成) を用います．入力画像の画素数は，2,600 万画素〜16 億画素と，PC のモニタ解像度やデジタルカメラの撮影解像度と比較して膨大なものを用いています．このような規模になると，一般の PC では主記憶容量が不足し，演算ができない状況が考えられます．

CPU 向けの画像処理プログラムは，以下に示すリスト 9 を適用しました．リスト 6 中の GPGPU に依存する部分について，OpenCV による処理を行うよう書き換えたプログラムとなります．

リスト 9: メインプログラム:MPI，OpenCV，GPGPU 併用 (main.cc)

```
#include <mpi.h>

#include <stdio.h>
#include <time.h>

#include <fstream>
#include <string>
#include <sstream>

#include <opencv2/opencv.hpp>

int main(int argc,char **argv)
{
  timespec tp;
```

```
int my_rank, my_size;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&my_size);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

printf ("%d␣:␣size␣(numprocs)␣=␣%d,␣my␣rank␣(id)␣=␣%d\n", my_rank, my_size,
    my_rank);

//all rank read img
cv::Mat src;
src = cv::imread("./lena.jpg");

//on all rank gen subimg
long sibsize = src.cols * src.rows *src.elemSize() / my_size;
long  siboff = sibsize * my_rank;

long sirbsize = src.cols * src.rows / my_size;

unsigned char *subimg = new unsigned char[sibsize];
::memcpy(subimg,&src.data[siboff],sizeof(unsigned char)*sibsize);

unsigned char *subimgres = new unsigned char[sirbsize];

//time cnt start
clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣0␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

cv::Mat devmem_subimg(src.rows/my_size,src.cols,CV_8UC3);
cv::Mat devmem_subimgres(src.rows/my_size,src.cols,CV_8UC1);

::memcpy(&devmem_subimg.data[0],subimg,sizeof(unsigned char)*sibsize);

//kernel
cv::cvtColor(devmem_subimg,devmem_subimgres,CV_RGB2GRAY);

::memcpy(subimgres,&devmem_subimgres.data[0],sizeof(unsigned char)*sirbsize);

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣1␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

//wait for all node
MPI_Barrier(MPI_COMM_WORLD);

cv::Mat *result;

//if rank0, gen cv::mat
if(my_rank == 0)
  {
    result = new cv::Mat(src.size(),CV_8UC1);
  }

//if rank0 -> copy subimgres to result (local), else copy to rank0 node by MPI
if(my_rank == 0)
  {
    memcpy(&result->data[0],subimgres,sizeof(unsigned char)*sirbsize);
  }
```

```
else
  {
    MPI_Send(subimgres, sirbsize, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD);
  }

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣2␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

//if rank0 -> recv subimggres from other rank
if(my_rank == 0)
  {
    for(int i=1;i<my_size;i++)
      {
        MPI_Status status;
        MPI_Recv(&result->data[sirbsize*i],sirbsize,MPI_UNSIGNED_CHAR,i,0,
            MPI_COMM_WORLD,&status);
      }
  }

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣3␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

if(my_rank == 0)
  {
    cv::imwrite("./dest.jpg",*result);
  }

clock_gettime(CLOCK_MONOTONIC,&tp);
printf("%d␣:␣tick␣at␣4␣%lf\n",my_rank,tp.tv_sec+tp.tv_nsec*0.001*0.001*0.001);

delete result;

delete [] subimg;
delete [] subimgres;

MPI_Finalize();
```

考察　各演算環境における処理時間について，表 4 および図 2 に示します．

今回適用した画像処理アルゴリズムは，RGB 画像の輝度画像への変換であり，非常に単純な処理といえます．CPU による処理と GPU による処理の性能差が小さく，かつ GPU による演算は CPU→GPU 間のデータ転送が生じるため，CPU による処理が高速である状況も含まれました．

ただし,「**入力データが大規模になるほど，GPU による処理が高速となる**」,「**演算ノード数の増加に対応して，演算性能が向上する**」ことが確認できます．また，解像度 40960 × 40960 のデータを入力した場合，1 演算ノードでは演算の実行が不能となります．これは，入力データの規模が，1 プロセスに割り当て可能なメモリサイズや GPU に転送できるメモリサイズを超過することが原因であり，MPI による複数ノード処理が必要となります．

4　まとめ

本稿では，CX400 クラスタに新たに導入された，Intel コンパイラおよび GPGPU の活用例について解説を行いました．具体的には,「画像処理の高速化」をターゲットとして，そのプログラム作成について

表 4: 演算性能 (RGB→ 輝度変換：単位 sec)

