DD2360 Applied GPU Programming Final Project Report - PG20

Jian Wen
jianwe@kth.se

Heng Kang
hengkang@kth.se

Yuxiang Mao
yuxmao@kth.se

January, 2024

GitHub Repository Link

1 Group Member's Contribution

- **Jian Wen:** Implemented the basic unified memory based on Rodinia's Gaussian module and explored advise and prefetch versions.
- **Heng Kang:** Resposible for testing different versions of code, debugging, recording data and drawing figures for evaluation
- **Yuxiang Mao:** Analysis the Rodinia's Gaussian module.review the prefetch part and further explore the streaming.

Expected Grades: B

2 Introduction

In this project, we have implemented unified memory on Gaussian module of Rodinia Suite and explored possible optimizations including Advise and Prefetch.

Unified memory was first introduced to simplify memory management and to solve the problem in programming applications on the heterogeneous systems arises from the physically separate memories on the host (CPU) and the device (GPU), which means that kernel execution run on GPU will only access to its own GPU. It introduces a pool of managed memory which can be accessed by both the device and the host by the same address by using the virtual memory abstraction. Thus pages in the virtual address space in an application process may be mapped to physical pages either on CPU or GPU memory. In conclusion, the advantages of using unified memory include simplifying GPU programming and migrating data transparently.

Based on the original version, there are two optional mechanisms of unified memory:

Advise: This mechanism involves providing strategic guidance to the Unified Memory system regarding the anticipated usage of memory segments. It facilitates the optimization of memory allocation and access patterns, allowing for targeted data placement and migration between CPU and GPU.

Prefetch: This mechanism entails prioritized data migration to the designated processor's memory prior to actual computational demand. By prioritizing localizing data, it minimizes latency and enhances computational efficiency, ensuring immediate data availability at the onset of processing.

As part of our work, we first implement the basic unified memory function and explore the above mechanisms with Rodinia's Gaussian module.

3 Methodology

3.1 Application Benchmark

The Gaussian module in Rodinia solves systems of equations using the gaussian elimination method. The application analyzes an N×N matrix and an associated 1×N vector to solve a set of equations with N variables and N unknowns. Based on any given input matrix and vector, the application calculate a solution vector and estimate the execution time with and without considering data transferring. By comparing the time, we can evaluate the overall optimization of the unified memory and explore more mechanisms.

3.2 CUDA Profiling

For evaluating and optimizing the performance, the tool chosen is NVIDIA Visual Profiler. Profiling tools are important for understanding program behavior and especially the performance of optimized version of the program. NVIDIA Visual Profiler is a graphical profiling tool that displays the timeline for the program's CPU and GPU activities. The nvprof profiling tool enables programmers to collect and view profiling data from the command-line. Unified Memory is fully supported by both the Visual Profiler and nvprof.

3.3 Code Optimizations

3.3.1 Basic Version

For the basic unified memory implementation, we replace the original `malloc()` with `cudaMallocManaged()` to allocate memory for input and output matrices and vectors. This approach simplify memory management by allowing both the CPU and GPU to access the same memory without performing extra data transfer. This can significantly reduce the time spent for data transfers between devices. The modifications include all memory related statements in main functions, initialization and `ForwardSub(), BackSub()`. Additionally, the unified memory automatically handles data migration between GPU and CPU, ensuring that data is available on any processors. Therefore, in functions outside main function such as `ForwarSub()`, the memory will not get allocated again by CUDA. With unified memory, the variables can be accessed directly so the execution time can be reduced. However, this simplicity may cause extra cost of optimal performance because the system cannot guarantee the most efficient data migration strategy in every scenario.

3.3.2 Advise Version

As mentioned above, the basic unified memory automatically choose the device for data migration but is not able to guarantee the best strategy. Therefore, the basic unified memory solution can be optimized by taking advantages of the advise mechanism - better manual memory controls. In this advise version, we use `cudaMemAdvise()` in conjunction with `cudaMallocManaged()` to tell CUDA runtime on which target device the data migration should be performed. In other words, we can now inform the runtime which processor will primarily access specific memory blocks, which allows more efficient data migration and potentially reducing resource usage. In our implementations, we looked into the data usage on the two processors and decided to allocate the data memory on the device0, which is the Tesla T4 GPU in our experiment environment.

