Heterogeneous Platform Programming for High Performance Medical Imaging Processing

Renan Sales Barros[1], Sytse van Geldermalsen[2], Anna M.M. Boers[1,3,4], Adam S.Z. Belloum[2], Henk A. Marquering[1,3], and Silvia D. Olabarriaga[1]

[1] Biomedical Engineering & Physics, Academic Medical Center, University of Amsterdam, Meibergdreef 9, 1105 AZ, Amsterdam, The Netherlands
[2] Department of Computational Science, University of Amsterdam, Science Park 107, 1098 XG, Amsterdam, The Netherlands
[3] Department of Radiology, Academic Medical Center, University of Amsterdam, Meibergdreef 9, 1105 AZ, Amsterdam, The Netherlands
[4] University of Twente, Postbus 217, 7500 AE, Enschede, The Netherlands

Abstract. Medical imaging processing algorithms can be computationally very demanding. Currently, computers with multiple computing devices, such as multi-core CPUs, GPUs, and FPGAs, have emerged as powerful processing environments. These so called heterogeneous platforms have potential to significantly accelerate medical imaging applications. In this study, we evaluate the potential of heterogeneous platforms to improve the processing speed of medical imaging applications by using a new framework named FlowCL. This framework facilitates the development of parallel applications for heterogeneous platforms. We compared an implementation of region growing based method to automated cerebral infarct volume measurement with a new implementation targeted for heterogeneous platforms. The results of this new implementation agree well with the original implementation and they are obtained with significant speed-up comparing to the sequential implementation.

Keywords: dataflow, framework, heterogeneous computing, heterogeneous platforms, medical imaging processing, OpenCL, parallel programming.

1 Introduction

In medical imaging applications large amounts of data must be processed quickly and accurately, which requires the usage of high performance computing systems. Commodity computer architectures are rapidly developing into systems with multi-core CPUs and with additional accelerated hardware devices such as graphics processing units (GPUs) and field programmable gate arrays (FPGAs). These heterogeneous platforms provide a new alternative to design and implement computationally demanding applications. Consequently, the computation power provided by these heterogeneous platforms should be explored for medical image processing.

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 301–310, 2014.
© Springer-Verlag Berlin Heidelberg 2014

Expertise of new programming constructs and concepts is however required for application developers to effectively utilize these platforms. The OpenCL [5] technology was developed with the aim of facilitating heterogeneous platforms usage. OpenCL includes a language for writing functions, called *kernels*, that execute on diverse computing devices. It also includes an application programming interface (API) that is used to control the heterogeneous platforms. A benefit of OpenCL is that the kernels that are coded according to this standard can run on different devices without any modification. This makes it possible to take advantage of computationally powerful devices that are well suited for different tasks. Nevertheless, OpenCL still requires application developers to deal with low level concerns such as the overhead of the code, memory management, and synchronization. In order to evaluate the potential of heterogeneous platforms in medical imaging processing, we needed an easier programming platform. A new framework named FlowCL was developed to provide an intuitive way to create applications utilizing heterogeneous platforms. This framework eliminates the OpenCL API usage, but maintains the OpenCL programming language for writing kernels. A brief description of this framework is presented in Section 2.

We used the FlowCL framework to implement a modified version of a previously developed method of automated measurement of cerebral infarct volume of patients after acute ischemic stroke. This method, which was developed and validated by Boers et al. [3], was modified for heterogeneous platforms. In Section 3, we explain the automated cerebral infarct volume measurement method and the modification implemented in this study. We compared the execution times of the original (sequential) implementation with the new parallel implementation for heterogeneous platforms using FlowCL. We also evaluated the differences between the results of both implementations in Section 4. Finally, the conclusions regarding this work and future improvements are presented in Section 5.

2 FlowCL Framework

During the development of OpenCL applications, programmers have to deal with a low level C library, which requires specialized expertise for effective and efficient code development. There are arious frameworks to mitigate this problem. For example, the Many GPUs Package (MGP) [2] was built on top of the Virtual OpenCL Layer (VCL). VCL is a transparent layer that accesses and manages OpenCL devices in clusters and presents these devices as a single node . MGP is a layer that facilitates the programing using clusters by hiding low level functions. A library named Maestro [10] also tries to reduce the complexity of OpenCL applications development by providing functions for automatic data transfer and task decomposition across OpenCL devices. However, MGP and Maestro do not use extensive optimizations strategies. A more complete framework i is StarPU [1], which is a runtime system capable of scheduling tasks over heterogeneous devices using several optimizations strategies. However, the framework API uses the C programming language and this hampers the usage of high level programming concepts.

