Improving Cache Performance in Structured GPGPU Workloads via Specialized Thread Schedules

Naraenda Prasetya

2022

Abstract

Efficient cache utilization is critical in programs with high data throughput. Improving performance in this area often requires niche knowledge of computer architecture, extensive benchmarking, and algorithms that do more than intuively required. By changing the order in which tasks are executed, the order in which memory gets accessed gets changed. This way, we can manipulate how caches get used. This thesis proposes a column iterator which reschedules a 2D workload. We show that performance can be increased compared to the naive method by implementing the proposed method in C++ with CUDA and as an extension to the data parallel DSL Accelerate.

Contents

Chapter 1

Introduction

Graphics Processing Units (GPUs) are specialized hardware that can manipulate large amounts of data in a highly parallel manner. The hardware's main purpose is processing graphical data, however modern use of GPUs is in the form of general-pupose computing on the GPU (GPGPU) where we exploit the GPUs many cores to run computations normally run on CPUs in parallel.

High performance GPU computing has become very accessible via high-level programming languages and libraries, which provide a limited set of operators, such as stencil computations, permute, fold, scan, which can manipulate data on a GPU [1]. An obvious drawback of high-level languages is that low-level interfaces of the hardware become less available to the programmer as the compiler and libraries handle these. In most cases, the compiled assembly has inefficient memory accesses with lower cache hit rates and uncoalesced accesses compared to manually tweaked code which requires in-depth knowledge about the architecture. On the other hand, any optimizations done to the compiler or library will benefit most programs. An approach that fits between improving the compiler and programmer skill is to improve the thread scheduler. A smarter scheduler can achieve better cache and memory utilization [2]. We propose a simple schedule to improve performance by leveraging the structure of memory accesses in the high level GPGPU framework Accelerate [1].

1.1 Motivation

1.2 Contributions

This thesis shows that the cache efficiency for stencil and matrix multiplication can be improved compared to the naive implementation and the more common tiling approach, by rescheduling via index mapping. While the main focus lies in improving the performance on GPUs, the techniques presented can also be applied on a CPU.

The main contributions of this thesis are:

- An analysis of the theoretical cache utilization and efficiency of both linear and multithreaded (GPU) execution for naive and tiled implementations of stencil and matrix multiplication operations.
- A simple implementable schedule that improves cache efficiency and therefore performance compared to the naive and tiling implementation.
- A benchmark for stencil and matrix multiplication operations to measure performance to the new schedule to the naive implementation.
- An analysis of the theoretical cache utilization and efficiency of a column-based scheduler.

1.3 Outline

We will first introduce the architecture of GPUs in chapter 2: the organization of hardware, the memory hierarchy, the execution model, and the performance of certain access patterns. A few existing optimization techniques are analysed for both stencils and matrix multiplications in chapter 3. We present our alternative approach and its analysis to the aforementioned optimizations in chapter 4. The benchmarks and results from our method are presented in chapter 5.

1.4 Common variable names

Throughout the thesis will use these variables to describe the following:

- L Cache line size (number of elements)
- M Cache size (number of elements)
- M_L Cache size (number of cache lines)
- F Number of memory fetches
- I_w, I_h Input size (number of tasks to produce the output)
- S_w, S_w stencil size

Chapter 2

Background

2.1 GPU Architecture

The GPU backend of Accelerate only works with CUDA capable devices (see section 2.3). Therefore, we will mostly focus on the architecture of newer Nvidia GPUs.

2.1.1 Hardware

Modern Nvidia GPU architectures are composed of multiple GPU Processing Clusters (GPCs), Texture Processing Clusters (TPCs), Streaming Multiprocessors (SMs) and memory controllers. The main point of interest are the streaming multiprocessors which handle the data processing. The GPU uses a single instruction multiple threads (SIMT) execution model, and the scheduler in each SM dispatches instructions to multiple cores to the various specialized cores for each of the various execution pipelines (single and double precision computation, integer computation, etc.).

On the Turing, Volta and Ampera architectures, each SM has four warp schedulers and the accompanying pipelines and therefore Jia et al. [3] suggests that a threadblock should contain at least 128 threads due to SMs on Turing, Volta, and Ampere being split into four processing blocks so at least all 4 schedulures can be completely utilized. [3–6]. 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 [4, 6] (figure 2.1). Data can be shared between threads in the same cooperative thread array (CTA), and therefore on the same SM, via the L1 cache and between all threads via L2 cache.

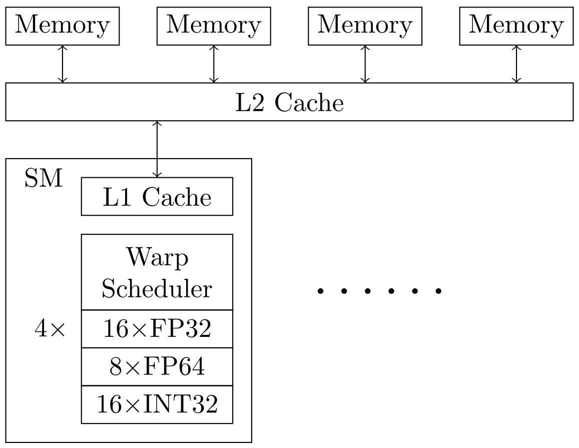


Figure 2.1: An overview of the memory hierarchy of the Ampere architecture. The L1 Cache is local to the SM which executes can 4 warps simultaneously

Aside from the L1 and L2 caches, GPUs also use a translation lookaside buffer (TLB) which caches recent translations from virtual address space to physical address space. By working in virtual memory space, multiple programs can use memory ranges independently of other programs which may want to occupy the same range.

	Architecture		Turing	Volta	Pascal	Maxwell
	GPU Board		T4	V100	P100	M60
	GPU Chip		TU104	GV100	GP100	GM204
	Year		2018	2017	2016	2014
L1 Cache	Size	KiB	32 or 64	32…128	24	24
	Line size	B	32	32	32	32
L2 Cache	Size	KiB	4096	6144	4096	2048
	Line size	B	64	64	32	32
L1 TLB	Coverage	MiB	32	32	~32	~2
	Page entry	KiB	2048	2048	2048	128
L2 TLB	Coverage	MiB	~8192	~8192	~2048	~128
	Page entry	MiB	32	32	32	2

Table 2.1: Summary of the cache specification on various Nvidia GPUs and architectures. Adapted from Jia et al.[3].

2.1.2 GPU Caches

To the programmer, the memory hierarchy is very simplified: there is the compute unit and the memory. Caches are hidden from this model as on most architectures they are managed by hardware.

Memory can become a significant bottleneck due to the large amount of threads running concurrently and the amount of data each thread processes. Caches are much faster than memory and are often on the same chip as the compute unit. However, they are limited in size, and a large enough problem can cause cache trashing – the premature eviction of data before any significant reuse [7]. To improve the efficiency of caches, caches asume spatial locality via cache lines. A cache line is the smallest unit of data that a cache can hold, and fetching data from memory also brings extra nearby data with it. The L1 cache on a Turing GPU (and most other modern architectures) uses 128 byte cachelines which plays well with the 32-thread warp size since executing a fetch for a single precision floating point takes $32 \times 4 = 128$ bytes which can fit in a single cache line. 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 between tasks, for example in scan operations. RAR has no data dependency and can be executed in any order [8].

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. Mei and Chu [9] presented a novel fine-grained pointer chasing (P-chase) microbenchmark to explore unknown GPU cache parameters. P-chase defines an array of indices where each element points to the next index to fetch from the array, thus chasing the pointer.

Jia et al. [3] 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 [9] fails to complete over the full L1 cache. Jia et al. conclude that newer architectures (Turing, Volta) use a non-LRU eviction policy [3, 9, 10]. 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 [3, 11].

On the Turing architecture Jia et al. [3] has found with the P-chase benchmark that the memory access latency for a:

- L1 cache hit (best case scenario) to have a latency of 32 cycles.
- L2 cache hit to have a latency of 188 cycles.
- L2 miss but a TLB hit to have a latency of 296 cycles.
- L2 cache miss and TLB miss to have a latency of 616 cycles.

