Optimizing a Biomedical Imaging Orientation Score Framework

Okwudire, C.G.U. Palatnik Minguell, M. Zhang, X. Kudchadker, T

Eindhoven University of Technology, The Netherlands

http://wwwooti.win.tue.nl/

{c.g.u.okwudire, m.palatnik.minguell, x.zhang, t.kudchadker}@tue.nl

Abstract

A branch of Biomedical image processing involves analyzing images containing elongates structures. The enhancement of these structures in noisy image data is often required to enable automatic image analysis. A framework for such noise reduction based on Coherence Enhancing Diffusion (CED) using Orientation Scores (OS) has been developed. However, owing to the high computational complexity and high memory consumption of this approach, the current implementation is not able to process sizeable images in reasonable time. This paper presents a GPU/CPU-based optimization of the OSCED framework. The primary goal of this work was to reduce the execution time of the framework by harnessing the processing capabilities offered by an existing GPU cluster. First, the bottlenecks were identified. These were subsequently improved by applying a number of CPU- and GPU-based optimizations. Using a set of reference images, we show that the performance of the framework improved by at least an order of magnitude following our optimizations. In addition, we present a 'split-and-merge' approach and illustrate its potential for further performance improvement using the existing GPU cluster as a reference. We conclude that significant performance gains can be obtained by applying our approach to a suitable cluster configuration. Furthermore, there is still room for optimizing the parts that are currently executed on the CPU.

1. Introduction

Nowadays, the use of Graphics Processing Units (GPUs) for general-purpose high-performance computing is gaining widespread prominence. One of the reasons for this trend is the fact that GPUs are becoming more programmable using high-level languages (e.g. CUDA) and are thus, more accessible to developers of high-performance applications. Furthermore, it is believed that this trend is here to stay, prompting owners of legacy applications to consider porting their code to GPUs.

In this paper, we describe the optimization of a framework for biomedical imaging using a heterogeneous cluster consisting of both Central Processing Units (CPUs) and GPUs.

1.1 Context and Motivation

Human beings are well-equipped to look at noisy images. However, for automatic analysis of images, noise can be really challenging to handle and can influence measurements in a negative way. Therefore, noise reduction in the form of diffusion is often used to remove noise from an image before further processing. In many cases, linear diffusion with a Gaussian kernel is used, but this also diffuses edges which is often not desired. Non-linear diffusion methods such as Coherence Enhancing Diffusion (CED) [6] can be used to overcome this problem. The reason is that at each point the structure is examined and only diffusion *along* an edge is performed and never *across* an edge.

One limitation of the classical CED is that only one direction is taken into account. With the Orientation Score approach [3–5], the image is split up in many orientations and CED is applied on the complete set of images. This solves the problem of only one orientation at each location and therefore, crossing and splitting structures can be handled in a better way. This is important in, for example, medical images such as fibers in microscopy images or blood vessels in X-ray images. An example of noise reduction using the OSCED technique can be seen in Figure 1.

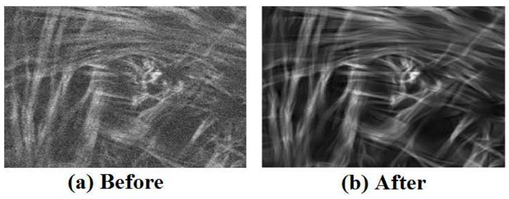

Figure 1. Noise reduction with the orientation score approach.

Unfortunately, the drawback of these algorithms is that their high computational complexity and high memory consumption limit the maximum image resolution that can be processed. For instance, many medical images have sizes of 1024x1024 (i.e., one Megapixel) or larger and need to be processed in a reasonable time, say seconds to minutes. The current implementation is not able to process these images at the desired rate.

Before describing the problem addressed in this paper as well as its solution, we first provide a context for the application.