FlowCL is a new high level framework that supports rapid prototyping and development with OpenCL, which makes it possible to closely control the execution across all OpenCL devices on one computer system. More details about FlowCL are found in [4]. It hides all low level calls to the OpenCL library API from the application developer. Only the OpenCL kernel code that is designed to run on a selected device must be provided to the framework. Also, FlowCL provides an object-oriented declarative API to easily build applications with the concept of dataflow. The programmer simply declares a set of memory objects and operations. Each operation runs each single kernel function on any available device. This framework automatically applies optimization strategies such as overlapping communication and computation, and asynchronous data transfers and kernel executions.

To use the FlowCL framework, application developers just have to deal with four classes of objects: memory, context, device, and operation. Figure 1 illustrates the relationships between these classes. By having only four classes with limited relationships, FlowCL provides a simpler approach that is easier to understand than OpenCL.

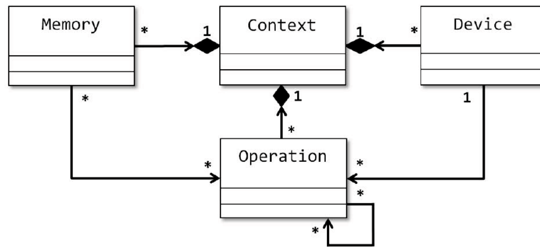

Fig. 1. FlowCL cardinality diagram. A **Context** is instantiated with kernel codes; these codes are usable on all available computing devices. **Memory** objects act as arguments for operations; they are created by a context with a given size and become available to all devices in this context. **Device** represents a computing device. **Operation** runs a specified kernel function on a selected device.

In short, FlowCL framework addresses the following key aspects: it facilitates application development with OpenCL; it provides an object oriented API to build applications with the dataflow concept; it eliminates the OpenCL API, except for kernel code; it automatically manages all devices on heterogeneous platforms; it supports concurrent kernel execution and asynchronous data transfers; and it supports multiple operating systems.

The framework is designed to run in the C++ language and only requires the FlowCL header file inclusion. To illustrate FlowCL usage, the following source code is shown:

```
#include "FlowCL.hpp"
using namespace FlowCL;

int main()
{
    Context con;
    con.CompileFile("source.cl");

    Memory memcpu = con.CreateMemory(1e8*sizeof(int));
    Memory memgpu = con.CreateMemory(1e8*sizeof(int));

    Operation genrand = con.CreateOperation(con.GetCPUDevice(), "GenRand");
    genrand.SetArg(0, memcpu); // CPU already has access to memory
    genrand.SetArg(1, memgpu);
    genrand.SetWorkSize(1e8); // Set finest granularity

    Operation sortcpu = con.CreateOperation(con.GetCPUDevice(), "SortCPU");
    sortcpu.SetArgDependency(genrand, 0, memcpu); // Wait for genrand
    sortcpu.SetWorkSize(1e8);

    Operation sortgpu = con.CreateOperation(con.GetGPUDevice(), "SortGPU");
    sortgpu.SetArgDependency(genrand, 0, memgpu); // Wait for genrand
    sortgpu.SetArgOutput(0, memgpu);
    sortgpu.SetWorkSize(1e8);

    con.Run(); // Blocking run
}
```

This example is visualized in Figure 2. The first operation executes on the CPU and generates random numbers that are sorted in the next two operations that execute in parallel on the GPU and the CPU. There is no data transferred to the sort operation that runs on the CPU because this data is readily available. Once the GPU operation is completed, the data is transferred back and the execution is finished.

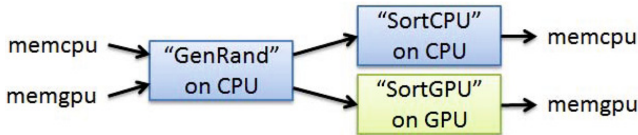

Fig. 2. Sample code visual representation

3 Case Study

In this section we present a case study using heterogeneous computing for measurement of cerebral infarct volume (CIV) of patients with acute ischemic stroke. The CIV has been suggested as an important measure for the effective treatment of these patients [9]. This volume can be manually measured in early follow-up non-contrast CT scans by the delineation of the whole infarct volume. An alternative is to estimate this volume by using the ABC/2 formula, which was originally designed for the estimation of hemorrhage volume [6]. The manual delineation is a tedious and time-intensive task, and the CIV estimation based

on the ABC/2 rule only approximates the total CIV. Aiming to address these problems, Boers et al. [3] proposed a method to automatically calculate the CIV in follow-up non-contrast CT scans. This method was validated by comparing it with manual delineations performed by experienced radiologists.

