MRI Image Processing with OpenCL

Juan Luis Soler Ferrer　Kiminori Matsuzaki

Functional magnetic resonance imaging (fMRI) is now widely used in brain science, and it takes brain images in four dimensions, usual three dimension images over time. The goal of this project is to obtain higher spatio-resolution images from the original ones, making use of super-resolution techniques in two dimensional, three dimensional and four dimensional scopes. In this study, we employ the open standard for parallel programming of heterogeneous systems OpenCL. With this approach of accelerated computing the code can be executed on diverse CPUs and GPUs, this is usually a great scenario for image processing, benefiting of the multiple cores and the balancing of the workload to get a higher performance of the hardware. We report the experiment results comparing the sequential implementation with the parallel implementation on a PC with two GPU devices, as well as lessons obtained through this project.

近年，脳科学において機能的磁気共鳴画像化法（fMRI）が広く利用されている．fMRI で撮像される脳画像は 4 次元のデータである，すなわち，通常の 3 次元画像の時系列である．本プロジェクトの目標は，超解像手法を 2 次元，3 次元および 4 次元データに適用することで，通常の fMRI 画像から高空間解像度の画像を得ることである．本研究では，ヘテロジニアス環境における並列プログラミングのための標準のひとつである OpenCL を適用する．このアクセラレータを用いた計算手法により，プログラムコードをさまざまな CPU や GPU 上で動作させることができる．これは，画像処理に適しており，ハードウェアの複数のコアを利用することで高いパフォーマンスを得ることが期待できる．2 つの GPU デバイスを持つ計算機上で逐次プログラムと並列プログラムを用いた実験の結果ならびに，本プロジェクトにおいて得た知見について報告する．

1 Introduction

Functional magnetic resonance imaging (fMRI) is now widely used in brain science. In fMRI, we often use a imaging method that takes an image of the (whole) brain in short time, 2–3 seconds. However, compared to the high temporal-resolution, the spatio-resolution of fMRI images are low: typical fMRI images have voxels of 3mm each. We are studying to cope with this rather low spatio-resolution of fMRI images [9, 10], especially based on the so-called super-resolution technique [12].

In this paper, we improved the performance of the super-resolution program by using the general-purpose GPU computing (GPGPU) technique. Especially, we developed a program for heterogeneous GPU environment with OpenCL [6]. We will report how we can develop a program for super-resolving fMRI images with OpenCL and the experiment results.

The rest of the paper is organized as follows. In Section 2, we introduce some important technologies used in this study. In Section 3, we show how we developed a parallel program with OpenCL. In Section 4, we report the experiment results conducted on a laptop PC with two GPU devices. In Section 5, we discuss how we learned in this project and our future directions.

2 Preliminaries

2.1 Functional Magnetic Resonance Imaging

Recently functional magnetic resonance imaging (fMRI) is widely used in brain science. Using

* OpenCL を用いた MRI 画像処理

Juan Luis Soler Ferrer, 高知工科大学, Universitat Politècnica de València, Kochi University of Technology, Universitat Politècnica de València.

Kiminori Matsuzaki, 高知工科大学, Kochi University of Technology.

fMRI for brain science has two important advantages: fMRI is non-invasive and we can take images of interior portion. In clinical use of MRI, we often take structural images with high spatio-resolution of 0.5–1 mm, which take several minutes for the whole brain image. In fMRI, we take functional images with the time-resolution of 2–3 seconds but the image has low spatio-resolution of 3 mm. Since the difference of signals is so small (often less than several percents) that we need statistical post-processing to get an activation map and judge the brain activities.

Images taken by fMRI have rather low space-resolution. Increasing the space-resolution is helpful for the growth of brain science. One approach is to apply post-processing to obtain images with higher space-resolution, which is the super-resolution technique [12]. There have been a lot of studies for the super-resolution technique applied to the structural images (MRI) [4, 11]. However, there have been only a few studies for the functional images (fMRI) [1, 5, 8].

