Improving Performance in Structured GPGPU Workloads via Specialized Thread Schedules

Narendra Prasetya

2021

[TODO] Maybe change title & abstract to reflect we're mostly focussing on grouped column ordering of threads?

Abstract

High performance GPU computing has become very accessible via high level frameworks. These frameworks provide a limited set of operators, such as stencil, permute, fold, scan, which can manipulate data on a GPU. However, in most cases the compiled assembly has inefficient memory accesses with lower cache hit rates and uncoalesced accesses. Threads can be scheduled in such a way to minimize these inefficiencies. We propose several specialized thread schedules to improve performance by leveraging the structure of memory accesses in the high level GPGPU framework Accelerate.

1 Introduction

Modern parallel libraries, such as Accelerate, can often exploit the GPU to achieve greater parallelization and efficiency compared to traditional multithreading on CPUs. [1] However these methods do not fully exploit the locality between threads with a modified scheduler. With a smarter scheduler, programs can achieve better cache and memory utilization. [2]

1.1 GPU Architecture

Massive parallel workloads are executed on numerous cores clustered in streaming multiprocessors (SMs). The memory is structured in a multi-level hierarchy containing a L1 cache for each SM, a shared L2 cache for all SMs and multiple banks of DRAM. [3, 4]

Memory can become a significant bottleneck due to the large amount of threads running concurrently. Caches can alleviate this but is limited in size, and given a large enough problem can cause cache trashing – the premature eviction of cache lines before any significant reuse. [5]

Data shared between threads through the cache can happen read-after-write (RAW) or read-after-read (RAR). RAW has data dependency among tasks, for example in scan operations. RAR has no data dependency and can be executed in any order. [6]

The L1 cache in older Nvidia GPU architectures (Maxwell, Pascal) uses the least recently used (LRU) eviction policy, new architectures (Turing, Volta) uses a non-LRU eviction policy. [7–9] Jia et al. [7] has shown that in Turing and Volta GPU's, the P-chase benchmark presented by Mei and Chu [9] fails to detect the full L1 cache. This is due to the new L1 eviction policy introduced with Volta, where cache lines can be assigned a priority. [7, 10]

A program instructs what a single thread most do, given certain runtime and constant variables. Executing a program spawns multiple threads, each with their own thread id and corresponding block id. These threads are grouped into cooperative thread arrays (CTA), sometimes also called a thread block. A CTA gets executed in SIMT in groups called warps which typically contain 32 threads. There is a maximum number of threads that can fit a CTA, so we often need multiple.

1.2 Accelerate

Accelerate is an embedded purely functional array language in Haskell. [1] Accelerate has a frontend containing the embedded language, and the backend which handles code generation and execution. The PTX backend implements a series of CUDA skeletons.

[TODO] Explain Accelerate execution model

2 Related Work

2.1 Cache Locality

Accessing memory on a GPU can be categorized as deterministic and non-deterministic loads. An access is deterministic when the referenced address is generated from parameterized data such as CTA ids, thread ids, and constant parameters. This classification can be done via backward data flow analysis. The structure of these deterministic loads can be exploited to generate better schedules for threads. Deterministic loads are more likely to have coalesced memory access patterns. Non-deterministic loads will generate more memory requests due to uncoalesced memory accesses. Koo et al. suggests that smarter CTA scheduling can improve performance. [11]

2.2 CTA Clustering

Not all applications have inter-CTA locality and intra-CTA locality can be solved within warps. Inter-CTA locality can be categorized into:

- Algorithm related locality present promising opportunities for inter-CTA reuse.
- Cache-line related locality result from non-aligned memory accesses or coalesced.
- Locality stemming from irregular data structures (pointers) often happens by accident and is thus difficult to account for.
- Write related applications may suffer when multiple CTAs write to the same cache line causing an eviction in the L1 cache.
- Streaming applications are coalesced and aligned.

Li et al. proposes a clustering algorithm for CTAs. For algorithmic locality, the partitioning is based on a dependency analysis on the array references. The CTAs are then remapped according to a set of patterns. These are executed normally or via an agent based system. [12]

2.3 Locality Graph-Based Scheduling

[TODO] Explain PAVER [6]

3 Research Question

> **Research Question 1** *What are the thread schedules for common structured GPGPU operations that result in the fastest execution?*

A schedule optimal for one operation may not be optimal for another operation. The optimal schedule is also dependent on the shape of the input data. We are interested in the theoretical bounds of amounts of cache loads for a given schedule. However, the scheduling can result in extra computation time, so we must also check with real world performance metrics.

> **Research Question 2** *Is specialized structured scheduling equally or more performant than locality graph-based scheduling for structured GPGPU operations?*

[TODO] Text

4 Approach

The built-in GPU scheduler allocates threads and CTAs to streaming multiprocessors. While the architecture does not guarantee an exact execution order of threads, it does group threads into CTAs deterministically and allocates these CTAs in a round-robin fashion. We can therefore manipulate the scheduling by transforming the thread id.

The transformation can be implemented during the code generation phase in Accelerate, specifically by modifying the CUDA skeletons. This allows us to manipulate the schedule depending on various (runtime) parameters such as input size, dimensionality, and GPU-architecture.

4.1 Specialized Scheduling for Stencil Operations

Stencil operations update an N-dimensional array according to a fixed pattern surrounding the updated element (fig. 1).

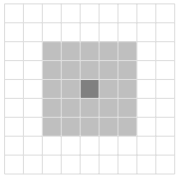


Figure 1: The access pattern for stencil operations on a single element ■ that accesses ■.