Which shows that having all relevant data in cache, especially L1, is key to high data throughput.

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 written 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 [11].

2.1.3 Kernel Execution

When working with CUDA, the programmer produces *kernels*, define the functions that should be executed on the GPU. This is normally done with CUDA C++, an extension on C++ programming language, but other methods of compiling for the GPU is also possible. In our case Accelerate will handle the generation of kernels (section 2.3) [11].

Executions on a GPU are directed on both the host and device (GPU).

The host side controls how these kernels should be executed, namely how the threads should be launched and executed. Threads are grouped and defined on a 2 level hierarchy: threads are grouped together in *cooperative thread arrays* (CTAs), also known as thread blocks, and multiple CTAs can be queued for the execution of a single kernel. Both can be controlled upon executing a kernel: the number of threads per CTA (threadblock size) and the total number of CTAs (gridsize). CTAs get assigned to SMs in an arbitrary manner.

When an SM executes a CTA, it splits the work into warps, a grouping of 32-threads. On architecture before Volta, a single warp is executed in a single instruction, multiple threads fashion, where a single program counter is shared amongst the 32 threads. With the volta architecture, independent thread scheduling allows full concurency between threads and the scheduler can group multiple threads into SIMT units.

SMs want to work on multiple warps since this can hide latency. After dispatching an instruction to an execution unit, the scheduler looks for the next warp that can execute an instruction (for example, not waiting for data)

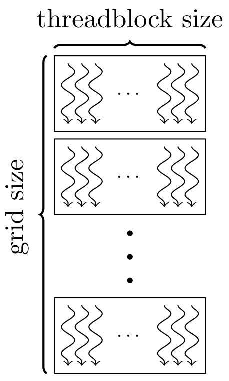


Figure 2.2: GPU workloads are defined by the threadblock size and grid size.

2.1.4 Performance of Access Patterns

Lam et al. and Meyer et al. describe two types of reference reuse [12, 13]:

- **Spatial reuse** occurs when accessing data from the same cache line, increasing spatial locality.
- **Temporal reuse** occurs when the same data is accessed at a later time, increasing temporal locality.

The reuse factor can be kept track of by counting the two types of reuse. Loading data horizontally (sequentially) in 2-dimensional array exploits spatial locality and is therefore cheaper than loading data vertically. Additionally, temporal reuse can only happen when other memory accesses do not displace reusable data from the cache.

Lam et al. proposed a method to model cache interference. In the simplest case where all data used is cached in different locations (and therefore no data is evicted), the number of misses per variable v is described by $D(v)/R(v)$, where $D(v)$ is the total number of references (loads) of v and $R(v)$ is the reuse factor. However, with interference misses when data gets displaced from cache, the total number of misses for v is

$$D(v)\left(\frac{1}{R(v)} + \frac{R(v)-1}{R(v)}M(v)\right)$$

With missrate being

$$M(v) = 1\,(1 - S(v)) \prod_{u \in V - \{v\}} (1 - F(u))$$

$F(u)$ is defined as the fraction of cache used by variable u, and self interference $S(v)$ is defined to be the fraction of accesses that map to non-unique location in the cache. In the context of blocking, the largest block size where no self-interference occurs is called the critical blocking factor. From Lam et al.'s observations with matrix multiplications, the total amount of cache misses gradually declines with larger block sizes until the critical blocking factor is reached.

2.2 CPU vs GPU based multi-threading

CPUs and GPUs differ in multiple ways. GPUs consist out of many more cores compared to CPUs, however this also comes in a reduction in core complexity and capability. Due to the CPUs being focused on single threads, it can exploit better branch prediction. Another hardware specific optimization available is prefetching, where instructions and data can be loaded into cache that are likely to be needed in the near future.

While GPUs can exploit its SIMT nature for parallelization, CPUs may implement single instruction, multiple data (SIMD). For example, Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX, AVX2, and AVX-512) allow CPUs to process multiple data with singular instructions. SSE can handle four 32-bit single-precision floating point numbers while AVX and AVX2 handle eight. AVX-512 can handle sixteen, but is not widely implemented.

Caches also share the same hierarchical structure as the cores, however this differs per manufacturer and architecture. Often the L1 cache is local to each thread, and L2 local to each core with 2 threads per core.

2.3 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 [14, 15]. Further hardware specific optimization is handled on the various backends. There are two LLVM [16] 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 [11, 16, 17]. 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.

2.4 Commonly Applicable Cache Improvements

2.4.1 Optimizations of Blocked Algorithms

Lam et al. [12] expands on the well known idea of working on blocks instead of entire rows or columns.

If all data fits onto cache without eviction, the misses that occur are *intrinsic misses*. In the real world however, data can be evicted by other memory accesses. This interference of reuse is categorized between two cases: *cross interference* and *self interference*. Cross interference assumes the location of data in memory is unrelated to the location in cache, and instead is measured by probablity that the reuse falls within the footprint of the variable. Self interference extends this by taking the cache locations of variables into account, which can happen when the data for a single iteration no longer fits in cache.

2.4.2 CTA Clustering

Li et al. [18] presented a clustering algorithm for thread reordering to improve cache performance on GPUs. It replaces a kernel with a new one with predefined clustering rules. CTAs with inter-CTA locality are clustered together which can then be assigned to SMs. The work in these clusters can be bounded to SMs in two ways: Round-Robin Binding which asumes the scheduler assigns CTAs to SMs in a strict Round Robin policy, and SM-based binding which extracts the current executing SM for a

CTA which it uses to devide the work it will do. The former results in redirection based clustering while the latter forms agent based clustering.

Additionally, Li et al. implemented three optimizations into their clustering framework: **i) CTA Throttling** which limits the number of concurrent CTAs on an SM to reduce resource contention. **ii) Cache Bypassing** to avoid unnecessary cache pollution. **iii) CTA Prefetching using Reshaped Order**

CTA clustering observed no significant speedup for normal and dense matrix multiplications. For convolutional neural networks (similar memory access patterns as stencils), a 1.4× speedup was observed for redirection based clustering, and a 1.2× speedup for other clustering algorithms of the Fermi architecture. However, on the Pascal architecture, convolutional neural networks no speedup was observed, and on Maxwell and Kepler this improvement is limited to 1.1×.

2.4.3 PAVER

Tripathy et al. [8] presented a novel threadblock scheduling method which analyses GPU code and generates a data dependency graph between threadblocks. A cache optimal program is then the clustering of threadblocks which have the least amount of dependency between clusters. Tripathy et al. have observed a 20-40% performance improvement on matrix multiplication depending on the architecture and configuration. However, this method is not easily implementable and no publicly available implemention exists so far.

Chapter 3

Analysis of Existing Approaches

3.1 Assumptions

During the analysis, we will asume a cache with an LRU eviction policy. Additionally, we will not take into consideration cache associativity to simplify the analysis.

3.2 Spatial Temporal Analysis

The memory accesses of an algorithm can be plotted in a spatial-temporal diagram, with the address space on the spatial axis and order of access on the temporal axis. Since only the order of tasks can be manipulated, we do not need to record the individual memory accesses within a thread. Instead, we record the state of the cache after executing a singular task. This also avoids the problem of threads being concurrent and puts the focus on the temporal locality between threads. A different thread order shuffles the columns within this diagram: the same data is accessed but simply in a different order.

Additionally, we annotate this diagram with the cache level (L1, L2, RAM) of each memory address by simulating memory. For the simulation we will use a simple LRU eviction policy which most Nvidia GPUs use and is similar to newer variation on the newer architectures like Turing and Volta, (see section 2.1.2).

The resulting spatial temporal diagrams (for example, figure 3.1) have the vertical axis describing the location in a 2D array which is mapped to 1D address space and the horizontal axis describing time. 🟥 are addresses of cache lines that are brought into cache. ■ are addresses being accessed. ▨ are addresses in cache.