2.2 Super-resolution Images

It is known as super-resolution (SR) [12] the set of techniques and algorithms designed to increase the spatial resolution of an image, typically from a sequence of lower resolution images. They differ from traditional techniques of image scaling that the latter ones only use an image for the resolution increase, focusing its objective to maintain sharp edges, without the appearance of new details. In contrast, in super-resolution the case is to merge information from several images taken from the same scene, to represent details that initially are not significant in the original images.

In this project we use the reconstruction approach that it takes a set of low-resolution images with sub-pixel shift and estimates the high-resolution image so that the error between the input images and simulated images is minimized utilizing the MAP algorithm.

Maximum a posteriori (MAP) Estimation [2] is a widely-used algorithm for reconstruction-based super-resolution. In this MAP estimation, we reconstruct the high-resolution image by maximizing the posterior probabilities based on the Bayesian estimation.

Here, we denote the high-resolution image x, the number of observed images n, the observed images y_i, and the transformation matrices $A_i = D_i B_i M_i$. A typical model in MAP estimation assumes that the noises follow the normal distribution and that the image is somehow smooth. Under this assumption, it minimizes the following objective function.

$$E(x) = \left[\sum_{i=1}^{n} ||y_i - A_i x||\right] + \alpha ||Hx||$$

In this equation, α is a parameter that control the smoothness of the image, H is a high-pass filter. The first term decreases if the estimated image is close to observed images. The second term decreases if the estimated image is smooth.

2.3 OpenCL

Open Computing Language (OpenCL) [6] is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors.

OpenCL applications consist of two parts. The OpenCL host program is a pure software routine written in standard C/C++ that runs on any sort of microprocessor. At a certain point during the execution of this host software routine, there is likely to be a function that is computationally expensive and can benefit from the highly parallel acceleration on a more parallel device: a CPU, GPU, FPGA, etc. This function to be accelerated is referred to as an OpenCL kernel. These kernels are written in standard C; however, they are annotated with constructs to specify parallelism and memory hierarchy. A Kernel is executed in the compute devices that typically consists of several compute units, which in turn comprise multiple processing elements (PEs). A single kernel execution can run on all or many of the PEs in parallel.

SR needs to apply an algorithm per pixel in multiple iterations, this scenario is perfect for parallelizing using the OpenCL technology.

2.4 OpenGL and SFML

Open Graphics Library (OpenGL) [7] is a cross-language, multi-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. For this project it is used because two reasons. The first one because can work directly with OpenCL, for example they

can share texture or image buffers without using any intermediate command. And the second one, for better user interface, being capable of checking the final results with an image, or an animation or a 3d model voxel rendering improves the debugging process.

Simple and Fast Multimedia Library (SFML) [3] is a cross-platform software development library designed to provide a simple interface to various multimedia components in computers. Also handles window creation and input as well as the creation and management of OpenGL contexts. In this project it is used as auxiliary library for dealing with particularities of the different OS like the time measurement or the OpenGL handling that can differ totally depending in the system that is running.

3 Implementation

We took the fMRI images under the following conditions. The subject was a man in his thirties. The task consisted of 15 sets of 30 seconds resting state and 30 seconds right-hand tapping motion, in turn. The whole brain was scanned every three seconds, and each scan took 36 slice images (64×64 resolution, 12-bit depth).

In total we have 310 3D images of $64 \times 64 \times 36$, making it 11160 slices that will be processed using the techniques previously explained for the purpose of improving the resolution from 64×64 to 128×128.

We have the algorithm implemented two times, one in C++ in sequential code and the other in C99 for OpenCL. Both implementations are practically identical with a few exceptions in their respective idiosyncrasies. For the sequential code implementation it is straight forward, and does not need any kind of preparation.

For the OpenCL implementation needs some preparation prior to execution. Before creating the context for OpenCL it is necessary to know in what platform (usually vendor dependent) to create the context. First, it is mandatory to check the OpenCL compatible platforms in the host computer. When it is decided the platform to use, then we can create the working context. A platform consists of one or multiple devices, where our code will be executed, such as CPU, GPU, FPGA, DSP, etc. We compile the program that is going to be executed in the device. It is important to stress that this part is done on runtime.

