

Performance Evaluation of the NVIDIA Tesla V100: Block Level Pipelining vs. Kernel Level Pipelining

Extended Abstract

Xuewen Cui
Virginia Tech
xuewenc@vt.edu

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

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

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

ABSTRACT

As accelerators become more common [2], expressive and performant interfaces for them become ever more important. 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. Achieving performance can require extensive hand-tuning to apply optimizations such as pipelining [1]. To pipeline a task, users must manually partition the task into multiple chunks then launch multiple sub-kernels. This approach can suffer from high kernel launch overhead. Also, the hyperparameters must be carefully tuned to achieve optimal performance. To ameliorate this issue, we propose a block-level pipeline approach that overlaps data transfers and computation in one kernel handled by different streaming multiprocessors on GPUs. Our results show that, without exhaustive tuning, our approach can provide 95% to 108% stable performance compared to the best tuned results with traditional kernel-level pipelining on NVIDIA V100 GPUs.

1 INTRODUCTION

OpenMP [3, 4] currently requires users to pipeline their code manually for optimal performance. Users must split the task into chunks and launch multiple sub-kernels bound to different GPU streams. This manual task partitioning and sub-kernel launch entails extra code refactoring and can introduce significant programming errors. Further, splitting into multiple chunks may cause extra function call overhead that reduces performance while hyperparameters (e.g., #streams and #chunks) must be carefully tuned to achieve optimal performance. Poor choices may even decrease performance.

We solve these issues through a block-level pipelining extension that performs data transfers and computation on different streaming multiprocessors in one GPU kernel. We evaluate a prototype of our approach and compare it to traditional kernel-level pipelining on NVIDIA V100 GPUs. Our results show that, with no parameter-tuning effort, our approach delivers competitive, stable performance compared to exhaustively tuned, kernel-level pipelining.

2 DESIGN AND IMPLEMENTATION

In our block-level pipelining approach, we first partition the GPU streaming multiprocessors into three groups to handle host-to-device memory copy, main computation and device-to-host memory copy. A persistent thread block of 1024 GPU threads occupies each

streaming multiprocessor. Each thread of the persistent blocks that handle data movement may simply read data from host (GPU) memory and then write it to GPU (host) memory for the host-to-device (device-to-host) direction. A flag array is maintained on GPU memory to label if the data of a block is ready to use as input or output. The computation or data copy thread blocks are triggered by the corresponding flags. We leverage atomic operations to set or to check flags to guarantee the modifications of the flags are visible to all streaming multiprocessors.

3 V100 BANDWIDTH EVALUATION

Our approach requires the streaming multiprocessors to read and to write values directly on host and device memory. We first evaluate contiguous and non-contiguous memory copy bandwidth compared to the traditional memory copy APIs on NVIDIA V100 GPUs. Figure 1 shows the `cudaMemcpy()` bandwidth on a Power9, V100, NVLink2 node with varying number of chunks. The maximum bandwidth is around 60 GB/s with only one chunk. Splitting the array into chunks drops the bandwidth substantially – 99% bandwidth with 1024 chunks. Figure 2 shows the direct access memory bandwidth with a varying number of copy threads. Direct kernel access provides similar bandwidth compared to the maximum bandwidth provided by `cudaMemcpy()` with large numbers of threads. We also leverage vectorized instructions to pack multiple elements into one load/store SIMD instruction. Although this optimization does not improve the maximum peak bandwidth, it reduces the number of copy threads required to achieve it. Also, we save compute resources for kernel computation. Figure 1 shows the `cudaMemcpy2d()` bandwidth with a varying number of chunks. We achieve nearly the maximum bandwidth of the previous test with only two chunks. However, increasing the number of chunks again reduces the bandwidth – by 90% with 64 chunks.

Figure 3 shows the direct access chunk bandwidth. For a fair comparison, we use a 32X32 thread block and copy the array column-wise. Thus, we copy the array by 32 columns each time, which is similar to 64 chunks with `cudaMemcpy2d()` with the same datasets. Direct access achieves nearly the maximum bandwidth as long as at least four streaming multiprocessors are used.

4 PROTOTYPE PERFORMANCE ON V100

We evaluate a prototype of our approach with two benchmarks, 2D Convolution and Generic Matrix-Matrix Multiplication (GEMM),

