DD2360 Final Project Report

Yeqi Wang yeqi@kth.se
Yaoyu Zhang yaoyuz@kth.se
Zeyang Lyu zeyang@kth.se

January 2023

Link: https://github.com/Morgan-LeFay/DD2360HT22/tree/main/Project

Group member's contributions

Yeqi Wang: Responsible for Implementation and Test of the Benchmark and the Basic UM Version, Draft of report;
Yaoyu Zhang: Responsible for Implementation and Test of UM Prefetch Version, Revision of report;
Zeyang Lyu: Responsible for Implementation and Test of UM Advise Version, Revision of report;
Expected Grade : B

Introduction

In this project, unified memory is used to replace explicit memory allocation and data migration in bfs(Breadth-first search), one application of the Rodinia suite.
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.
In the project we successfully apply the unified memory to and evaluate the effectiveness of unified memory. The evaluation of performance in execution time and CPU Page faults are plotted in the *Results* part of the report and *Results* also compare the performance of different version of UM application.

Methodology

A. Benchmark

Graph algorithms are fundamental and widely used in many disciplines and application areas. Large graphs involving millions of vertices are common in scientific and engineering applications. Thus one essential challenge is to optimize the algorithm for graph traversal. In the project unified memory is applied to bfs suite. Breadth-first search is one kind of graph algorithms. This benchmark suite provides the GPU implementations of breadth-first search (BFS) algorithm which traverses all the connected components in a graph.

B. Profiling Tool

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.

C. Change of scope of the source code

When we use Unified Memory, all *cudamemcpy* as explicit data migration are deleted and all *cudamalloc* are replaced by *cudamallocmanaged*. As a result, all the explicit data allocation and migration are replaced with the managed unified memory and both CPU and GPU can use these data.
As for Advise version based on Unified Memory, CUDA 8 provides a new *cudaMemAdvise()* API which provides a set of memory usage hints that allow finer grain control over managed allocations. The same data is accessed by two processors many times within a single multigrid cycle. There is no reuse once the data is moved and the overhead of processing page faults outweighs any benefits from keeping the data local to the processor. We can optimize this phase because it is possible to pin regions to CPU memory and establish a direct mapping from the GPU by using a combination of *cudaMemAdviseSetPreferredLocation* and *cudaMemAdviseSetAccessedBy* respectively. All the variables that we allocate in the unified memory are often used by the GPU so we just set the preferred location as device and also set accessed by CPU in order to establish a remote mapping on CPU. Another useful hint is *cudaMemAdviseSetReadMostly*. It will automatically duplicate data on a specified processor. Constant variables like graph_nodes and graph_edges are set read mostly as they won't be modified in the kernel execution. However, doing so will invalidate all the copies, so it's still an expensive operation.
As for Prefetch version based on Unified Memory, it is to use Unified Memory prefetching to move the data to the GPU after initializing it. CUDA provides *cudaMemPrefetchAsync()* for this purpose. We prefetch large data structures that will be accessed by GPU kernels in a background stream while the GPU kernel is launched in the default stream.

Experimental Setup

The platform used in the project is Google Colab since it is easy to use and convenient for cooperation. The hardware of Google Colab is NVIDIA Tesla T4 and the CUDA Version installed in Google Colab is 11.2.

Results

Using NVIDIA Visual Profiler, we get performance of different optimization including the execution time of GPU activities and total CPU Page fault.
Figure 1 shows the execution time of GPU activities when different optimizations is applied to the application. As for Unified Memory, it is obvious that UM eliminates the process of transferring data between CPU and GPU which accounts for a large fraction of execution time in the original application. However the two kernel consume much more time than the original version. Using advise and prefetch based on UM sees a little advantage when it comes to the execution time, they all reduce the time two kernel execute. In comparison, the prefetch version surpass the advise version in performance.
As for total CPU page fault occur in the process of different optimization. The usage of unified memory introduces CPU page fault in the application. To deal with the problem, the prefetch function shows better performance in this aspect of performance as it reduce total CPU page fault from 146 to 138. In the advise version, by using *cudaMemAdviseSetReadMostly* we can reduce the page faults occurs when getting access to variable of graph_nodes and graph_edges as only the first time of access will introduce CPU page fault.

