Optimization of a 3D Scene Reconstruction Application

Alex Loizidis, Diana Ahogado, Fanis Grollios, Ivana Kostadinovska, Huy Nguyen

Eindhoven University of Technology, The Netherlands

http://parse.ele.tue.nl

{a.loizidis, d.c.ahogado.alvarez, t.grollios, i.kostadinovska, x.h.nguyen}@tue.nl

Abstract

The work presented in this paper is a solution for optimizing an existing 3D Scene Reconstruction application [1]. The application is aimed to match and reconstruct scenes from extremely large unstructured collections of photographs. The reconstruction process, which performs complex algorithms from image matching to large scale optimization, can take days. In this paper we propose a way to speed up the performance by porting the application on GPU computing. Our experimental results demonstrate that it is now possible to reconstruct 3D objects five times faster.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; C.4 [*Performance of Systems*]: Modeling Techniques

General Terms Performance

Keywords Single-GPU, Multi-GPU, parallelization

1. Introduction

Internet provides access to a vast, ever-growing collection of photographs of cities and landmarks from all over the world. A simple search for a city name in Flickr returns millions of photographs capturing different parts of the city from various viewpoints. Creating accurate 3D models of cities is a problem of great interest and with broad applications. The 3D Scene Reconstruction application that we are using is a software aimed to reconstruct a 3D model from an unstructured collections of photographs. It is achieved by applying new computer vision techniques. This application has a pipeline architecture which consists of four parts, as shown in Figure 1.

The input for the 3D reconstruction is a collection of unstructured images. The processing goes through four main steps: *SiftGPU*, *KeyMatchFull*, *Bundler* and *Pmvs*. Each of these steps represents an independent application that keeps the communication in the pipeline through input and output files. The final output is a file in a *ply* format that can be visualized as a 3D model.

One problem that occurs when using huge collections of images to reconstruct a 3D model is that it takes a big amount of time (from several hours to several days). In this paper we propose one possible solution for reducing this time. Our focus is optimization of the first three steps of the pipeline: *SiftGPU*, *KeyMatchFull* and *Bundler*, which implement complex algorithms for creating a sparse 3D model from a collection of images. We use the last step (PMVS) only to create the output file that we need in order to visually compare the quality of the results. The optimization is done by porting the source code to one or multiple GPUs, using CUDA [6] programming and OpenMP.

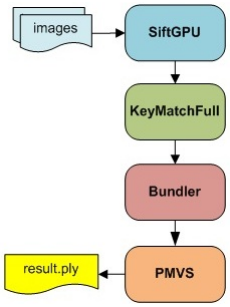

Figure 1. Application pipeline.

1.1 Paper outline

In section 2 of this paper we give more details about the application pipeline. We also describe the data sets and the machine configurations that we used to run the experiments. Further, in section 3 we analyze the approaches we applied for optimizing and we present the results from the measurements we performed. In the last section we conclude with our findings and possible approaches for future improvement.

2. Context

The solution that we propose in this paper for optimizing the 3D Scene Reconstruction application involves optimizing each of the three main steps of the pipeline: *SiftGPU*, *KeyMatchFull* and *Bundler*.

SiftGPU is the first step of the application pipeline. Sift [4], which stands for Scale Invariant Feature Transform, is a method for extracting distinctive invariant features from images that can be used to perform reliable matching between different views of an object or scene. *SiftGPU* [7] is an implementation of Sift for GPU computing.

KeyMatchFull [1] is an intermediate step between the extraction of keys in *SiftGPU* and the *Bundler*. It associates images with common features. It compares two images by calculating the nearest neighbors for all the keys of the images and produces a list of matches.

The *Bundler* application is an implementation of a Structure-from-Motion (SfM) algorithm that takes the set of images, a list of images features and a list of image matches as input, and produces

a 3D reconstruction of cameras and (sparse) scene reconstruction as output [3].

