UNIVERSITY OF COPENHAGEN
DEPARTMENT OF MATHEMATICAL SCIENCES

Master's thesis

Kristoffer August Kortbæk and Rune Ejnar Bang Lejbølle

Utilizing Tensor Cores in Futhark

A case study in programming new heterogeneous hardware

20-12-2024

Advisor: Cosmin Eugen Oancea

Abstract

Modern hardware has become more heterogeneous, and with the AI boom, specialized hardware for especially performing matrix multiplication has become readily available. In NVIDIA graphical processing units (GPUs), Tensor Cores allow for efficient execution of matrix multiplication routines that can significantly speed up AI and deep learning operations, as well as other programs containing matrix multiplication.

However, programming for the Tensor Cores is not straightforward, and often requires adapting code to restrictions and performance guidelines unique to this hardware. The hardware is made more accessible through application specific libraries such as cuBLAS and cuDNN, but for more general use specialized CUDA or PTX code targeting the Tensor Cores must be written. In order to ease the use of Tensor Cores in general, we propose to integrate the use of Tensor Cores into Futhark, a data parallel array language and highly optimizing compiler that generates efficient GPU code.

Our main contribution is to allow Futhark programs with matrix multiplication in an intragroup kernel to use the Tensor Cores. We evaluate the Futhark compiler output against handwritten CUDA programs using the Tensor Cores and the stock unmodified compiler on benchmarks such as the matrix multiplication routine from LU-Decomposition in Rodinia [1], FlashAttention [2] like programs, and other matrix multiplication programs. The results show that our modified compiler is still considerably slower than handwritten implementations, but compared to the stock Futhark compiler we see speedups between 1.9× and 60×.

Table of content

1 Introduction **5**
- 1.1 Related work . . . 6
- 1.2 Terminology . . . 7

2 Background **8**
- 2.1 GPUs . . . 8
  - 2.1.1 Hardware . . . 9
  - 2.1.2 Thread hierarchy . . . 10
  - 2.1.3 Memory hierarchy . . . 11
  - 2.1.4 Tensor Cores . . . 12
  - 2.1.5 CUDA . . . 13
- 2.2 GPU Optimizations . . . 17
  - 2.2.1 Coalesced Global Memory Access . . . 17
  - 2.2.2 Avoiding Shared Memory Bank Conflicts . . . 19
  - 2.2.3 Vectorized Loads and Stores . . . 25
  - 2.2.4 Pipelining . . . 27
  - 2.2.5 Block-(Warp)-Register Tiling . . . 30
  - 2.2.6 Tuning for Occupancy . . . 33
- 2.3 Futhark . . . 34
  - 2.3.1 The language . . . 34
  - 2.3.2 Writing Futhark Programs . . . 36
  - 2.3.3 A quick compiler overview . . . 37

3 Prototypes **38**
- 3.1 Pure CUDA . . . 38
- 3.2 Cutlass/CuTe . . . 39
- 3.3 Evaluation . . . 42

4 Compiler Modifications **45**
- 4.1 Code Generation Strategy . . . 45
- 4.2 Tensor Core Header Library . . . 47
  - 4.2.1 CuTe Copy from Global to Shared . . . 49
  - 4.2.2 Matrix Multiplication . . . 50
  - 4.2.3 Cute Copy Registers Shared . . . 51
- 4.3 High Level IR transformations . . . 51
  - 4.3.1 Copy global to shared . . . 56
  - 4.3.2 Setting the Block Size for Matrix Multiplication . . . 58
  - 4.3.3 Tensor Core Matrix Multiplication . . . 59
  - 4.3.4 Storing the Result in shared memory . . . 60
- 4.4 Transformations on IR With Memory . . . 61
  - 4.4.1 Modifying the Default Allocation Space . . . 62
  - 4.4.2 Reduce memory copies . . . 63
- 4.5 Using the Modified Compiler . . . 65

5 Experimental Evaluation **66**
5.1 Testing Methodology . . . 66
5.2 Benchmark Methodology and Hardware . . . 66
5.3 Benchmark Limitations . . . 67
5.4 Benchmark Results . . . 67
5.4.1 Batched matrix multiplication . . . 67
5.4.2 Custom attention like . . . 68
5.4.3 Attention Like Program . . . 71
5.4.4 Rodinia LUD matrix multiplication . . . 72
5.4.5 Large matrix multiplication . . . 74

6 Future Work **76**
6.1 Accumulation in registers . . . 76
6.2 Double-buffering or pipelining . . . 79
6.3 Interoperability with incremental flattening and autotuning . . . 79
6.4 Better integration of swizzling . . . 80
6.5 Avoiding bank conflicts when writing matrix multiplication results (C arrays) . . . 80
6.6 Generalizing the Solution . . . 81
6.6.1 Supporting other matrix multiplication-like patterns . . . 82
6.6.2 Supporting more data types . . . 82
6.6.3 Supporting and optimizing for more architectures . . . 82

7 Conclusion **83**

References **85**

A Source Code **85**
A.1 `preludeTensorCores.cu` . . . 85
A.2 `TensorCores.hs` . . . 90
A.3 `ExtractTensorCores.hs` . . . 90
A.4 `TensorCoreMemFixup.hs` . . . 106
A.5 `Utils.hs` . . . 112

1 Introduction

Historically, the clock rate at which instructions execute has increased exponentially in the period 1990-2004[3]. Programs designed around a uni-processor architecture would tremendously benefit from this, with program performance scaling with the clock rate. Often this has been contributed to a reinterpretation of Moore's Law:

Compute power doubles every 19-24 months, while the cost effectiveness keeps up.

Cost effectiveness keeping up means that the computational power increase essentially comes for free at no additional price increase. For some time, this meant that single-core and single-threaded software could at no additional programming effort benefit from the rapid increase in transistor count and hardware innovations. The above statement is however not quite the original observation made by Gordon E. Moore. The original observation more closely states that:

The number of transistors on an integrated circuit doubles every 19-24 months.

The additional transistor count predicted by Moore's Law was used to increase the clock rate of single-CPU systems in 1990-2004. This is the so called *killer-micro* effect[3]. Figure 1 shows on a log scale the microprocessor trend in the period between 1975-2010. From around 1990 to 2005, the increase in transistor count and single threaded clock frequency and performance are closely correlated. The period around 2005 turns out to be a paradigm shift. The transistor count still follows an exponential trend, but the clock frequency can no long keep up, and comes to a halt. To keep Moore's Law alive, future architecture must use the additional transistor budget for some form of massive parallelism [3]. Figure 1 shows how CPU manufacturers implement this by increasing the core core count of their CPU architectures from around 2005.

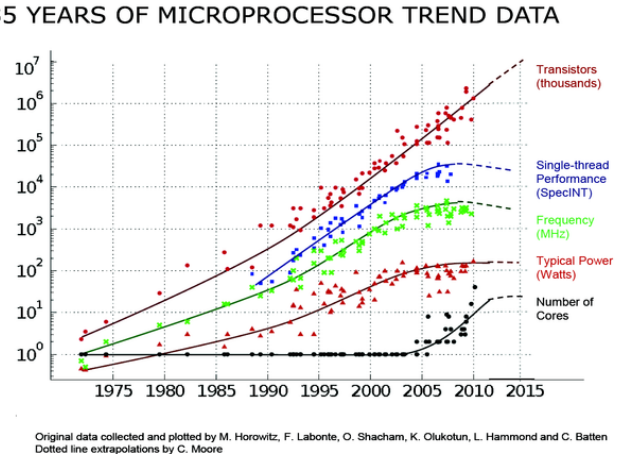

Figure 1: Microprocessor hardware trends. Figure taken from [4]

The shift from uni-core processor with ever increasing clock rates to a multi-core design was prompted by examples such as the *power wall* and *memory wall*:

- **Power wall**: The processor power is proportional to the cube of the frequency. At some point, further increasing processor frequency therefore becomes impractical from a temperature and power perspective. Increasing the frequency by 2× requires a power increase about 8×, but increasing the processor count by 2× only increases the power by 2×.

- **Memory wall**: Historically, the increase in processor speed has been around 50 %, while the DRAM speed has only increased by 7 % each year. This made an ever larger gap between the processor speed and waiting for data. Multi-core systems has rendered the memory wall obsolete, but at the cost of a bandwidth problem [3].

Both the power- and memory wall are good motivations for moving to parallel systems. GPUs are a canonical example of a modern massively parallel system. They provide thousands of simple, small cores, and tens to hundreds of thousands of threads [3], but often times with a lower processor frequency.

Other ways of using the increased transistor count provided by Moore's Law is in the realm of application specific hardware. Examples include the Tensor Cores on modern Nvidia GPUs. These are specialized cores for computing matrix multiplication. Other examples are custom deep learning hardware such as Google's TPUs, and Apple's Neural Engine. In all cases, the direction of modern hardware seems to be *heterogeneous systems* that subsume several level of heterogeneous hardware. NVIDIA graphical processing units (GPUs) such as the A100 and H100, also called *Tensor Core GPU*s by NVIDIA, provide at least two levels of heterogeneous hardware with the Tensor Cores, in that: (1) the general-purpose (graphical processing) unit (GPU) is in itself specialized hardware for accelerating massively parallel computation, but also (2) contains specialized hardware, namely the Tensor Cores, aimed at accelerating matrix multiplication, which is an operation at the core of training and inference of neural networks, and other compute intensive applications. These new computing systems also carry with them a multi-level memory hierarchy that has to be used efficiently to unlock the performance that the new hardware provides. Open source libraries such as PyTorch and TensorFlow provide a domain specific language for operating on multi-dimensional matrices known as tensors that take advantage of the new hardware. Similarly, application specific libraries such as cuBLAS and cuDNN also provide interfaces to utilize the hardware in their respective domains. However, in general use, the increasingly heterogeneous hardware has tended to also increase the heterogeneity of code used to program it, with languages or programming models specialized for specific hardware, such as CUDA C++ for programming NVIDIA GPUs. This brings a need for new programming models that make modern heterogeneous hardware more generally accessible and easier to program outside of the realm of deep learning and AI.

In this work we investigate if Futhark, a data parallel array language, can utilize some of this new hardware by modifying the compiler to use the NVIDIA Tensor Cores. Our focus is on prototyping the compiler to use the Tensor Cores for matrix multiplication computations, and we do not try to change the general compiler infrastructure to efficiently use the Tensor Cores. In doing so, we hope to investigate the constraints and problems related with utilizing this new specialized hardware in an existing programming language.

1.1 Related work

We start this section with a brief overview of the evolution of compilers and scheduling languages. This has relevance in the way new heterogeneous hardware is programmed, and we end up discussing existing compilers and languages for programming the NVIDIA Tensor Cores.

Before the 1990s, compilers were generally seen as black boxes only controlled with optimization flags and a small set of directives for analysis and optimization [5]. In the 1990s, compiler research is mostly focused on optimizing locality in caches. It became obvious that the best sequence

of program transformations is heavily dependent on the architecture and input data. To guide architecture specific optimization, cache models were developed. As these models got sufficiently complicated, another idea arouse. The program could be run on different target architectures, and it can be tuned or adjusted for the best performance on that platform. Automatically Tuned Linear Algebra Software (ATLAS)[6] introduced the notion of *autotuning* for linear algebra routines. Other approaches to tune computation kernels to other domains was developed in the 1990s and 2000s. Simultaneously, *iterative compilation* compiler infrastructure was developed to explore alternative sequences of transformations, execute the code on hardware and use this to decide on the best performing version. The crucial point is that is was still the compilers responsibility to decide on what to explore. Following this, systems were introduced that allowed the programmer to express different semantically equivalent program variants, and then leave it to auto tuning to pick the fastest. Other systems enabled the expert programmer to specify a sequence of transformation to apply to the code with the specification written either through annotations in the code or by a separate transformation recipe. Halide [7], a domain specific language (DSL) for image processing, popularized this idea where a high performance implementation is derived by a two stage approach. First, a simple high level DSL is used by the domain expert to express the program. Next, in a separate language, called a *scheduling language*, a transformation recipe is written by a compiler expert.

Just as Halide optimizes image processing pipelines, future scheduling languages might also be used to optimize the data movement needed for heterogeneous hardware such as the Tensor Cores. Graphene [8] provides scheduling for thread mapping to prepare data for computation on the Tensor Cores. In similar spirit, CuTe [1], a cuda C++ header only library, provides template abstractions for describing multidimensional layouts of threads and data directly in the C++ type system. Although CuTe is not a scheduling language, it still decouples the mapping of threads and data from the GPU algorithm implementation of linear algebra routines. With CuTe, there is still a need for tuning the data layout and data movement depending on the architecture for best performance. Specific types of data movement operations are also hardware specific, and CuTe also decouples this loosely from the algorithm implementation.

Scheduling languages may also provide a way of specifying hardware specific features. This could be the specification of matrix multiplication operations using the Tensor Cores for the when available in the target hardware architectures. While not having a scheduling language, Triton [9] is a DSL and IR, mostly focused on accelerating deep learning computations, and it also performs block tiling. Triton will emit Tensor Core operations where possible [10]. The Futhark compiler also already has a pass that performs *block and register tiling* [11] whenever it can be determined that there is an opportunity of optimizing temporal locality. This in turn optimizes matrix multiplication, but it does not use the Tensor Cores.

Our contributions to the Futhark compiler are mostly similar to Triton in that we emit Tensor Core operations where possible. The compiler uses CuTe to specify the architecture specific type of Tensor Core operation as well as data movement and layouts.

1.2 Terminology

We will frequently use the below terms and notation. The terms will be explained whenever they are first used, but they are also stated here for quick reference.

[1] Cutlass and Cute is hosted at: `https://github.com/NVIDIA/cutlass`

- ***Intragroup kernel*** refers to the collection of threads in a CUDA block that collectively performs a computation in parallel or a `SegOp` at the level of threads in a CUDA block.
- $\mathcal{A}$ an uppercase bold calligraphic letter corresponds to an entire array.
- $\mathcal{A}_{mk}$ corresponds to a slice of an array $\mathcal{A}$ with size $m \times k$.
- `f16` and `f32` are 16-bit half precision and 32-bit single precision floating point numbers respectively.

2 Background

In this section we set out to give an overview of the relevant background information needed to provide support for Tensor Cores in the Futhark compiler. This starts by a discussion of NVIDIA GPUs and how they differentiate from CPUs. We briefly cover the GPU hardware and memory hierarchy and then go into how GPUs can be programmed in CUDA `C++`. Next, we show common GPU optimizations that are especially important for fast matrix multiplication using Tensor Cores, but are generally applicable for a variety of GPU programs.

We finish off with an introduction to the Futhark programming language and compiler. This introduces programs that are relevant for discussion about how they might take advantage of Tensor Cores and perhaps use the GPU optimizations that were introduced.

2.1 GPUs

The Graphics Processing Unit (GPU) is a massively parallel computation unit very different from a Central Processing Unit (CPU). Where the CPU has higher clock rates and excels at processing a sequence of instructions, the GPU provides thousands of simple "cores" that require the programmer to use tens to hundreds of thousands of threads, and excels in parallel processing. This lets the GPU achieve higher instruction throughput and bandwidth than a CPU at a similar price and power [12].

Figure 2 shows how the transistor resources are used between CPUs and GPUs. For the CPU, a large chunk of the transistors are spend on caches and control units. In comparison, fewer transistors are used for the arithmetic logical unit (ALU) that performs computation. A GPU will on the other hand allocate a much larger portion of the transistor count to raw computation (the ALU).

This lets the GPU hide the additional memory latency caused by having less cache, by overlapping memory transfers with computation. In other words, whenever a group threads stall waiting for data, it is more than likely that another group of threads can do some computation in the meantime. This in comparison to the CPU which instead relies on various cache layers to ensure memory can be accessed faster.

It has to be mentioned that not all GPUs are the same. Some of the big GPU manufacturers are AMD, NVIDIA, and Apple, and they each provide their own type of GPU with a different programming model, although these models typically overlap in some aspects. Therefore, in order to dive deeper into the hardware and how it can be utilized we have to scope the types of GPUs. For this work, we are fousing on NVIDA GPUs, and we will use the NVIDIA terminology to describe the GPU hardware and programming model in more detail.

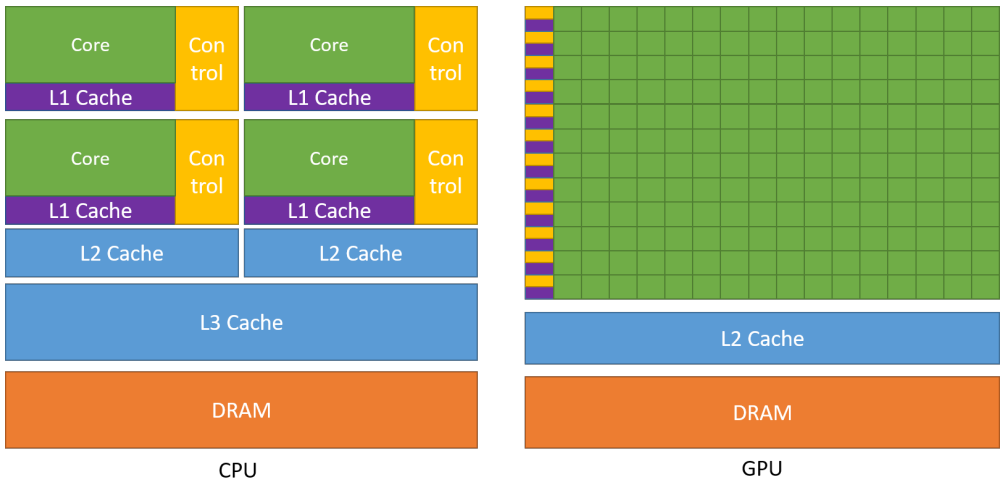

Figure 2: CPU architecture (left) and GPU architecture (right). The illustration was created by NVIDIA[12].

2.1.1 Hardware

Computation on NVIDIA GPUs is performed by a number of streaming multiprocessors (SMs), shown for the NVIDIA A100 in Figure 3. An A100 has 108 such SMs, which share a common, on-chip L2-cache, and have access to the same off-chip *device* memory. Each SM has its own L1-cache, and can be further subdivided into 4 units that can execute different instructions in parallel. Each of these units further use SIMD (Same Instruction - Multiple Data) execution to achieve even more parallelism. As shown in the schematic, each unit also has acess to a Tensor Core, which is what we make use of in this work.

Figure 3: Streaming multiprocessor for the NVIDIA A100. The illustration was created by NVIDIA[13].

2.1.2 Thread hierarchy

The programming model for Nvidia GPUs organizes its threads in a hierarchy. This organization defines how individual threads can communicate with one another, and what hardware resources they share.

At the lowest level of the hierarchy are the individual *threads*. This is the smallest unit of execution. Each thread has its own registers and program counter[2].

