Improving Performance in Structured GPGPU Workloads via Specialized Thread Schedules

Naraendra Prasetya

2021

1 Introduction

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 [1]. However, in most cases the compiled assembly has inefficient memory accesses with lower cache hit rates and uncoalesced accesses than manually tweaked code. Threads can be scheduled in such a way to minimize these inefficiencies. With a smarter scheduler, programs can achieve better cache and memory utilization [2]. We propose several specialized thread schedules to improve performance by leveraging the structure of memory accesses in the high level GPGPU framework Accelerate [1].

1.1 GPU Architecture

Since the GPU backend of Accelerate only works with CUDA capable devices (see section 1.2), we will mostly focus on the architecture of newer NVidia GPUs. Massive parallel workloads are executed on numerous cores clustered in streaming multiprocessors (SMs). The memory is structured in a multi-level hierarchy containing an 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 are limited in size, and given a large enough problem can cause cache trashing – the premature eviction of data before any significant reuse [5]. To improve efficiency of caches, caches asume spatial locality via cache lines. A cache line is the smallest unit of data that a cache can hold. For example, an L1 cache on a Turing GPU uses cache lines that hold 128 bytes of data. Therefore, fetching data from memory also brings extra nearby data with it.

Data shared between threads through the cache can happen in a read-after-write (RAW) or read-after-read (RAR) manner. 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. When caches become full, we need to remove data (a cache line) from the cache to allow newer data to be cached. An LRU eviction policy evicts data that is the least recently used. Jia et al. [7] have shown that in Turing and Volta GPUs, the P-chase benchmark that is used to detect the LRU eviction policy presented by Mei and Chu [8] fails to complete over the full L1 cache. Jia et al. conclude that newer architectures (Turing, Volta) uses a non-LRU eviction policy [7–9]. When the L1 cache in Turing and Volta GPU saturates, 4 consecutive cache lines are chosen randomly to be evicted. This is in line with a new eviction policy mechanism introduced with Volta, where cache lines can be assigned a priority [7, 10].

Modern Nvidia GPUs are able to handle various types cache operations and eviction hints. By default, loads are cached at all levels (L2, L1) with an LRU policy. This brings a problem with it: if data is writen to a cached value, we need to evict this cache line from all other L1 caches first, since that value is no longer up to date after our update. As an example, it is also possible to only cache on L2, bypassing L1. Another option is to hint cache streaming, where the loaded cache line will have an evict-first policy to prevent polution of the cache. Similar operations exist for writing data to memory. In both cases it is up to the compiler and programmer to exploit this for extra performance [10].

When working with CUDA the programmer defines a kernel. This is normally done with CUDA C++, an extension on C++ programming language, but in our case Accelerate will handle the generation of kernels (section 1.2). The kernel 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. CTAs get assigned to SMs in round-robin. An SM executes an assigned CTA in a single instruction multiple thread (SIMT) fashion, 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 [10].

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 frontend handles general optimizations such as sharing recovery and array fusion [11, 12]. Further hardware specific optimization is handled on the various backends. There are two LLVM [13] backends provided: one that targets multicore CPUs `accelerate-llvm-native` and one that targets Nvidia GPUs `accelerate-llvm-ptx`. In both backends we compile Accelerate code to LLVM IR. When we want to run Accelerate on a GPU, LLVM will handle the compilation from LLVM IR to PTX, the instructions set for Nvidia's CUDA programming environment [10, 13, 14]. The GPU backend implements a series of skeletons which implement primitive operations such as stencils, generate, permute, and scan. These skeletons define how a program should be compiled and is the part where a custom thread scheduler can be implemented. Further customizations to the scheduler can be done on the executing side of the backend as it controls how kernels are launched.

2 Related Work

2.1 Cache Locality

Meyer et al. describes two principles of locality[15]:

- **Spatial locality:** When accessing a block of data (i.e. a cache line), it should contain as much useful data as possible.
- **Temporal locality:** Once a data block is cached, as much work with it should be done before it is evicted.

Furthermore, if a program is memory bound, we want to increase either or both types of locality depending on the application and hardware.

However, to generate cache-aware programs, we need to know when what gets accessed. Koo et al. [16] proposes a categorization between deterministic and non-deterministic loads. A memory 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. [16] concludes that a smart CTA scheduler can improve performance.