3.3.3 Prefetch Version

Based on the previous advise version, we further explored the prefetch functions of unified memory. We have added `cudaMemPrefetchAsync` after original `cudaMallocManaged` to prefetch data to our GPU. Then we can actively move data to our GPU for later calculations and reduce page faults as well latency caused by on-demand data migration. By prefetching data to the GPU before it's needed by the GPU kernels, and back to the CPU if necessary for post-processing, this approach provides a proactive way to optimize memory usage. We implemented this method with one default stream and four streams respectively to compare their influences to the runtime of unified memory. Theoretically, by using more streams, we are able to allow overlapping data transfers with computation. However, in our evaluation the difference between the two approaches is not significant because the efficiency and performance of Rodinia's Gaussian module basically cannot get improved by changing from serial to parallel. Detailed result analyzes will be discussed in Section 5.

4 Experimental Setup

Google Colab is the platform of this project. The GPU of the Colab is NVIDIA Tesla T4 and its version is 12.2. All the experimental results are the average value of three times measurements.

5 Results

Results are shown in Figure 1 and Figure 2. In our project, four versions of the code were implemented, and their execution times were recorded. The left side of Figure 1 shows that, compared with the original code, the implementation of unified memory significantly reduces the total execution time. In addition, there is a marked decrease in the time excluding kernel execution, which includes memory transfer time. As illustrated in the right side of Figure 1, the unified memory version has the longest execution time, followed by the advised version. The prefetched version is the fastest. In Figure 2, UM has the largest total CPU page faults while UM prefetched has the least number of faults.

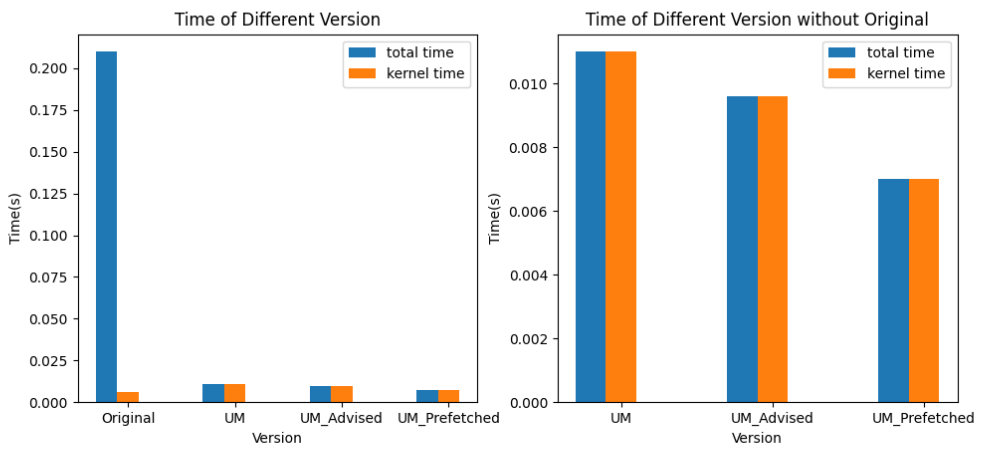

Figure 1: Time of Different Versions

6 Discussion and Conclusion

From figure1, We can see that the basic version, advise version and prefetched version all achieve a better performance in the total time. It is easy to find that the total time has been reduced to one twentieth of the original running time. Also, we can find that the total time is very similar to the kernel time. All of these is because we apply unified memory using cudaMallocManaged and then accessing the memory on the CPU and GPU without memcpy calls, letting the system page the allocations back and forth between CPU and GPU as needed. It means that the memory management can largely improve the performance and decrease the time of data transfer between CPU and GPU.

When comparing the UM, UM-device and UM-prefetched, we can find that the the UM-prefetched has the best performance. It has a 20 percent time reduction compared with UM-advised. It proves that our optimization for the prefetched memory works. Also, the improvements in the advised version by using better manual memory controls also works.