[2]for NVIDIA Volta and newer architectures

Threads are grouped together in groups of 32 to form a *warp*. Warps execute in lockstep, which means all active threads in a warp execute a single common instruction in parallel. However, since each thread has its own registers and program counter, threads in a warp can diverge in their execution path if a conditional evaluates differently across the threads in the warp. If this occurs, each taken branch of the conditional will be executed in sequence by the whole warp, with threads not on each path disabled. This means that for maximum performance, conditionals should evaluate equally across each warp. A number of primitives exist allowing more efficient communication and cooperation between threads in the same warp than is possible across threads of different warps. One such primitive is the family of `mma` PTX instructions, which will be described in more detail below.

Warps of threads are grouped together in *blocks*. The number of threads per block can be decided by the programmer, up to a maximum of 1024 threads for current architectures. Threads within a block can access a common chunk of *shared* memory, and use this for communication and cooperation. Additionally, threads within a block can synchronize their execution to avoid race conditions, using the `__syncthreads()` primitive.

Blocks of threads are organized in a *grid*. A grid of thread blocks all execute the same *kernel*, but individual thread blocks cannot communicate with one another, and must be able to run in any order. A kernel is a special type of function that runs asynchronously on the GPU.

NVIDIA Compute Capability 9.0, introduces an additional, optional, level of organization, such that thread blocks can be grouped in thread block *clusters*, which are then organized in a grid. We have not used this in our work. [12][14]

2.1.3 Memory hierarchy

Threads in a CUDA kernel cannot access ordinary host memory (RAM), and must instead use memory resident on the GPU. There are a number of different spaces of memory on the GPU. These spaces are organized in a hierarchy closely related to the hierarchy of threads. Each individual thread has exclusive access to its own registers and its own *local* memory. All threads in a thread block can access the same chunk of *shared* memory. Finally, all threads across all blocks can access the same global memory. All threads in all blocks can also access some additional read-only memory spaces called *constant*, *texture* and *surface* memory, which are optimized for special access patterns often used in graphics. We have not used these read-only memory spaces, and will therefore not address them further here.

Data coming from *host* code, i.e. code running on the CPU must initially be moved to global memory in order to be used on the GPU. When a variable e.g. an array is declared in *device* code, i.e. code running on the GPU, it can can be explicitly declared to be in global or shared memory, using just the `__device__` or `__shared__` specifiers, respectively. If neither is used, the variable will typically reside in registers. However, for various reasons this is not always possible. In such cases the variable will instead reside in local memory. Typical reasons for this is register spilling, variables simply being too large to fit in registers, or arrays that are not always accessed by static indexes. For example, an array indexed by a loop variable in a `for` loop which is not unrolled, would have to reside in local memory.

Other than the registers, shared memory offers the fastest access times, since it resides on-chip. However, it is fairly limited in size. In fact, for all architectures so far, the maximum total size of the register file for a thread block has been larger than the maximum amount of shared memory available to a thread block. Shared memory can be seen as a form of programmable cache, and in fact shares its hardware resources with the L1 cache[2].

Both local and global memory resides in off-chip *device* memory, and is thus much slower than shared memory, but offers much more abundant space. To make up for the slower speed, both local and global memory are cached in on-chip memory. This caching consists of an L2 cache shared by all SMs, and an additional L1-cache for each individual SM. [12]

2.1.4 Tensor Cores

GPUs have been used to accelerate a variety of computational problems. One of the largest use cases has been to train large deep neural networks (DNN). A key operation when either training or running inference on a DNN is matrix multiplication. This is such a common operation that modern NVIDIA GPUs are equipped with a new hardware component called the Tensor Core that, for matrices of specific, relatively small, sizes, can perform in-hardware matrix multiplication and accumulate (MMA) operations of the form:

$$\mathcal{D} = \mathcal{A}\mathcal{B} + \mathcal{C} \tag{1}$$

Using tiling as explained in subsubsection 2.2.5, these basic operations can be used repeatedly to implement MMA for larger matrices. The operations can also be used to implement other operations containing matrix multiplication, such as generalized matrix multiplication (gemm) of the form:

$$\mathcal{D} = \alpha\mathcal{A}\mathcal{B} + \beta\mathcal{C} \tag{2}$$

The MMA operation is often performed with a mixed precision of 16-bit (`f16`) and 32-bit (`f32`) floating point numbers. I.e. when performing the matrix multiplication $\mathcal{C} = \mathcal{A}\mathcal{B}$ with matrices of size $M \times K$ and $K \times N$ respectively, the multiplication $a_{ik}b_{kj}$ is done using `f16` and the accumulation

$$c_{mn} = \sum_{k=0}^{K} a_{mk}b_{kn}$$

is done with `f32` precision. Since the accumulation step is done with `f32`, the precision loss is minimized.

The first NVIDIA architecture to introduce Tensor Cores was the Volta series of GPUs. The architecture has 8 Tensor Cores per SM , and each can perform 64 `f16`/`f32` mixed precision fused multiply and add (FMA) per clock[13]. The more recent Ampere and Hopper architectures only have 4 Tensor cores per SM, but each Tensor Core can perform more FMAs per clock. Depending on the NVIDIA architecture, the tensor cores are used slightly differently:
For Volta, 4 groups of 8 threads, called quad pairs, each perform their own MMA operation of size $M \times N \times K = 8 \times 8 \times 4$.
In Ampere, the entire warp instead collaborates to do the MMA. The size computed by the warp is $16 \times 8 \times 16$ as a single warp-wide result. The programmer must, however, still coordinate the work to be done between warps in a block for a block level result.
For Hopper, an entire warpgroup of four warps can perform the MMA operation.

Besides producing different output tiles, the Tensor Cores also expect the input matrices to be in a very specific format depending on the architecture. In general, the input matrices must be in registers, but Hopper also allows them to be in shared memory. The details of this is listed in [14]. To combat the issue that every architecture is programmed differently, NVIDIA provides helper libraries to abstract these details away. This is further discussed in section 2.1.5.

With proper tiling strategy and optimization, Tensor Cores can be used to speed up matrix multiplication, as indicated by Table 1. This speedup is even greater if one takes into account the ability of the Tensor Cores to effeciently do mixed-precision computations, using e.g. `f16` for multiplication and `f32` for accumulation, with the same throughput of full `f16` computation. The general and simple matrix multiplication algorithm should be compute bound since $\mathcal{O}(N^3)$ multiplications and additions are performed compared to $\mathcal{O}(N^2)$ memory reads and writes. However, since the Tensor Cores have such a high computational throughput, almost all programming effort ends up being used in optimizing the memory pipeline to feed the Tensor Cores with data. In section 3 we show a matrix multiplication program using Tensor Cores, while subsection 2.2 details the optimizations used to keep the Tensor Cores occupied with data.

GPU	Peak FP16 [TFLOPS]	Peak FP16 Tensor Core [TFLOPS]	Speedup
NVIDIA A100	78	312	4x
NVIDIA H100	133.8	989.4	7.4x

Table 1: Theoretical peak performance for the A100 and H100 using FP16, with and without use of Tensor Cores[13][15].

2.1.5 CUDA

A simple CUDA program

For NVIDIA GPUs, CUDA is the programming model used to write thread parallel programs. It defines the thread hierarchy described in section 2.1.2, the memory hierarchy described in section 2.1.3, synchronization mechanisms between threads and groups of threads (thread blocks) and a CUDA runtime API. The CUDA program programs themselves are written in CUDA C++, a set of language extensions to C++. CUDA programs are split into code that runs on the CPU called *host code*, and code that runs asynchronously on the GPU called *device-* or *kernel code*.

To demonstrate the different aspects of the CUDA programming interface, the host code for a small CUDA program is written below. The special `<<<grid, block>>>` syntax is one of the CUDA language extensions to C++. It defines a *kernel launch*. This is special type of function call that tells the CUDA runtime to execute the `do_square_gpu` function on the GPU[12]. In this case, the launch is specified to run on a kernel grid with size $128 \cdot 128 \times 121 \times 1$ and each grid node consists of a thread block of 256 threads. In case the block size exceeds the maximum 1024 threads, the kernel fails to launch.

Listing 1: Host code to launch a CUDA kernel

```
void square()
{
    int *host_array = // filled by the cpu somewhere
    size_t n = 128 * 128 * 256; // number of elements on host_array
    int *device_array;
    cudaMalloc(&device_array, n);
    cudaMemcpy(device_array, host_array, n, cudaMemcpyHostToDevice);
    dim3 grid(128 * 128, 1, 1);
    dim3 block(256, 1, 1);
    do_square_gpu<<<grid, block>>>(device_array, n);
    cudaDeviceSynchronize();
    cudaMemcpy(host_array, device_array, n, cudaMemcpyDeviceToHost);
}
```

The example also demonstrates how the CUDA runtime API handles synchronization and memory between the host (CPU) and the device (GPU). To copy a block of memory to the device, the memory is generally first allocated by the host and filled with data. Before any memory can be copied to the GPU, memory must first be allocated on the GPU as well. The CUDA runtime API defines a `malloc` equivalent called `cudaMalloc` to allocate memory on the device. Likewise, a `memcpy` equivalent called `cudaMemcpy` exists to then copy the host data to the device.
As previously mentioned, the kernel runs asynchronously on the GPU, and the function call to `do_square_gpu` returns immediately. To wait for the result, a synchronization primitive must therefore be used. In this case `cudaDeviceSynchronize` is used. This will block, waiting for the kernel to finish execution and the result can be copied back to the CPU.

The actual kernel code for `do_square_gpu` is also written in CUDA C++, and the implementation can even be written in the same file as the host code. Below, the corresponding kernel code is given:

Listing 2: Kernel code to square elements of an array.

```
__global__ void square(int* array, int n) {
    int blockOffset = blockDim.x * blockIdx.x;
    int tid = blockOffset + threadIdx.x;
    if (tid < n) // only compute on in bounds elements
        array[tid] = array[tid] * array[tid];
}
```

A CUDA kernel is denoted with the `__global__` declaration specifier. This allows it to be called by the host using the `<<<>>>` syntax, and tells the compiler that this should run on the GPU. The input array has $128 \cdot 128 \cdot 256$ elements, and therefore the kernel was launched with a grid and block size such that each thread needs to compute exactly one element of the input. This is a very typical way to program GPUs. Instead of having a loop iterating with dimensions proportional to the problem size, a kernel is simply launched with enough threads to do the computation in one go.

The kernel code above has some logic for computing an `offset`. Recall that threads are organized into blocks in a kernel grid. Each thread must therefore compute an offset into the array where the block should start its computation, and then a local offset within the block. As shown above, the threads in a CUDA block can access their thread index in the block by the `threadIdx` `dim3` variable. For this kernel launch, the block was specified to have threads only

in the x dimension, and therefore we have `blockDim.x=256`. To compute the block offset, each thread needs to use the `blockDim` and `blockIdx` variables to get the dimensions of the block and the index of the block in the CUDA grid.

The NVCC compiler

The above program will get compiled by the `nvcc` CUDA compiler into both assembly and an assembly like code called PTX (Parallel Thread Execution) for the host and kernel code respectively. Not all PTX instructions supported by the GPU can necessarily be expressed by CUDA C++ programs. This sometimes requires one to resort to using inline PTX assembly instead. Using Tensor Cores is one such case where it can become preferable to include inline PTX assembly. The CUDA C++ standard library does expose a warp level Tensor Core API known as `wmma`. This is however a poor API since it gives bad control over the memory layout and quickly leads to performance issues. Therefore, for maximal performance, it is necessary to use the `mma` family of PTX instructions. For the Ampere series of GPUs, a $16 \times 8 \times 16$ mixed precision `mma` can be performed. The instruction can be wrapped into a function that can be called by each thread:

Listing 3: Wrapper function to call a $16 \times 8 \times 16$ Tensor Core operation.

```
__forceinline__ __device__ void mma_m16n8k16(
    uint32_t d[4], uint32_t a[4], uint32_t b[2], uint32_t c[4]) {
    asm volatile("mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
                 {%0, %1, %2, %3},
                 {%4, %5, %6, %7},
                 {%8, %9},
                 {%10, %11, %12, %13};"
                 : "=r"(d[0]), "=r"(d[1]), "=r"(d[2]), "=r"(d[3])
                 : "r"(a[0]), "r"(a[1]), "r"(a[2]), "r"(a[3])
                 , "r"(b[0]), "r"(b[1])
                 , "r"(c[0]), "r"(c[1]), "r"(c[2]), "r"(c[3]));
}
```

The `__device__` qualifier tells the NVCC compiler that this a GPU only function, and it can therefore only be called by kernel code. The instruction included in the body of the function is fairly complicated to use and it has strict requirements for its inputs. We will do a brief overview of the instruction as a motivation for why an abstraction for using the Tensor Cores is needed, and also to demonstrate the difficulty required to use the instruction.

As it was mentioned earlier, for Ampere GPUs, the entire warp of 32 threads collectively performs the MMA. For a $16 \times 8 \times 16$ mixed precision MMA of the form $\mathcal{C} \mathrel{+}= \mathcal{AB}$, the warp will provide a 16×16 matrix for $\mathcal{A}$ and a 16×8 matrix for $\mathcal{B}$ with each element being `f16` and residing in registers. For matrix $\mathcal{A}$, the warp is responsible for $16 \cdot 16 \cdot 2 = 512$ bytes of input data and each thread is responsible for a 16-byte slice. This fits with the function signature where the input `a` is an array of 4 `uint32_t` elements, corresponding to 16 bytes of data supplied by each thread. For `f16` input matrices 8 `f16` elements per thread are thereby supplied. Figure 4 shows the mapping from matrix elements to registers for this kind of MMA operation. Thread 0 is denoted as T0 in the figure. We see that thread 0 has to load two elements from each of the indices (0,0), (0,8), (8,0) and (8,8) for a total of 8 elements. The elements are stored in the same order in the `uint32_t[4]` input. This mapping is perhaps a bit unexpected since each thread does not load contiguous elements from the source matrix. If the source matrix is not loaded into registers exactly as specified in figure 4, the instruction computes the wrong result. The mapping

of threads to values for matrix B is equally convoluted, and uses a column-major layout. In short, to effectively use one of the `mma` instructions requires a very careful arrangement of the source elements into registers. The exact arrangement will also depend on the input element type, since the Tensor Cores can also work with other data types than f16. Handling this mapping across different architectures and data types correctly is a very tedious process that can easily lead to unexpected and hard to debug errors.

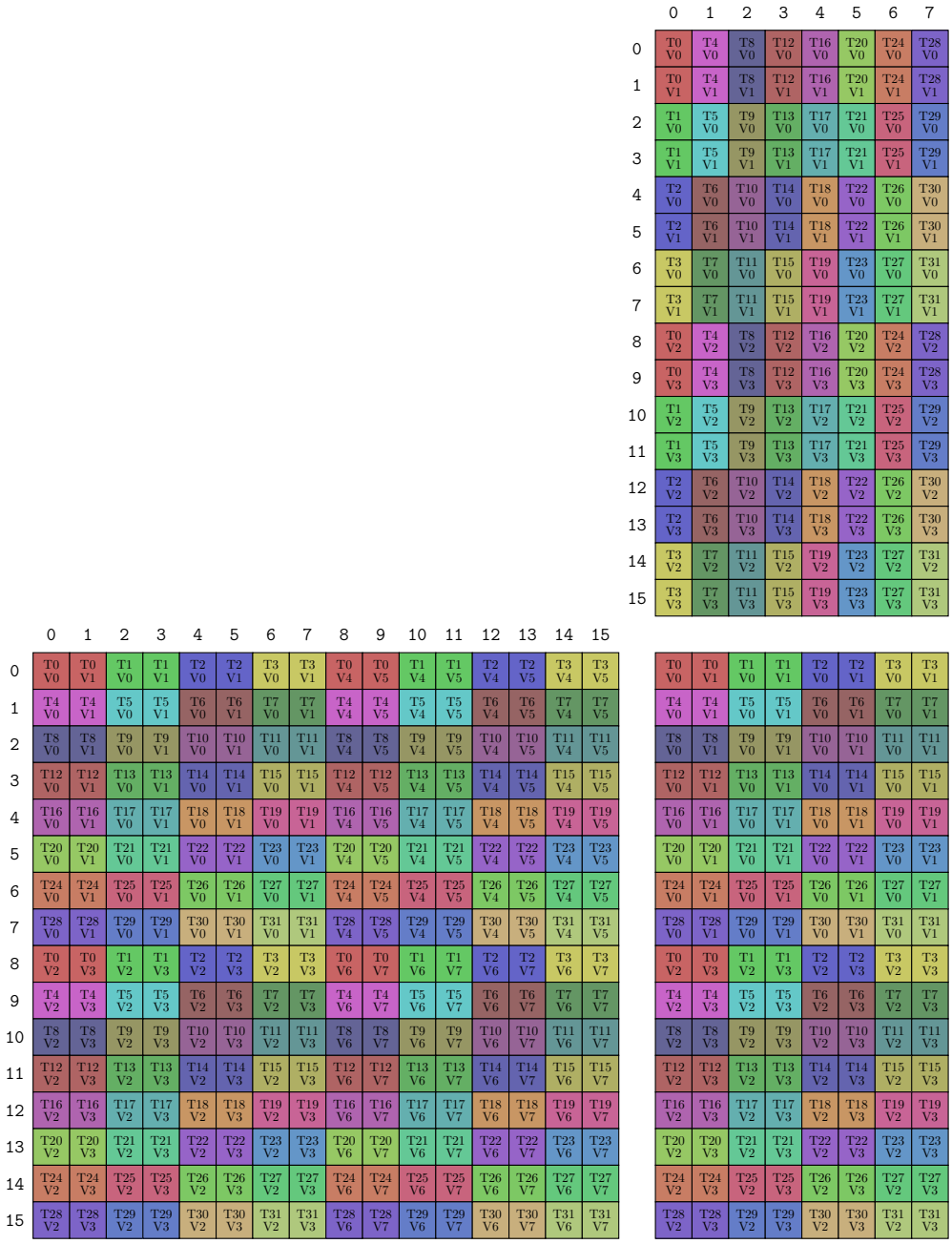

Figure 4: Mapping of elements into registers of the 32 threads in the warp for $16 \times 8 \times 16$ mixed precision MMA. The left shows the mapping for matrix A of a $C = AB$ matrix multiplication, the top right shows B and the lower left shows C Each thread holds 8 elements of A, and 4 elements of B and C.

2.2 GPU Optimizations

This section sets out to describe a set of optimizations a CUDA kernel can implement in order to come close to the peak performance for matrix multiplication kernels. These optimizations are all implemented in our prototype matrix multiplication kernel, and some of them also make their way into the futhark compiler when adding Tensor Core support.