1.1.1 The Orientation Score Framework

The orientation score framework is a tool that has been developed to implement the orientation score approach. It consists of two parts namely: (i) `mathvisioncpp`, a complex and generic application-independent C++ library for mathematical image processing and (ii) `os_ced`, an application that is a C++ implementation of the orientation score approach for processing and analyzing images containing elongated structures. Both the library and application were developed within the Biomedical Imaging Analysis (BMIA) group at the Eindhoven University of Technology as there were no suitable add-ons to perform complex image analysis in Mathematica [4].

1.1.2 The GPU Computer Cluster

For the purpose of this study, a cluster of computers was set up. As shown in Figure 2, the cluster consists of five physical machines.

The `gpucluster` machine is defined as the 'global host' via which the other machines can be accessed. The four `gpunode` machines are the computation 'power houses' of the cluster, each equipped with quad-core Intel i7 CPU and four NVIDIA GeForce GTX570 GPUs.

1.1.3 Definition of Terms

To avoid ambiguity, we define the following terms as used in this paper:

- *CPU* refers to any of the CPU cores on the `gpucluster` or `gpunode` machines.
- *GPU* refers to any of the GPU cores on the `gpunode` machines.
- *The (OSCED) application*, except as otherwise stated, refers to the combination of `mathvisioncpp` and `os_ced` (see Section 1.1.1).
- *Machine*, except as otherwise stated, refers to one of the `gpunode` machines. *Cluster machine* is used when reference to the `gpucluster` machine is intended.

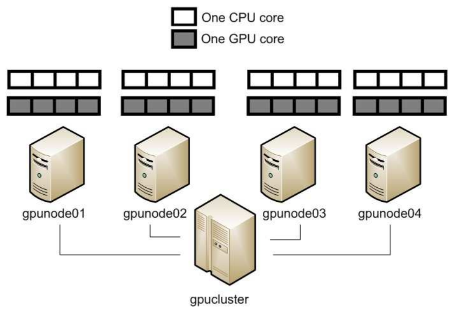

Figure 2. The GPU cluster with four computer nodes.

1.2 Problem Description

The goal of this study is to improve the execution-time performance of the OSCED application (described in Section 1.1.1) by porting it to the GPU cluster (described in Section 1.1.2).

Although GPUs are gaining prominence for high-performance general-purpose computing, not all legacy applications are well-suited for execution on GPUs. Thus, a secondary objective of this work is to investigate the suitability of the application for GPUs and to propose a cluster configuration for optimal performance.

1.3 Contributions

The main contributions of this work are:

- We identify the main bottlenecks of the application.
- We show how these bottlenecks can be addressed by porting the application to a heterogeneous cluster, thereby improving the application's execution time by nearly an order of magnitude.
- We show the benefits of our proposed 'split-and-merge' approach for further performance improvement.
- We predict the effects of different cluster configurations on the application's performance.

The rest of this paper is organized as follows. Section 2 introduces the application and identifies bottlenecks. Section 3 presents the different optimizations that were performed to address these bottlenecks. Section 4 provides an evaluation of the results obtained, discusses the implications of the cluster configuration on the application's performance, and discusses future work. Section 5 concludes the paper.

2. Profiled Application

A logical first step in optimizing any application is to first identify the bottleneck, i.e. operations with a relatively long execution time and/or high invocation count. Optimizing these so-called 'hotspots' often results in significant performance improvements. In this section, we describe the profiled application.

Table 1. Single GPU profiled application

Function	Time for boat	Time for noisy_fibers
`Convolve::calculateLevel_`	102.48s	2499.39s
`ScaleSpace::ExplicitDiffuser::Calculate`	4.85s	154.37s
`OrientationScores::OSConvert-DiffusionTermsToCartesian`	2.30s	71.69s
`OrientationScores::SteerOrien-tationScoreDerivatives`	2.04s	63.44s
Rest	29.11s	886.29s
Total	140.78s	3675.18s

For the purpose of this study, two reference images were used as benchmarks for quantitative analysis of performance improvements. These reference images, `boat` and `noisy_fibers`, took approximately 147 seconds and one hour, respectively, to execute with the original code on one of the `gpunode` machines. Whereas `boat` was selected for its relatively short execution time (useful for fast testing), `noisy_fibers` was selected for its image characteristics as well as for its execution time which is representative of typical medium-sized images.