2.2 CTA Clustering

Because CTAs are assigned to SMs in a round-robin fashion, nearby CTAs may not be able to exploit the L1 cache as they can be assigned to different SMs. We call this sharing between CTAs inter-CTA locality. Not every program has inter-CTA locality, for example: a program that increments every element of an array by a fixed amount. Inter-CTA locality opertunities and issues can be categorized [17]:

- Algorithm related locality presents promising opportunities for inter-CTA reuse.
- Cache-line related locality resulting from non-aligned memory accesses or coalesced accesses.
- 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. While we can pass the remapped CTAs to the GPU, the round-robin scheduler may still assign

```
plus3x3s :: Stencil3x3 a -> Exp a
plus3x3s ((_,t,_)
         ,(l,c,r)
         ,(_,b,_)) = Prelude.sum $ Prelude.zipWith (*) kernel [t,l,c,r,b]

plus3x3 :: Num a => Acc (Matrix a) -> Acc (Matrix a)
plus3x3 = stencil plus3x3s clamp
```

Listing 1: An example of defining the stencil function `plus3x3s` and turning it into a stencil operation `plus3x3`.

things unoptimally. Alternatively, an agent based approach may be more benificial: workers are fixed to an SM, so we can more finely control which CTA gets executed on which SM [17].

2.3 Locality Graph-Based Scheduling

Tripathy et al. presented a priority aware vertex scheduler (PAVER) for read-after-read programs which expands on the previous work [6]. PAVER does not work with read-after-write programs, as those might result in a deadlock with the implemented scheduler. It derives a locality graph from PTX code which can be partitioned to maximize data sharing between SMs. The partitioned work is stored as queue of thread block groups. Each SM runs a dedicated worker which gets a thread block group from the global queue and then goes through the thread blocks within the group. If an SM has no more work, it steals work from other SMs by taking from the tail of the thread block group.

2.4 Stencil Computations

Stencil operations update an N-dimensional array according to a fixed pattern surrounding the updated element (fig. 1). Accelerate has stencils as primitive operations and allows these to be easily used (listing 1).

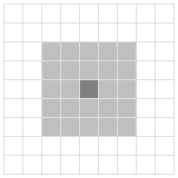


Figure 1: The access pattern for a 5x5 stencil operation on a single element ■ that accesses ■.

Improving the performance of stencil computations has been studied extensively. Kamil et al. [18] presented optimizations targeting temporal cache reuse across multiple stencil sweeps via time skewing. By blocking in the time axis, more data is available to be cached which can help us utilize the larger caches on newer hardware [19].

Zhao et al. [20] presented a vectorization approach and improvements regarding instruction cache hits. However, none of the presented optimizations try to tackle temporal locality within a stencil sweep.

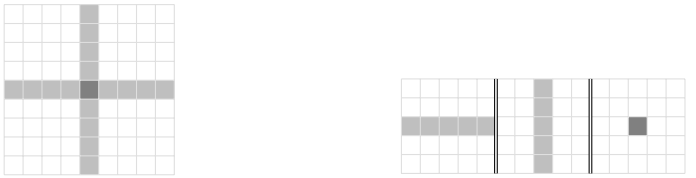
2.5 Matrix Computations

Matrix multiplication take two 2D input matrices to produce a new output matrix $C = AB$ and 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).

Optimizing cache performance for matrix multiplications has been researched extensively. Implementations either use a form of tiling, a divide and conquer strategy or different array structures to improve performance [21, 22].



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

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

The lower bound of matrix multiplication is not $O(n^3)$. Strassen's original algorithm [23] and Winograd's variation [23] runs in $O(n^{2.81})$. Note that the best bound is $\omega < 2.373$ for $O(n^\omega)$ [25]. Strassen's algorithm has also been implemented on the GPU by Li et al. [26].

3 Research Question

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

We want to know what thread schedule on the GPU is optimal for structured operations such as stencils and matrix multiplications. Ideally such a schedule is generalizeable, and we can derive such a schedule from a program's access pattern. For this study, we will mainly look at improving performance on Accelerate as it uses skeleton code to generate the code for various operators. We are interested in the theoretical bounds of amounts of cache loads for such a schedule and its comparison to the naive schedule. Furthermore, the generation and execution of schedules 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?*

The PAVER method by Tripathy et al. [6] is the current state of the art regarding GPU scheduling as it can adapt to various problems similarly to the proposed method. Since we are tackling a smaller set of problems, namely structured operations, more specialized tweaks can be applied and our proposed method may perform better.

