Performance Evaluation of the NVIDIA Tesla P100: Our Directive-Based Partitioning and Pipelining vs. NVIDIA's Unified Memory

Extended Abstract

Xuewen Cui
Viginia Tech
Blacksburg, Virginia 24060
xuewenc@vt.edu

Thomas R. W. Scogland
Lawrence Livermore National Laboratory
Livermore, California 94550
scogland1@llnl.gov

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Livermore, California 94550
bronis@llnl.gov

Wu-chun Feng
Virginia Tech
Blacksburg, Virginia 24060
feng@cs.vt.edu

ABSTRACT

We need simpler mechanisms to leverage the performance of accelerators, such as GPUs, in supercomputers. Programming models like OpenMP offer simple-to-use but powerful directive-based offload mechanisms. By default, these models naively copy data to or from the device without overlapping computation. So, achieving better performance can require extensive hand-tuning to apply optimizations, such as pipelining. Users must manually partition data whenever it exceeds device memory. Our directive-based partitioning and pipelining extension for accelerators [2] overlaps data transfers and kernel computation without explicit user data-splitting. We compare a prototype implementation of our extension to NVIDIA's Unified Memory on the Pascal P100 GPU and find that our extension outperforms Unified Memory on average by 68% for data sets that fit into GPU memory and 550% for those that do not.

1 INTRODUCTION

Systems with accelerators, particularly GPUs, are becoming prominent on the Top500 [3]. Many programming models support these systems, but rather than grapple with unfamiliar programming models, scientists often prefer to keep their existing verified C, C++ or Fortran code. OpenMP, since version 4.0 [5, 6], allows for the straightforward adoption of that existing code. Without extensively rewriting code, users can add directives to offload their computation.

We extend OpenMP to automatically partition data and overlap transfers with computation through pipelining. Our extensions allow data to be mapped into a small buffer to reduce memory use. This simple interface can reduce memory use and improve performance significantly [2].

Recently, NVIDIA announced the latest update to their Unified Memory (UM) [4] feature with CUDA 8 on Pascal GPUs [1]. Because GPU page faults are supported, UM offers an alternative memory oversubscription mechanism. Furthermore, CUDA 8 includes several UM optimizations, e.g., prefetching and duplication.

We evaluate a prototype of our partitioning and pipelining extension using two applications and compare it to UM on the Pascal P100 GPU using multiple data sets. The results show that our partitioning and pipelining extension can dramatically reduce memory use while delivering competitive performance. Our extension outperforms NVIDIA's Unified Memory in most cases, particularly when the dataset size significantly exceeds the available GPU memory.

2 DESIGN

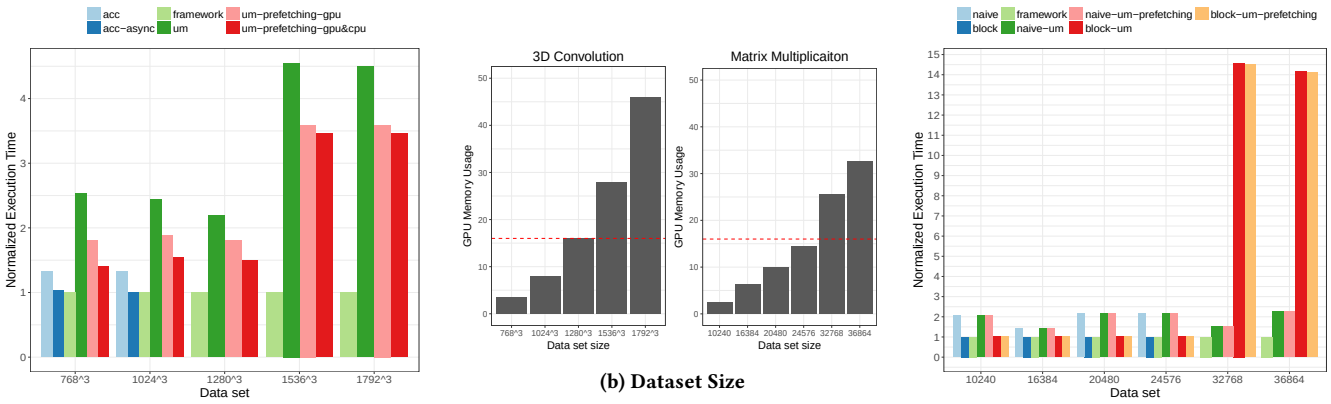
We briefly introduce the clauses that our extension adds. The `pipeline_map` clause extends the semantics of the `map` clause, which makes all data available at the beginning and/or at the end of the region. Our `pipeline_map` clause splits the data updates and subsequent loop computation into multiple subtasks. As with `map`, the `map_type` specifies the data transfer direction. The `array_split_list` defines how to split the arrays. The format of this parameter is `<var>[split_iter:size][0:m]`. The `<var>` is the variable or base pointer of an array. The `[split_iter:size]` parameter identifies the dimension to split while `split_iter` is a function of the loop variable of the subsequent loop. The function defines the split starting offset in that dimension while the `size` defines the range. The `chunk_size` is the number of indices in the subsequent loop that we handle in each device buffer (potentially fewer in the last chunk). The `num_stream` parameter determines the number of GPU streams used, which is the number of chunks that we launch asynchronously. We can also limit memory use with the `pipeline_mem_limit()` clause.