Table 1 shows the identified hotspots for the two reference images used in this study[1]. The profiling was done by using the CPU timing utility already present in the `mathvisioncpp` library (at an earlier stage, the Intel VTune Amplifier XE 2011 was very useful especially for identifying control flow).

As the profiling results show, the `calculateLevel_()` function dominates the execution for both images. Actually, this function is at the heart of the orientation score approach. It is used to perform convolution and correlation on lines from the different image orientations. For example, it is invoked 282 times and 1402 times for the `boat` and `noisy_fibers` images, respectively. The execution time per invocation varies depending on the size of the input data, the size of the convolution/correlation kernel as well as on whether the input data is real- or complex-valued.

Furthermore, we remark that the 'Rest' in Table 1 represents the hundreds of small functions which are invoked during the complex orientation score computations but each having a relatively low execution cost of a few milliseconds.

3. Optimizations

In this section, we briefly describe the optimizations that we performed in order to address the bottlenecks identified in Section 2.

3.1 CPU-based Optimizations

Although the application was initially designed with high performance in mind and subsequently improved for multi-CPU execution in an earlier work, we identified a number of CPU-based optimizations.

The optimizations employed were: (1) loop unrolling; (2) code simplification based on fixed template and/or input parameters;

[1] Unless otherwise stated, all results reported in this paper were averaged over five executions

Table 2. Performance gains due to optimizations

Image	Original	After CPU Optimizations	Speedup	Time for Single GPU-Core Version	Time for Four-GPU-Core Version	Speedup
boat	140.78s	40.51s	3.48	17.90s	17.76s	1.0
noisy_fibers	3675.18s	1121.67s	3.28	523.28s	473.17s	1.11

(3) exploitation of the four CPU cores on a single machine using OpenMP, an API for multi-threaded, shared memory multiprocessing; and (4) forced in-lining of frequently executed code to avoid function call overhead. The results are shown in Table 2, and were largely due to OpenMP and in-lining optimizations applied to `calculateLevel_()`; details are excluded owing to space constraints.

3.2 Single-Machine Single-GPU Implementation

In this step, we added a single GPU node to the execution of the multi-CPU-optimized code from the previous step i.e., we employed four CPU cores (as before) and one GPU on a single `gpunode` machine. The GPU was programmed using CUDA.

The resulting performance gains for the ported functions are shown in Table 3. In order to maximize the gains, we attempted, while porting, to maximize GPU occupancy, to minimize access to global memory, as well as to apply other best practices recommended by the GPU vendor, NVIDIA [1].

Table 3. Performance gains per kernel for the boat image due to single GPU port

Ported Function	Time before	Time after	Speed-up
`Convolve::calculateLevel_`	19.21s	5.87s	3.27
`ScaleSpace::ExplicitDiffuser::Calculate`	4.85s	0.44s	10.95
`OrientationScores::OSConvertDiffusionTermsToCartesian`	2.30s	0.19s	12.00
`OrientationScores::SteerOrientationScoreDerivatives`	2.04s	0.32s	6.37

These results show the benefit of using GPUs for computationally-intensive and/or frequently executed loops. This benefit arises from the streaming Single Instruction, Multiple Thread (SIMT) architecture of the GPUs that allows the processing of multiple blocks of data at the same time.

We note that similar improvements were measured for the `noisy_fibers` image. We do not include those results here for the sake of brevity; instead we reflect their cumulative effect on the total execution time of the image in Table 2.

3.3 Single-Machine Multi-GPU Implementation

In this step, we further parallelize the bottleneck function, `calculateLevel_()`, by utilizing all four GPUs on the single machine for the processing (i.e., the actual convolution/correlation).