The method proposed by Boers et al. was implemented using MATLAB [7] and took a long time to run (in the order of 10 min). It uses an intensity-based region growing (IRG) algorithm, which is responsible for more than 95% of the total processing time. To evaluate the potential benefit of heterogeneous computing for this application, we replaced this method with a new version of the IRG algorithm developed with the FlowCL framework. The integration of the new IRG algorithm with the previous MATLAB implementation was straightforward because MATLAB allows external code calls.

In short, the objective of this case study is to understand how heterogeneous platforms can be used and what is their potential value for medical imaging applications. To achieve this objective, we run the method for automated CIV calculation with two different implementations of the IRG algorithm, one based on the original MATLAB code and the other using FlowCL. Below we provide an overview of the complete method for automated CIV measurement used in this case study, and then we describe both the sequential and the parallel IRG implementations.

3.1 Automated Cerebral Infarct Volume Measurement

The automated CIV measurement proposed by Boers et al. was designed to process non-contrast CT scans of the whole brain of the patients. The volume measurements are performed for a part of the brain that is segmented from the image using a region growing algorithm (IRG). In this algorithm, a voxel is added to the segmented volume if the difference between its intensity and the average intensity of the segmented volume so far is smaller than a specific threshold. To compute the CIV, this algorithm was repeated for 7 different values of threshold, going from 1.5 until 4.5 with steps of 0.5 Hounsfield units (HU). The algorithm requires a position as starting point (called seed point) in the infarcted lesion. The seed position is set by an experienced radiologist and this assures that the correct infarcted area was selected.

The brain midline is used to prevent the segmented region from leaking to into the contralateral hemisphere, i.e., the region cannot grow into the other side of the brain. This midline is detected based on the geometric center and the most extreme mid-sagittal bone or nasal cartilage structures present on the scan. Also, to avoid leaking into the hypo-attenuated ventricles, the hypo-attenuated region close to the geometric center is segmented and excluded from the segmentation of the infarcted area. All the steps of the segmentation process are illustrated in Figure 3. This process is repeated for 7 thresholds. In the end of this process, the observer must select the best result that most agrees with the scans.

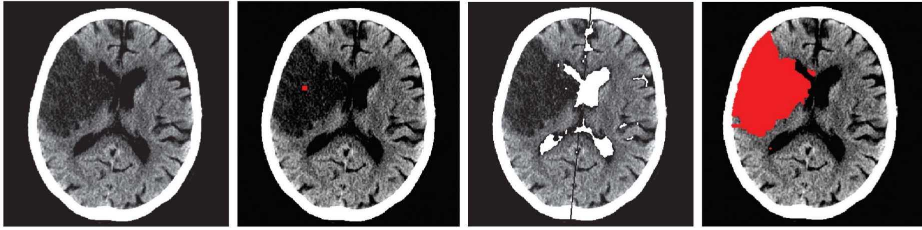

Fig. 3. CIV segmentation steps. From the left to right: a CT scan showing an infarct in the right hemisphere (left of the image); the seed position defined by a radiologist; the determined midline and the ventricles segmentation; and the final segmentation representing the CIV.

3.2 Intensity-Based Region Growing Algorithm

Region growing is a segmentation technique to select an image area that is connected according to a specific condition [8]. In intensity-based region growing (IRG), the intensity of the voxel is used as criterion to include or not a voxel to the region. Starting from a given seed point, the IRG algorithm iteratively adds voxels to the region such that the following condition is satisfied: $|I - A| \leq T$, where I is the intensity of the processed voxel and A is the average intensity of the selected image area. The voxel I must be in the neighborhood of the selected area and it is included in this area when its intensity is smaller than or equal to the threshold T.

Different neighborhood definitions can be applied (e.g., for 3D images it can take 9 or 26 neighbors into account), and the order in which the voxels are considered for inclusion may influence the final result. The IRG is also sensitive to the chosen threshold T; for this reason, 7 different thresholds are used in the CIV method, and the user can pick the best result.

The sequential implementation of IRG in the original CIV measurement method updates the average of the selected image area immediately after the inclusion of each voxel, and the updated average is used in the test to include the next neighboring voxel. In the parallel implementation of IRG, computing devices in the heterogeneous platform simultaneously process the voxels based on the same value of A. The average intensity A is only updated after all the neighbors of a given voxel are considered for inclusion. Therefore the sequential and the parallel algorithms perform inclusion tests based on potentially different values of A, and can deliver different results.