Fig. 1 is the function that we call in the host program for the SR algorithm. The input is a vector of 4 low-resolution images and the output is the image processed.

Line 1: The class i2D is a container of 2d image in the format used by fMRI.

Lines 2–4: Create 3 buffers, 2 read-only for input data and 1 write-only for output data.

Lines 6–10: Create the kernel that manage the calls to the function "superResolution" (Fig. 2) in the devices, also put the buffers previously created as arguments for the kernel.

Lines 12–13: Create the command queue that it will our medium to sending commands to the device. All the commands processed by the device they may past through the command queue.

Lines 15–18: Create two new images that will be used for intermediate processing by the SR algorithm, and other auxiliary variables for the kernel execution.

Lines 20–23: We write the vector with the 4 low-resolution images to one of the input buffers, and optimize using only one concatenating the 4 images.

Line 25: Start the iteration loop.

Lines 26–27: Write the image in the buffer that we have worked on.

Lines 28–29: Execute the kernel program.

Lines 30–31: Copy the output buffer directly to the input buffer for the iteration process.

Lines 34–35: Read the last output buffer as a result image.

Lines 37–41: Release OpenCL resources from the device.

Line 43: Return the processed image and finish the function.

4 Results

The results obtained were gathered from a MSI GE60 2PE Apache Pro Laptop Computer, this computer has an Intel Core i7 (8 cores) and two graphic cards, a Low consumption Intel graphic card and a high performance NVidia GTX860M graphic card. That implies that we have 3 OpenCL compatible devices in one computer for testing.

Table 1 shows the experiment results where we measured the execution time and calculated the rel-

Table 1 Experiment results

Device	Time (s)	Speedup
Intel CPU (Sequential)	535.5	—
Intel GPU (OpenCL)	223.9	2.39
NVidia GPU (OpenCL)	108.5	4.93

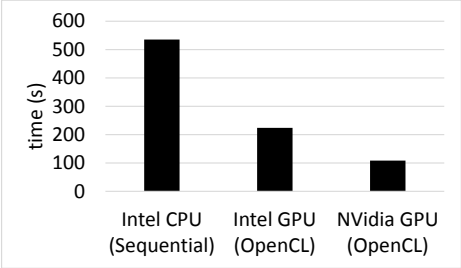

Fig. 3 Process time

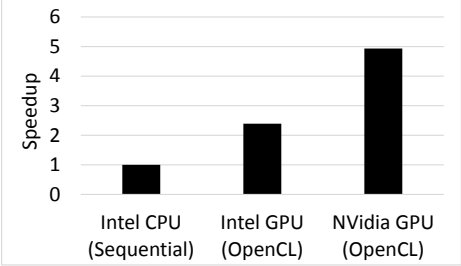

Fig. 4 Speedup

ative speedup with respect to the sequential code executed on CPU. Figures 3 and 4 plot the execution time and speedup. From this results, we can see that the OpenCL version runs 2–5 times faster with a GPU.

5 Discussion

The results of the project are very satisfactory, the utilization of the OpenCL technology in this type of case is a great scenario of parallel computing. Having to process each pixel/voxel individually without dependencies with each other makes it a very optimizable algorithm in this kind of context.

A negative part is that, although OpenCL is a good technology, in the end it depends on the vendors or manufacturers of the devices and their implementations. The closed-source nature of the drivers makes it hard to work with it in Linux, while in the Windows counterpart there is a newer version of the drivers. Also some vendors like NVidia put OpenCL implementation under the CUDA technology, a similar but proprietary technology.

This project can be expanded in directions: for example we may get better performance using an Altera or Xilinx FPGA; we should analyze the results expanding with more criteria like power consumption and power efficiency; and conclude which technology is more appropriate for end-user usage.

References

[1] Ben-Eliezer, N., Goerke, U., Ugurbil, K., and Frydman, L.: Functional MRI using super-resolved spatiotemporal encoding, *Magnetic Resonance Imaging*, Vol. 30, No. 10(2012), pp. 1401–1408.