2.2.1 Coalesced Global Memory Access

Global memory is accessed via naturally aligned 32-, 64-, or 128-byte memory transactions. That is, 32 consecutive bytes, starting at an address which is a multiple of 32, can be accessed in a single transaction, and similarly for 64 and 128. When a warp executes an instruction accessing global memory, the memory accesses of all the threads in the warp are coalesced into a number of transactions of this size. If, for example, each thread in a warp accesses a separate 4 byte value, each in a different 128 byte chunk of memory, this would then result in 32 separate transactions. Additionally at least 32 * 32B = 1024B of data would be transferred, when only 32 * 4B = 128B was really needed. This kind of access pattern causing excessive transfers is called *uncoalesced* memory access, and results in inefficient execution. If, on the other hand, the 32 threads accessed 32 consecutive 4 byte values, starting at a 32B aligned address, only a single 128B transaction would be needed, resulting in optimal performance, and the access would be said to be *fully coalesced*. Due to the high latency of global memory accesses, it is very important for performance to ensure that accesses to global memory are coalesced, so that the total number and size of transactions is kept at a minimum. [12]

The figures below show some examples in which each of the 32 threads in a warp access a distinct 4 byte word in global memory. Each arrow represents an access by a thread.

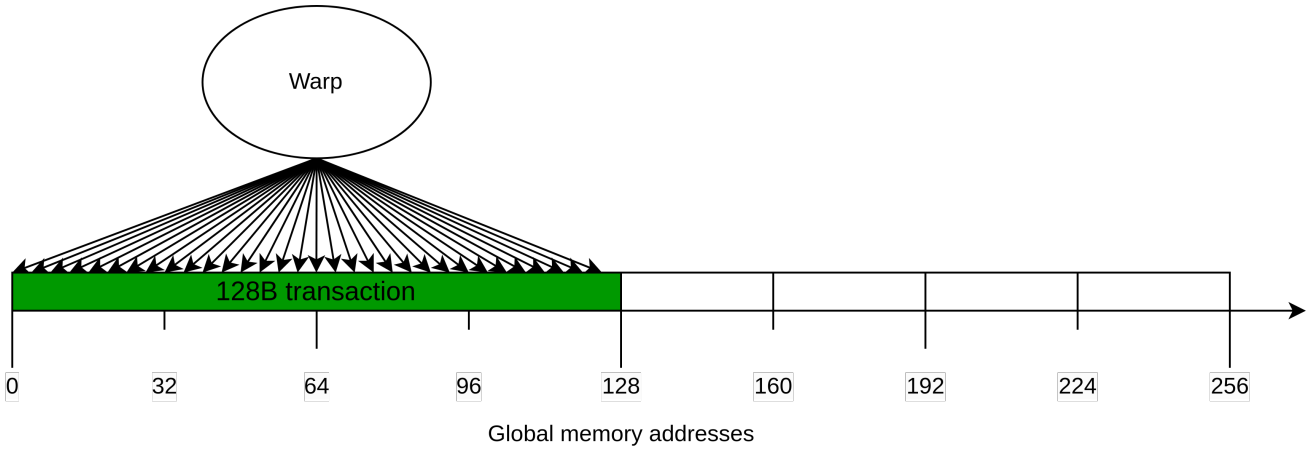

Figure 5: Fully coalesced access resulting in a single 128B transaction.

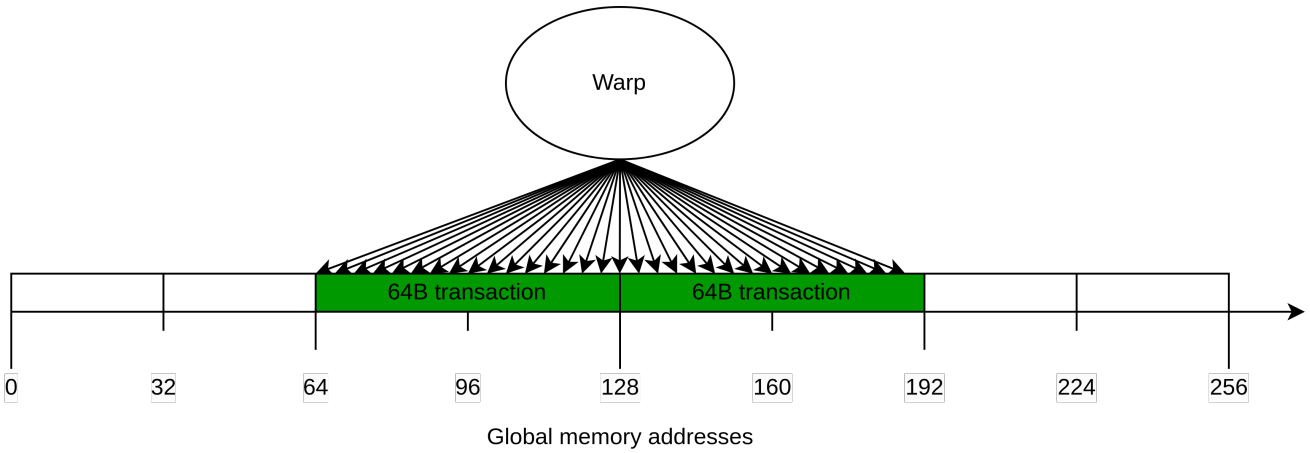

Figure 6: Unaligned access requiring 2 separate 64B transactions.

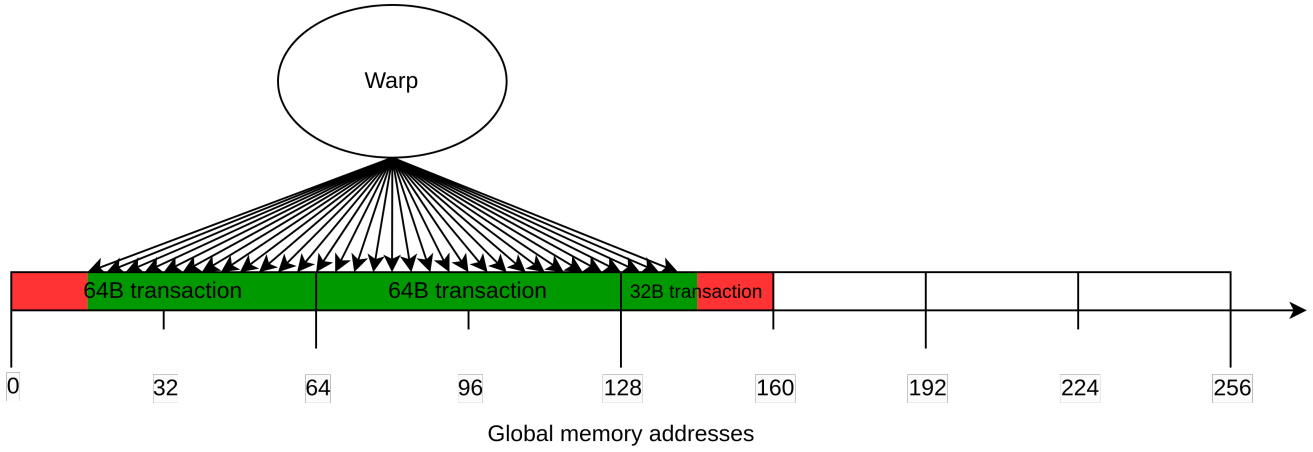

Figure 7: Unaligned access causing 3 separate transactions and excessive movement of data.

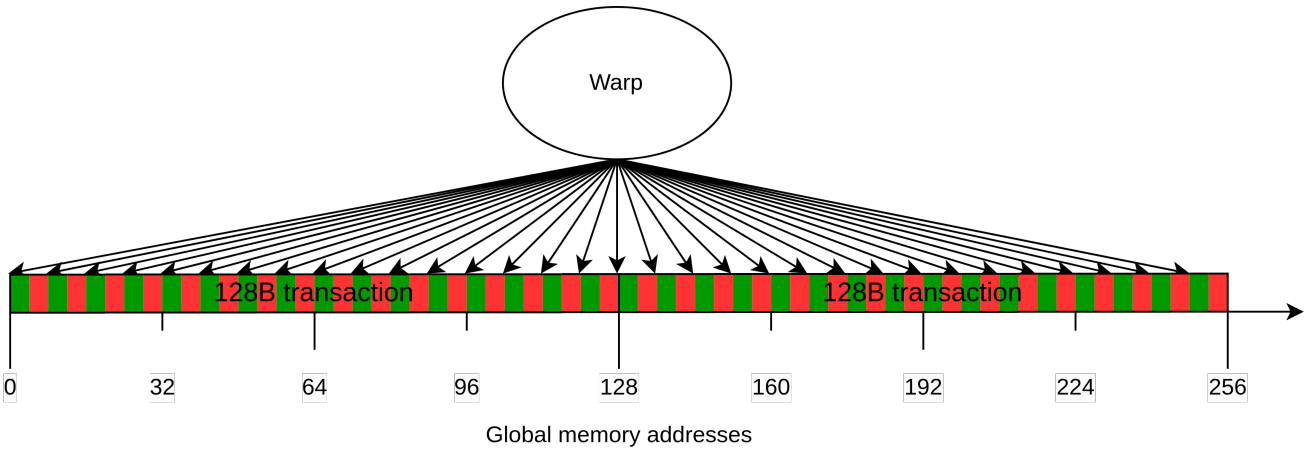

Figure 8: Strided access causing excessive movement of data. It should be noted that larger strides would result in even more transactions and excessive movement of data, even worse if combined with misalignment.

2.2.2 Avoiding Shared Memory Bank Conflicts

Shared memory is divided into 32 memory *banks*[3] of equal size, which can be accessed simultaneously by each thread in a warp. The addresses of of shared memory are divided in 4 byte words, with consecutive 4 byte words residing in consecutive banks. If each thread in a warp accesses addresses of memory in separate banks, all accesses can be serviced simultaneously, and the request is said to be *bank conflict* free. This is also the case even if some threads access data within the same 4 byte word, within the same bank[3]. However, if multiple threads within a warp access addresses in distinct 4 byte words within the same bank, this is a *bank conflict*, and these accesses must be serialized. Bank conflicts only occur within warps. [12]

For optimal performance, one must therefore be mindful to avoid bank conflicts. When working with multidimensional data, bank conflicts can often result from ”naive” code. This is in part due to the fact that it can often be desirable to use 2 or more distinct access patterns for the same data in shared memory. For example one might want to write data to shared memory in a row-major order, but read it in a column-major order.

A prime example, which is not at all far fetched from our uses, is that data in shared memory is stored in a row-major 2D array with a width which is a multiple of 128B, so that consecutive columns are in consecutive banks. This is illustrated in Figure 9.

In this case, a warp can access distinct elements of a single row of data without bank conflicts. However, if a warp accesses distinct elements of a single column, this will result in a 32-way bank conflict. This means that the 32 accesses made by the threads in the warp must be fully serialized, rather than being able to run in parallel. In fact, any access of data from less than 32 distinct columns by a warp must necessarily result in bank conflicts, which means we also cannot access sub-arrays without bank conflicts.

[3]for NVIDIA Maxwell and newer architectures

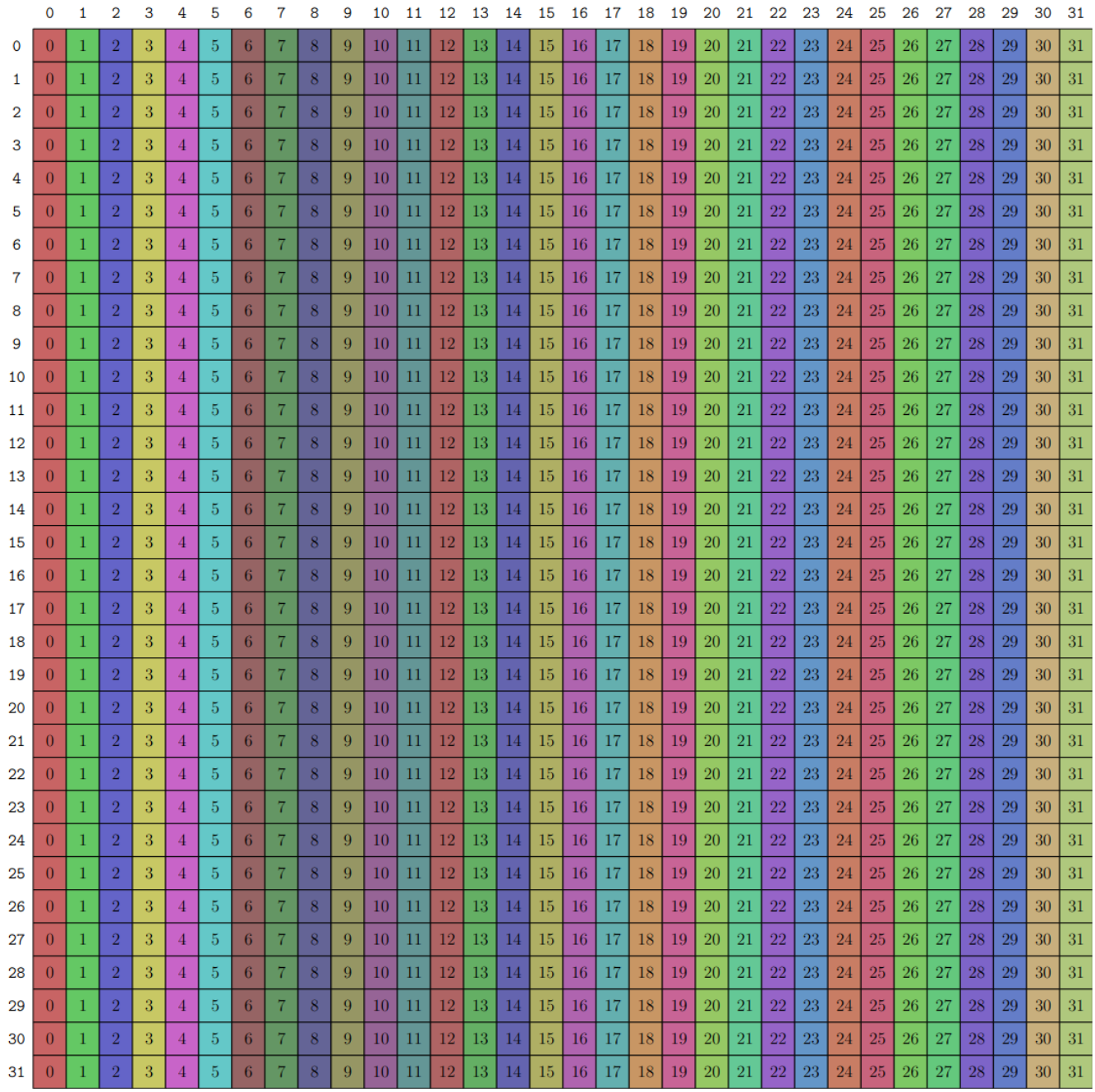

Figure 9: Naive layout for a 32 x 32 row-major array of 4 byte elements, with each cell showing the index of the memory bank in which the corresponding element will reside.

One way to reduce the number of bank conflicts is *padding* the leading dimension of multi-dimensional arrays in shared memory. If the goal is to be able to read columns without bank conflicts, a single element can be used for padding. In the previous example of a 128B multiple width array, it is easy to see from figure Figure 10 that padding with a single 4 byte element would ensure bank conflict free access to both single rows and single columns. The fact that single row access also remains bank-conflict free is important if the data is stored row-major in global memory, since data can then be transferred between global and shared memory in a row-major fashion, ensuring both fully coalesced accesses to global memory, and bank conflict free access to

shared memory, while also allowing bank conflict free column major access in shared memory.

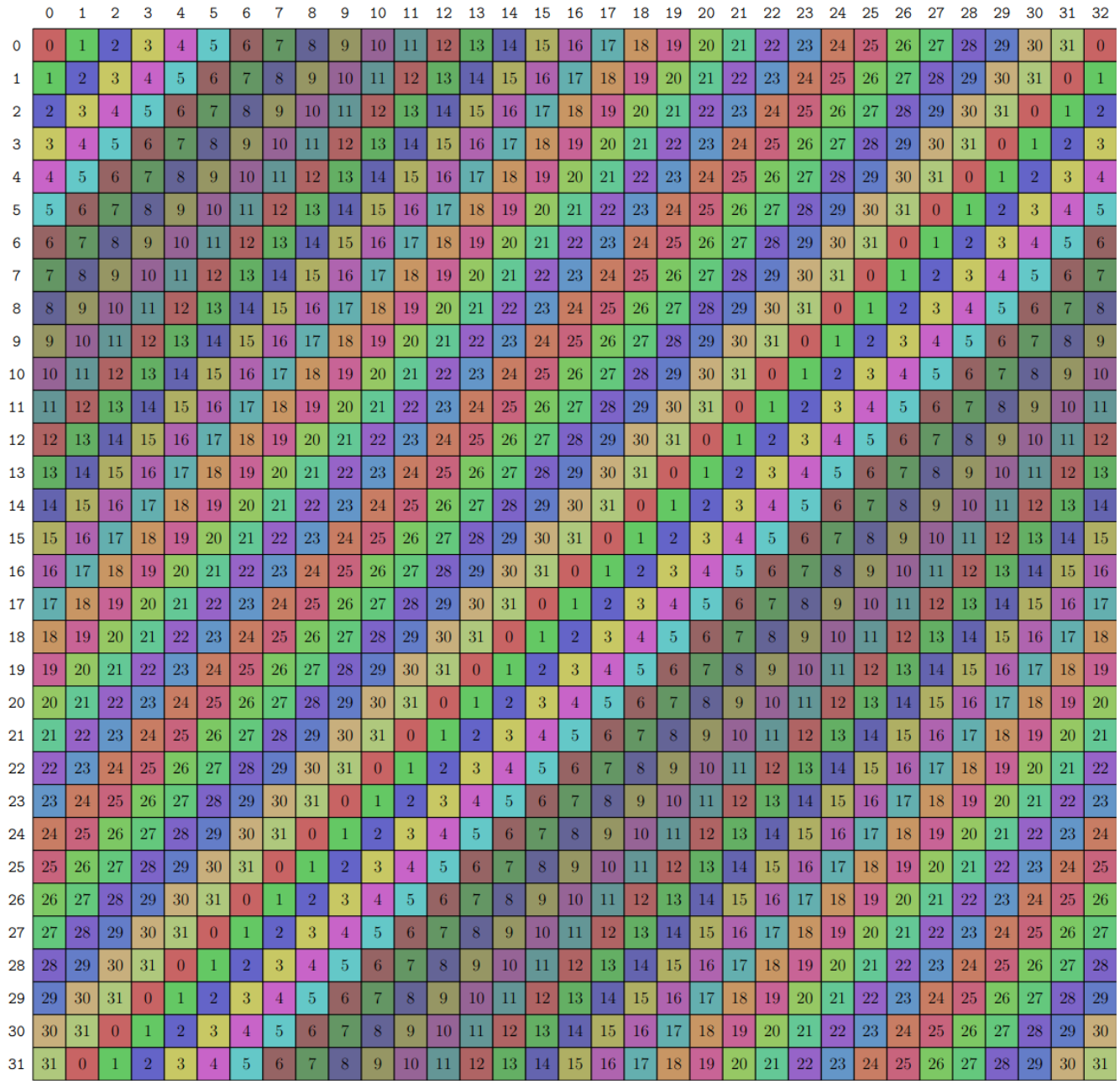

Figure 10: Single element padded layout for a 32 x 32 row-major array of 4 byte elements, with each cell showing the index of the memory bank in which the corresponding element will reside. Note the additional column.