3.3 Stencil Operations

A stencil operation produces an N-dimensional array from a same sized input. It consists of $I_w \times I_h$ tasks with I_w being the first dimension of the input and I_h being the product over the other dimensions. For each element it reads a fixed sized $S_w \times S_h$ neighborhood and writes a single element. Stencils are used for image operations (edge detection, filters, noise reduction), but can also find their use in other fields such as approximating partial differentiation[19] and cellular automata. For example, a 2-dimensional 5×5 box blur filter over an input matrix A can be mathemathically defined as:

$$\texttt{Stencil}(x, y) = \sum_{i=-2}^{2} \sum_{j=-2}^{2} \frac{A[x+i, y+j]}{25} \tag{3.1}$$

Stencils in Accelerate, as shown in listing 1, define the stencil function as `stencil -> Exp b` that takes an N-dimensional tuple to produce a single value. The boundary condition `Boundary (Array sh a)` defines how values outside the limits of the array are handled, either as a predefined function such as `clamp` and `mirror`, or as a user defined function.

```haskell
stencil :: forall sh stencil a b. (Stencil sh a stencil, Elt b)
    -- stencil function
    => (stencil -> Exp b)
    -- boundary condition
    -> Boundary (Array sh a)
    -- source array
    -> Acc (Array sh a)
    -- destination array
    -> Acc (Array sh b)
```

Listing 1: The type signature of the stenciling function in Accelerate.

To implement equation 3.1 in Accelerate, we define a function to sum and divide all elements (listing 2).

```haskell
-- Stencil types: included in Data.Array.Accelerate
type Stencil5 a = (Exp a, Exp a, Exp a, Exp a, Exp a)
type Stencil5x5 a = (Stencil5 a, Stencil5 a, Stencil5 a, Stencil5 a, Stencil5 a)

-- Example of a 5x5 box blurring stencil filter
boxblur5x5 :: Acc (Matrix Float) -> Acc (Matrix Float)
boxblur5x5 = A.stencil s A.clamp
  where
    -- Take a 5x5 array, then select each element and concatenate them
    s :: Stencil5x5 Float -> Exp Float
    s = average . concatMap (^..each) . (^..each)

    -- average all the elements in a list
    average :: Fractional a => [a] -> a
    average xs = sum xs / genericLength xs
```

Listing 2: How to use the stenciling function (listing 1) in Accelerate to produce a 5×5 box blur filter.

3.3.1 Naive

The naive implementation iterates over each of the outputs linearly, horizontally first. The temporal linearity translates to parallelism on GPUs where multiple threads work concurrently on each element of the output array. The naive implementation of the 5×5 box blur filter (equation 3.1 and listing 2) is compareable to the following naive CUDA C++ implementation (listing 3).

While the naive implementation of stencil operations works well enough when enough rows of the input fit in the cache, it begins to fall in performance on larger inputs. More specifically, inputs that are horizontally wide.

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

Table 3.1: The input parameters to generate the spatial-temporal diagrams for stencil operations.

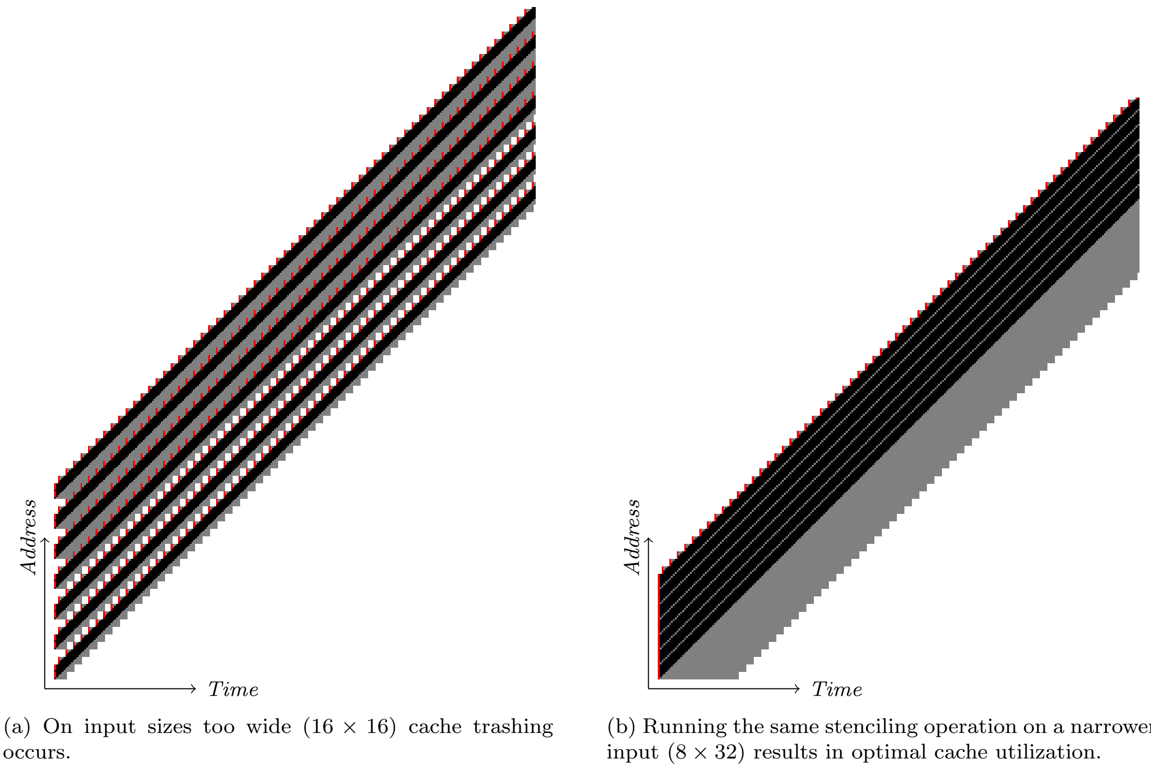
```
#define STENCIL_SIZE 5

__global__ void stencil_naive(float *output, float *input, int width, int height, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int x = tid % width;
    int y = tid / width;

    if (tid >= N) {
        return;
    }

    float result = 0;
    for (int d_i = -STENCIL_SIZE/2; d_i <= STENCIL_SIZE/2; d_i++) {
        int i = min(height-1, max(0, y + d_i));
        for (int d_j = -STENCIL_SIZE/2; d_j <= STENCIL_SIZE/2; d_j++) {
            int j = min(width-1, max(0, x + d_j));
            result += input[i * width + j];
        }
    }
    output[tid] = result / (STENCIL_SIZE * STENCIL_SIZE);
}
```

Listing 3: The naive CUDA C++ equivelant to listing 2



(a) On input sizes too wide (16 × 16) cache trashing occurs.

(b) Running the same stenciling operation on a narrower input (8 × 32) results in optimal cache utilization.

Figure 3.1: The spatial temporal diagram of a 7x7 stencil with linear ordering. See section 3.2.

The lower bound of cache in amount of cache lines needed M_l is bounded by the total horizontal footprint $I_w + S_w$ in amount of cache lines L, multiplied by the stencil height S_h

$$M_l \geq \left\lceil \frac{I_w + S_w}{L} \right\rceil S_h \tag{3.2}$$

While in most cases (such as a 9×9 stencil on a 2048×2048 array of floating point numbers which takes roughly $\tilde{8}0$ KiB of cache.) this enough to fully exploit L2 caches, this can be unoptimal in regard to the L1 cache (64 KiB max on the Turing architecture). The amount of cache lines needed to be fetched from memory F is bound by the worse case (eq. 3.2 is not satisfied) where we consistently evict data from cache before we can reuse:

$$F \leq \left\lceil \frac{I_w + S_w}{L} \right\rceil I_h S_h$$

If equation 3.2 is satisfied, the amount of fetches F is no longer depedent on the stencil height S_h

$$F = \left\lceil \frac{I_w + S_w}{L} \right\rceil I_h$$

With multiple threads active, even more data is required to be kept in cache for optimal usuage. In the best case, all threads are cohesive with overlapping accesses, and in the worst case, threads will be spread out more with less overlapping accesses. Threads in GPUs are grouped by warps, threads contained within are always cohered, and therefore a guarantee for overlapping accesses. Therefore, only when multiple warps are executed on the same SM, divergence in accesses can occur. A single warp of 32 threads uses $\left\lceil \frac{32+S_w}{l} \right\rceil S_h$ cache lines when the threads cover a single rows. When the warp is split between 2 rows, the cache needs to be slightly bigger: $\left\lceil \frac{32+2S_w}{l} \right\rceil S_h$.