[2] Bishop, C.: *Pattern Recognition and Machine Learning*, Springer, 2006.

[3] Gomila, L.: Simple and Fast Multimedia Library (SFML), `http://www.sfml-dev.org/`.

[4] Greenspan, H.: Super-resolution in medical imaging, *The Computer Journal*, Vol. 52, No. 1(2009), pp. 43–63.

[5] Joshi, S., Marquina, A., Osher, S., Dinov, I., Darrell, J., Horn, V., and Toga, A.: Image resolution enhancement and its applications to medical image processing, Technical Report TechReport CAM08-62, Department of Mathematics, UCLA, 2008.

[6] Khronos group: OpenCL, `https://www.khronos.org/opencl/`.

[7] Khronos group: OpenGL, `https://www.khronos.org/opengl/`.

[8] Kornprobst, P., Peeters, R., Nikolova, M., Deriche, R., Ng, M., and Hecke, P. V.: A superresolution framework for fMRI sequences and its impact on resulting activation maps, *Medical Image Computing and Computer-Assisted Intervention (MICCAI03)*, 2003, pp. 117–125.

[9] Matsuzaki, K. and Miyazaki, R.: Evaluation of Super-Resolution for fMRI Images, *Proceedings of 5th International Symposium of Frontier Technology (ISFT 2015)*, 2015.

[10] Matsuzaki, K. and Miyazaki, R.: Super-Resolution for fMRI Images and its Evaluation, *Kochi University of Technology Research Bulletin*, Vol. 12, No. 1(2015), pp. 131–138.

[11] Reeth, E., Tham, I., Tan, C., and Poh, C.: Super-resolution in magnetic resonance imaging: A review, *Concepts in Magnetic Resonance*, Vol. 40A, No. 6(2012), pp. 306–325.

[12] Simpkins, J. and Stevenson, R.: *An introduction to super-resolution imaging, Mathematical Optics: Classical, Quantum, and Computational Methods*, CRC Press, 2012.

```
1  i2D superResolutionOpenCL(vector<i2D> imas) {
2    cl_mem _in_imas = CL_CHECK_ERR(clCreateBuffer(OCL.context, CL_MEM_READ_ONLY, sizeof(usint)
         *64*64*4, NULL, &_err));
3    cl_mem _in_res = CL_CHECK_ERR(clCreateBuffer(OCL.context, CL_MEM_READ_ONLY, sizeof(usint)
         *128*128, NULL, &_err));
4    cl_mem _out_next = CL_CHECK_ERR(clCreateBuffer(OCL.context, CL_MEM_WRITE_ONLY, sizeof(usint)
         *128*128, NULL, &_err));
5
6    cl_kernel kernel;
7    kernel = CL_CHECK_ERR(clCreateKernel(OCL.program, "superResolution", &_err));
8    CL_CHECK(clSetKernelArg(kernel, 0, sizeof(_in_imas), &_in_imas));
9    CL_CHECK(clSetKernelArg(kernel, 1, sizeof(_in_res), &_in_res));
10   CL_CHECK(clSetKernelArg(kernel, 2, sizeof(_out_next), &_out_next));
11
12   cl_command_queue queue;
13   queue = CL_CHECK_ERR(clCreateCommandQueue(OCL.context, OCL.devices[0], 0, &_err));
14
15   i2D res;res.setSize(128,128,2047);
16   i2D next;next.setSize(128,128);
17   cl_event kernel_completion;
18   size_t global_work_size[1] = { 128*128 };
19
20   CL_CHECK(clEnqueueWriteBuffer(queue, _in_imas, CL_TRUE, 0*sizeof(usint)*64*64, sizeof(usint)
         *64*64, &imas[0].datos[0], 0, NULL, NULL));
21   CL_CHECK(clEnqueueWriteBuffer(queue, _in_imas, CL_TRUE, 1*sizeof(usint)*64*64, sizeof(usint)
         *64*64, &imas[1].datos[0], 0, NULL, NULL));
22   CL_CHECK(clEnqueueWriteBuffer(queue, _in_imas, CL_TRUE, 2*sizeof(usint)*64*64, sizeof(usint)
         *64*64, &imas[2].datos[0], 0, NULL, NULL));
23   CL_CHECK(clEnqueueWriteBuffer(queue, _in_imas, CL_TRUE, 3*sizeof(usint)*64*64, sizeof(usint)
         *64*64, &imas[3].datos[0], 0, NULL, NULL));
24
25   for (int i=0;i<50;i++) {
26     CL_CHECK(clEnqueueWriteBuffer(queue, _in_res, CL_TRUE, 0, sizeof(usint)*128*128, &res.datos
           [0], 0, NULL, &kernel_completion));
27     CL_CHECK(clWaitForEvents(1, &kernel_completion));CL_CHECK(clReleaseEvent(kernel_completion));
28     CL_CHECK(clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, &
           kernel_completion));
29     CL_CHECK(clWaitForEvents(1, &kernel_completion));CL_CHECK(clReleaseEvent(kernel_completion));
30     CL_CHECK(clEnqueueCopyBuffer(queue,_out_next,_in_res,0,0,sizeof(usint)*128*128,NULL,NULL,&
           kernel_completion));
31     CL_CHECK(clWaitForEvents(1, &kernel_completion));CL_CHECK(clReleaseEvent(kernel_completion));
32   }
33
34   CL_CHECK(clEnqueueReadBuffer(queue, _out_next, CL_TRUE, 0, sizeof(usint)*128*128, &next.datos
         [0], 0, NULL, &kernel_completion));
35   CL_CHECK(clWaitForEvents(1, &kernel_completion));CL_CHECK(clReleaseEvent(kernel_completion));
36
37   clReleaseCommandQueue(queue);
38   clReleaseKernel(kernel);
39   clReleaseMemObject(_in_imas);
40   clReleaseMemObject(_in_res);
41   clReleaseMemObject(_out_next);
42
43   return next;
44 }
```