4 Approach

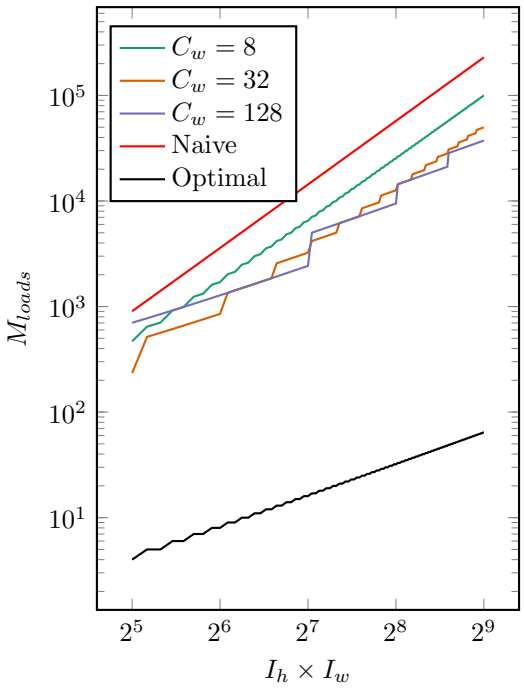
The built-in GPU scheduler allocates threads and CTAs to streaming multiprocessors. While the architecture does not guarantee an exact execution order of threads, it does group threads into CTAs deterministically and allocates these CTAs in a round-robin fashion. We can therefore manipulate the scheduling 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 (run-time) parameters such as input size, dimensionality, and GPU-architecture.

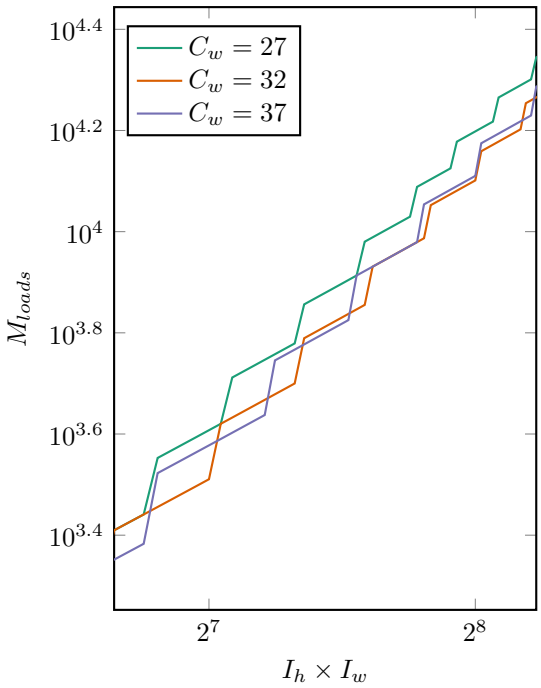
4.1 Specialized Scheduling for Stencil Operations

Naive implementations of 2D stencil operations go through the workload on a row-by-row basis. If the input matrix is wide enough, data from the previous row can be unloaded before we can reuse it in the next row. Rivera and Tseng [27] suggest using tiling, including for higher dimension stencil operations. However, we propose an alternative method: *column grouping*. By splitting the workload into fixed width columns, we can keep as much data of the previous row(s) into the cache as needed. It would add an extra initial load when threads start working on new columns, but is negligible with sufficiently long columns.

In this case, the amount of cache needed correlates to the footprint of the stencil operation running on two rows of the column. When working with multiple threads, an operation can start further ahead



(a) The M_{loads} for a given c_w that is a multiple of M_{width}. Naive describes the case where the cache is too small to share between stencil rows. Optimal describes the absolute minimum of loads possible, assuming the cache is large enough to fit the input array.



(b) c_w that are not a multiple of M_{width} may perform better, but a multiple of M_{width} may be preferred.

Figure 3: The number of cache line load needed for a 7x7 stencil operation with various column widths c_w for $M_{width} = 8$ and $s_h = s_w = 7$.

on the column. We can estimate the desired size of the cache M_{cache} with the stencil height s_h and width s_w, column width c_w, number of threads t and element size e:

$$M_{cache} = e(c_w + s_w)\left(s_h + \left\lceil \frac{t}{c_w} \right\rceil\right)$$