Ideally, the whole input array would fit on the cache, but a sufficiently large input (e.g. a 2048×2048 32-bit floating point array uses 16 MiB) will not fit on the L2 caches of modern GPUs ($\approx$ 6MiB of L2 data cache, Volta V100) and cache misses are unavoidable. Even if data would fit on the L2 cache, there would still be potential cache misses at the L1 cache (128 KiB, Volta V100).

The model by Lam et al. [12] and described in section 2.1.2 can be used to estimate the cache misses of the naive implementation. However, this requires a simulation to factor in self-interference.

3.3.2 Tiling

A common used optimization is by dividing the work into works as described in 2.4.1 and this can be applied to stencil operations as well.

The cache size M_l for optimal tiling is lower bound by the area of memory required to compute a single row within a tile of size t.

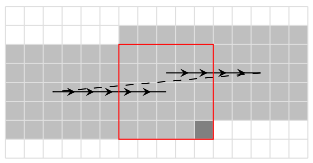
$$M_l \geq \left\lceil \frac{t + S_w}{L} \right\rceil S_h \tag{3.3}$$

The largest possible tiling size t is derived by inverting equation 3.3

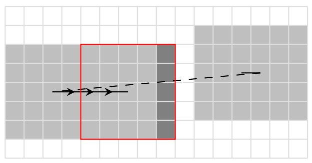
$$t \leq \frac{LM_l}{S_h} - S_w \tag{3.4}$$

In practise, equation 3.3 can be satisfied by adjusting t, we can have a lower upper bound on the amount of cache line fetches F:

$$F \leq \left\lceil \frac{I_w}{t} \right\rceil \left\lceil \frac{t + S_w}{L} \right\rceil \left\lceil \frac{I_h}{t} \right\rceil (t + S_h) \tag{3.5}$$



(a) Ideal cache size, only minimal amount of loading is required.



(b) Cache too small, data gets evicted before any potential reuse.

Figure 3.2: Stencil operations require enough cache to prevent chasing.

3.4 Matrix Multiplication

3.4.1 Naive

The matrix multiplication of $C = AB$ can be described as $c_{i,j} = \sum_k a_{i,k} b_{k,j}$ and can be implemented naively as in listing 4.

```
__global__ void matrix_naive(float *o, float *a, float *b, int depth, int width, int height) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int x = tid % width;
    int y = tid / width;

    if (tid >= width * height) {
        return;
    }

    float result = 0;
    for (int i = 0; i < depth; i++) {
        result += a[y * depth + i] * b[x + i * width];
    }
    o[tid] = result;

}
```

Listing 4: The naive CUDA C++ implementation of matrix multiplication

We run a similar simulation as in section 3.3.1 with the parameters described in table 3.2.

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

Table 3.2: The input parameters to generate the spatial-temporal diagrams for matrix multiplications.

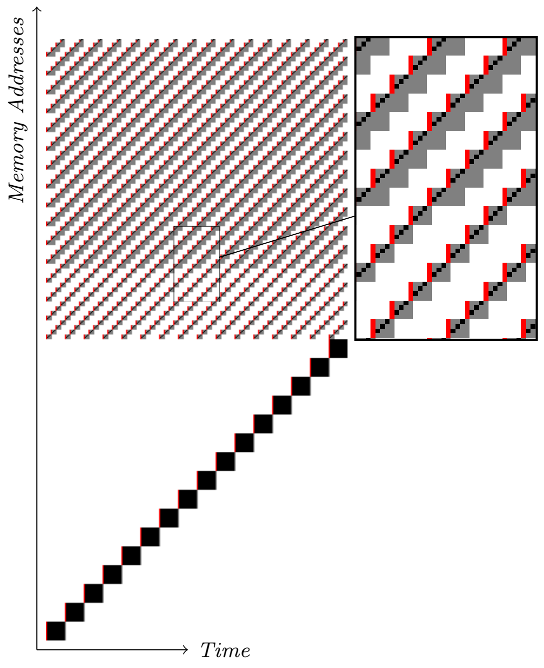


Figure 3.3: The spatial temporal diagram of matrix multiplication with linear ordering. See section 3.2.

Given the output matrix C of size $I_w \times I_h$ with input matrices A and B of size $I_w \times D$ and $D \times I_h$ respectively, data from B can be reused in the naive case only if we can both keep one row of B and one column of A loaded.

$$M_l > \left\lceil \frac{D}{L} \right\rceil + D \tag{3.6}$$

If equation 3.6 is satisfied, the amount of cacheloads is roughly

$$F \approx \left\lceil \frac{D}{L} \right\rceil I_h + D I_w I_h \tag{3.7}$$

In the case that equation 3.6 is not satisfied, there is no reuse of data possible, and the amount of cache line fetches is

$$F \approx \left(\left\lceil \frac{D}{L} \right\rceil + D \right) I_w I_h \tag{3.8}$$

To reuse date from both matrix A and B requires to keep the entirity of A cached until we work on the second row.

$$M_l > \left\lceil \frac{D}{L} \right\rceil + \left\lceil \frac{I_w D}{L} \right\rceil \tag{3.9}$$

This lowers the amount of cache loads significantly

$$F \approx \left\lceil \frac{D}{L} \right\rceil I_h + \left\lceil \frac{I_w}{L} \right\rceil D \tag{3.10}$$

3.4.2 Tiling

Tiling is a common optimization technique for matrix multiplication which divides the work both horizontally and vertically as described in section 2.4.1. Lam et al. [12] suggests a tiling size (Lam et al. use the term blocking factor) of $\sqrt{C/2}$ since at that point self interference is minimum.

Large scale matrix multiplication (such as machine learning) is often implemented as Single (Precision) General Matrix Multiplication (SGEMM). Nvidia's implementation of Basic Linear Algebra Subroutines (BLAS), named cuBLAS, is a closed sourced library that implements a highly optimized GPU based matrix multiplication. We can, however, look into implementation of GEMM in CUTLASS, an open source alternative to cuBLAS which performs within 80-100% of cuBLAS depending on platform and workload. CUTLASS uses a hierarchial structure for tiling putting the outer product first and foremost.

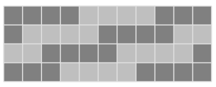
Chapter 4

Column Based Iteration

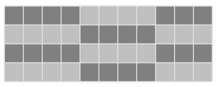
While tiling itself is a well-known optimization technique, we must consider the parts of why it works. Temporal locality is increased because the scope of the work is smaller, which is done by sacraficing a bit of spatial locality: we no longer load an entire row, only a part. However, this only explains the horizontal tiling, vertical tiling only reduces the scope as we work on a smaller tile but when considering the entire workload, it ultimately does not matter. It can even be argued that by tiling vertically, we fragment the the workload as temporal locality cannot be guaranteed between tiles.

4.1 Theory

In a sense, grouping columns is similar to striding clusters of data, except in the case when a row of work can't be perfectly strided. When the width of a multidimensional array is not a multiple of the stride, the loads will not align column wise (figure 4.1). To work around this, in the column approach we allow the last column to be narrower.



(a) Striding does not work well when the width of the task is a non-multiple of the striding factor.



(b) Allowing the last column to be flexible allows columns of thread groups to stay cohesive.

Figure 4.1: Striding does not work if the stride does not fit perfectly in the input width. Flexibility is required.

The index mapping $i \mapsto j$ to get the columning order consists out of four parts (figure 4.2):

- The starting offset of the column o.
- The column width w'.
- The index within a column i'.
- The position within the column (x, y).