4 Experimental Results and Comparisons

To evaluate the speed-up obtained with the heterogeneous platform, 53 CT scans were processed in two different hardware configurations with the original and the GPU implementations of the automated CIV measurement method. The complete method was executed in both cases, however for timing purposes only

Table 1. Hardware configurations

Hardware Detail	Configuration 1	Configuration 2
CPU Name	Intel Core i7-3930K	Intel Xeon E5-2620
CPU Clock	3.20 GHz	2.00 GHz
GPU Name	NVIDIA GeForce GTX 550 Ti	NVIDIA Quadro K600
GPU Clock	900 MHz	876 MHz
GPU Memory Clock	4104 MHz	1782 MHz
GPU Driver Version	9.18.13.1106	9.18.13.2000

the IRG part was considered. Both hardware configurations have 12 CPU cores and 192 GPU cores, however one is slower than the other – see Table 1.

The CT scans include the entire brain and were performed with thin-section acquisition by using 8 different multi-section CT scanners with at least 16 sections, but mostly with 64 or more sections. The 32-bit MATLAB version 8.0.0.783 (R2012b) was used to run the MATLAB code. Microsoft Visual C++ 2010 was used to compile the region growing algorithm for heterogeneous platforms. These software were executed on 64-bit Microsoft Windows 7 Enterprise operating system on both hardware configurations. Execution times were measure before and after calling the IRG function in the MATLAB code. Note that this time includes overhead of internal function call for the sequential implementation, as well as for the external call of the program for the heterogeneous platform implementation.

All the CT scans were analyzed using exactly the same parameters for 7 threshold values $(1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5)$. Parameters that must be manually configured, such as the region growing seed position, were defined only once and used in all runs of the method. Because different threshold values have a great influence in the size of the segmented volume and, consequently, also in the algorithm execution time, we compare the execution time separately for each threshold value.

Figure 4 shows the speed-up factor for the parallel respectively to the original IRG implementation. As we can see, speed-ups of 36 times were obtained. In general, larger gains are obtained for higher threshold values. Higher thresholds produce bigger segmented volumes, which require more computations and, consequently, result in more expressive speed gains. For lower thresholds, the execution time varies among different scans (larger standard deviation). Small thresholds generate smaller segmented volumes, which are more sensitive to the inclusion of neighboring voxels.

Also note that due to this sensitivity regarding the small volumes, the implementation for heterogeneous platforms can be slower than the original implementation. Note however that only a minor performance reduction is noticed. In most situations the performance of the new implementation is better as presented in figure 5. We must highlight that the automated CIV measurement requires the region growing algorithm to run with 7 different threshold values and, because of this, the performance gains obtained for higher thresholds values

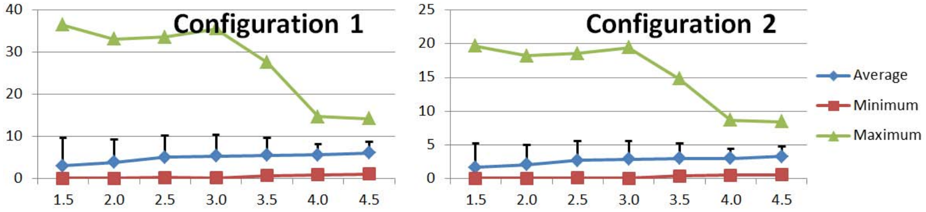

Fig. 4. Speed-up on heterogeneous platforms (vertical axis) for each threshold value (horizontal axis). Bars indicate standard deviation from the mean speed-up calculated for 53 scans.

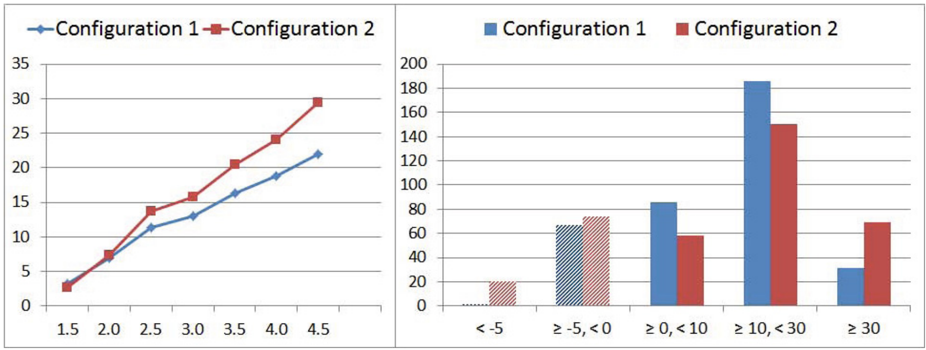

Fig. 5. Differences in execution time between both implementations (new - old). *Left:* Average difference in seconds (vertical axis) for each threshold value (horizontal axis) calculated for 53 scans. *Right:* Histogram of differences in execution times in seconds for all thresholds. The negative ranges indicate the runs where the original implementation was faster than the new implementation.

compensate for the loss for smaller thresholds in the total processing time. In no case the new implementation had a total processing time slower than the original implementation when all 7 thresholds are considered. The new implementation was faster in 82% of the scans using the hardware configuration 1 and in 75% of the scans using the hardware configuration 2.