These three applications are independent and they communicate through input and output files. Therefore in each of them we apply different approaches to reduce the execution time, which are described in more details in section 3.

For running the experiments and measuring the performance of the application we used three different data sets:

Xu31 Consists of 31 images of size 3456x5184 pixels.

Car31 Consists of 31 images of size 2048x1536 pixels.

Car61 Consists of 61 images of size 2048x1536 pixels.

The data sets Xu31 and Car31 have the same number of images with different content, whereas Car31 is a subset of Car61 (with 31 and 61 images correspondingly). The reason why we chose these data sets is to measure the impact of the number of images and the content of the images on the results.

The configuration on which we ran the experiments is:

1 compute node
4x NVIDIA GeForce GTX570 boards
1920 CUDA cores
5.5TFLOPs GPU compute power
0.6TB/s GPU memory bandwidth
1KW power supply

Specifications of the GPUs:

GF110, CC2.0
480 CUDA cores
1280MB GDDR5 memory, 320-bit interface
Core clock: 1250MHz (downclocked from 1464MHz)
Memory clock: 1615MHz (downclocked from 1900MHz)

Since there is no available scientific tool to measure the accuracy of the results created in the end of the application pipeline, we compared the 3D models visually. Therefore, the quality of the final result is based on our subjective judgements.

3. Analysis and results

In this section we present the measurements we conducted on the three data sets explained before. We also give the results for the overal performance improvements as well as for every step of the 3D Reconstruction application pipeline separately. Figure 2 shows the improvement that was achieved over the total time of the pipeline.

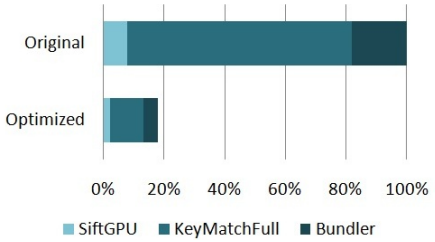

Figure 2. Comparison of the execution time for each step of the original and the optimized version of the application (given in percentage of the total execution time)

By observing the chart we can summarize that the total amount of time in the optimized version is reduced by factor of five. Additionally, the chart gives information about the amount of time each step takes from the total execution time of the pipeline: *SiftGPU* takes less that 10%, *KeyMatchFull* takes around 75% and *Bundle* takes less than 20%. Further in this section we explain the results for each of these steps in more details.

3.1 SiftGPU

SiftGPU processes pixels in parallel to build Gaussian pyramids and detects Difference of Gaussian key points. The result of the process contains the orientations and descriptors of the key points. The original implementation of *SiftGPU* extracts key points for every image on a single GPU. By using the maximum number (4) of GPUs on a node, we modified the *SiftGPU* implementation by processing four images in parallel. We used OpenMP for thread manipulation.

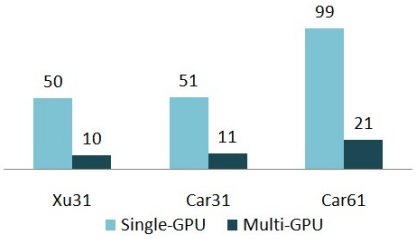

Figure 3. Execution time (in seconds) for the two implementations of *SiftGPU* for the three data sets.

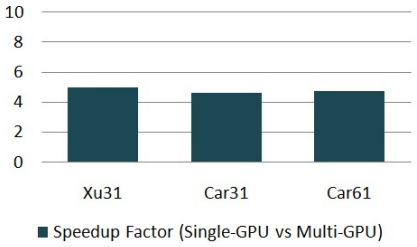

Figure 4. The speedup factors of the new implementation of *SiftGPU*(Multi-GPU) compared to the initial one(Single-CPU), for the three datasets.