An alternative method for preventing bank conflicts is permutation of indices, or *swizzling*. With this technique, data in shared memory is permuted rather than padded, which means no space is wasted. Additionally, this technique is very flexible, and can allow bank conflict free access for many different sets of access patterns. Finding permutations that allow bank conflict free access for a given set of access patterns in general requires the solving of a sudoku-like problem and can have many or no solutions.

A common technique for permutation of row-major arrays is to only permute the row-index of each element, calculating the new column-index by using an XOR operation between some of the bits of the column-index and the same number of bits from the row-index. Using this technique allows permuted indices to be calculated efficiently using fast bit-level operations. Which bits to use depends on the access pattern one wishes to remove bank conflicts from.

Figure 11 shows how one might compute a swizzled flat index from a given row-major flat index of the example 32 x 32 array in Figure 9. Since the width of the array is a power of 2, the bits of the flat index can be split such that the 5 least significant bits represent the column index, while the 5 most significant bits represent the row index. The XOR of these 2 sequences of bits is then used in place of the 5 least significant bits, i.e. as the new column index.

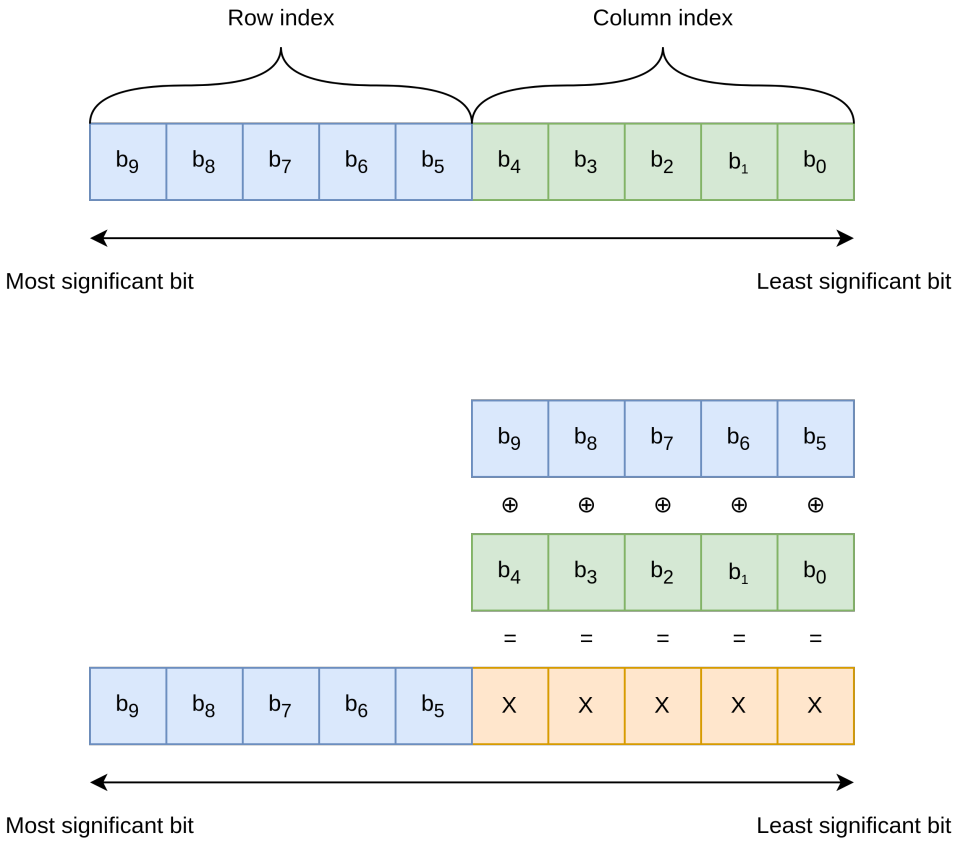

Figure 11: Example of computing a swizzled flat index using XOR, denoted as $\oplus$. Top: original index. Bottom: swizzled index.

It is clear that this transformation preserves the row index of every element. Additionally, it should clear that any 2 distinct elements that originally had the same column index, but distinct row indexes, i.e. 2 distinct elements from the same column, now also have distinct column indexes, since a different row index was then used in the XOR operation with the same column index. This means that a warp can now access both single rows and single columns of the original layout without bank conflicts using the new layout. The full result of this permutation can be seen in Figure 12.

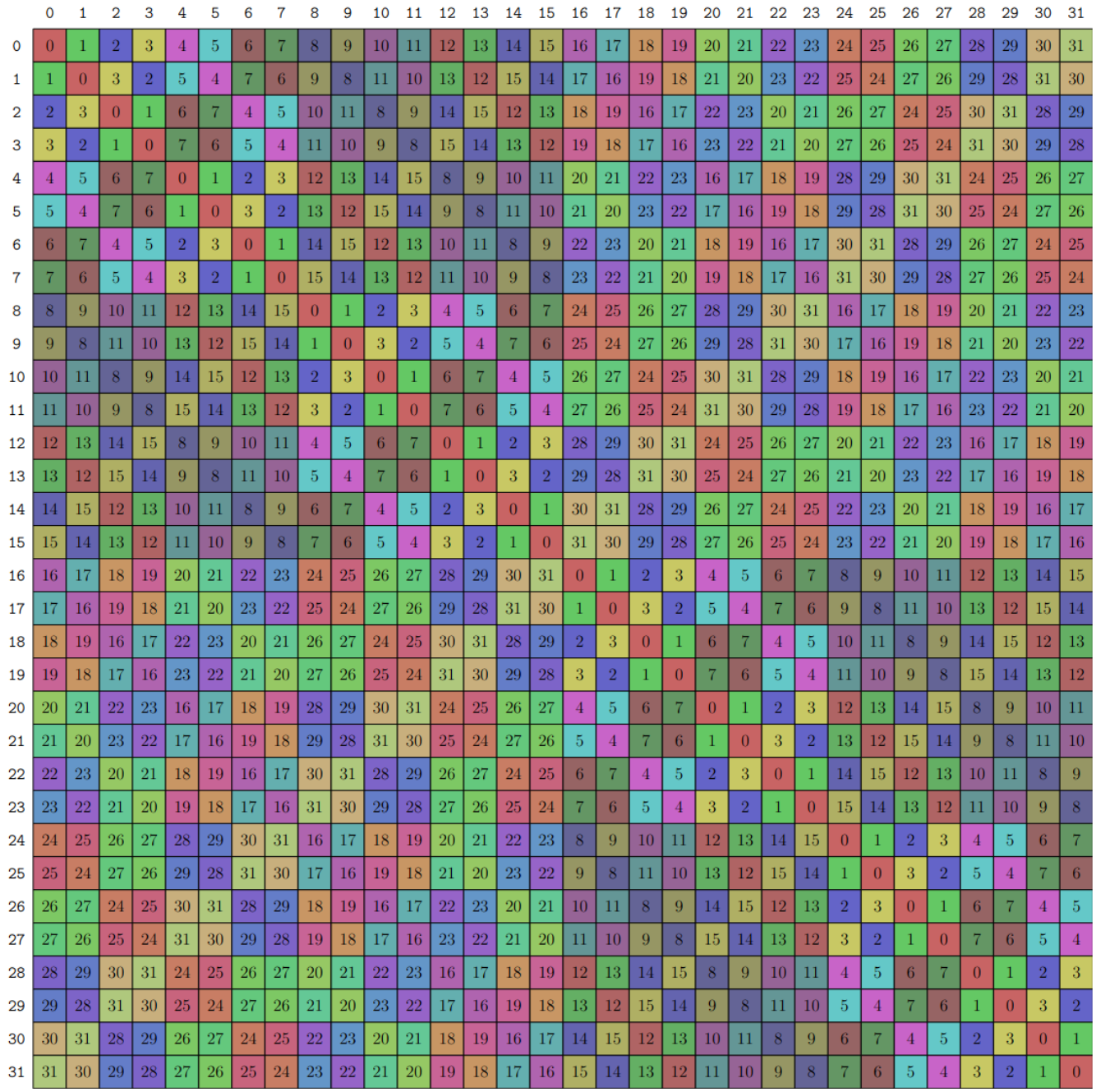

Figure 12: Example of a swizzled layout for a 32 x 32 row-major array of 4 byte elements, with each cell showing the index of the memory bank in which the corresponding element will reside.

The flexibility of swizzling also allows achieving bank conflict free access to sub-arrays. If, for example, each warp must access a 8 x 4 sub-array of the 32 x 32 array of the previous examples, the permutation shown in Figure 13 can be used. It should be clear that this permutation allows bank conflict free access to both entire rows and 8 x 4 sub-arrays.

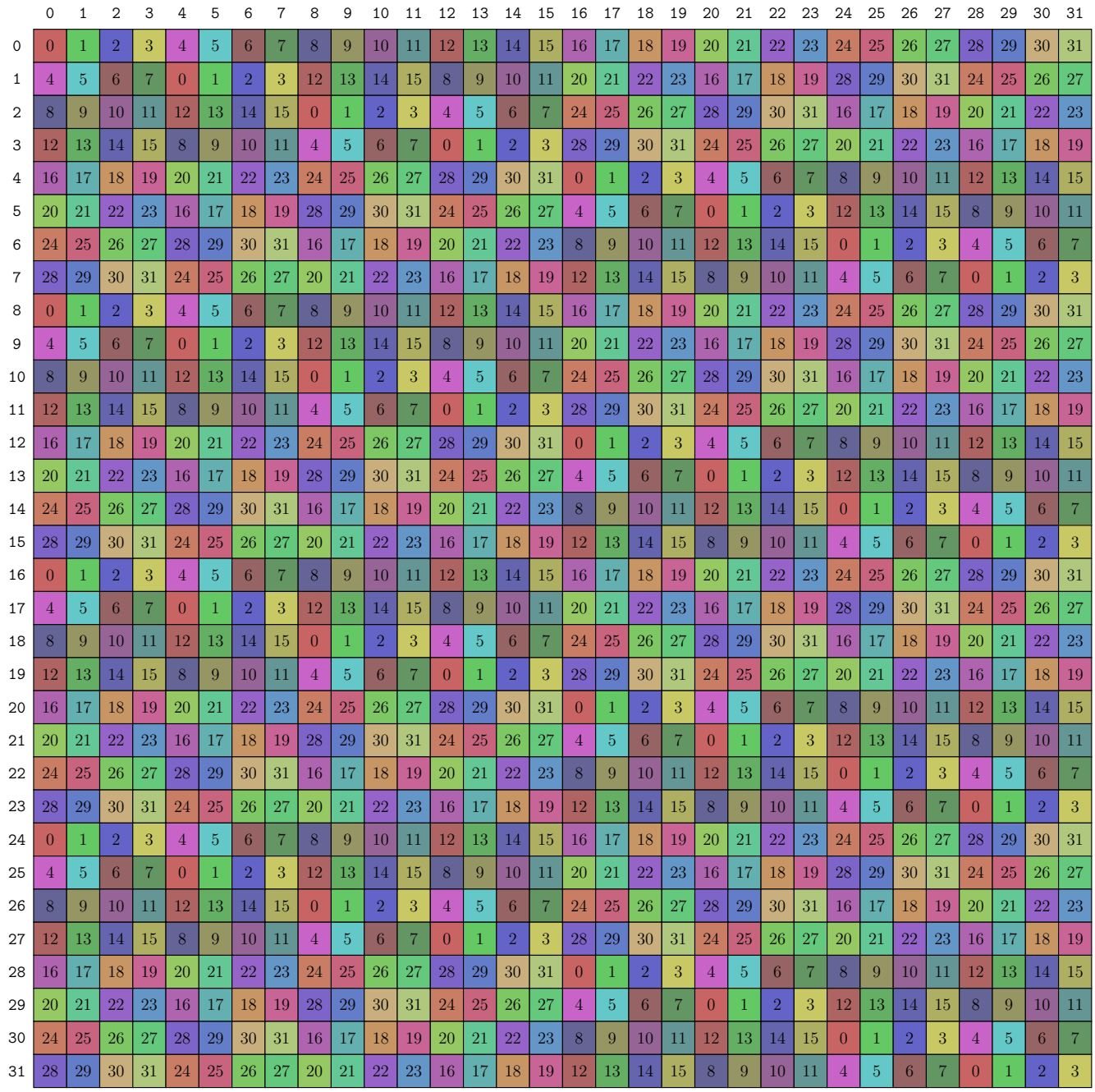

Figure 13: Second example of a swizzled layout for a 32 x 32 row-major array of 4 byte elements, with each cell showing the index of the memory bank in which the corresponding element will reside.

The computation used to calculate swizzled indexes in this case is shown in Figure 14.

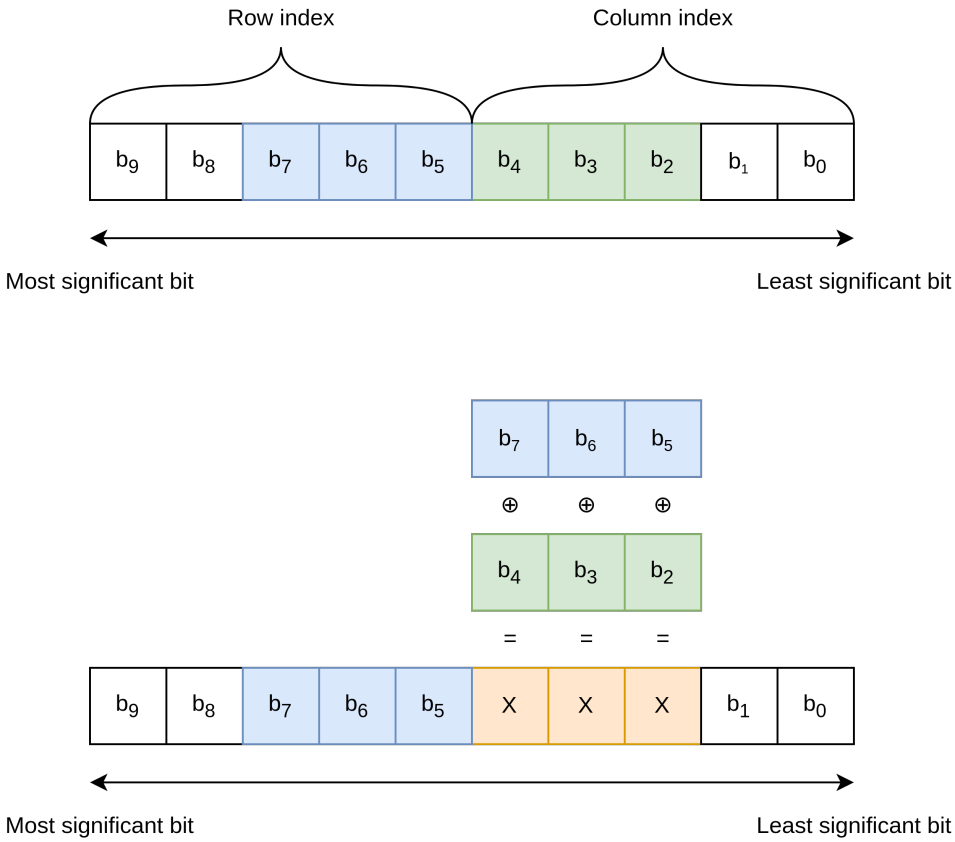

Figure 14: Second example of computing a swizzled flat index using XOR, denoted as $\oplus$. Top: original index. Bottom: swizzled index.

In this case, the 2 least significant bits and the 2 most significant bits are left unchanged. This means that sequences of $2^2 = 4$ consecutive elements will remain consecutive after the permutation, and that the overall pattern will repeat itself every $2^{5-2} = 8$ rows. Similarly to the previous pattern, the intuition as to why this computation allows bank conflict free access to 8 x 4 sub-arrays, is that elements of distinct rows in such a sub-array will also have distinct bits in the positions marked blue in Figure 14. Meanwhile, the bits in green will be the same for all sub-array elements. Therefore elements of distinct rows within the same sub-array will also have distinct bits resulting from the XOR operation. Additionally, elements of distinct columns within the sub-array will also have distinct bits in the 2 least significant positions, and as such there is bank conflict free access to all 8 x 4 sub-arrays.

The fact that sequences of 4 consecutive indexes, i.e. groups as large as the sub-array width, are still consecutive in the permutation, also allows for *vectorized accesses*, as will be described below.

In order to ensure bank conflict free access to sub-arrays of a given width using padding instead, one would have to use an amount of padding equal to this width, which could result in large amounts of wasted space.

2.2.3 Vectorized Loads and Stores

Vectorized loads and stores are another important optimization to ensure good performance. The concept is simple. Instead of having each thread request a single element from global memory at

a time, the threads will instead requests multiple elements simultaneously. In Figure 15, a slice of 8 elements of a larger array is to be loaded from global memory. The top of Figure 15 shows a scalar load with each thread reading one element. Meanwhile, at the bottom, each thread requests four elements to be loaded, and only two threads are therefore needed to copy 8 elements.

The transformation is most useful when each thread needs to access more than one element anyway, and would do so using multiple sequential scalar accesses. If this is not the case, the amount parallelism would have to be reduced in order to use the transformation, which may not always be beneficial for performance.

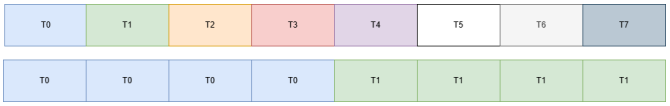

Figure 15: **Top**: Scalar load using 8 threads to load 8 elements. **Bottom**: Vectorized load with 2 threads loading 4 elements each.