The results of this optimization are shown in Table 2. An interesting observation is that the optimization did not yield a speedup in the order of 3 to 4 as would be expected by intuition (since we move from one GPU to four). In order to explain these results, two factors must be considered. First, the execution of the `calculateLevel_()` function is split into three parts, namely: reading, processing and writing. This was done because memory access was highly strided (i.e., there were huge skips in successive read and/or write locations). As an example, a stride of 230400 was observed for the `boat` image in some cases. Strided access to GPU global memory has a very huge performance penalty [1]. Therefore, it must be avoided as much as possible. Hence, in the reading part, the input data was copied to contiguous blocks in memory before passing it to the GPU. Similarly, the contiguous output data was written back to the appropriate locations during the writing part. Reading and writing had to be performed on the CPU because they required access to CPU memory.

Another benefit of splitting the function into three parts was that it enabled us to accurately determine the size of data structure instead of having to deal directly with the rather complicated pointer arithmetic in the original implementation. In fact, our success in finally porting this function after several attempts was largely due to this approach of splitting it up.

Despite optimization of these parts in the previous step using OpenMP, they still accounted for about 60% of the total execution time of the `calculateLevel_()` function. Hence, according to Amdahl's law [2], the maximum expected speedup would be around 2.5 times.

The second limiting factor is the fact that the processing part, which is performed on the GPU, is memory-bounded. In other words, getting the data in and out of the GPU memory takes more time than the actual computation. Therefore, although the task of processing was split between four GPU nodes, each node is still limited by the PCI Express bus' bandwidth. This also explains why the speedup for the smaller image is smaller: Less data means less processing; hence, the effect of the memory access bottleneck becomes more prominent.

3.4 GPU Cluster Implementations

As shown in Figure 2, the provided cluster includes four `gpunode` machines. Nonetheless, we only used one of them up to this point. Thus, in this section, we exploit the entire cluster.

3.4.1 Four-node Implementation

In this step, we execute the single-machine multi-GPU implementation on all four `gpunode` machines. We achieved this by splitting the input image into four parts, each processed simultaneously by one of the machines. Rather than first dividing the input image and sending parts to each of the machines for processing at *runtime*, we upload the entire image to all the machines at *compile-time*. Each machine, based on its unique ID, then determines what portion of the image to read and process. The assignment of machine IDs was realized using the Message Passing Interface (MPI), a widely used intercommunication library. By using this approach, we save the overhead of transferring image parts between machines at runtime as the images are transferred before the execution of the program. Such overhead could become significant for large input images.

After the processing is completed, the output image parts are recombined on the `gpucluster` machine. For this purpose, we implemented a simple image merging tool, `imagemerger`. The results of this optimization are shown in Table 4. As can be seen from the table, the time required for merging the image is negligible compared to the execution times of the image parts. Also, the data copying time measured includes the time expended on copying intermediate results from the `gpunode` machines. In any case, the output image has to be copied back from the gpunodes and since this time was not included in earlier optimization steps, we also ignore it in this case. Hence, we take the (average) execution times of the slowest input image parts (i.e., the first row in Table 4) as the total execution time, ignoring both data copying and image recombination times.

Using Overlaps to Smoothen Boundaries As is common to most divide-and-conquer algorithms, artifacts may occur around boundaries between the recombined parts; these artifacts need to be taken care of. To this end, an overlap of some pixels was added to each image part to allow for smoothing at the edges. In the results presented in Table 4, we used overlaps of 20 pixels on each side for the boat image and 40 pixels for the noisy_fibers image. The choice of these overlaps in this initial investigation of the approach was a guesstimate of approximately 20 - 25% of the original image size. In principle, it is possible, based on a number of input parameters, to determine the minimum padding required for acceptable outputs. However, such a calculation would not be trivial and fell outside the scope of this study.

There was also some change in the overall brightness of the output images. This expected effect was due to the removal of the mean of the image before processing that occurred when the image was split into parts. However, a linear scaling of the end result will solve these problems and can be easily implemented. Moreover, since the outputs preserved the desired characteristics (i.e., the elongated structures in the presence of image noise), they were considered to be acceptable as the noise reduction is achieved and the quality of the resulting image is comparable to the one produced by the original code.