The results obtained in Figure 3 show significant improvements between the two implementations. The performance of the key point extraction process was speeded up to 4.53 - 5.1 times as shown in Figure 4. In addition, it proves the advantage of using OpenMP correctly for splitting the *SiftGPU* process into multiple processes. Moreover, Figure 3 also depicts the different execution time of *SiftGPU* for the same number (31) of images for data sets Xu31 and Car31 because the dimension and the key points of the images used in those data sets are not the same.

The achieved result was even more than expected (more than four times) because the way of applying multiple processes reduces the execution time for loading *SiftGPU* initialization and releasing the memory.

3.2 KeyMatchFull

We can identify three phases in the process of key match: unzipping and loading the keys of all images, comparing each pair of images to find the nearest neighbors of each point, and writing the results to file as output. From these, the comparing procedure scales exponentially, when the number of images increases, while the loading/saving procedure scales linearly. For this reason, and also because nearest neighbor search was the most time consuming task, we focused our analysis in this phase.

The initial implementation of the *KeyMatchFull* uses ANN library [5] to match the key points between two images. Although ANN is highly optimized to minimize the number of calculations needed for finding the nearest neighbors, this task can be ported on GPU. In order to do it, we decided to use the Fast K-Nearest Neighbor (KNN) Search integrated GPU Computing [2], a CUDA library that performs the same task. It offers a search algorithm that is much simpler, resulting in many more calculations, but nevertheless adds a big benefit from the parallelization. KNN was a very promising option since it a performance that can be 64 times faster than ANN. We managed to replace the KNN library with the ANN library in the *KeyMatchFull* implementation, since the KNN version that we used is highly customized to meet our needs. The output results we obtained by using KNN are slightly different from the initial ones but this doesn't affect significantly the whole process and is a fair trade-off for the speed up that we achieved.

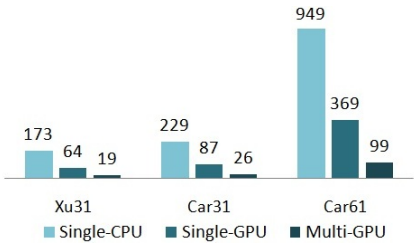

Figure 5. Execution time (in seconds) for the three implementations of *KeyMatchFull* for the three data sets.

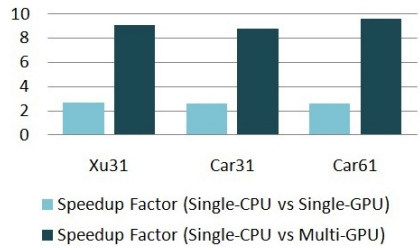

Figure 6. The speedup factors of the two new implementations of *KeyMatchFull*(Single-GPU and Multi-GPU) compared to the initial one(Single-CPU), for the three datasets.

Figure 5 shows the timing results we obtained running the three versions of *KeyMatchFull* (ANN-CPU, KNN-GPU, KNN-4 GPU) with our testing data sets, and Figure 6 presents the speedup factors between these versions. The measurements do not include the loading phase but contain the saving time. It is easily noticed that the GPU version is about 2.5 times faster than the initial one. This final speedup factor differs from the one originally promised by KNN because the testing data that were used in KNN's research paper [2] differ significantly from our case.

Porting *KeyMatchFull* to multi-GPUs was achieved by using task parallelization, performing multiple matching processes in parallel. The reason that the execution results are not four times faster is because the matches on different GPUs are synchronized at the end for writing the output file.

In total, for large data sets, we reached almost one order of magnitude speed-up. This speed-up can be further expanded by using more computer nodes. The calculations in this part are really intensive and cover the cost of memory transfers between different machines making *KeyMatchFull* suitable for multi-node parallelization.

3.3 Bundler

The first step in the optimization of the *Bundler* was to profile it in order to analyze the performance of every routine in the code. From the profiler results, we noticed that the step that takes most time during the execution in the CPU is the Levenberg-Marquardt (LM) algorithm. It is aimed to solve a linear system of equations for bundle adjustment created out of the parameters given as input for the Structure-from-Motion (SfM) process. The function that solves this equations system is in the library sba-1.5 [3], and it takes 22.1% of the execution time.