To evaluate the differences in the quality of results obtained with both implementations we calculate the Dice coefficient for each threshold value individually - see Table 2. Similarly to what we found in the execution time analysis, the greater differences are measured for smaller thresholds. However, this variation in the results does not have a great impact in the method, because the final segmentation must be selected by a human observer which will filter out the segmentations that are not consistent with the images. Moreover, usually the selected segmentation is generated with one of the middle threshold values. The extreme threshold values are used as a safe margin to assure that the most adequate segmentation will be inside this interval.

Table 2. Dice coefficients for each threshold

Threshold Value:	1.5	2.0	2.5	3.0	3.5	4.0	4.5
Maximum:	0.99994	1.00000	1.00000	1.00000	0.99247	0.99141	0.97592
Average:	0.85936	0.89620	0.94856	0.93499	0.91252	0.91199	0.89875
Minimum:	0.34349	0.25384	0.68682	0.84418	0.82026	0.76991	0.75164
Standard Deviation:	0.18170	0.16985	0.07482	0.05082	0.05052	0.06906	0.07345

As shown in Table 2, the Dice coefficients for the threshold between 2.5 and 4 are higher than 0.9. These results indicate good agreement between segmentations when compared to variations found in results obtained with manual segmentation by experts. For example, during the validation of the original method, [3] found that the Dice coefficient for segmentations manually defined by two experienced radiologists were 0.84 ± 0.08 ranged from 0.63 to 0.94 [3].

5 Conclusions and Future Work

In this work we presented how the FlowCL framework, which was developed for intuitive heterogeneous platform programing, was used in a medical imaging application. Part of a previously developed and validated method for automated cerebral infarct volume measurement was adapted for heterogeneous platforms using the FlowCL framework. Only the code related with the intensity-based region growing algorithm was modified. All other pieces of code and software used in the method were not modified. We compared the two implementations of the automated CIV measurement method in order to investigate the potential of heterogeneous platform in medical imaging applications. The results of the implementation for heterogeneous platform were obtained faster and were also consistent with the results of the original implementation. This study shows that heterogeneous platforms can increase performance in medical imaging applications. This indicates that other computationally demanding medical imaging algorithms could also be adapted to run on heterogeneous platforms in a straightforward manner with the FlowCL framework.

In the present study, only GPUs and multicore CPUs were used as computing devices. However, there are other different computing devices, such as FPGAs, that were not included in this study and which can be also used in a more comprehensive future study.

Acknowledgements. Part of this work has been funded by ITEA2 10004: MEDUSA.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)

2. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A package for opencl based heterogeneous computing on clusters with many gpu devices. In: 2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), Heraklion, pp. 1–7 (2010)
3. Boers, A., Marquering, H., Jochem, J., Besselink, N., Berkhemer, O., van der Lugt, A., Beenen, L., Majoi, C.: Automated cerebral infarct volume measurement in follow-up noncontrast ct scans of patients with acute ischemic stroke. American Journal of Neuroradiology (2013)
4. van Geldermalsen, S.: Work In Progress Thesis - FlowCL. Master's thesis, University of Amsterdam (2013)
5. Khronos OpenCL Working Group: The opencl specification (2012)
6. Kothari, R.U., Brott, T., Broderick, J.P., Barsan, W.G., Sauerbeck, L.R., Zuccarello, M., Khoury, J.: The abcs of measuring intracerebral hemorrhage volumes. Stroke 27(8), 1304–1305 (1996)
7. MATLAB: version 8.0.0.783 (R2012b). The MathWorks, Inc., Natick, Massachusetts (2012)
8. Pham, D.L., Xu, C., Prince, J.L.: Current methods in medical image segmentation. Annual Review of Biomedical Engineering 2(1), 315–337 (2000)
9. Saver, J.L., Johnston, K.C., Homer, D., Wityk, R., Koroshetz, W., Truskowski, L.L., Haley, E.C., et al.: Infarct volume as a surrogate or auxiliary outcome measure in ischemic stroke clinical trials. Stroke 30(2), 293–298 (1999)
10. Spafford, K., Meredith, J., Vetter, J.: Maestro: Data orchestration and tuning for opencl devices (2010)