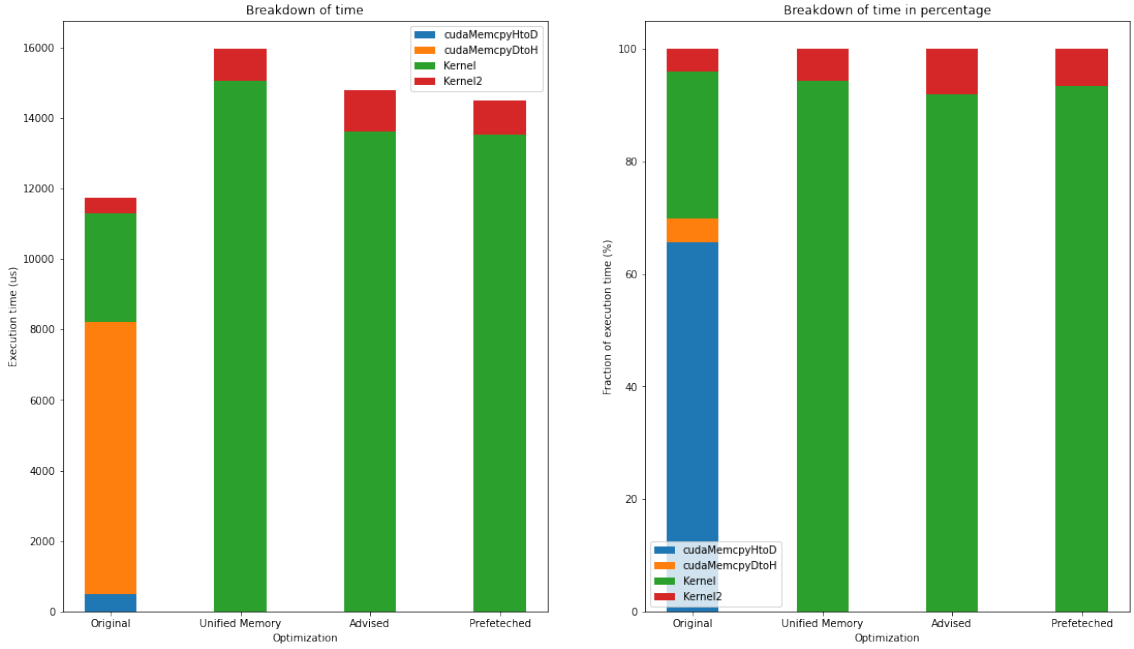

Figure 1: Execution time of GPU activities when different optimization is applied

Discussion and Conclusion

Usage of unified memory eliminate the process of transferring data between the host and the device, which is the most time-consuming process in the application. However, it also burden the kernel to get access the variable that allocated in the unified memory as it dramatically increases the time for kernel execution. The usage of UM advise improves the performance in execution time as it uses *cudaMemAdviseSetPreferredLocation* to allocate physically the specified memory that often used by GPU close to the GPU which is much more time-saving. Similarly, the usage of the prefetching mechanism with the help of *cudaMemPrefetchAsync* to pose the data imgration which could avoid stalling the computation threads in a background CUDA stream.
As for the further work, we will take ways reduce the CPU trash and page fault.

References

[1]S. Chien, I. Peng and S. Markidis, "Performance Evaluation of Advanced Features in CUDA Unified Memory," 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), 2019, pp. 50-57, doi: 10.1109/MCHPC49590.2019.00014.
[2]W. Li, G. Jin, X. Cui and S. See, "An Evaluation of Unified Memory Technology on NVIDIA GPUs," 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 1092-1098, doi: 10.1109/CCGrid.2015.105.
[3]Mark Harris, Unified Memory for CUDA Beginners.https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
[4]Nikolay Sakharnykh, Beyond GPU Memory Limits with Unified Memory on Pascal.https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/