Based on the described result we decided that the LM procedure should be parallelized. In order to do it, an existing implementation of the *Bundler* that uses Multi-Core CPU and Single-GPU implementations of the SfM was used [8]. That implementation is focused on reducing the use of memory and time in the execution of the Hessian, Schur complement and Jacobian, which are usually used by LM to reduce the size of the linear system [1]. The code is contained in the library *libpba*, which was integrated in the application and used instead of sba-1.5 [3].

After the integration was performed, we ran experiments in order to compare the performance between the Single-CPU, Multi-CPU and Single-GPU approaches. The three data sets described in the introduction of this paper were used. In every experiment two measurements were taken, the time of execution of the SfM routine (which was parallelized) and the total time of execution of the *Bundler*. The results obtained are presented in Figure 7.

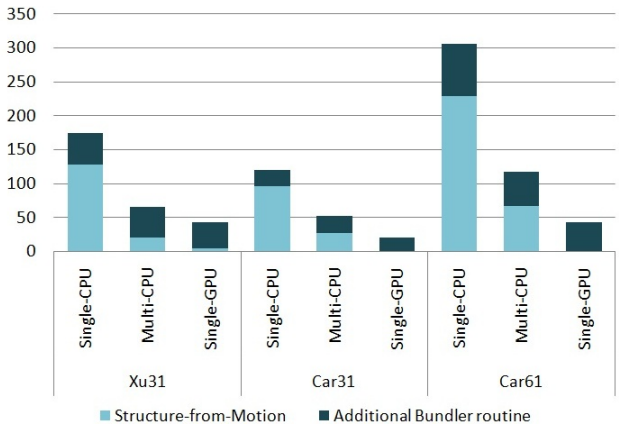

Figure 7. Execution time (in seconds) for the three implementations of *Bundler* for the three data sets.

As it can be observed in the Figure 7, after the parallelization of the Structure-from-Motion the reduction in the execution time of this routine influenced the total execution time of the *Bundler*. In Figure 8 we can see the speedup factors of the Structure-from-Motion routine for each data set, which goes up to more than 200 depending on the data set.

Figure 9 depicts the speedup factors of the total execution of the *Bundler*. We can notice that the speedup factors of the total execution of the *Bundler* are lower due to the part of the *Bundler* that was not parallelized.

By comparing the speedup factors and getting some understanding of the algorithms used in the SfM routine, we concluded that four aspects influencing the execution time are: the number of images in the data set, the number of cameras detected, the number of key points per image, and the number of matches previously detected among the images. In the case of the used data sets, the number of matches found is presented in Table 1.

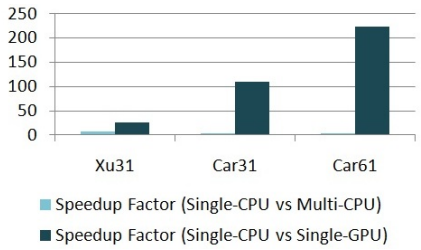

Figure 8. The speedup factors of the two new implementations of the SfM routine in *Bundler*(Multi-CPU and Single-GPU) compared to the initial one(Single-CPU), for the three datasets.

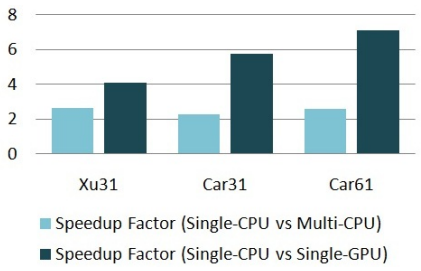

Figure 9. The speedup factors of the two new implementations of the *Bundler*(Multi-CPU and Single-GPU) compared to the initial one(Single-CPU), for the three datasets.