環境	画像幅	画像高	画素数	1node	2nodes	4nodes	8nodes	16nodes
CX400(CPU)	5,120	5,120	26,214,400	0.09337	**0.04615**	**0.02252**	**0.01090**	**0.005795**
CX400(K20m)	5,120	5,120	26,214,400	0.2505	0.2282	0.217	0.2076	0.2065
PC(CPU)	5,120	5,120	26,214,400	**0.046**	N/A	N/A	N/A	N/A
PC(GTX780)	5,120	5,120	26,214,400	0.157	N/A	N/A	N/A	N/A
CX400(CPU)	10,240	10,240	104,857,600	0.3834	**0.1903**	**0.09451**	**0.04683**	**0.02350**
CX400(K20m)	10,240	10,240	104,857,600	0.3506	0.2967	0.2495	0.2252	0.2134
PC(CPU)	10,240	10,240	104,857,600	**0.172**	N/A	N/A	N/A	N/A
PC(GTX780)	10,240	10,240	104,857,600	0.2660	N/A	N/A	N/A	N/A
CX400(CPU)	20,480	20,480	419,430,400	1.525	0.7588	0.3820	**0.1900**	**0.09401**
CX400(K20m)	20,480	20,480	419,430,400	0.8197	**0.5635**	**0.3818**	0.2958	0.2443
PC(CPU)	20,480	20,480	419,430,400	1.280	N/A	N/A	N/A	N/A
PC(GTX780)	20,480	20,480	419,430,400	**0.7030**	N/A	N/A	N/A	N/A
CX400(CPU)	40,960	40,960	1,677,721,600	実行不能	3.010	1.523	0.7583	0.3815
CX400(K20m)	40,960	40,960	1,677,721,600	実行不能	**1.377**	**0.7828**	**0.5112**	**0.3448**
PC(CPU)	40,960	40,960	1,677,721,600	実行不能	N/A	N/A	N/A	N/A
PC(GTX780)	40,960	40,960	1,677,721,600	実行不能	N/A	N/A	N/A	N/A

検討を行いました.「OpenCV を Intel コンパイラを用いてセルフビルド・プログラム中で利用」,「GPGPU による画像処理プログラムの記述」,「CUDA の Intel MPI 並列下での利用」が特徴となります.

本稿に掲載した設定ファイルやプログラムを参照いただければ，本環境をテストすることが可能です. 本稿が，CX400 クラスタでは「MPI を用いることにより複数の GPGPU ボードを演算に適用できること」,「セルフビルドにより様々なライブラリが適用できること」の理解に繋がれば幸いです.

また，九大研究用計算機システムの利用支援全般についてのお問い合わせ・ご相談等がありましたら，メールアドレス res-system @ isc.kyutech.ac.jp までお問い合わせください.

参考文献

[1] 九州大学研究用計算機システム 高性能演算サーバ利用法，http://www2.cc.kyushu-u.ac.jp/scp/system/general/CX/how_to_use/

[2] Intel Developer Zone：Intel Compilers, https://software.intel.com/en-us/intel-compilers

[3] NVIDIA CUDA Zone，https://developer.nvidia.com/cuda-zone

[4] OPENCV：Open Source Computer Vision，http://opencv.org/

[5] NVIDIA Tesla Server Solutions，http://www.nvidia.com/object/tesla-servers.html

[6] Message Passing Interface Forum，http://www.mpi-forum.org/

[7] 林 豊洋，PC 向けグラフィックカードを活用した画像処理の高速化，情報科学センター広報 第 22 号，http://www.isc.kyutech.ac.jp/kouhou/kouho22/pdf/koho22-kaisetu3.pdf，2010

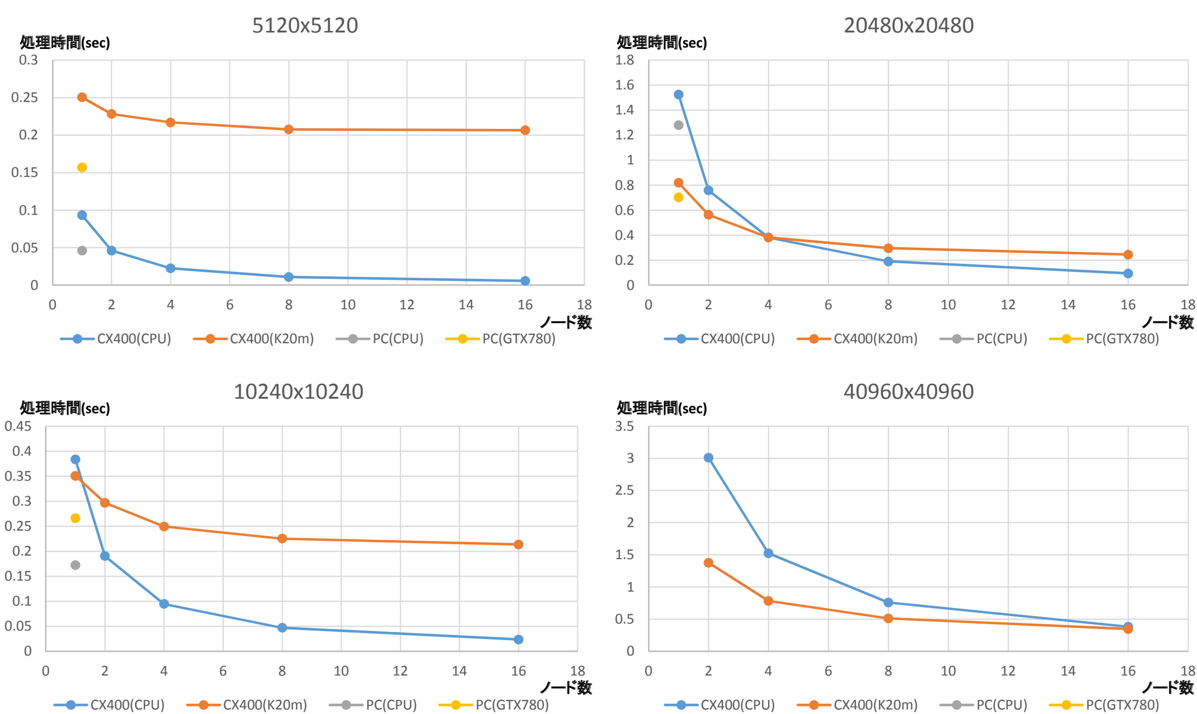

図 2: 演算性能 (RGB→ 輝度変換)