First, we calculate the offset for the starting index of the column we need to map to:

$$o = \left\lfloor \frac{i}{I_h w} \right\rfloor w \tag{4.1}$$

Then, modify the width value such that the last column does not exceed the input width:

$$w' = \begin{cases} I_w - o & \text{if last column} \\ w & \text{otherwise} \end{cases} \tag{4.2}$$

And take i' as the index within a column:

$$i' = i \bmod I_h w' \tag{4.3}$$

Calculate the position (x, y) within the column:

$$x = i' \bmod w' \tag{4.4}$$

$$y = \left\lfloor \frac{i'}{w'} \right\rfloor \tag{4.5}$$

So that we can calculate j:

$$j = x + yI_w + o \tag{4.6}$$

On GPUs threads in threadblocks are batched in warps of 32 threads and these warps may be executed in an arbitrary order (section 2.1.3). As a result we might not be able to exploit the cache as much since column orders are highly dependent on thread order and on the higher level, CTAs may be assigned in a completely arbitrary way.

4.1.1 Zigzagging variation

When working in warps of 32-threads and having a non-multiple of 32 wide columns, the workload for a warp may be spread out due to the tail end of our task being put at the beginning of the next row. By reversing the order of every other row, we can keep accesses local to each other when they span multiple rows. We can modify equation 4.6:

$$j = \left(\begin{cases} x & y \text{ is uneven} \\ w' - x & y \text{ is even} \end{cases} \right) + yI_w + o \tag{4.7}$$

4.1.2 Higher dimensions

Since works only gets split horizontally, namely the one dimension that benefits from spatial locality, this algorithm can work on higher dimensions. All other dimensions can only exploit temporal locality and do not benefit from spatial locaity. We can therefore encapsulate an N-dimensional problem into 2-dimensional (horziontal and vertical) one, by mapping all but the first dimension to the vertical dimension.

4.2 Implementation

The remapping algorithm described in section 4.1 can be directly transcribed into C code (listing 5). Note that we use a minimum operator on line 35 instead of a condition the avoid branching.

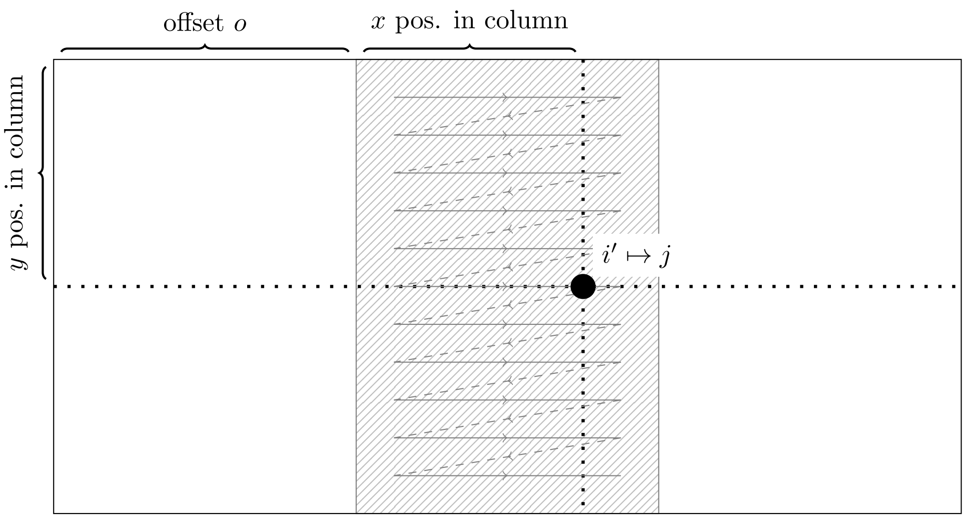


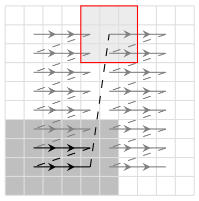
Figure 4.2: The anatomy of the column based index remapping.

For the implementation in Accelerate, we first define the type signature of a thread id mapping function (listing 6). It takes the size of the task (`ShapeR sh` and `Operands sh`), the original thread index (`Operands Int`) and produces the machinecode to compute the new thread index (`CodeGen arch (Operands Int)`). We then implement the algorithm as described in section 4.1 (listing 7) However, in Accelerate a single thread can work on multiple elements if the work is larger than the maximum number of threads. Part of the calculation can be moved out of the loop in the thread that handles the multiple elements to avoid unneeded work. Instead of computing the complete remap every new index, we can precompute part of the algorithm (listing 8). This precomputation is done by calling `precomp_column`. The function returns a lambda (listing 8, lines 16-34) that generates the machine code for the new index given the old index which can be called within the loop that enumerates over the elements.

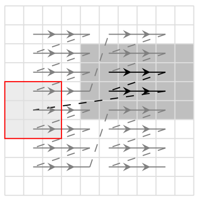
4.3 Stencil Operation

The naive stencil operations had problems of chasing the cache on sufficiently large input sizes which tiling does resolve (section 3.3.2). Consider what the tiling implies: we split the work on all axis, to improve locality. However, all tiling except horizontal is can be considered unnecessary because vertical tiling only causes to fragmentize the workload.

Let us first consider the single threaded case of column based iteration: by controlling the column width we can force the ideal scenario of the naive stencil implementation (figure 3.2a) to occur, similarly to tiling. The column based approach is similar to tiling, but also allows the ideal memory access pattern to continue accross tiles vertically. In GPUs this translates to less cohesive threads as threadblocks get assigned in a round-robin fashion to SMs, eliminating posibilities of L1 cache reuse.



(a) Grouping by column only incurs a heavy load every time a new column is started.



(b) Tiling also incurs a heavy load when starting a new row of tiles, and due to having more rows also has more column starts.

Figure 4.3: Visualisation of cache bottlenecks for both column based and tiling approaches. 🟥 Memory required by the next task. ⬛ Tasks that still have their memory in cache. ▨ Memory in cache. ⬜ Previous and upcoming tasks.

The size of the cache needed M_{size} can be approximated given the units per cache line L, column width c, stencil width s_w and height s_h, and the number of active threads t. The required memory is independent of the size of the input data.

$$M_{size} = u\,(c + s_w)\left(s_h + \left\lceil \frac{t}{c} \right\rceil\right)$$

Solving for c gives an unneedly complex solution, so instead we approximate M_{size} using the asymptote.

$$M_{size} = u\,(t + s_h\,(c + s_w))$$

Solving for column width c gives

$$c = \frac{M_{size} - u(s_h s_w + t)}{s_h u}$$

The amount of fetches is similar to equation 3.5, however, since we do not divide the work horizontally anymore, the number of cache line fetches is slightly lower:

$$F \leq \left\lceil \frac{I_w}{c} \right\rceil \left\lceil \frac{c + S_w}{L} \right\rceil (I_h + S_h) \tag{4.8}$$

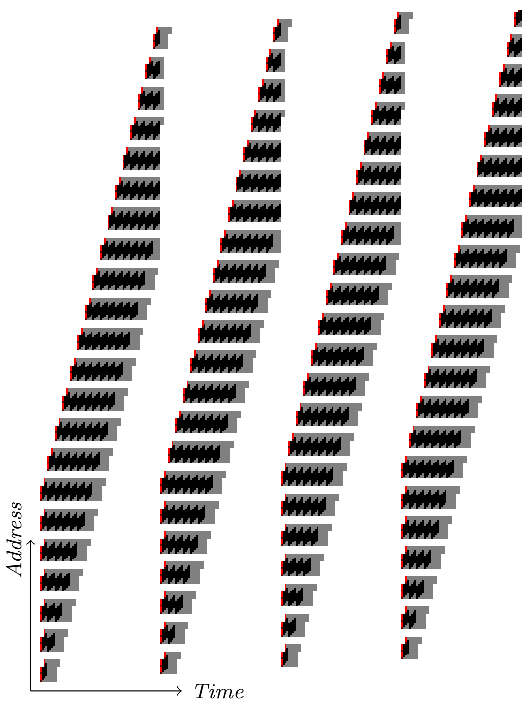


Figure 4.4: The spatial temporal diagram of a 7x7 stencil with a column of size 4 ordering. See section 3.2.

4.4 Matrix Multiplication

Matrix multiplication can benefit from a column based approach due to many vertical reads. Section 2.1.4 showed that continuous vertical reads are more expensive than continuous horizontal reads (streaming). Therefore, we want to keep data that is read vertically in cache for as long as possible. The naive implementation (section 3.4.1) showed that working on an entire row of outputs requires to load the whole $I_w \times D$ data By limiting the horizontal space we work on, we can control what stays in case, increasing reuse.