Data set	Number of matches
Xu31	51624
Car31	33602
Car61	51970

Table 1. Number of matches per data set.

We can notice from these results that the speedup factor is higher when the data set size is bigger. This happen, even when the number of matches among images in a large data set is similar to the number of matches in a smaller data set (Car61 and Xu31 in Table 1). That could be due to the number of features in the images, which is bigger in the larger data sets. A bigger number of features leads to generation of a larger number of key points (from *SiftGPU*) that are input for the bundler. It is necessary to make clear that given the high complexity of the algorithms in this paper we are not explaining in details the way all the previously mentioned factors directly influence the execution time.

Regarding improvements to the code by porting the Single-GPU implementation to multiple GPUs, we decided to profile the *libpba* library to find out the function that takes more time during execution. As we found that the function that solves the normal equation system takes 80% of the time and has several several calls to CUDA kernels, we decided that it would be the appropriate point to make an optimization.

After analyzing the code, we made analysis of two approaches to create a Multi-GPU version changing the Single-GPU:

Parallelization by data. In the Single-GPU implementation, a copy of the data is made in the GPU before starting with the execution of all the kernels. Then, after the execution of every kernel, the final result is copied back to the host. In order to make a Multi-GPU implementation it would be necessary to make an initial distribution of the data in the four GPUs, and then change the calls to the kernels in all functions to process the data in parallel. For every kernel, once the data is processed the final results obtained in every GPU must be copied back to the host. There is a trade-off between the time that is gained in the parallel processing of the data and the time that is added copying data back and forth. Since the execution time of the kernels in the Single-GPU version is already very short and the number of calls is big (around 40000 for the data set Xu31), the time added for making copies of the data would result in shorter time of execution.

Parallelization by task. In this case there are dependencies between the inputs and outputs of some of the kernels, which makes it impossible to perform a parallel execution of them. Therefore is not possible to make a complete parallelization of the code by executing the kernels in different GPUs at the same time.

The conclusion after performing the previously explained analysis is that in order to parallelize the *Bundler* application on multiple GPUs, it is necessary to change the architecture.

4. Conclusion

The 3D scene reconstruction application that we optimized was initially designed to run in clusters of multiple CPUs. By using very large input sets and many CPUs, execution time can take up to days. The purpose of our research was to investigate if we can speed-up the whole application by using a number of GPUs and thus decrease the number CPUs needed. Despite the fact that in our analysis we used one CPU, the results we obtained can be generalized and prove our initial assumption. Moreover there is still room for further optimization and better utilization of the GPU capabilities.

We encountered a lot of challenges while trying to parallelize the code-base of the application. Possibilities for parallelism where sometimes limited by the nature of the initial design or the structure of the algorithms. Re-design and use of alternative algorithms was needed. Moreover, the code itself was complex and purely documented making the adaptations even more intricate. In addition, the way the input data sets affect execution time is not always apparent. The latest was also a challenge when trying to evaluate our results. However, we believe that the numbers we presented in this paper are very representative.

Our analysis shows that GPU computing can significantly reduce the execution time of this application. We have already achieved an overall speed-up of about 5 times and with further improvements this number can be increased.

References

[1] S. Agarwal, N. Snavely, I. Simon, S. M.Steven, and R. Szeliski. Building rome in a day, 2009.

[2] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *ICIP*, pages 3757–3760. IEEE, 2010.

[3] M. A. Lourakis and A. Argyros. SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Trans. Math. Software*, 36(1):1–30, 2009.

[4] D. G. Lowe. Distinctive image features from scale-invariant keypoints, 2003.

[5] D. M. Mount. *ANN Programming Manual*, 2010.

[6] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[7] C. Wu. SiftGPU: A GPU implementation of scale invariant feature transform (SIFT). `http://cs.unc.edu/~ccwu/siftgpu`, 2007.

[8] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz. Multicore bundle adjustment.