Figure 16 shows a very contrived example demonstrating scalar and vectorized loads in CUDA C++. Each thread block has to copy $4 \cdot 1024$ elements into shared memory, and then needs to do some additional work on the elements in shared memory. Note that the`__shared__` qualifier makes the memory be allocated as shared. The kernel on the left uses scalar loads with coalesced access to global memory. Since the kernel needs to copy more elements than the block size, it has to sequentially copy the elements to shared. In the end, each thread copies 4 elements to shared memory using four load instructions. The kernel on the right instead performs a pointer cast for the global and shared memory pointers with the `reinterpret_cast<float 4*>` statement. In effect, each thread loads a `float4` of 16 bytes into shared memory. This forces the compiler to generate vectorized loads from global memory and a vectorized store into shared shared memory. As a result, only a single global memory load instruction is needed while each thread of the kernel still copies 4 elements from global memory to shared. The `__syncthreads()` is a CUDA intrinsic function used for block level synchronization. It acts as a barrier, forcing all threads in the block to reach this point before they can continue execution.

```cpp
__global__ void scalar_load(float* gmem, int n){
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  __shared__ float smem[4 * 1024];
  for(int i = 0; i < 4; i++){
    int offset = i * 1024;
    smem[offset + tid] = gmem[offset + tid];
  }
  __syncthreads();
  // do some more work in shared memory
}
```

```cpp
__global__ void vector_load(float* array, int n){
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  __shared__ float smem[4 * 1024];
  float4 *vec_gmem = reinterpret_cast<float4 *>(
    array);
  float4 *vec_smem = reinterpret_cast<float4 *>(
    smem);
  vec_smem[tid] = vec_gmem[tid];
  __syncthreads();
  // do some more work in shared memory
}
```

Figure 16: The left and right CUDA snippets show a copy from global to shared memory. The left uses scalar loads and the right shows vectorized loads.

Another form of ”vectorized load” is the `ldmatrix` family of PTX instructions, which allows loading up to four 8 x 8 matrices of 16-bit elements from shared memory to registers using a single instruction, which can result in significant speedups.

2.2.4 Pipelining

Most arithmetic instructions in CUDA have a latency of 4 clock cycles. Comparatively, the latency of global memory accesses is in the hundreds of clock cycles[12]. Hiding this latency, by interleaving memory transfers with computation, is therefore an important step to ensuring efficient execution. A common method for achieving this is *double buffering*, or more generally *pipelining*.

A common pattern is that, in a loop, data is transferred from some slower memory into a faster ”buffer”, after which the data in the buffer is then used in some computation. In CUDA, one example of this could be moving data from host memory to device memory, and then using it for computation on the device. Another example is moving data from global memory to shared memory, and then using it for a computation within the thread block. The same ideas can even be applied when moving data from shared memory to registers, and then doing computation with the data in registers, with the registers taking the role of the ”buffer”.

Note that these examples can and will often be nested, such that an outer loop first moves data from host to device, and then runs a computation on the device, that in a loop moves data from global to shared memory, and then runs a computation in each thread that in a loop moves data from shared memory to registers and runs a computation using these. Additionally, data will often also have to be moved back to slower memories. Such cases are omitted here for simplicity, but similar optimizations as the ones described below can also be used for these transfers.

Below, a simple code snippet exhibiting the pattern of data transfer followed by computation in a simple single-threaded setting is given:

Listing 4:

```
1  for (i = 0; i < N; i++) {
2      X = Y[i];
3      f(X);
4  }
```

It should be clear that there is a read-after-write dependence from the statement on line 2 to the statement on line 3 within each loop iteration, which means the two statements cannot run in parallel.

Additionally, if we assume that `X` refers to the same ”resource” i.e. register or memory location in all loop iterations, there are also cross iteration dependencies, which means the individual loop iterations also cannot run in parallel.

Figure 17 shows a simplified view of how a single buffer implementation like the code snippet above will be executed, with no interleaving of data transfers and computation.

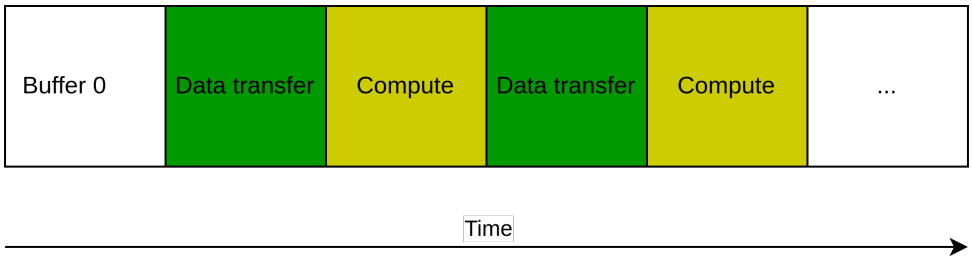

Figure 17: Single buffer loop.

Double buffering involves using 2 buffers instead of the single buffer from the previous example.

With 2 buffers, we can start by writing into one buffer outside a loop, and then let each loop iteration write into one buffer while reading from the buffer written into in the previous iteration.

The transformation of the previously shown code snippet using this optimization is shown below:

Listing 5:

```
X0 = Y[0]

for (i = 1; i < N; i++) {
    if (i % 2 == 1) {
        X1 = Y[i];
        f(X0);
    } else {
        X0 = Y[i];
        f(X1);
    }
}

if ((N - 1) % 2 == 1) {
        f(X1);
} else {
        f(X0);
}
```

Since the writing and reading of each loop iteration uses separate buffers, the data transfer and computation within each loop iteration can now be run in parallel. It should, however, be clear that there are still cross-iteration dependencies, since each iteration reads data written in the previous iteration, giving a read-after-write dependency. Additionally, each iteration also writes to the same buffer that was read from in the previous iteration, giving a write after read dependency. Therefore, distinct loop iterations still cannot be run in parallel.

Figure 18 illustrates how execution of the double buffered program could progress. This shows that in a best case scenario, with no overhead, fully parallel execution, and equal time required for data transfer and compute, it is theoretically possible to achieve doubled throughput after the first data transfer, by using double buffering.

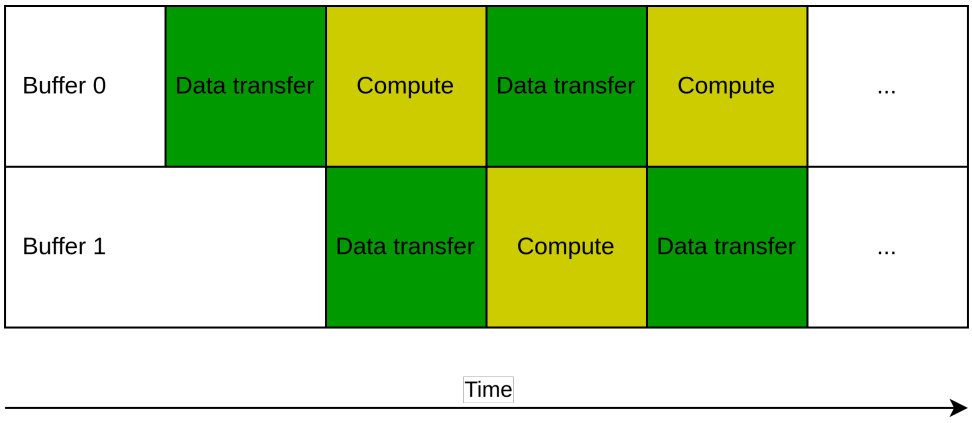

Figure 18: Double buffered loop (2-stage pipeline).

Double buffering can be generalized to *n-stage pipelining*. This is especially useful when the

data transfer has a much higher latency than the computation in each loop iteration. Figure 19 illustrates the execution of a triple buffered loop, in which which the latency of the data transfer is twice the latency of the compute operations. Note that exactly one compute operation takes place at any point in time after the initial data transfers, which means that no 2 computation operations need to run in parallel, so each will have full access to the computational units of the processor. Also note that all operations are still initiated at the same rate as in the double buffered example, corresponding to a single data transfer and compute operation per loop iteration.

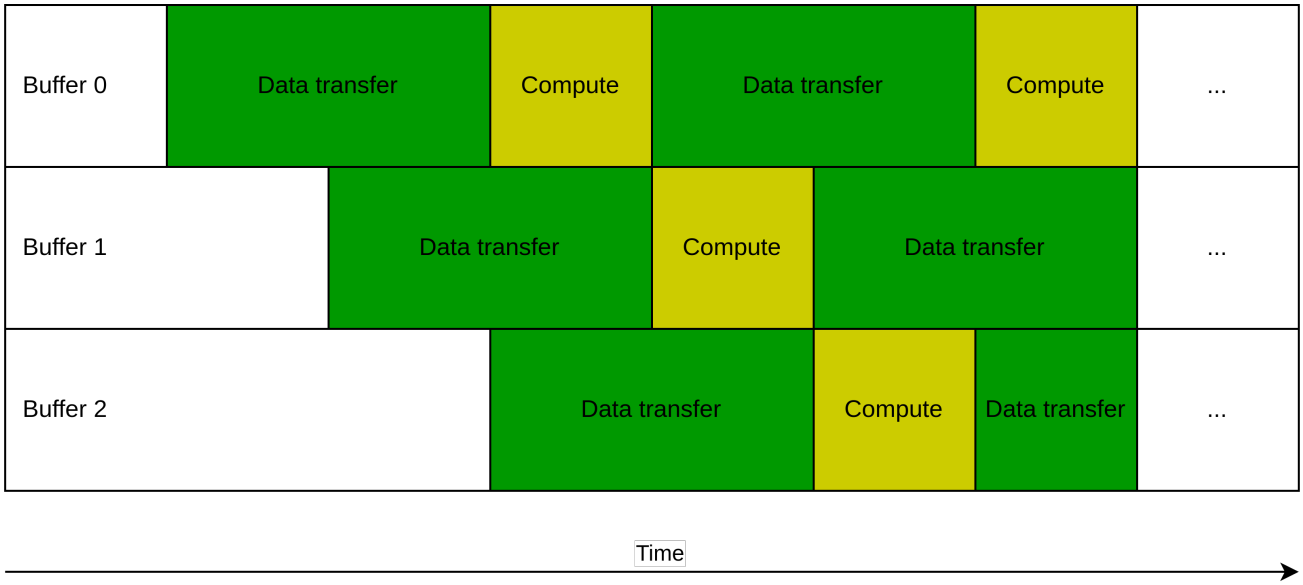

Figure 19: Triple buffered loop (3-stage pipeline) with doubled latency of data transfers.

In general, parallel execution of the data transfer and compute for each iteration could be achieved using instruction level parallelism. However, in the context of CUDA, each data transfer could be a copy from global to shared memory done by a group of threads, that all need access to this data for a computation operation afterwards. In this case, double buffering can also enable thread level parallelism. The listing below shows a naive CUDA implementation without double buffering, largely resembling the previous single buffered code snippet:

Listing 6:

```
1       __shared__ int S[blockDim.x];
2
3       for (int i = 0; i < N; i++) {
4           S[threadIdx.x] = G[i * blockDim.x + threadIdx.x];
5           __syncthreads();
6           f(S, threadIdx.x);
7           __syncthreads();
8       }
```

Note that the first `__syncthreads()` is needed due to the intra-iteration read-after-write dependency, while the second is needed due to cross-iteration dependencies, as described in the previous single buffer example. It is therefore clear that neither instruction-level parallelism nor

thread-level parallelism can be used to interleave data transfer and computation within a thread block in this example.

Similarly to the previous example we can rewrite this example to use 2 separate buffers, as shown below:

Listing 7:

```
__shared__ int S[2][blockDim.x];

S[0][threadIdx.x] = G[0 * blockDim.x + threadIdx.x];

for (int i = 1; i < K; i++) {
    int writeBuffer = i % 2;
    int readBuffer = (i + 1) % 2;

    S[writeBuffer][threadIdx.x] = G[i * blockDim.x + threadIdx.x];
    f(&S[readBuffer], threadIdx.x);
    __syncthreads();
}

f(&S[(K - 1) % 2], threadIdx.x);
```

One `__syncthreads()` statement has now been removed, allowing both thread-level and instruction-level parallelism within each loop iteration, allowing the interleaving of data transfer and computation.

As noted previously, it can be beneficial to generalize this optimization to n-stage pipelining. However, the use of `__syncthreads()` requires all memory writes to complete before that point in each iteration[12], which prevents having multiple data transfers in-flight during each iteration. One can get around this by explicitly first reading from global memory to registers, then writing from registers to shared memory in the iteration where the data is needed in shared memory.

A better solution, also for simple double buffering, available for the NVIDIA Ampere architecture and newer, is to use asynchronous copies from global to shared memory. These instructions have the benefit of not using any intermediate registers for data transfers, and their completion can be explicitly waited for when it is needed, instead of having to complete at the use of `__syncthreads()`.

2.2.5 Block-(Warp)-Register Tiling

Block tiling is a very general transformation that can be used for optimizing temporal locality of reference [3]. It is general in the sense that it is not a transformation that is unique to matrix multiplication, but such programs are often good illustrative examples of the transformation. We use this as the base transformation for achieving good performance on our matrix multiplication using Tensor Cores prototype in section 3. Therefore, we choose to show the transformation can be applied on a matrix multiplication program, and then in section 3 show how the transformation can be adapted to handle Tensor Cores. The walkthrough of the transformation is based on the work in [3].

Loop stripmining is a basic transformation needed for block tiling. It is always safe to do and it looks like the following:

```
for(i = 0; i < N; i++){                    ⇒    for(ii = 0; ii < N/T; ii += T){
    //loop_body                                     for(i = ii; i < min(ii + T, N); i++) {
}                                                     //loop_body
                                                  }
                                              }
```

Figure 20: Loop stripmining of a normalized loop.

The transformation splits up a *normalized* loop into two *loop nests* with the outer loop having stride T and the inner loop with stride 1. A normalized loop is a loop starting from 0 going to some variable bound N with stride 1.

Block tiling does *loop stripmining* on consecutive loops in a perfect loop nest and then performs *loop interchange* inwards on the resulting loops of stride 1. The transformation is safe if loop interchange is safe. Below we show a matrix multiplication program in C-like pseudo code. `forall` denotes parallel loops and `for` denotes sequential loops. We use matrix multiplication as the illustrative example of the full block register tiling transformation:

Listing 8: Capttion

```
1 forall(i = 0; i < N; i++) {
2     forall(j = 0; j < M; j++) {
3         float c = 0
4         for(k = 0; k < K; k++){
5             c += A[i,k] * B[k, j]
6         }
7         C[i,j] = c
8     }
9 }
```

We can notice that matrix `A` is invariant to the j loop and `B` is invariant to the i loop. This makes it possible to optimize the temporal locality of the program. We can also use that parallel loops are always safe to interchange inwards [3], and that the two outer loops of matrix multiplication are parallel loops in a perfect loop nest. The program then meets the requirements for block tiling, and we can apply the transformation with a stripmining factor of `T` on the two outer loops, and we can also do loop stripmining on the k loop with a factor of `T`:

Listing 9: Capttion

```
1  forall(ii = 0; ii < N; ii += T) { //grid.y
2      forall(jj = 0; kj < M; jj += T) { // grid.x
3          forall(i = ii; i < ii + T; i++) { //block.y
4              forall(j = jj; j < jj + T; j++) { //block.x
5                  float c = 0
6                  forall(kk = 0; kk < K; kk += T){
7                      // shared memory copy
8                      //...
9                      // shared memory copy done
10                     forall(k = kk; k < kk + T; k++){
11                         c += A[i,k] * B[k, j]
12                 }   }
13                 C[i, j] = c
14 }   }   }   }
```

On the transformed program, a comment indicates how the parallel loops can be mapped to the threads in a CUDA program. We can spawn a $N/T \times M/T$ grid of blocks, with each block

consisting of a 2D group of $T \times T$ threads. Now we can reason about the access patterns of the threads in a CUDA block. Within the inner loop of index `k`, each thread in the $T \times T$ block does $2T$ accesses to global memory to load the values from A and B. Across all threads, this a total of $2T^3$ accesses to global memory. By inspecting the indexing and the loop bounds for the loop variable i, j, k we can see that we only read T^2 distinct elements from `A` and `B`. With this in mind, we can copy the T^2 distinct elements of `A` and `B` into shared memory just before the `k` loop, and instead use shared memory to read the values of `A` and `B` in the innermost loop. This has a reuse factor of T and temporal locality has thereby increased by reducing the number of accesses to global memory.

As previously mentioned in subsubsection 2.1.3 there is an additional layer of on chip memory for GPUs; namely the thread local registers. This allows for an additional level of tiling for better register reuse. The exact tiling strategy will however depend on whether the Tensor Cores are used and also which type of Tensor Core to use. For the sake of simplicity. we will show how register tiling can be applied a the thread level without Tensor Cores, and adapt the strategy to handle Tensor Cores later on. Instead of stripmining the two outer parallel loops with a tiling factor T we will instead double stripmine them with a tile size $T \cdot R$ and stride R and then by a tile of size R and stride 1. The stride 1 loops are then interchanged inwards. Just as before, T denotes the block tile size and R denotes the register tile size. Similarly, the sequential k loop in Listing 8 is also double stripmined with the same tile and stride sizes. The result becomes:

Listing 10: Capttion

```
1  forall(iii = 0; iii < N; iii += T*R) { //grid.y
2      forall(jjj = 0; jjj < M; jjj += T*R) { // grid.x
3          forall(ii = iii; ii < iii + T*R; ii += R) { //block.y
4              forall(jj = jjj; jj < jjj + T*R; jj += R) { //block.x
5                  forall(i = ii; i < ii + R; i++) {
6                      forall(j = jj; j < jj + R; j++) {
7                          float c = 0
8                          forall(kkk = 0; kkk < K; kkk += T*R){
9                              forall(kk = kkk; kk < kkk + T*R; kk += R){
10                                 forall(k = kk; k < kk + R; k++){
11                                     c += A[i,k] * B[k, j]
12                         }   }   }
13                         C[i,j] = c
14 }   }   }   }   }   }
```

To complete the block register tiling transformation, we need to distribute all, but the parallel grid, loops across its statements and perform *array expansion* for the accumulator `c`. The final result of the transformation is shown in Listing 11. Notice that the stripmined loops with indices i, j have been sequentialized, and that the indexing to `A` is invariant to the j loop. We could therefore hoist the indexing into `A` out of the j loop into a register variable `float a = A[i,k]`, and then compute `float c_ = a * B[k,j]`. The sequentialization of the two innermost loop makes each thread compute a register tile of size $R \times R$ as part of the final output. Also notice that as a side effect of register tiling, each thread has to load R elements into shared memory. This copy could therefore benefit from vectorized loads and stores as described in subsubsection 2.2.3. As a last side note, the copy to shared can also be subject to pipelining in conjunction with vectorization as a way to hide the memory load latency with overlapping compute. The details of this will be discussed in section 3.