Naive implementations of 2D stencil operations will go through the workload on a row-by-row basis. However, if the input matrix is wide enough, data from the previous row can be unloaded before we can reuse it in the next row. While a zigzagging pattern can help alleviate this, the issue would still persist when working on wide enough matrices. By splitting workload into fixed width columns we can keep as much data of the previous row(s) into cache as needed. It would add an extra initial load when threads start working on new columns, but is negligible with sufficiently long columns.

Given the stencil height s_h, the column width c_w and the element size e, the required cache M_{cache} is estimated with:

$$M_{cache} = (s_h + 1)ec_w$$

Given the input matrix with width I_w and height I_h, stencil width s_w, the amount of loads into cache L_{cache} is estimated by:

$$L_{cache} = (c_w + s_w - 1)(I_h + s_h - 1)\left\lceil \frac{I_w}{c_w} \right\rceil$$

We want to minimize L_{cache} by increasing the column width. However, we need to keep M_{cache} small enough that so everything fits on our hardware.

Earlier research suggest using tiling [13], including for higher dimension stencil operations, and needs to be compared to the column method.

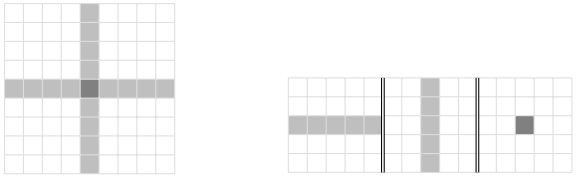
[TODO] Column wise access figures

4.2 Specialized Scheduling for Matrix Multiplication

Matrix multiplication $C = AB$ is defined as

$$c_{ij} = \sum_{k}^{n} a_{ik} b_{kj}$$

In terms of memory accesses, for each element, a row and a column needs to be accessed (fig. 2).



(a) Different arrays stacked. (b) Different arrays separated.

Figure 2: The access pattern for matrix multiplications on a single element ■ that accesses ■.

In the ideal world, caches would be large enough to contain all the relevant matrices. However, with inputs large enough we need to load data into cache multiple times. By splitting the input matrices into blocks (block matrix multiplication), we can limit the amount of times a value has to be loaded into cache.
[TODO] Spatial-temporal graphs for MM
[TODO] Column wise access for MM
[TODO] Strassen's MM on GPUs. [14]

4.3 Generate and Permute

The permutation primitive makes initializes an array with default values and then combines it with the input array according to a combination function and an index map. From a memory point of view we are interested in the predictable memory accesses which can be defined in terms of execution parameters per thread. These can be categorized as: **[TODO] Do I really need to differentiate local? Grouped column ordering does seem to care only about how much cache is needed and is available.**

- **Localized horizontal** accesses are confined to a small horizontal area. Ordering threads linearly is the optimal pattern as this preserves spatial locality.
- **Localized vertical** accesses are confined to a small vertical area. Threads should be grouped horizontally to allow a single cache line to provide for multiple threads.
- **Arbitrarily horizontal [TODO] Description + Effect**
- **Arbitrarily vertical [TODO] Description + Effect**
- **Singular** values are only read once and are a prime target for cache bypassing allowing other fused operations to use more cache.
- **Random** access are either values dependent on an earlier read value or both arbitrarily horizontal and vertical. While cache bypassing my help cache trashing when fused with another operator, it might reduce performance when accessing similar addresses.

[TODO] Propose a way to calculate column width given the accessed offsets and cache specification.

4.4 Fused Operations

[TODO] Fused [15]

4.5 Scan and Fold

Scan primitives, also known as prefix sums, have many parallel algorithms. However, due to the RAW relationship between threads there is less freedom to tweak the scheduling.

Fold primitives are similar to scan primitives, but instead only return a single element. The same (or partial) strategies used in scan can be used for folds.

4.6 Planning

[TODO] Planning

4.7 Preliminary Results

To test the viability of thread scheduling, zigzagging has been implemented for stencil operations. For a 9x9 box averaging stencil on a 4k matrix it resulted in a 13% decrease from 20.8ms to 1.79ms in kernel run times on a RTX 2080 Super.

References

[1] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, 2011.

[2] Cedric Nugteren, Gert-Jan van den Braak, and Henk Corporaal. A study of the potential of locality-aware thread scheduling for gpus. In *European Conference on Parallel Processing*, pages 146–157. Springer, 2014.

[3] NVIDIA. Nvidia volta v100 gpu architecture. 2017.

[4] NVIDIA. Nvidia a100 tensore core gpu architecture. 2020.

[5] Hongwen Dai, Chao Li, Huiyang Zhou, Saurabh Gupta, Christos Kartsaklis, and Mike Mantor. A model-driven approach to warp/thread-block level gpu cache bypassing. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[6] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. Paver: Locality graph-based thread block scheduling for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(3):1–26, 2021.

[7] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[8] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[9] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[10] NVIDIA. Cuda toolkit documentation v11.5.0. URL `https://docs.nvidia.com/cuda/index.html`.

[11] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. Revealing critical loads and hidden data locality in gpgpu applications. In *2015 IEEE International Symposium on Workload Characterization*, pages 120–129. IEEE, 2015.

[12] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware cta clustering for modern gpus. *ACM SIGARCH Computer Architecture News*, 45(1):297–311, 2017.

[13] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pages 32–32. IEEE, 2000.

[14] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Strassen's matrix multiplication on gpus. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 157–164. IEEE, 2011.

[15] DP van Balen. Optimal fusion in data-parallel languages: From diagonal fusion to code generation. Master's thesis, 2020.