3 EXPERIMENTS AND EVALUATION

We evaluate two applications (3D Convolution and Matrix Multiplication) using our prototype runtime framework that implements our pipelining extension.

3.1 3D Convolution Benchmark

We implemented six versions of 3D Convolution. First, our baseline OpenACC version, which is denoted as "acc", uses that model's naive offload method. Our "acc-async" version uses OpenACC APIs to pipeline the data transfer and kernel computation by splitting the task into multiple chunks. The "framework" version uses our prototype runtime framework. We also implement three versions



(a) 3D Convolution

(b) Dataset Size

(c) Matrix Multiplication

Figure 1: Normalized Execution Time and Dataset Size

that use UM; “um” uses the managed memory for the baseline version. The other two use GPU prefetching and CPU prefetching.

Figure 1b shows the data sizes that we use with the 3D Convolution benchmark. Our two small data sets fit into GPU memory. The third data set size is exactly 16 GB but does not fit into the P100 memory because of runtime memory overhead. The input data of our two large problems exceed the memory limit.

Figure 1a shows the execution times of 3D Convolution versions, normalized to the "framework" version. Pipelining the computational kernel and data transfers can provide 1.4X speedup over naive and the same performance as "acc-async" when the data set fits into GPU memory. However, the naive version and “acc-async” fail with larger data sets. The UM executes all test cases correctly but GPU and/or CPU prefetching improve its performance significantly. The UM version is about 1.5X slower than the framework version for small test cases and 3.5X slower for large sets. We found that although we assign multiple GPU streams for the UM prefetching versions, they only use 1 GPU stream because PGI disabled asynchronous support for P100 and oldr GPUs when managed memory is used to avoid segment faults that can arise if the host and device access the same address.

3.2 Matrix Multiplication Benchmark

Figure 1b shows the data sizes that we use for Matrix Multiplication. We have two small and two medium data sets that fit into the GPU memory, one large data set that slightly exceeds the GPU memory and one large data set that significantly exceeds the GPU memory.

Figure 1c shows the normalized execution times of the Matrix Multiplication benchmark. First, we can observe that the block matrix multiplication algorithm can provide 1.5X to 2X speedup over the naive matrix multiplication algorithm. We also find that prefetching technique is not effective as we observe little performance improvement (usually less than 2%) after applying it. If the data set fits into the GPU memory, the UM versions provide about 96% to 99% performance compared to their corresponding version without using UM, which appears to indicate that the compiler applies optimization that automatically alleviate the GPU page-fault overhead, which makes prefetching less effective. However, if we continue to increase the data set size, making it much larger than the 16-GB GPU memory, the UM version of the naive matrix multiplication algorithm maintains the same performance difference. However, the Block UM version shows huge performance degradation (14X), which appears to indicate that the compiler optimizations result in additional data transfers due to non-contiguous data.

4 CONCLUSION

We present a directive-based pipelining extension for offload models such as OpenMP 4.X. Our extension allows GPU programmers to pipeline data transfers without major refactoring, thus automating the overlap of computation and communication. Furthermore, mapping subsections of the host array to a device buffer reduces memory requirements. Our results show that our extension can significantly reduce memory consumption and deliver excellent performance. It outperforms NVIDIA’s Unified Memory versions, achieving speedups on average of 68% for data sets that fit into GPU memory and 550% for those that do not.

REFERENCES

[1] Ian Buck. 2015. Nvidia’s next-gen Pascal GPU architecture to provide 10x speedup for deep learning apps. (2015).
[2] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-chun Feng. 2017. Directive-Based Partitioning and Pipelining for Graphics Processing Units. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 575–584.
[3] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. 1994. Top500 Supercomputer Sites. (1994).
[4] Mark Harris. 2013. Unified memory in CUDA 6. *GTC On-Demand, NVIDIA* (2013).
[5] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. 2013. Early Experiences with the OpenMP Accelerator Model. In *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 84–98.
[6] OpenMP ARB. 2015. OpenMP Application Program Interface Version 4.5. (2015).