A matrix multiplication of $A \cdot B = C$ with C being of size $I_w \times I_h$, and A and B being size $I_w \times D$ and $D \times I_h$ respectively, we compute the lower bound on the required cache by seperating the horizontal and vertical reads.

The cachelines required M_l for optimal cache usuage of a single row within a column of the matrix multiplication can be computed from the size of the matrices multiplied and the column width

$$M_l = \left\lceil \frac{I_w}{L} \right\rceil + I_h \left\lceil \frac{c}{L} \right\rceil$$

Using cache size M_{size} instead of cachelines gives a simpler equation, albeit less accurate

$$M_{size} = uI_hc$$

However the simpler approximation allows us to solve for the column width c much more easier

$$c = \frac{M_{size}}{uI_h}$$

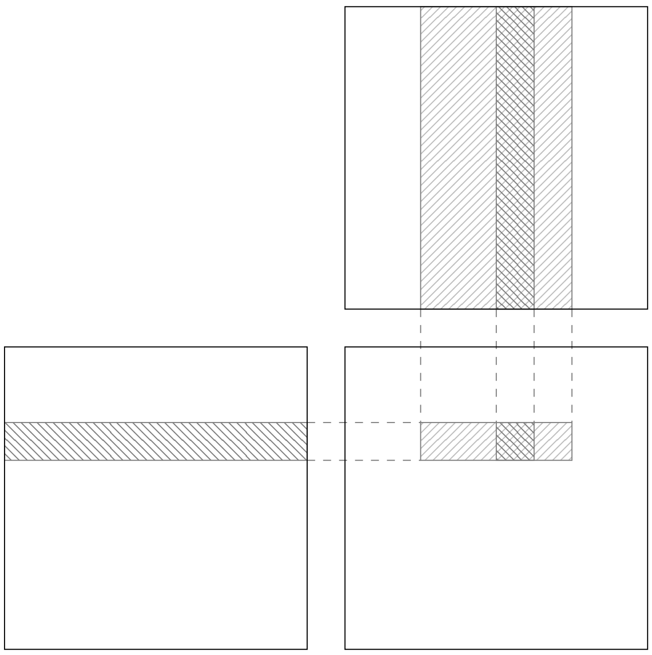


Figure 4.5: Calculating a single element in a matrix multiplication can exploit the cheaper memory accesses on the same cachelines.

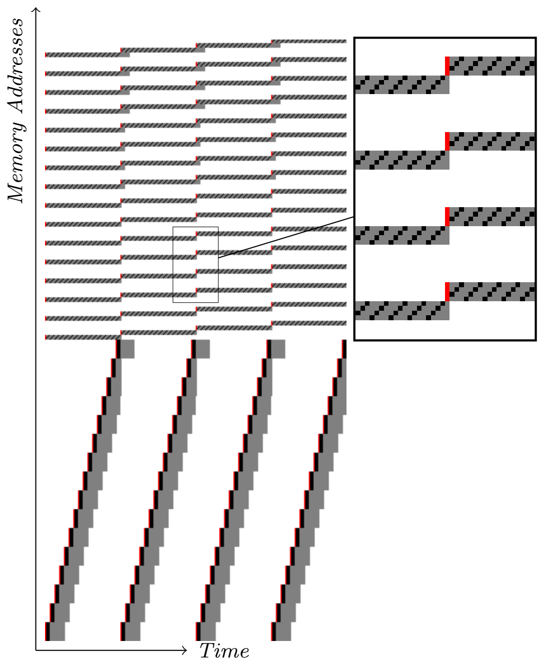


Figure 4.6: The spatial temporal diagram of a matrix multiplication with a column of size 4 ordering. See section 3.2.

```
__global__ void my_kernel(int width, [...]){
    // For a given thread ID we can compute the
    // new thread ID and it's x, y position:
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int x; // originally x = tid % width;
    int y; // originally y = tid / width;

    if (tid >= width * height) {
        return;
    }

    int col_size = height * col_w;

    // Column identifier
    int col_id  = tid / col_size;

    // Thread id within column
    int col_tid = tid % col_size;

    // Column offset
    int col_off = col_id * col_w;

    // True width of column
    //
    // int col_tw  = col_w;
    // if (col_id == width  / col_w) {
    //     col_tw  = width - col_id * col_w;
    // }
    //
    // A shorter version of  the above since col_tw is
    // always bounded by col_w. It removes the need of
    // an expensive  divide operation  and saves 2 PTX
    // instructions overal.

    int col_tw = min(width - col_id * col_w, col_w);

    x = (col_tid % col_tw) + col_off;
    y =  col_tid / col_tw;
    tid = x + (y * width);

    do_things_with(tid, x, y);
}
```

Listing 5: The CUDA C++ implementation column based remapping.

```
type ThreadMapping sh e arch
    =  ShapeR sh
    -> Operands sh
    -> Operands Int
    -> CodeGen arch (Operands Int)
```

Listing 6: The type signature of a thread remapping function which takes the input dimension sizes and thread index and produces a new thread index.

```haskell
column :: Operands Int -> ThreadMapping sh e arch
column column_width shr sh index = do
    -- input sizes
    input_width  <- widthOp shr sh
    input_size   <- sizeOp  shr sh
    input_height <- input_size |//| input_width

    column_size  <- column_width |*|  input_height
    column_count <- ((input_size |+| column_size) |-| (1::Int)) |//| column_size

    column_index      <- index |//| column_size
    column_last_index <- column_count |-| (1::Int)
    column_offset     <- column_index |*| column_width

    index' <- index |%| column_size

    column_current_width <- ifThenElse (
        TupRsingle $ SingleScalarType $ NumSingleType (numType @Int),
        column_index |==| column_last_index
      ) ( do
        input_width |-| (column_last_index |*| column_width)
      ) ( do
        return column_width
      )

    x_in_column <- index' |%|  column_current_width
    y_in_column <- index' |//| column_current_width
    result      <- x_in_column |+| (y_in_column |*| input_width) |+| column_offset

    return result
```

Listing 7: The thread remapping function for Accelerate implemented in Haskell.

```haskell
precomp_column :: Operands Int
               -> ShapeR sh
               -> Operands sh
               -> CodeGen arch (Operands Int -> CodeGen arch (Operands Int))
precomp_column column_width shr sh = do
    -- input sizes
    input_width  <- widthOp shr sh
    input_size   <- sizeOp  shr sh
    input_height <- input_size |//| input_width

    column_size  <- column_width |*|  input_height
    column_count <- ((input_size |+| column_size) |-| (1::Int)) |//| column_size

    column_last_index <- column_count |-| (1::Int)

    return \index -> do
        column_index  <- index |//| column_size
        column_offset <- column_index |*| column_width
        index' <- index |%| column_size

        column_current_width <- ifThenElse (
            TupRsingle $ SingleScalarType $ NumSingleType (numType @Int),
            column_index |==| column_last_index
        ) ( do
            input_width |-| (column_last_index |*| column_width)
        ) ( do
            return column_width
        )

        x_in_column <- index' |%|  column_current_width
        y_in_column <- index' |//| column_current_width
        result      <- x_in_column |+| (y_in_column |*| input_width) |+| column_offset

        return result
```

Listing 8: The column mapping from listing 7 with precomputing the fixed parameters.

Chapter 5

Results

The optimization described in chapter 4 has been implemented as both a native CUDA program and a modification to the CUDA Accelerate backend. Both implementations are benchmarking stencil operations and matrix multiplication on a Nvidia RTX 2080 Super GPU with a AMD Ryzen 5800x3D as CPU.