Given image height I_h, width I_w and cache line width M_{width}, we can deduce the amount of cache misses. The number of columns needed to cover the output is

$$\left\lceil \frac{I_w}{c_w} \right\rceil$$

The number of rows needed

$$(s_h + I_h - 1)$$

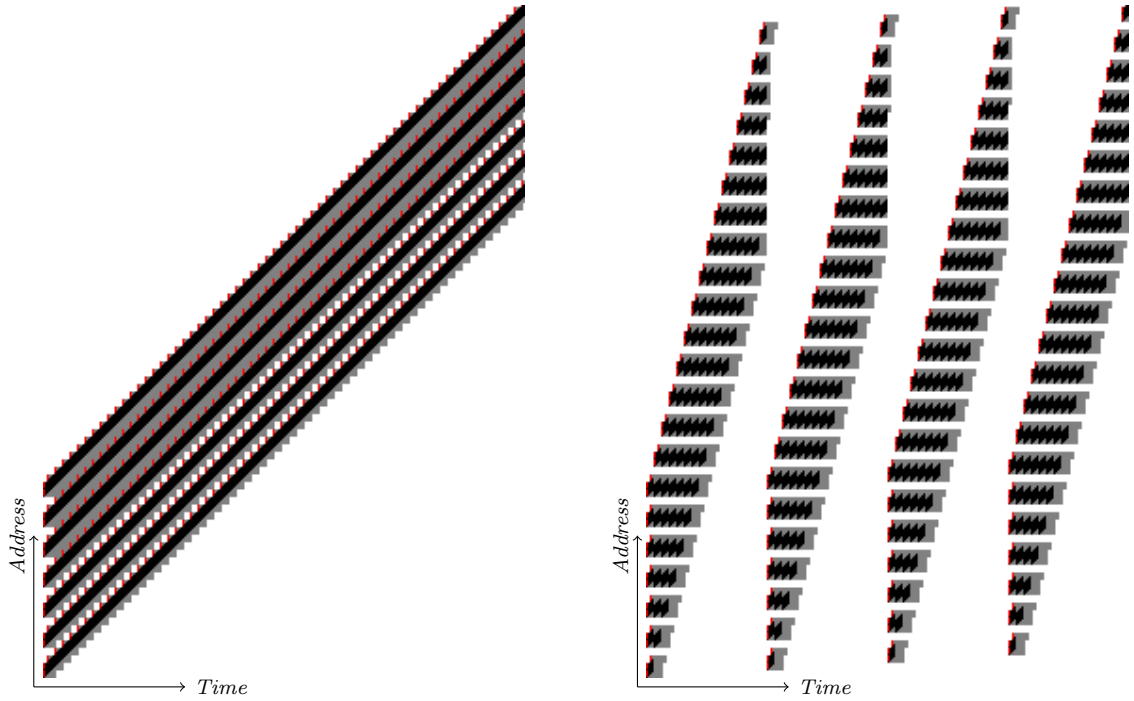
Each row may span multiple cache lines. In the worst case, even if our row is smaller than M_{width}, it still may cross a cache line border

$$\left\lceil \frac{s_w + c_w - 1}{M_{width}} + 1 \right\rceil$$

Which can be combined to estimate the number of cache lines that needs to be brought into memory M_{loads}

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

We want to minimize M_{loads} by increasing the column width, while still staying within the M_{cache} of our physical hardware. There is also an incentive to keep c_w to multiples of the cache line width to decrease the amount of cache line borders crossings. The number of loads for various column configurations is compared to the best and worst case of the naive implementations in figure 3.



(a) The spatial temporal diagram of a 7x7 stencil with linear ordering.

(b) The spatial temporal diagram of a 7x7 stencil with a column of size 4 ordering.

Figure 4: The spatial temporal diagrams of a 7x7 stencil. The vertical axis describes the location in a 2D array which is mapped to 1D address space. The horizontal axis describes time. ■ are addresses of cache lines that are brought into cache. ■ are addresses being accessed. ■ are addresses in cache.

4.1.1 Preliminary Simulation

We analyze the stencil pattern on a simulator with a simple LRU cache with the parameters in table 1. A time step consists of the calculation of a single element of the stencil in which we record the accessed addresses. The state of each address at every time step is plot in a spatial-temporal diagram (fig. 4) which can show us spatial and temporal locality. Spatial locality can be identified by ■ vertical regions where accessed addresses can share the same cache line. Temporal locality can be identified by ■ and ■ horizontal regions where cached data stays in cache long enough to being reused. ■ regions are data that is brought into cache from memory, either due to being unloaded or being evicted earlier.