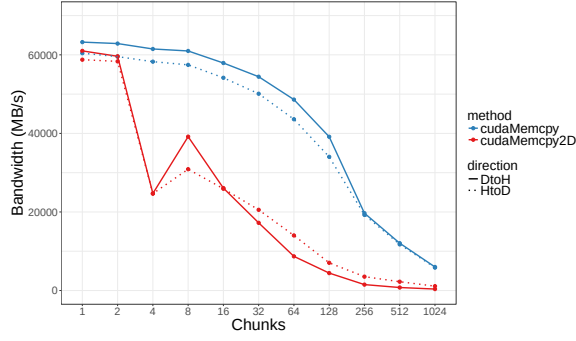


Figure 1: cudaMemcpy & cudaMemcpy2D Bandwidth

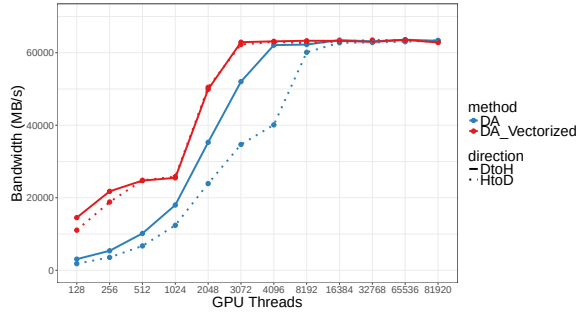


Figure 2: Direct Access Bandwidth

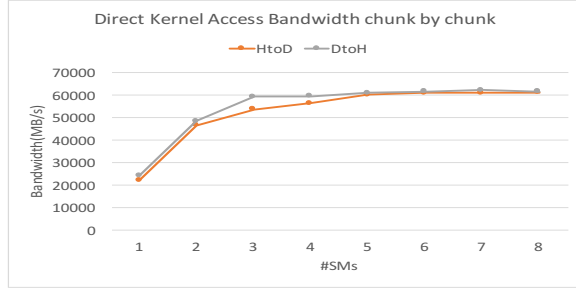


Figure 3: Direct Chunk Access Bandwidth

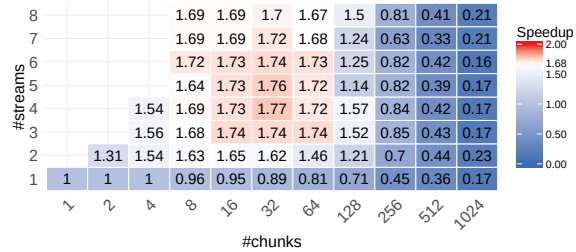


Figure 4: Speedup of Async 2D Convolution (NVLink2)

from the Polybench suite [5] on V100 GPUs with NVLink or PCI-E. Figure 4 shows the speed up with NVLink2 of the async version of

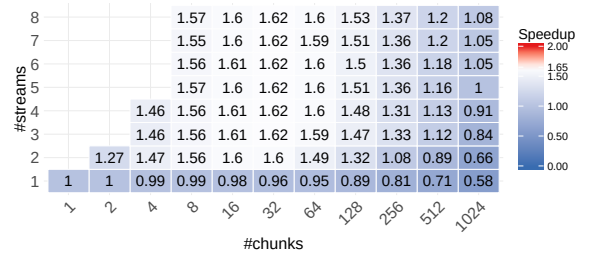


Figure 5: Speedup of Async 2D Convolution (PCI-E)

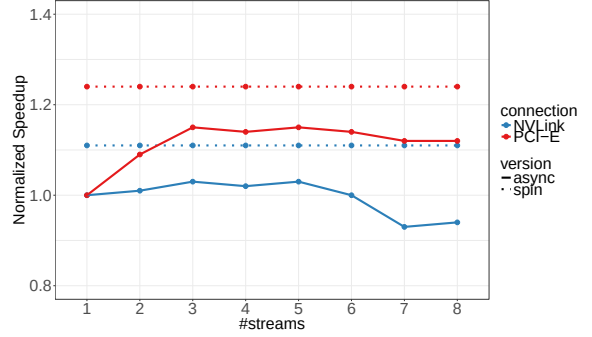


Figure 6: Normalized Speedup of GEMM on V100

the 2D convolution benchmark compared to the sync version. We exhaustively enumerate all hyperparameter combinations (#streams, #chunks) to tune async performance to achieve speed up as high as 1.77X. However, poorly chosen hyperparameters can reduce performance by up to 83%. Our prototype achieves 1.68X speed up, which is 95% of the best case async settings. The blue blocks indicate hyperparameter choices that are slower than our prototype, while red ones indicate faster settings, which occurs rarely. Since Figure 5 shows no red blocks, our prototype outperforms all async hyperparameter settings with PCI-E.

Figure 6 shows GEMM performance. We compare our prototype *spin* version to the async version. We leverage the block matrix multiplication algorithm such that shared memory improves performance, which requires non-contiguous memory copy to pipeline the task. We use a 32X32 block size so the default chunk size is 32 elements. We only vary the #streams to tune async performance and compare to our prototype *Spin* version. We are limited to under 3% speedup on NVLink2 due to the huge bandwidth drop provided by cudaMemcpy2d() with non-contiguous memory copy. Our prototype provides about 10% stable performance speedup over the baseline. On V100 with PCI-E, the speedup increases because the ratio of the data transfer and kernel computation is larger thus increasing the benefit of pipelining. Also, the bandwidth drop of cudaMemcpy2d() is limited by PCI-E bandwidth thus it still provides good performance. In this case, our prototype still outperforms the best async version by about 8%.

REFERENCES

- [1] 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.
- [2] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. 1994. Top500 Supercomputer Sites.
 - [3] 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.
 - [4] OpenMP ARB. 2015. OpenMP Application Program Interface Version 4.5.
 - [5] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. URL: <http://www.cs.ucla.edu/pouchet/software/polybench> (2012).