5.1 CUDA Benchmarks

Two simple problems are implemented in two CUDA programs: a 9×9 stencil operation on a 4096×4096 input, and a matrix multiplication on two 1024×1024 arrays of 32-bit floating point numbers. We execute the program naively, with the no remapping, and with our optimization: column based iteration. The kernel is analyzed with Nvidia profiler (`nvprof`) and Nvidia Nsight Compute CLI (`ncu`). Contrary to our expectations, the kernel execution time seems to be not correlated to our estimated column width at all, instead prefering lower multiple of 32 widths regardless of block size (figure 5.1).

bench	b32	b64	b128	b256	b512	b1024
naive	2.02	1.42	1.42	1.46	1.51	1.62
4	7.61	7.55	7.53	7.48	7.47	7.6
8	4.09	3.91	3.88	3.84	3.84	3.84
16	2.17	2.05	2.02	1.99	2	2.1
30	2.27	2.14	2.09	2.05	2.04	2.19
31	2.28	2.13	2.09	2.06	2.05	2.2
32	1.48	1.36	1.32	1.29	1.37	1.62
33	2.23	2.03	2	1.95	1.92	2.08
34	2.14	1.98	1.95	1.9	1.87	2.04
48	1.69	1.68	1.66	1.58	1.55	1.81
64	1.46	1.39	1.38	1.31	1.33	1.59
96	1.47	1.44	1.44	1.36	1.34	1.61
128	1.47	1.47	1.47	1.38	1.34	1.6
256	1.47	1.46	1.47	1.46	1.38	1.6
512	1.49	1.47	1.46	1.45	1.47	1.61
1024	1.49	1.46	1.46	1.45	1.47	1.68
2048	1.49	1.45	1.45	1.44	1.47	1.67

(a) The execution time in milliseconds of a 9×9 stenciling kernel on a 4096×4096 input array with various column sizes and block sizes (prefixed with b).

bench	b32	b64	b128	b256	b512	b1024
naive	2.36	2.31	2.31	2.31	2.34	2.38
4	6.55	6.55	6.55	6.57	6.58	6.58
8	3.67	3.66	3.66	3.66	3.65	3.75
16	2.35	2.34	2.34	2.32	2.29	2.29
30	2.43	2.44	2.44	2.43	2.35	2.37
31	2.51	2.49	2.48	2.53	2.55	2.89
32	1.83	1.85	1.85	1.85	1.74	1.74
33	3	3.01	3	3.07	3.07	3.46
34	2.88	2.89	2.89	2.91	2.8	2.89
48	2.22	2.23	2.22	2.21	2.1	2.09
64	1.81	1.81	1.81	1.81	1.72	1.68
96	2.03	1.97	1.99	1.8	1.75	1.68
128	2.13	2.15	2.3	1.77	1.71	1.66
256	2.26	2.24	2.3	2.31	1.74	1.68
512	2.32	2.28	2.31	2.31	2.34	1.82
1024	2.35	2.3	2.31	2.31	2.32	2.38
2048	2.36	2.31	2.3	2.31	2.32	2.38

(b) The execution time in milliseconds of matrix multiplications on two 1024×1024 input arrays with various column sizes and block sizes (prefixed with b).

Figure 5.1: The execution time of stenciling operations and matrix multiplication on a power-of-two input. Lower is better.

The factor of 32 seems to be independent of input size and are not affected by no power of 2 inputs (figure 5.2). Changing the data type to half precision (16-bit) floating points had effect on the overal improvements. Using 64-bit double precision floating points made the kernel compute bound instead of memory bound and therefore no tangible difference between the naive implementation and any column remapping is observed. This confirms our earlier assumption (section 4.1) that the 32 thread wide warps might play a role.

bench	b32	b64	b128	b256	b512	b1024
naive	2.04	1.36	1.37	1.41	1.46	1.58
4	4.71	4.45	4.37	4.29	4.27	4.81
8	3.71	3.32	3.26	3.17	3.16	3.26
16	2.45	2.23	2.15	2.09	2.08	2.23
30	2.43	2.24	2.16	2.07	2.04	2.22
31	2.42	2.23	2.14	2.07	2.04	2.22
32	1.46	1.31	1.27	1.25	1.31	1.63
33	2.3	2.16	2.06	2	1.98	2.17
34	2.24	2.09	2.02	1.96	1.91	2.13
48	1.66	1.61	1.6	1.53	1.49	1.8
64	1.46	1.34	1.33	1.28	1.31	1.62
96	1.47	1.38	1.38	1.3	1.32	1.63
128	1.47	1.39	1.4	1.32	1.31	1.61
256	1.45	1.39	1.38	1.39	1.34	1.62
512	1.46	1.38	1.38	1.38	1.41	1.63
1024	1.47	1.38	1.38	1.38	1.41	1.67
2048	1.51	1.38	1.37	1.37	1.41	1.67

Figure 5.2: The execution time in milliseconds of a 9×9 stenciling kernel on a 4037×4037 input array with various column sizes and block sizes (prefixed with b). Even on non power of two's, column widths of multiples of 32 produce more optimal results.

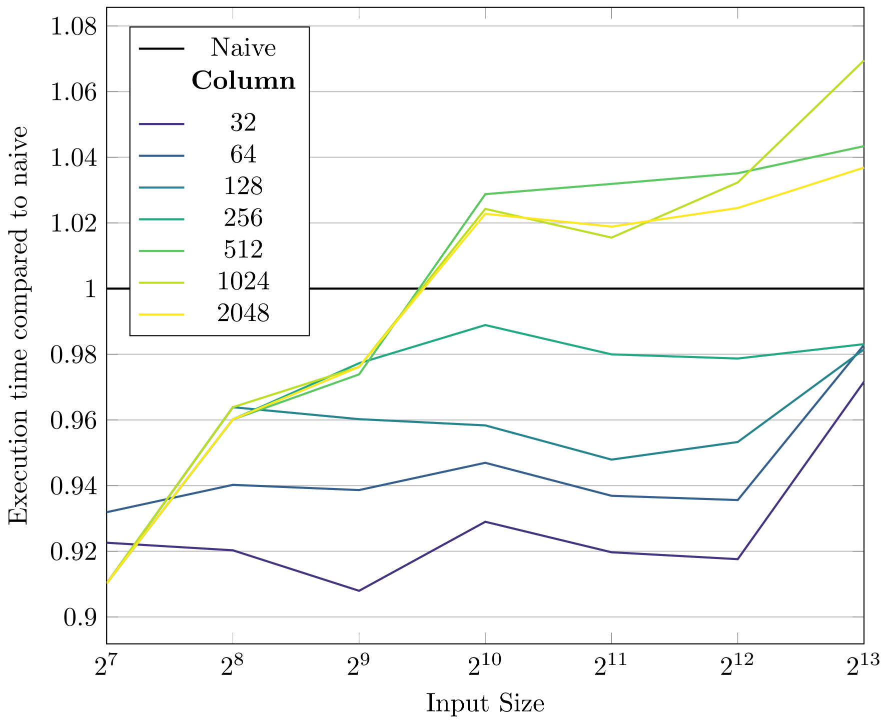


Figure 5.3: The relative kernel execution time of various column widths compared to the naive implementation for a 9×9 stencil operation on an $N \times N$ matrix. Lower is better.

Execution time for matrix multiplication with CBI compared to naive only appears to improve after a certain input size (2^8) and has diminishing returns after (2^10), see figure 5.4. A possible explanation is that after this points caches are completely saturated and fully utilized.