Listing 11: Caption

```
forall(iii = 0; iii < N; iii += T*R) { //grid.y
    forall(jjj = 0; jjj < M; jjj += T*R) { // grid.x
        float c[T][T][R][R]
        forall(ii = iii; ii < iii + T*R; ii += R) { //block.y
            forall(jj = jjj; jj < jjj + T*R; jj += R) { //block.x
                for(i = ii; i < ii + R; i++) {
                    for(j = jj; j < jj + R; j++) {
                        c[ii - iii][jj - jjj][i - ii][j - jj] = 0
        forall(ii = iii; ii < iii + T*R; ii += R) { //block.y
            forall(jj = jjj; jj < jjj + T*R; jj += R) { //block.x
                forall(kkk = 0; kkk < K; kkk += T*R){
                    // Copy to shared here. Could use vectorization.
                    forall(kk = kkk; kk < kkk + T*R; kk += R){
                        for(k = kk; k < kk + R; k++){
                            for(i = ii; i < ii + R; i++) {
                                for(j = jj; j < jj + R; j++) {
                                    float c_ = A[i,k] + B[k,j]
                                    c[ii - iii][jj - jjj][i - ii][j - jj] =
                                        c_
        }   }   }   }   }   }   }
        forall(ii = iii; ii < iii + T*R; ii += R) { //block.y
            forall(jj = jjj; jj < jjj + T*R; jj += R) { //block.x
                forall(i = ii; i < ii + R; i++) {
                    forall(j = jj; j < jj + R; j++) {
                        C[i,j] = c[ii - iii][jj - jjj][i - ii][j - jj]
}   }   }   }   }   }
```

Especially when using Tensor Cores, it makes sense to add an additional layer of tiling between blocks and threads, namely warp tiling. This is analogous to the other forms of tiling, and it is used to ensure reuse of the values stored in the registers of each thread participating in the warp-level Tensor Core operations.

2.2.6 Tuning for Occupancy

Occupancy is measured as the ratio between active warps on an SM and the maximum number of active warps running on the SM [16]. According to [16], a warp is considered active from the time its threads begin execution to the time they have all finished execution. The theoretical maximum amount of active warps depends on the compute capabilities of the hardware and can be found in [12]. It was mentioned in subsection 2.1 that GPUs can hide latency with compute. This might however not be possible if the occupancy is low, since there may be no warps eligible to hide the latency incurred by dependent instructions. For instance, a warp could be waiting for some data to arrive, but due to poor occupancy, there is no other warp that can run and thereby hide the latency.

On the other hand, the ability to have multiple thread blocks resident on a single SM can potentially allow latency hiding not possible with a single blocks per SM. For example, in cases like Listing 6, having a single block per SM would allow no parallelism between data movement and compute within the SM. However, having multiple blocks per SM would allow thread-level parallelism between threads of different blocks within the SM, since blocks are independent of each other by definition. Running multiple blocks per SM is not always possible, however, as it requires limiting resource usage.

Each SM has a limit on the number of blocks that can reside on it, the number of threads/warps that can reside on it, the register file size, and the amount of shared memory. These limits are shown for the NVIDIA A100 in Table 2.

Thread Blocks	Threads/Warps	Register file size	Shared memory size
32	2048/64	262144 B	167936 B

Table 2: Resource limits for a single SM on an NVIDIA A100[12].

These resources are shared between the threads in the thread block(s) running on the SM, and these restricted resources can therefore limit the achieved occupancy of a given kernel launch. For example, to achieve 100% occupancy on the A100, there must be 64 active warps per SM. Since there is a limit of 32 warps per thread block, this can be achieved by having e.g. 2 thread blocks of 32 warps resident on a single SM. These thread blocks must then use at most $167936B/2 = 83968B$ of shared memory, and each thread must use at most $262144B/2048 = 128B$ bytes of the register file, corresponding to 32 32-bit registers. Note that using more than 48 KB of shared memory in a single block requires the use of *dynamic* shared memory, which has to be requested explicitly at the kernel launch.

In most cases, configuring a kernel launch to achieve 100% occupancy does not guarantee that this configuration also achieves the best performance, since this severely limits the resources available to each thread, and could for example require register spilling. Instead, there is often a trade-off between achieving higher occupancy, and doing work more efficiently in each thread.

For example, when doing block-register tiling as described above, increasing the size of block tiles increases the amount of shared memory reuse, but also increases the amount of shared memory used per thread block, which can limit the amount of thread blocks that can reside on each SM. Similarly, increasing the size of the register tile increases the amount of register reuse, but also increases the amount of registers used per thread, which limits the amount of threads that can run on a single SM without spilling. The pipelining optimization discussed earlier also comes with this kind of trade-off, as it also increases the amount of shared memory per block or registers per thread.

These kinds of trade-offs means that in order to achieve maximal performance in CUDA, the launch configurations for a given program must be fine tuned to the resources available on the GPU it will be run on. Although it is fairly straightforward to calculate theoretical occupancy, finding optimal configurations in terms of overall performance is often best done by simply running different configurations and benchmarking these, similar to what is done by `futhark autotune`. However, understanding the hardware limits allows quickly ruling out some configurations, and can help explain why some configurations perform better than others.

2.3 Futhark

This section aims to demonstrate both Futhark the language and the underlying compiler. We will show how parallelism is expressed in Futhark, and introduce a few small example programs that will later be reused to show how the parallel constructs can take advantage of tensor cores. A brief overview of the compiler is also given.

2.3.1 The language

Futhark is a statically typed, data parallel and purely functional array language [17]. It is in the ML family of languages, and in many aspects feel very similar to languages like SML, OCaml, Haskell or similar. All parallelism is made explicit by means of second order array combinators (SOACS) that have implicit parallel semantics. The compiler will not try to automatically parallelize something

that could be run in parallel, such as loops. This skeleton based approach of expressing parallelism by SOACS enables simple, elegant, and optimized code generation such as single pass `scan` [18] or the multi-histogram implementation [19]. We will later see how a matrix multiplication routine can be written in Futhark using these parallel building blocks. Below, all the SOACS are given together with a brief and informal algorithmic overview of their semantics. Although `reduce` and `scan` might look similar to a `fold` from the ML languages, they are very different as they have parallel semantics and require that the binary operator is *associative* and has a neutral element, and `fold` has sequential semantics with no restrictions on the operator. `reduce_by_index` puts further restrictions on the operator and requires it to be *commutative*.

- **`reduce`** $\oplus$ e $[xs]$: $(\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha$ computes a generalized reduction using the associative binary operator $\oplus$ with neutral element e. In C-like pseudo-code using addition as the operator it computes:

  Listing 12: Caption

  ```
  1      int res = e
  2      for(i = 0; i < n; i++) {
  3          res = res + xs[i]
  4      }
  ```

- **`map`** $f[xs]$: $(\alpha \to \beta) \to [n]\alpha \to [n]\beta$ applies its function f to each element of the input array. This is like in any functional language, but f is required to be pure[4].

- **`scan`** $\oplus$ e $[xs]$: $(\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$ computes a generalized prefix sum. Again, instantiating the associative binary operator with addition, `scan` computes the prefix sum:

  Listing 13: Caption

  ```
  1 int res[n]
  2 for (i = 0; i < n; i++) {
  3     if (i == 0)
  4         res[i] = e + xs[i]
  5     else
  6         res[i] = res[i-1] + xs[i]
  7 }
  ```

- **`reduce_by_index`** $[dest]$ $\oplus$ e $[is]$ $[xs]$:
  ${}^*[m]\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \to [n]i32 \to [n]\alpha \to {}^*[m]\alpha$
  At a first glance, `reduce_by_index` looks like an unusual operator. It is a generalized histogram computation, where the elements in xs are reduced into the m distinct buckets using the indices is with out of bounds indices being discarded. Using addition as the commutative binary operator, `reduce_by_index` does a histogram computation:

  Listing 14: Caption

  ```
  1 int dest[m] = {e} // all elements of outputs initialized to e
  2 for (i = 0; i < n; i++){
  3     int index = is[i]
  4     int x = xs[i]
  5     if (index < m && index > 0)
  6         dest[index] = dest[index] + x
  7 }
  ```

[4]Futhark is a pure language, and this requirement is therefore trivial.

2.3.2 Writing Futhark Programs

Parallel programs written in Futhark are made up of a nested compositions of data parallel operators like `scan`, `reduce` and `map` as introduced previously. Futhark programs are much more high level than CUDA - there is no direct manipulation of memory as the language uses reference counting and automatically manages memory on the host and device. In fact, the notion of memory is introduced in very late compilation stages together with related memory optimizations [20]. The language also guarantees that data races do not exist by construction of all parallelism being expressed through SOACS. This makes writing a matrix multiplication routine almost trivial in futhark through a `map-reduce` composition:

Listing 15: caption

```
let dotproduct [n] (x: [n]f32) (y: [n]f32) =
    map2 (*) x y |> reduce (+) 0
let matmul [m][n][q]  (A: [m][q]f32) (B: [q][n]f32) : [m][n]f32 =
  map (\ Arow ->
         map (\Bcol ->
                dotproduct Arow Bcol)
             (transpose B)
      ) A
```

The program expresses matrix multiplication as the dot product between each row of $\mathcal{A}$ with each column of $\mathcal{B}$. Notice that the reduction uses the monoid[5] $(+, 0)$ to reduce the mapped result to a single `f32` result. The Futhark compiler is a highly optimizing compiler, and it is up to the compiler to generate as efficient code as possible given a program using SOACs to express all parallelism. For this program, the compiler applies a block register tiling transformation like the one described in subsubsection 2.2.5.
Below we give a slightly different program. It uses the Futhark `loop` construct. This is a sequential loop that iteratively accumulates a result similar to tail recursive functions, where the result of each loop iteration is the input to the next iteration. The program is similar to flash attention in that matrix multiplication is repeatedly computed in a loop:

Listing 16:

```
def attention_like [q] (A: [m][k]f16) (B: [q][k][n]f16) : [m][n]f32 =
  let acc_init : *[m][n]f32 = replicate (m * n) 0.0f32 |> unflatten in
  loop (acc : *[m][n]f32) = (acc_init: *[m][n]f32) for i < q do
    let C' = matmul A B[i]
    in map2(map2 (+)) acc C'
def main [q][p] (A: [p][m][k]f16) (B: [p][q][k][n]f16) =
  #[incremental_flattening(only_intra)]map2 attention_like A B
```

The program has a special attribute, namely `#[incremental_flattening(only_intra)]`, instructing the compiler on the flattening strategy to use. The futhark compiler makes use of regular flattening [21] to map regular nested parallelism efficiently to the different layers of parallelism on the GPU. It was therefore a simplification to say that matrix multiplication programs automatically are subject to block register tiling. In fact, the block register version is one manifestation of the program that takes advantage of the two levels of parallelism on a GPU: An outer map over the blocks on the kernel grid and an *intragroup kernel* computing a block register tile. By intragroup kernel, we refer to the the parallel operation being mapped to the threads in a cuda thread block. These can often be executed efficiently since the threads has accessed to shared

[5] A monoid is the pair of an associative operator $\oplus$ with neutral element e.

resourced such as shared memory. For the full tiling strategy generated by the compiler see [11]. A fully flattened version is also created that fully exploits all parallelism in the program. Full flattening will ruin any hopes of good locality, but it might be beneficial if the two levels of parallelism cannot saturate the GPU. For this reason, the Futhark runtime system will choose the version to execute depending on the size information of the input arrays. There is much more to be said about flattening and how it is implemented in Futhark. We refer to [21] for more details about flattening, and [22] for the background about how Futhark uses *autotuning* to select the most appropriate program version at runtime.

In the attention like Futhark program above, the annotation tells the compiler to only generate the version where the parallelism inside the `attention_like` function is exploited using blocks of threads. This way, the outer `map2` is mapped to the CUDA kernel grid and the parallelism inside the `attention_like` function becomes an *intragroup kernel* mapped to the CUDA block that does matrix multiplication. Since the `attention_like` function will execute at the block level, we can reason that if the dimensions of the arrays for `matmul` are exactly those required for a Tensor Core matrix multiplication, so the compiler could map the computation directly to the Tensor Cores.

2.3.3 A quick compiler overview

The Futhark compiler has a very typical compiler architecture consisting of three main parts: the frontend, middle-end and a backend. Figure 21 gives an overview of the different stages of the compiler when compiling a GPU program.

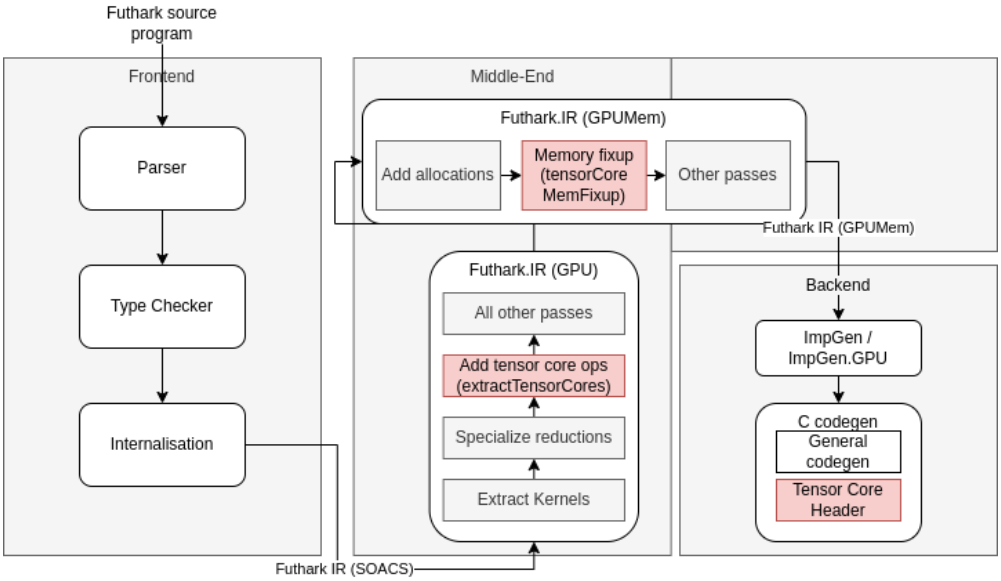

Figure 21: Overview of the Futhark compiler stages. The red square boxes correspond to passes that were added to the middle-end or new code generation in the backend. Named arrows show the current IR representation.

Frontend

The frontend is responsible for parsing and type checking a Futhark source program. An *internalisation* is then performed that turns the source program into core IR. The core IR is parameterized over a representation and, depending on the compilation pipeline, this representation will gener-

ally get transformed into other kinds of representation. In all cases however, the output of the frontend is Futhark core IR on the SOACS representation.

Middle-end

Most optimizations of the compiler take place in the middle-end. Multiple passes on the IR can be composed into a pipeline. Figure 21 shows two pipelines for generating GPU code: a GPU pipeline on the `GPU` representation and a GPU pipeline that adds memory information using the `GPUMem` representation. We refer to [20] for details about the memory representation in Futhark IR.

Backend

The backend takes an IR program with memory information such as `GPUMem` and eventually converts it to a C-program for the host code and CUDA program for device code. As a stepping stone for code generation, the backend first converts the `GPUMem` IR program to a high level imperative code form, *ImpCode*, through the `ImpGen` pass.

3 Prototypes

Before beginning to modify the Futhark compiler, we wanted to write, by hand, a prototype demonstrating efficient matrix multiplication using Tensor Cores in CUDA. The main purpose of this was to first of all get first-hand experience with programming the Tensor Cores, and through this learn the tricks and optimizations required to use them to their full potential. We also wanted to identify a set of core "components" or "primitives" that could be inserted by a compiler, enabling it to flexibly and efficiently use the Tensor Cores in its output code.

3.1 Pure CUDA

The first prototype was done in "pure" CUDA, i.e. no non-core libraries. One of the first challenges with this approach was that many Tensor Core specific operations are not exposed by the CUDA API.

In particular, the *swizzling* optimization mentioned in subsubsection 2.2.2 cannot be used in combination with the only function exposed by the CUDA API for loading Tensor Core arguments, `load_matrix_sync`, corresponding to the `wmma.load` family of PTX instructions. This is due to the fact that these instructions require a constant stride between the rows of the matrix being loaded, which is not the case after swizzling.

The `ldmatrix` family of PTX instructions are much more flexible in this regard, since they take as arguments a pointer to each row of the matrix to be loaded. However, these instructions are not exposed by the CUDA API, thus requiring the use of inline PTX.

The `mma_sync` function is the only function exposed by the CUDA API that uses the Tensor Cores for matrix multiplication. This function takes as arguments objects initialized by `load_matrix_sync`, and the exact representation of these objects is opaque according to the documentation[12]. Therefore, to use `mma_sync`, it is necessary to also use `load_matrix_sync` and we cannot load the registers to use in the mma computation using regular loads. Therefore, we again had to resort to inline PTX and use the `mma` family of PTX instructions.

We also ran into some unexpected performance issues when using asynchronous copies for transferring data from global to shared memory, as described in subsubsection 2.2.4. Through some testing and debugging, we were able to fix these issues by removing some conditionals around the asynchronous copies related to bounds checking.

As far as we can tell, the performance issues were due to the fact that the conditionals prevented the asynchronous copies from being fully coalesced, even though synchronous copies with the same conditionals and access patterns were fully coalesced. It may be the case that this is simply a restriction one must keep in mind when using these asynchronous copies, however if this is the case, these restrictions are not well documented in [12].

The `cp.async` family of PTX instructions does provide alternatives to regular bounds checking, perhaps due to the exact performance issues described above. However, these alternatives are not accessible through the `memcpy_async` API available in CUDA, and one must instead use the special primitive `__pipeline_memcpy_async`, which provides limited access to this functionality with an interface more closely resembling the PTX instructions. Alternatively, we can again use inline PTX to gain full access to this functionality.

In the end, we did manage to get a working and fairly efficient implementation in pure CUDA, but in order to achieve this high performance, the code ended up being quite complex, which would in turn make it more complex to generate similar code in the compiler. The complexity was in part due to the use of inline PTX as described above. Much of the complexity also came from index calculations, which were especially complicated by the use of the tiling and swizzling optimizations described earlier. For these reasons, we decided to implement another prototype, this time using NVIDIA's open source library Cutlass.

The full implementation of the pure CUDA prototype can be seen at `https://github.com/caymand/mma/blob/main/src/cuda_prototype/matmul-tensor.cuh`.

3.2 Cutlass/CuTe

Cutlass contains full implementations of matrix multiplication using e.g. a whole device, thread block, or warp. However it also contains CuTe, a "sub-library" of abstractions for working with multidimensional data in CUDA, and in particular also for programming Tensor Cores. This library is used to provide building blocks for many of the linear algebra program implementations in Cutlass.

Using this library allows us to write much simpler code, since it abstracts away much of the complicated index calculations and calling of inline PTX. Additionally, the library's rich usage of template arguments makes it easier to parameterize our code over things like types, sizes, and architectures, which might otherwise require a lot of code to enable support for each possible option. It also gives us a higher level of control and flexibility, and allows us to better modularize our code, compared to using a full standalone implementation. These features are very useful, if not necessary, when we want to generate code like this in the Futhark compiler.