Table 4. Mapping the application to the entire computer cluster

Processing Step	boat (20 pixels overlap)	noisy_fibers (40 pixels overlap)
Max. exec. time per part	8.333s	174.230s
Data copying time	2.200s	6.000s
Image merging time	0.002s	0.028s

3.4.2 Sixteen-node Optimization

As a final optimization step, we considered using the computer cluster in a different way: Instead of splitting the image into four parts, we would split it into sixteen parts. However, for this to work, we would have to disable any parallel execution both on CPU and GPU. In other words, each part would be executed on a CPU-GPU pair on one of the machines.

The rationale behind this approach is that, as seen so far, the execution time does not linearly depend on the size of an image (cf. total execution times in Table 1 compared to image dimension of 160x160 and 400x400 for the two images, respectively). This means that the smaller the image the higher the speedup relative to the original execution time. Hence, by splitting into more (smaller) parts, it may be possible to achieve better performance despite using a single-CPU single-GPU optimized version of the application. Unfortunately, this approach could not be implemented owing to time constraints.

4. Evaluation

In this section, we briefly summarize the results obtained, discuss the suitability of the application for execution on a GPU, and suggest possible directions for future work.

4.1 Overview of Results

Table 5 presents an overview of all versions of the application as discussed in Section 3, including the original application.

Table 5. Summary of results

#	Version	Time for boat	Speedup (w.r.t. previous)	Time for noisy fibers	Speedup (w.r.t. previous)
1	Original code	140.78s	-	3675.18s	-
2	CPU improved	40.51s	3.48	1121.67s	3.28
3	Single machine single GPU	17.90s	2.26	523.28s	2.14
4	Single machine multi GPU	17.76s	1.01	473.17s	1.11
5	4-node cluster	8.33s	2.13	174.23s	2.72
	Overall speedup	16.9		21.1	

4.2 Discussion and Future Work

By looking at the results, we can make some interesting observations. Firstly, CPU optimizations (version 2) yielded some significant performance improvements. Given the size and complexity of the mathvisioncpp library, we were unable to optimize all parts. Thus, there is still room for more such optimizations and it may pay off to focus on before or alongside further GPU porting.

In terms of suitability for execution on the GPU, the results show the benefit of using GPUs especially in going from versions 2 to 3. However, with the ‘split-and-merge’ approach, it is arguably possible to gain comparable speedup without GPUs by using a cluster of several CPU cores. At least, with the current results, it seems sufficient to have a single GPU per machine and instead invest in more CPUs. In the future, different configurations can be experimented with.

Another possibility for future work is to further improve the calculateLevel_() function by applying techniques such as asynchronous transfers and overlapping transfer with computation as explained in [1]. These were not implemented owing to time constraints.

5. Conclusion and Recommendations

In this paper, we described the results obtained by porting a computationally-intensive and complex mathematical imaging framework to a GPU cluster consisting of a number of CPUs and GPUs. Based upon the results obtained, a speed-up of 16.9 for the boat image and 21.1 for the noisy_fibers image, we conclude that there is room for more optimization. By employing the ‘split-and-merge’ approach described in this paper, we strongly believe that the desired performance of processing megapixel-sized images in seconds can be realized. To this end, we recommend that the identified directions for future work should be implemented.

References

[1] NVIDIA CUDA C programming guide. June 2011.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] R. Duits. *Perceptual Organization in Image Analysis: A Mathematical Approach Based on Scale, Orientation and Curvature*. PhD thesis, Eindhoven University of Technology, Eindhoven, September 2005.

[4] E. M. Franken. *Enhancement of Crossing Elongated Structures in Images*. PhD thesis, Eindhoven University of Technology, Eindhoven, December 2008.

[5] M. A. van Almsick. *Context Models of Lines and Contours*. PhD thesis, Eindhoven University of Technology, Eindhoven, September 2006.

[6] J. Weickert. Coherence-enhancing diffusion filtering. *Int. J. Comput. Vision*, 31:111–127, 1999.