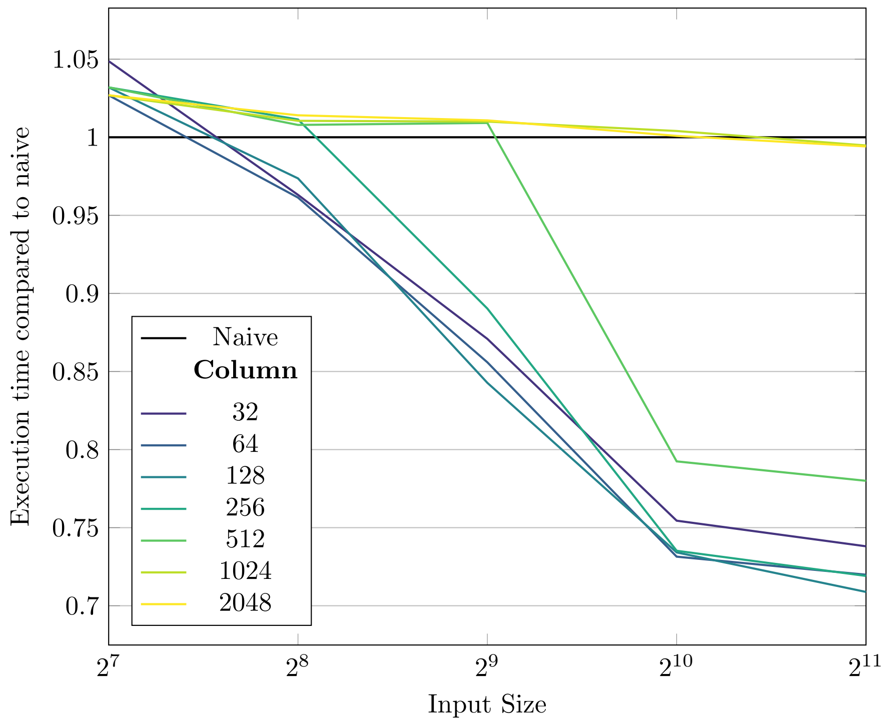


Figure 5.4: The relative kernel execution time of various column widths compared to the naive implementation for the matrix multiplication of two $N \times N$ matrices. Lower is better.

5.2 Hardware Utilization

On the 1024×1024 matrix multiplication, running a 32 wide column patterns resulted in an improvement to the L1 cache hit rate from 19.49% to 77.40%. This results in a drop in transferred data from L2 to L1 cache from 4.02 GB to 1.13 GB. The L2 hit rate dropped from 98.52% in the naive case, to 95.73% with CBI.

In both the naive and column pattern, Nvidia Nsight Compute reported that the scheduler is stalling due to the L1 instruction queue for local and global memory operation being full. However, the column pattern fix stalls on long scoreboard, meaning instruction could not be executed out of order due to a depedency on an unfinished L1TEX (local, global, surface, tex) operation.

5.3 Accelerate Benchmarks

The addition of CBI to Accelerate is also benchmarked, and because this modification applies only to Accelerate, it is quite easy to benchmark numerous configurations. We benchmark stencils of size 3×3 up to 9×9 and matrix multiplication on square power-of-two input sizes between 256×256 up to 4096×4096. The time is the average over multiple runs and the first run, which includes shader compilation, is discarded. The relative execution time of these benchmarks, calculated as $\frac{t_{optimized}}{t_{naive}}$ are compiled in figure 5.5. The results vary greatly and most noteable is the significantly longer execution time for `stencil/blur3x3/4096` (2.6 ms vs 0.6 ms). Contrarily, `matrix/4096` runs significantly fastr aftr CBI. A possible explanation for the former might be that the overhead for CBI within accelerate is excesively large compared to a naive implementation negating all performance gains. Investigataing this is difficult because Accelerate does not emit the right debug information for us to sensibly link the generate assembly back to the modified Accelerate backend. The overal trend of improvements does reflect the results from the earlier presented C++ implementation: stencils perform worse on larger inputs, while matrix multiplication performs better on larger inputs.

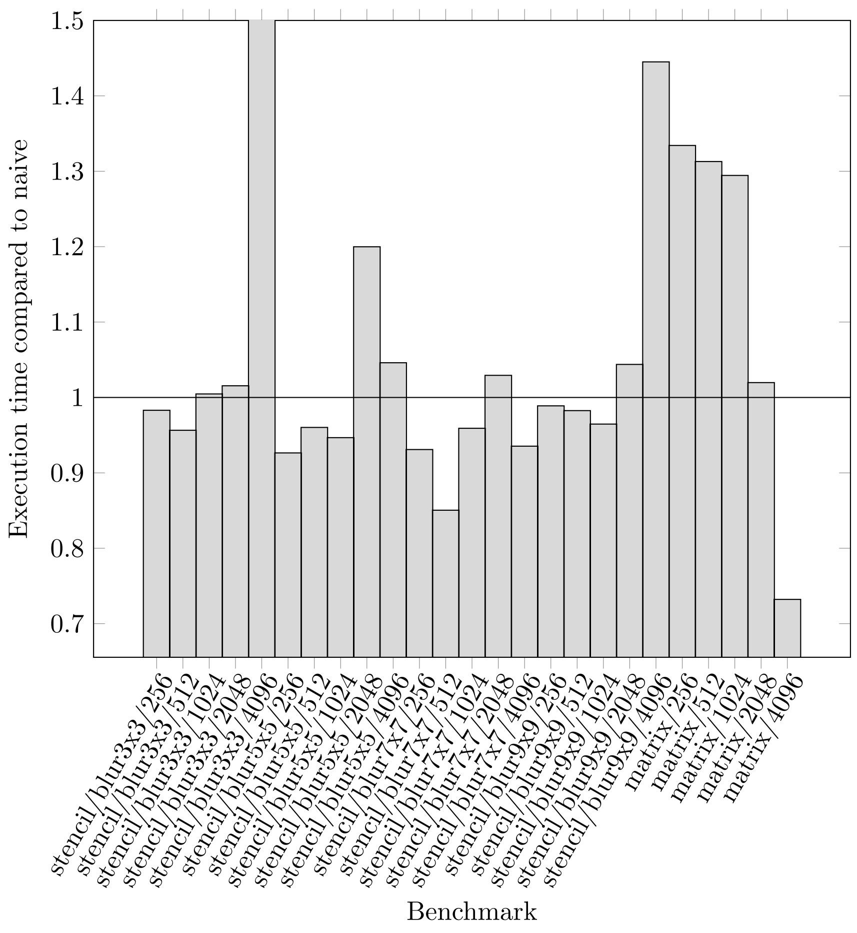


Figure 5.5: The relative kernel execution time of CBI with column width 32 compared to the naive implementation for various benchmarks. Each column is a different benchmark with the input size of $N \times N$ as the suffix. Lower is better.

Chapter 6

Conclusion

Column based iteration (CBI) is a variation on tiling where we only split work on one axis in a multidimensional workload. It improves the performance of naive stencil (up to 8%) and matrix multiplication (up to 28%) operations.

We improve hitrates of the L1 cache by a significant amount (from 19% to 77%). Usually, a column width of 32, 64, or 128 is ideal as this allows us to more properly utilize the warp (32 threads), cache lines (128 bytes = 32 × 4 bytes), and scheduler (4 warps). Wheras CBI improves performance over naive matrix multiplication, specializd algorithms (as implemnted in cuBLAS/CUTLASS) are faster than our thread rescheduling. Unfortunately, the performance increase within Accelerate is less consistent, and sometimes performs significantly worse (execution times of 50% or more), than our C++ and CUDA implementation which saw a more consistent trend.

6.1 Future Work

While column based iteration does improve performance there are still areas for possible improvements. Currently, CBI has only been tested and benchmarked on simple testcases, and it is unknown how it will behave with multiple programs running on the same GPU. Seeing if similar improvements hold when other programs also fight for cache resources would be interesting as this would imply these improvements can also be applied to more consumer oriented programs such as video games and image processing. CBI also may work with other structured workloads that are memory bound. Furthermore, after applying CBI we still observe cache related bottlenecks: stalls due instructions not being able to be executed out of order due to L1 data depedencies. An idea might be to increasing the threadblock size to hide the latency, but this will put too much pressure on the L1 cache. This can also include applying the optimization to generalized matrix multiplication. Furthermore, finding a better method for the spatial-temporal analysis that includes the non-linearity of parallel execution may allow us to gain more insight into faster thread patterns.

Another vector of improvement is seeing if this can be applied to optimizing CPU code as well. However, a problem is that often more complex patterns may impede the compiler's ability to vectorize the code into SSE/AVX instructions and also may impede with hardware based prefetching.

Bibliography

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

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

[5] NVIDIA. Nvidia turing gpu architecture. 2018.

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

[7] 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.

[8] 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.

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

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

[12] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

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

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

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

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

[17] 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.

[18] 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.

[19] Gerald Roth, Gerald Roth, John Mellor-crummey, John Mellor-crummey, Ken Kennedy, Ken Kennedy, R. Gregg Brickner, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*, pages 1–20. ACM Press, 1997.