Fig. 1 The C++ function that manages the OpenCL algorithm implementation

```
__kernel void superResolution(__global unsigned short int *_in_imas, __global unsigned short int
    *_in_res, __global unsigned short int *_out_next)
{
  int i = get_global_id(0);
  int x=i%128;
  int y=i/128;
  int alpha = 4;

  int dd[4][2] = {{0,0}, {0,1}, {1,0}, {1,1}};

  int dEdp=0;

  for (int l=0;l<4;l++) {
    int dx = dd[l][0];
    int dy = dd[l][1];
    int newx = (x + dx) / 2; if (newx < 0) continue; if (newx >= 64) continue;
    int newy = (y + dy) / 2; if (newy < 0) continue; if (newy >= 64) continue;
    int pl=0;

    for ( int lx = newx * 2 - dx; lx < (newx + 1) * 2 - dx; lx++) {
      for (int ly = newy * 2 - dy; ly < (newy + 1) * 2 - dy; ly++) {
        if (lx < 0 || lx >= 128 || ly < 0 || ly >= 128) {
          //pl += 0;
        } else {
          pl += _in_res[lx+(ly*128)];
        }
      }
    }
    pl /= 4;
    int ql = _in_imas[newx+(newy*64)+(l*64*64)];
    dEdp += pl - ql;

    if (x > 0 && y > 0 && x < 128 - 1 && y < 128 - 1) {
      int pn = _in_res[(x-1)+(y*128)] + _in_res[(x+1)+(y*128)] + _in_res[(x)+(y*128)-128] +
          _in_res[(x)+(y*128)+128];
      dEdp += alpha * (4 * _in_res[x+y*128] - pn) / 4;
    }
    int pix;
    if (dEdp > 0) {
      pix = _in_res[x+y*128] - (dEdp + 2) / 4;
    } else {
      pix = _in_res[x+y*128] - (dEdp - 2) / 4;
    }
    _out_next[x+y*128]=clamp(pix,0,2047);
  }
  if (x == 0 || y == 0 || x == 127 || y == 127) {
    _out_next[x+y*128]=0;
  }
}
```

Fig. 2 The OpenCL algorithm implementation