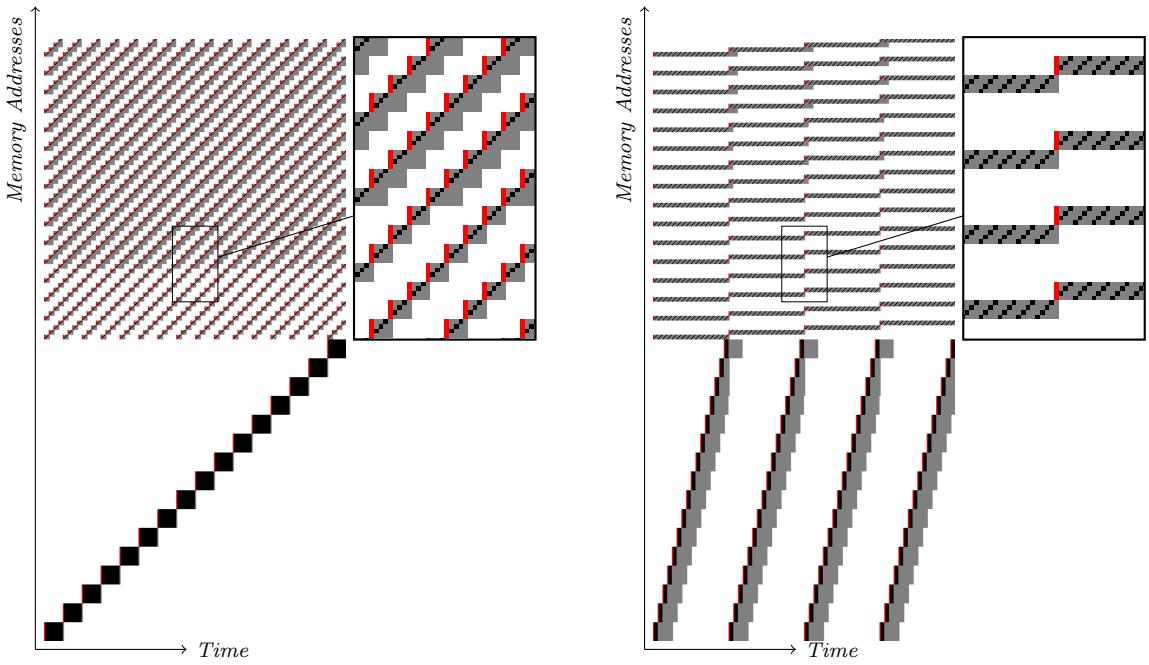
Cache lines	24
Cache line width	4
Eviction policy	LRU
Stencil size	7x7
Input Size	16x16
Column size	8

Table 1: The input parameters to generate figure 4.

For the naive implementation, we observe a large amount of cache misses (fig. 4a). This is because the cache is not large enough to allow for reuse when wrapping around to the next row of calculations. The proposed method solves by also ensuring temporal locality (fig 4b). The limited horizontal range from the columns can be observed in the zigzagging in the diagram.

4.2 Specialized Scheduling for Matrix Multiplication

Let's look at improving cache performance on the naive $O(n^3)$ matrix multiplication. 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. A common optimization is to partition the work into



(a) The spatial temporal diagram of matrix multiplication with linear ordering.

(b) The spatial temporal diagram of a matrix multiplication with a column of size 4 ordering.

Figure 5: The spatial temporal diagrams of a matrix multiplication. The vertical axis describes the location in a 2D array which is mapped to 1D address space. The horizontal axis describes time. ■ are addresses of cache lines that are brought into cache. ■ are addresses being accessed. ■ are addresses in cache.

tiles, however the column based method described in section 4.1 also works.

4.2.1 Preliminary Simulation

We run a similar simulation as in section 4.1.1 with an adjusted amount of cache lines to account for the larger amount of memory processed (table 2).

Cache lines	32
Cache line width	4
Eviction policy	LRU
Input Size	16x16
Column size	8

Table 2: The input parameters to generate figure 5.

The results are compiled into spatial temporal diagram (figure 5). With the original ordering, accessing the columns for each resulting element can incur lots of cache misses (figure 5a). The proposed schedule improves reuse by ensuring that columns in the same cache lines are reused as much as possible (figure 5b). As a result we incur more cache misses on row accesses, but this is mitigated by the fact that these accesses are more efficient since more useful data is contained per cache line.

There are, however, a matrix multiplication algorithms with a lower computation complexity. Li et al. [26] has shown that it is possible to run Strassen's algorithm (see section 2.5) on the GPU. The implementation for a scheduler for Strassen's algorithm is lower priority and may or may not be implemented, since Accelerate does not have an implementation of it.