From figure 2, we can see that the total gpu page fault of UM is larger than UM-advised version and UM-prefetched version. And we guess it is the reason why there's a kernel running time reduction.

However, on the other hand, we can find that the kernel time doesn't change a lot. The kernel time even increases a little bit for the reasons of UM.

In conclusion, we realize the UM, UM with advised and UM with prefetched three version to successfully do a progressing optimization on the running time.

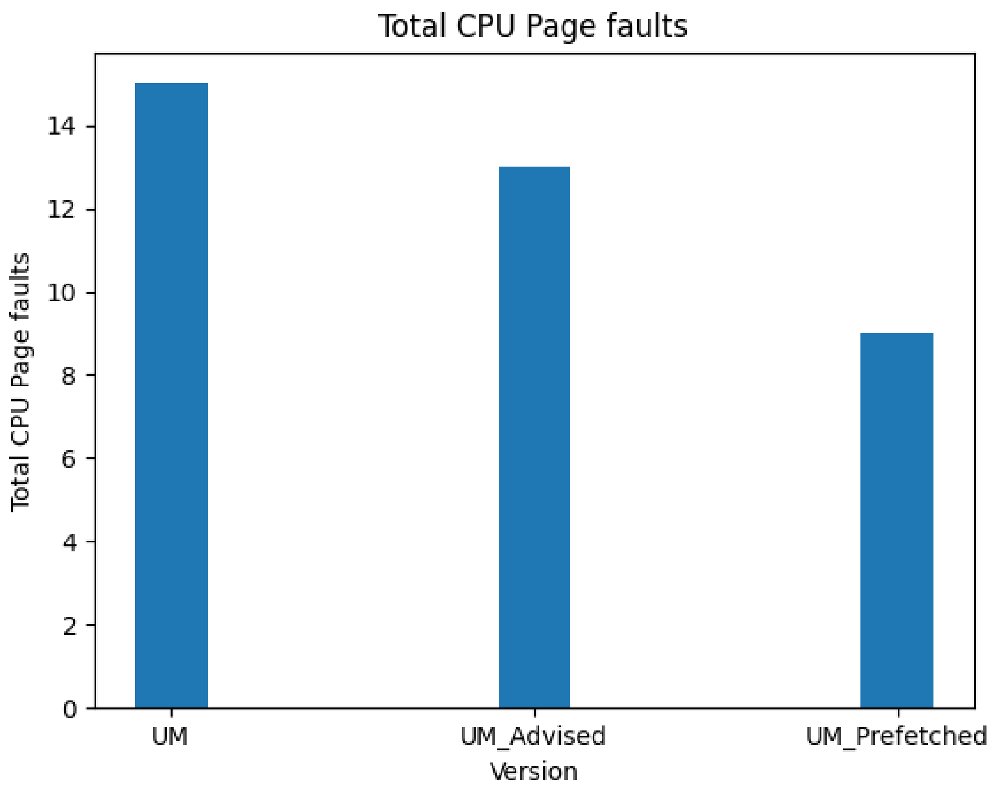

Figure 2: Total CPU Page faults

7 Future work

During this project, we also tried to use cuda streams to improve the performance. We tried to divide the data to achieve stream. But since Gaussian elimination is a two-step cuda kernel function with 1 input matrix, it is really hard to split matrix for several streams to run in parallel. On the other hand, The linear streaming operation for 2 continuous cuda kernel function is useless. We fail in this task in how to make a matrix transformation like this running in several streams. Maybe in the future we can find some new methods to work it out.

Also, another very intersting work is to use cuda cuBLAS library to substitue the cuda kernel function. But we don't have enough time to work on that. Maybe in the future we can work it out.

8 Reference

[1] Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing[C]//2009 IEEE international symposium on workload characterization (IISWC). IEEE, 2009: 44-54.

[2] Chien S, Peng I, Markidis S. Performance evaluation of advanced features in CUDA unified memory[C]//2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC). IEEE, 2019: 50-57.

[3] Mark Harris. Unified Memory for CUDA Beginners. [Online] Available at: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[4] Landaverde R, Zhang T, Coskun A K, et al. An investigation of unified memory access performance in cuda[C]//2014 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2014: 1-6.