Apart from using a full matrix multiplication implementation, the simplest way to do tiled matrix multiplication using CuTe, is to use a main loop like the one shown below.

Listing 17: Simple main loop for tiled matrix multiplication using CuTe

```
for (int k_tile = 0; k_tile < k_tile_max; k_tile++)
{
    // Copy global -> shared
    copy(global_shared_tiled_copy_A, tAgA(_,_,_,k_tile), tAsA);
    copy(global_shared_tiled_copy_B, tBgB(_,_,_,k_tile), tBsB);
    __syncthreads();

    // Tiled matrix multiplication with input in shared memory,
    // accumulated in registers
    gemm(tiled_mma, tCsA, tCsB, tCrC);
    __syncthreads();
}
```

Here, `A_global_shared_tiled_copy` and `B_global_shared_tiled_copy`, and `tiled_mma` are special CuTe structs that define how copying from global to shared, and matrix multiplication is done, respectively. The structs define both the PTX instructions to use for copying and matrix multiplication, and the mapping of threads and values. `copy` and `gemm` are CuTe functions that use these definitions to actually perform the operations using the given arguments.

The variables starting with a lower case "t" are all CuTe tensors, which are defined by a pointer to an array, and a *layout* of this array. The use of this tensor interface greatly simplifies indexing and tiling.

The main complexity in the above implementation is not shown in the code snippet, but involves initializing the tensors, and the copy and mma structs in a way that is both valid and allows for optimal performance. As a part of this initialization, we can apply most of the optimizations described in subsection 2.2. By configuring the copy structs and the layout of the tensors representing shared memory, we can use swizzling, and ensure fully coalesced and bank conflict free memory accesses. In a similar manner we can also make the kernel use both vectorized and asynchronous loads. If asynchronous loads are used, these also need to be committed and awaited, which can be done using the CuTe functions `cp_async_fence` and `cp_async_wait`, which simply calls the corresponding PTX instructions using inline PTX.

Listing 18 shows how the kernel can be configured to use a swizzled layout for A in shared, and to use asynchronous, vectorized copies from global to shared, using $16 \cdot 8 = 128$ threads to copy 64×64 elements.

Listing 18:

```
TiledCopy copyA_global_shared = make_tiled_copy(
    Copy_Atom<SM80_CP_ASYNC_CACHEGLOBAL<uint128_t>, half_t>{},
    Layout<
        Shape<_16, _8>,
        Stride<_8, _1>
    >{},
    Layout<Shape<_1, _8>{}
);

auto sA_layout = tile_to_shape(
    composition(
        Swizzle<3,3,3>{},
        Layout<
            Shape <_16, _64>,
            Stride<_64, _1>
        >{}
    ),
    make_shape(shared_M, shared_K)
);

Tensor sA = make_tensor(make_smem_ptr(smemA), sA_layout);

ThrCopy thr_copyA_global_shared = copyA_global_shared.get_slice(threadIdx.x);

Tensor tAgA = thr_copyA_global_shared.partition_S(gA);
Tensor tAsA = thr_copyA_global_shared.partition_D(sA);
```

It is also clear that Listing 17 resembles the code snippets shown in subsubsection 2.2.4, which means we can rewrite the main loop and apply the pipelining optimization, in order to hide the large latency of the loads from global memory.

Another possible optimization is to implement the copying of data from shared memory explicitly, and call `gemm` with elements of A and B in registers rather than in shared memory. This allows explicit pipelining of the movement of data from shared memory to registers, and allows using the `ldmatrix` family of PTX instructions to ”vectorize” the loads from shared memory to registers.

Listing 19 shows how `tiled_mma` can be instantiated using templating, and used to create some of the tensors used in Listing 17. The code shown defines a `TiledMMA` struct that uses the PTX instruction `mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32` to perform mixed precision matrix multiplication. The 2 x 2 x 1 `Layout` is the layout of warps that will be performing the instruction in parallel, i.e. a total of 4 warps or 128 threads in this case. The 32 x 32 x 16 `Tile` defines the total size of the tile in terms of elements in each dimension. It should be noted here that if each of the 2 x 2 x 1 warps only ran the `mma` instruction once the size of the computed tile would only be 32 x 16 x 16. The `Tile` parameter causes the `mma` instruction to instead be run 2 times in the N dimension in each warp, so that the total size of the computed tile is instead 32 x 32 x 16. This ensures that we can use the largest vectorized load possible for elements of both A and B, i.e. the version of `ldmatrix` that loads 16 x 16 elements in each warp. The matrix elements will be distributed into thread registers of each warp similarly to what is shown in Figure 4, except that this pattern will be extended in the N dimension so each warp computes a 16 x 16 tile of C.

Listing 19:

```
TiledMMA tiled_mma = make_tiled_mma(
    MMA_Atom<SM80_16x8x16_F32F16F16F32_TN>{},
    Layout<Shape<_2,_2,_1>>{},
    Tile<_32, _32, _16>{}
);

ThrMMA thr_mma = tiled_mma.get_slice(threadIdx.x);

Tensor tCgC = thr_mma.partition_C(gC);
Tensor tCrC = thr_mma.make_fragment_C(tCgC);

Tensor tCsA = thr_mma.partition_A(sA);
Tensor tCsB = thr_mma.partition_B(sB);
```

The full implementation of the prototype using CuTe can be seen at `https://github.com/caymand/mma/tree/main/src/cuda_prototype`. For the CuTe prototype we used 3 different kernels corresponding to 3 different main loops, these are in `matmul-cutlass-simple.cuh`, `matmul-cutlass-sync.cuh`, and `matmul-cutlass.cuh` in this repository.

3.3 Evaluation

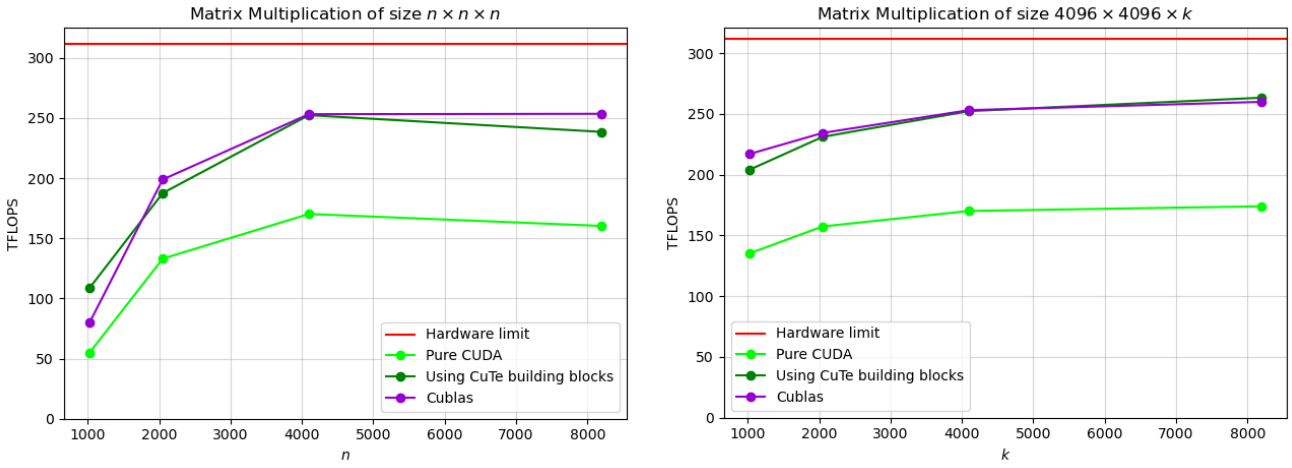

Figure 22: Achieved performance in TFLOPS using mixed precision tensor core operations on an NVIDIA A100 for the best configurations of our 2 prototypes, compared to cuBLAS. All results are averaged over 20 runs. **Left:** Performance in TFLOPS for matrix multiplication of size $n \times n \times n$, as a function of n. **Right:** Performance in TFLOPS for matrix multiplication of size $4096 \times 4096 \times k$, as a function of k.

Figure 22 compares the performance of our 2 prototype programs. The performance of cuBLAS is shown as a state-of-the-art reference. It should be noted that the performance shown for our programs is for the best configuration we were able to find manually for the given problem size. We did not explore the entire search space, so it is possible that better configurations exist for some problem sizes. The cuBLAS function used selects the configuration automatically based on

problem size and type, which could incur a very small runtime overhead. However, this overhead should be negligible for larger problem sizes.

Our pure CUDA implementation achieves 62-69% of the performance of cuBLAS, while the version using CuTe reaches 94-100%, and even beats cuBLAS for some problem sizes. In addition to the added performance, the implementation using CuTe allowed for both less, and much more readable code, compared to the pure CUDA implementation. This is mainly due to the fact that much of the index calculations and calls to PTX instructions are abstracted away. However, understanding all of the building block provided by CuTe did take some time, and although most where well documented, it was still not trivial to combine the building blocks into a solution with performance close to that of cuBLAS.

There is no precompiled "magic" in CuTe, as it is a fully open-source header-only library, so it should of course be possible to have a pure CUDA implementation reach similar performance. We believe there are multiple reasons why our pure CUDA version does not achieve this. For one, we can see from compiler output that the pure CUDA version generally uses more registers than the CuTe version even when using the same configuration in terms of block and shared memory buffer size. This means that some configurations cannot run without spilling registers, which is very bad for performance. This discrepancy could probably be removed by further optimizing the pure CUDA version to make more efficient use of registers.

The compiler output also shows that the pure CUDA version has many more instructions that are not directly used for data movement or MMA, compared to the CuTe version. For comparison the length of the SASS[6] program generated by the compiler is 1612 instructions for the pure CUDA version, while only 606 for the CuTe version, even though the number of instructions used for MMA are roughly the same. This could perhaps be caused by inefficient index calculations, or insufficient use of static calculations in the pure CUDA version.

In order to get an understanding of the effects of our various optimizations, we also ran simple benchmarks for different variations of our CuTe prototype. The benchmarks simply consisted of measuring the execution time for a number of matrix multiplications between relatively large matrices. The time used to move data to the GPU and back was not included. We convert the measured time into floating point operations per second (FLOPS) by dividing the total number of floating-point operations in the calculation, i.e. $2 \cdot M \cdot N \cdot K$ per matrix multiplication, by the execution time. Figure 23 shows an excerpt of these benchmarks.

[6]An even lower-level assembly language than PTX, which compiles to the binary microcode run natively on NVIDIA GPUs.

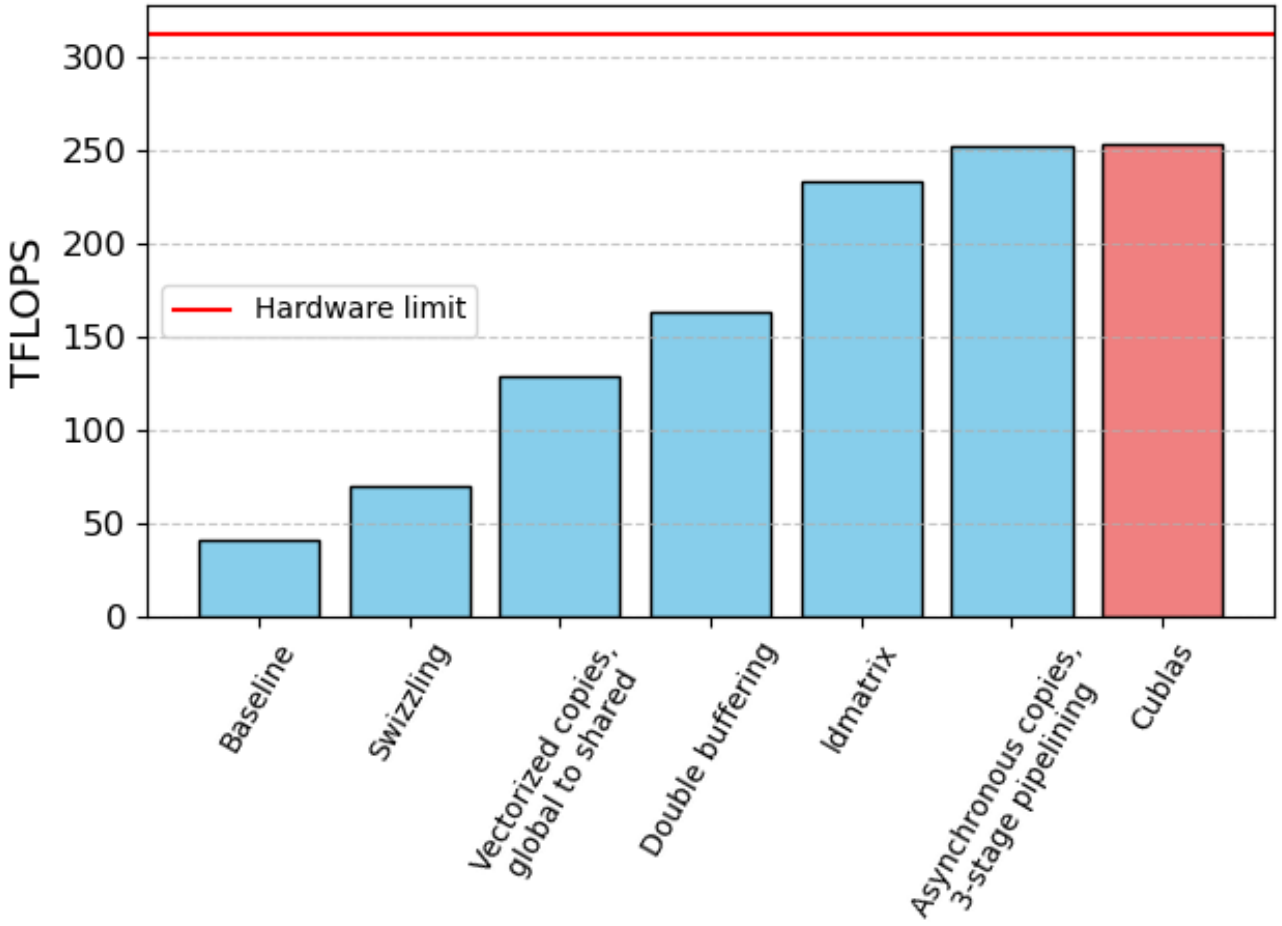

Figure 23: Comparison of performance in TFLOPS, i.e. 10^{12} FLOPS using various optimizations. The performance is averaged over 20 runs of mixed-precision matrix multiplication with $M = N = K = 4096$. The benchmarks were run on an NVIDIA A100. The performance of cuBLAS is shown as a state-of-the-art reference. Note that the optimizations shown are incremental, such that each version uses all optimizations listed to the left of it. The order was chosen by at each step applying the single optimization that improved performance the most. The baseline version uses a main loop very similar to the one shown in Listing 17. All versions use 256 threads per block and shared memory buffers with $m = 128$ $n = 256$ and $k = 64$.

The above plot can be used as a form of step-by-step guide on how to best improve performance of matrix multiplication using Tensor Cores, and the performance to be expected at each step. However, it is of course possible, and could also be beneficial, to implement the optimizations in a different order, or only use a subset of optimizations not shown in the graph. For example, our benchmarks showed that the combination of swizzling, vectorized copies from global to shared, and asynchronous copies resulted in almost the same performance as the version labeled ”Double buffering” in the plot, while still allowing it to use the very simple main loop from Listing 17. The reason why this is relatively efficient even without double buffering is likely that the use of asynchronous copies frees up some registers which can result in better occupancy, and prevent register spilling,

It should be noted that occupancy optimization is not taken fully into account in the above plot, and some versions did perform slightly better with different sizes of blocks and shared memory buffers. However, we tried to pick a size for the blocks and shared memory buffers that generally performed well for all versions, and prevented large amounts of register spilling. The best way to do this sort of benchmarking would probably be to find the best configuration of each version under some resource constraints and compare these to each other. However, as noted earlier, this sort of optimization can be very tedious and require lots of trial and error.

4 Compiler Modifications

To investigate the feasibility for Tensor Core operations in the CUDA backend of the Futhark Compiler, we try to implement the least amount of modifications to the compiler. The general strategy is to have the compiler recognize matrix multiplication inside of an intragroup kernel, and replace the matrix multiplication code with special function calls that are opaque to the compiler. Instead of having the compiler generate code for our special functions, we include a header file with their implementation. To gauge the feasibility of such a solution our first step was to perform these code transformations by hand on the Futhark IR. After this had shown promising results, we continued with modifying the compiler.
In the next subsection we start by giving an overview of how we have modified the compiler to support operations using tensor cores. Afterwards, the remaining subsections go into details about the added compiler passes and modifications.

4.1 Code Generation Strategy

At a very hight level, our compiler modifications take the following approach to utilizing the Tensor Cores:

1. Make the compiler recognize matrix multiplications that are suitable to execute on the Tensor Cores.

2. Transform the IR to make use of special functions that abstract out hardware specific Tensor Core operations.

3. Massage the compiler to accept the new special functions.

4. Have the code generation step generate CUDA code utilizing Tensor Cores within the special functions.

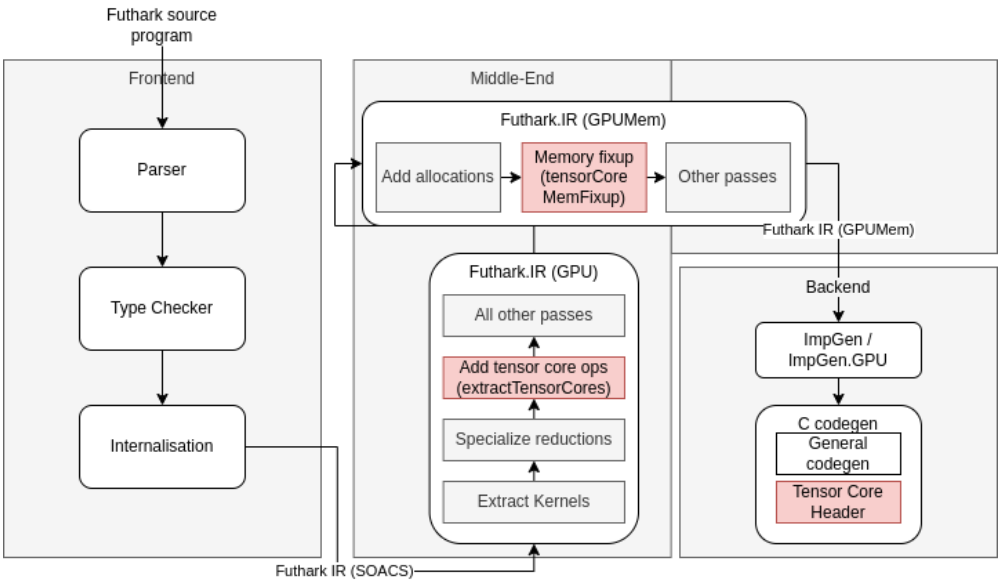

Figure 24: Overview of the compilation pipeline for a GPU program.