4.3 Generate and Permute

The generate operation makes an array of a given shape and maps each index to a value (see listing 2). Backwards permutation is a more specific instance of this: the value of each index from the resulting

array is derived from a given index mapping from a source array. The permutation primitive 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. However, attempting to optimize for read performance may hurt performance when multiple threads need to write to the same element using the combination function which needs to be synchronized with other threads.

```
generate :: (Shape sh, Elt a)
-- Shape of array:
=> Exp sh
-- Function that generates values from indices:
-> (Exp sh -> Exp a)
-- Resulting array:
-> Acc (Array sh a)

backpermute :: (Shape sh, Shape sh', Elt a)
-- Shape of the result array:
=> Exp sh'
-- Index permutation function:
-> (Exp sh' -> Exp sh)
-- Source array:
-> Acc (Array sh a)
-- Resulting array:
-> Acc (Array sh' a)

permute :: (Shape sh, Shape sh', Elt a)
-- Combination function:
=> (Exp a -> Exp a -> Exp a)
-- Array of default values (shape is also defined):
-> Acc (Array sh' a)
-- Index permutation function:
-> (Exp sh -> Exp sh')
-- Array of source values to be permuted:
-> Acc (Array sh a)
-- Resulting array:
-> Acc (Array sh' a)
```

Listing 2: The type signatures of `generate`, `backpermute` and `permute`.

To generalize the thread ordering method described in section 4.1, we categorize memory accesses as following:

- **Horizontal** accesses that describe spatial locality. While these can exploit cache lines the most, they may also cause inefficiencies by crossing cache line boundaries.
- **Vertical** accesses that describe temporal locality, due to any thread on the same column being able to reuse accessed data. From a memory perspective we simply have a fixed offset.
- **Streaming** accesses 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 may prevent cache trashing when fused with another operator, it might reduce performance when accessing similar addresses.

Loading horizontally aligned data is much cheaper than loading vertically aligned data, since the latter may span multiple cache lines. We therefore prioritize optimizing for vertical locality. To exploit vertical locality, data must be kept in cache long enough. For the column ordering described in section 4.1, we can control how long data is stored in cache by increase or decrease c_w. Having a generic method to determine c_w allows us to incorporate fused operators since we only care about the relative offset and range of each memory access and the input of fused operators are thread indices.

4.4 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. Due to both primitives having high data dependency between threads and there being multiple algorithms, we will not look at the scan and fold primitives unless time allows.

4.5 Planning

The idea is to first work on simpler discrete problems to get familiar with Accelerate and setting a baseline of improvements. Then generalize the scheduler to work for any structured schedule where threads may work independently of each other. The planning is as following:

- **Week 1:** Stencil scheduler implementation.
- **Week 2:** Stencil benchmarks.
- **Week 3:** Matrix multiplication scheduler implementation.
- **Week 4:** Matrix benchmarks.
- **Week 5, 6:** Analysis for required column width for any structured program.
- **Week 7, 8:** Implementation of analysis.
- **Week 9:** Scheduler implementation for any structured program.
- **Week 10, 11, 12:** PAVER implementation.
- **Week 12, 13, 14:** Paper writing.
- **Week 15, 16:** Reserved for delays or further improvements.

4.6 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 2.08ms 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] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

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

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

[11] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *ACM SIGPLAN Notices*, 48(9):49–60, 2013.

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

[13] The llvm compiler infrastructure. URL `https://llvm.org/`.

[14] Trevor L McDonell, Manuel MT Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe runtime code generation: accelerate to llvm. *ACM SIGPLAN Notices*, 50(12):201–212, 2015.

[15] Ulrich Meyer, Peter Sanders, et al. *Algorithms for memory hierarchies: advanced lectures*, volume 2625. Springer Science & Business Media, 2003.

[16] 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.

[17] 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.

[18] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, 2006.

[19] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache accurate time skewing in iterative stencil computations. In *2011 International Conference on Parallel Processing*, pages 571–581. IEEE, 2011.

[20] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–44, 2019.

[21] Michael Bader and Christoph Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and its Applications*, 417(2-3):301–313, 2006.

[22] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 285–297. IEEE, 1999.

[23] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[24] Shmuel Winograd. On multiplication of 2× 2 matrices. *Linear algebra and its applications*, 4(4): 381–388, 1971.

[25] Josh Alman and Virginia Vassilevska Williams. Limits on all known (and some unknown) approaches to matrix multiplication. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–591. IEEE, 2018.

[26] 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.

[27] 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.