The first and second part is most conveniently done on a high level IR program representation without any notion of memory. In the compiler, this IR is known as the `GPU` IR. We implement a new module called `TensorCores` with a pass named `extractTensorCores` working on the high level `GPU` IR. See the corresponding pass in Figure 24. The pass pattern matches intragroup kernels[7], checking if they correspond to matrix multiplication, and transforms the intrgroup kernel into code that uses the Tensor Cores by calling special functions. Whenever the pattern match fails, the code is left unmodified. We go into more detail about how we recognize possible tensor core operations in subsection 4.3 and subsection 4.3 shows each step of our transformation.

Later stages of the Futhark compiler work on an IR representation with memory information. This IR is called `GPUMem` (see Figure 24), and it is at this stage of the compiler that memory allocations are added. For the third part about *massaging* the compiler into accepting our special functions, the generated functions are opaque to the compiler, and the memory allocations might therefore be suboptimal. As an example, it might happen that the compiler allocates the arguments for our special functions in global memory instead of shared memory. This happens because the compiler does not know where our special functions are being called from, and conservatively the arguments are allocated in global memory. We however know exactly how our special functions are to be used, and we can therefore implement another pass `tensorCoreMemFixup` in the same `TensorCores` module that tries to fix up the memory problems created by the compiler. This pass works on the `GPUMem` representation of the IR, and it will run in the GPU memory pipeline of Figure 24. We describe this pass in more detail in subsection 4.4.

For generating code utilizing Tensor Cores, we mostly rely on CUDA C++ templating to generate code for specific problem instances, rather than implementing this part of the code generation in Haskell, as is done in the rest of the Futhark compiler. We do this by adding a CUDA header file with the implementation of three special template functions, and we can then compile this together with the rest of the generated code. Our only modification to the code generation is simply to ignore any function definition with the same name as one of our special functions. Imple-

[7]Code that runs at the level of threads in a CUDA block

menting this part of the code generation using CUDA C++ templating rather than Haskell allows for easier integration with the templating used in CuTe, while still allowing us to parameterize our code generation over e.g. matrix and block sizes determined by our the pattern matching of our initial compiler pass. In subsection 4.2 we discuss the implemented header library in more detail.

We will be using the below code listing as a running example to show our code transformations and the transformation criteria we use. The program represents a batched matrix multiplication with an outer map containing an intragroup kernel corresponding to matrix multiplication. As a brief reminder, when we mention an intragroup kernel, we simply refer to a parallel operation executed by the threads in a CUDA block.

Listing 20:

```
let dotproduct x y =
  map2 (*) x y |> map f32.f16 |> reduce (+) 0
let matmul [m][n][k] (A: [m][k]f16) (B: [k][n]f16) =
    map (\ Arow ->
        map (\Bcol ->
            dotproduct Arow Bcol)
        (transpose B)
    ) A
def matmul_intra [q] (A: [q][16][16]f16) (B: [q][16][16]f16) =
  #[incremental_flattening(only_intra)]map2 matmul A B
```

4.2 Tensor Core Header Library

Using our prototype implementations as a reference, we created 3 functions that could be used in the CUDA code output by the Futhark compiler, allowing it to take advantage of the Tensor Cores. One function for copying data from global to shared memory using special PTX instructions, one function for performing matrix multiplication on the data in shared memory using Tensor Cores and accumulating the result in registers through another PTX instruction, and one for copying the results from registers to shared memory. Figure 25 shows where in the backend the special functions are implemented.

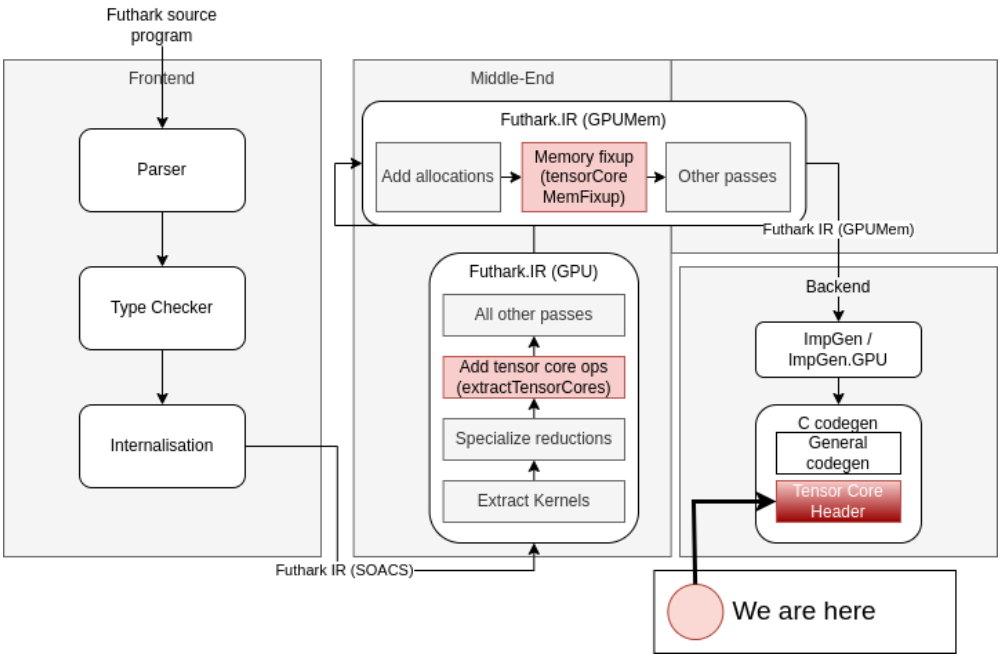

Figure 25: Depiction of where the header library is implemented in the compiler.

The purpose of the copy functions is to allow us to manage the layouts of registers and shared memory using CUDA/CuTe code rather than Futhark IR. This makes it easier to handle architecture- and type specific register layouts, such as the one shown in Figure 26, as well as swizzled shared memory layouts, which would be more complex to do in the Futhark IR. Additionally the copy functions give us full control over the optimizations applied when copying.

T0 V0	T0 V1	T1 V0	T1 V1	T2 V0	T2 V1	T3 V0	T3 V1
T4 V0	T4 V1	T5 V0	T5 V1	T6 V0	T6 V1	T7 V0	T7 V1
T8 V0	T8 V1	T9 V0	T9 V1	T10 V0	T10 V1	T11 V0	T11 V1
T12 V0	T12 V1	T13 V0	T13 V1	T14 V0	T14 V1	T15 V0	T15 V1
T16 V0	T16 V1	T17 V0	T17 V1	T18 V0	T18 V1	T19 V0	T19 V1
T20 V0	T20 V1	T21 V0	T21 V1	T22 V0	T22 V1	T23 V0	T23 V1
T24 V0	T24 V1	T25 V0	T25 V1	T26 V0	T26 V1	T27 V0	T27 V1
T28 V0	T28 V1	T29 V0	T29 V1	T30 V0	T30 V1	T31 V0	T31 V1
T0 V2	T0 V3	T1 V2	T1 V3	T2 V2	T2 V3	T3 V2	T3 V3
T4 V2	T4 V3	T5 V2	T5 V3	T6 V2	T6 V3	T7 V2	T7 V3
T8 V2	T8 V3	T9 V2	T9 V3	T10 V2	T10 V3	T11 V2	T11 V3
T12 V2	T12 V3	T13 V2	T13 V3	T14 V2	T14 V3	T15 V2	T15 V3
T16 V2	T16 V3	T17 V2	T17 V3	T18 V2	T18 V3	T19 V2	T19 V3
T20 V2	T20 V3	T21 V2	T21 V3	T22 V2	T22 V3	T23 V2	T23 V3
T24 V2	T24 V3	T25 V2	T25 V3	T26 V2	T26 V3	T27 V2	T27 V3
T28 V2	T28 V3	T29 V2	T29 V3	T30 V2	T30 V3	T31 V2	T31 V3

Figure 26: Layout of threads and values in registers after $16 \times 8 \times 16$ `f16` or `f16/f32` mixed precision `mma` computation.

When composed, the 3 functions can be used to efficiently implement a range of different programs containing matrix multiplication, including both batched matrix multiplication of relatively small matrices and tiled matrix multiplication of larger matrices. The functions can apply many of the optimizations listed in subsection 2.2, but pipelining and block tiling cannot be performed from within these functions, and must instead be done in the calling code.

In addition to the 3 special functions, the header library also contains some templated structs, used for selecting configurations at compile time, based on the available information of e.g. types and sizes. As an example, Listing 21 shows how the type of `TiledMMA` used is instantiated for `f16/f32` mixed precision `mma` computations. This configuration is specific for the types used in the computation, but general in the size of the output matrix, `SizeM` × `SizeN` and the layout of warps used in the computation, which is `WarpsM` × `WarpsN`.

Listing 21: Mixed precision mma configuration

```
template<class SizeM, class SizeN, class WarpsM, class WarpsN>
struct get_mma_config<half_t, half_t, float, SizeM, SizeN, WarpsM, WarpsN>{
    using MMATraits = MMA_Traits<SM80_16x8x16_F32F16F16F32_TN>;
    using ACopyOpSharedRegisters = SM75_U32x4_LDSM_N;
    using BCopyOpSharedRegisters = SM75_U16x8_LDSM_T;
    using MMATile = Tile<Int<16 * WarpsM{}>, Int<16 * WarpsN{}>, _16>;
    using TiledMMA = TiledMMA<
        MMA_Atom<MMATraits>,
        Layout<Shape<WarpsM,WarpsN,_1>>,
        MMATile
    >;
};
```

The implementation of the functions is otherwise very similar to the code shown in subsection 3.2, but uses this additional templating in an attempt to make it general in the size and types of matrices. Below, we give an overview of the type signatures for our special functions and discuss how they are to be called by the generated `C` code.

4.2.1 CuTe Copy from Global to Shared

Listing 22: Copy global to shared memory

```
template<class ElmTypeIn,
         class SizeY, class SizeX, class WarpsM, class WarpsN>
FUTHARK_FUN_ATTR void futrts_copyGlobalShared(
    unsigned char **mem_out_p, // alias for shared_mem pointer
    unsigned char *global_mem, unsigned char *shared_mem, int64_t offset,
    // Satically known type information needed to use CuTe
    ElmTypeIn, SizeY, SizeX, WarpsM, WarpsN);
```

We implement copying from global to shared with the CUDA C++ function shown in Listing 22. The first template arguments is used to get the element type of the array and the `SizeX` and `SizeY` is used to get the statically known sizes for the matrix to copy. The `WarpM` and `WarpsN` are parameters dictating the layout of the warps used and thereby also the number of warps used. In Listing 19 these values would be `WarpsM=WarpsN=2` for 4 total warps, or 128 threads. These template parameters are all that is needed in order to have the function implement copying from global to shared using structs similar to the ones presented in Listing 18.

For the function parameters, the first argument, `mem_out_p`, is used for the return value of the function. The function writes to the buffer pointed to by `shared_mem` in-place, and the value of the `shared_mem` pointer will be written to the memory location pointed to by `mem_out_p`. The `global_mem` pointer argument points to the array in global memory to read from, while `offset` is the offset within this array that the thread block must load its data from. Note that the remaining function arguments have no parameter names, but are simply the types of the template list. This is because we are only interested in the static type information stored in these types. When the Futhark code generator needs to generate a call to this function it will still pass these function arguments, and the templates are thereby deduced. This means we do not have to modify the code generator with any code that adds explicit template instantiation.

In this function we make use of asynchronous, vectorized copies, and swizzle the layout of the shared memory buffers. This allows fully coalesced access to global memory, and conflict free access to shared memory, using the minimal number of transactions and registers. Both the use of asynchronous copies and a swizzled layout means that code generated by other parts of the compiler cannot use the output of this function. It is therefore implicitly assumed that the result is only used by our special `tensorMMM` function that will be introduced next. It is the job of our compiler passes to make sure that this assumption is adhered to.

4.2.2 Matrix Multiplication

Listing 23:

```
template<class ElmTypeAIn, class ElmTypeBIn, class ElmTypeCIn,
         class SizeM, class SizeN, class SizeK, class WarpsM, class WarpsN,
         class ASwizzled, class BSwizzled, int numRegs>
FUTHARK_FUN_ATTR void futrts_tensorMMM(
    ElmTypeCIn (*mem_out_p)[numRegs], unsigned char *A_mem,
    unsigned char *B_mem, ElmTypeCIn (&C_mem)[numRegs],
    // Type information needed to use CuTe
    ElmTypeAIn, ElmTypeBIn, SizeM, SizeN, SizeK, WarpsM, WarpsN,
    ASwizzled, BSwizzled);
```

The CuTe matrix multiplication function takes more template arguments than the global to shared memory copy. Now the the element type of the input matrices $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ has to be known as well as the type of the output matrix $\mathcal{C}_{mn}$. The corresponding types are stored in the first three template arguments. Next, the `SizeM, SizeN` and `SizeK` template arguments are the statically known sizes m, n and k from the input matrices. We again take two arguments, `WarpsM` and `WarpsN`, for the tiling strategy of the warps in the block. The `ASwizzled` and `BSwizzled` arguments are used as statically known boolean arguemnts, indicating if the $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ arrays, respectively, have a swizzled layout. Finally, we take a `numRegs` argument, storing the number of results computed by each thread.

The arguments follow a similar pattern as the global to shared copy function. The first argument, `mem_out_p` is is used to output the output memory buffer. The output memory buffer will be the `C_mem` input argument, which is used to store the matrix multiplication result. The `A_mem` and `B_mem` arguments are pointers to the shared memory buffers storing $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$. The remaining arguments are again simply used to infer the template arguments. Again, the function implementation looks similar to the example given in Listing 19.

In this function, warp and register tiling is performed on the data in shared memory. Additionally, assuming the $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ matrices contain elements of type `f16`, the `ldmatrix` PTX

instruction is used to load data from shared memory to registers efficiently. If the input arguments use a swizzled layout, i.e. are the result of the `copyGlobalShared` function, as indicated by the `ASwizzled` and `BSwizzled` arguments, loads from shared memory are bank conflict free.

4.2.3 Cute Copy Registers Shared

Listing 24: Copy shared memory to registers

```
template<class ElmTypeAIn, class ElmTypeBIn, class ElmTypeCIn,
         class SizeM, class SizeN, class WarpsM, class WarpsN, int numRegs>
FUTHARK_FUN_ATTR void futrts_copyRegistersShared(
    unsigned char **mem_out_p, ElmTypeCIn (&registers_mem)[numRegs],
    unsigned char *shared_mem, ElmTypeAIn,
    ElmTypeBIn, SizeM, SizeN, WarpsM, WarpsN)
```

The type signature for our copy from registers to shared is given in Listing 24. The template list includes first the element type for all matrices that were involved in the mma. At first, this might seem a bit strange, since the element type of $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ should not have anything to do with writing the result back to shared memory. However, in order for each thread to write back its result in shared memory at the correct location, we need to rebuild the thread to value mapping. This requires us know the element type of $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ in addition to $\mathcal{C}_{mn}$. An example of this thread-value mapping was shown in Figure 26. Note that not all values for each thread are stored consecutively, which makes it hard to express this pattern in Futhark IR.

Note that the thread to value mapping is encoded in C++ template types and it is therefore completely statically known with no runtime overhead. The remaining template arguments are statically known sizes of the output matrix, the layout of warps needed for the mma calculation and the number of registers per thread used. The first argument is again a pointer for where to store the output memory buffer. Similar to the other copy function, this will be set to the value of the `shared_mem` argument pointing to an allocated shared memory buffer. The `registers_mem` is the per thread result in registers. Like the previous copy function from global to shared, the remaining function arguments have no parameter name because we are only interested in automatically deducing the template arguments.

This function makes use of vectorized copies to shared memory, but does not use a swizzled layout, which means it will result in bank conflicts. The reason we do not swizzle the layout is to ensure that the result can be used by code generated by other parts of the Futhark compiler.

4.3 High Level IR transformations

For convenience, the compiler overview is shown again in Figure 27 below. As mentioned in the initial roadmap, we pattern match an intragroup kernel corresponding to matrix multiplication on the high level IR representation without memory. Figure 27 highlights that the transformation criteria runs within a new `extractTensorCores` pass on the `GPU` representation of the IR.

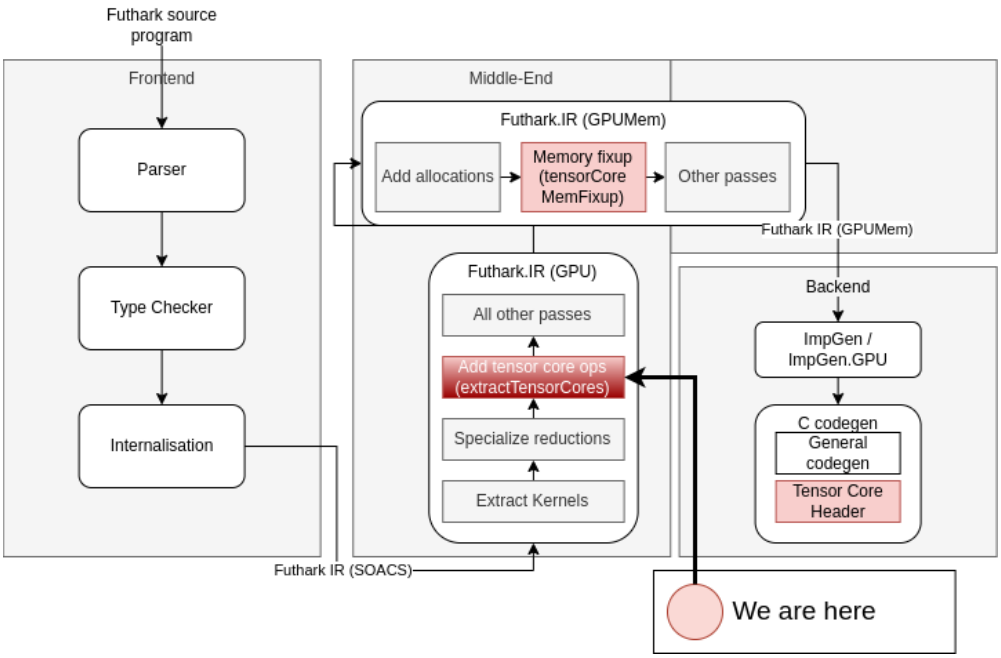

Figure 27: Compiler pass implementation within the compiler.

To understand how the pattern match for the transformation criteria works, and what our code transformations look like, it is necessary to have an understanding of the Futhark IR. At this stage of compilation, all parallelism is expressed in terms of four flat *segmented operations* (SegOps) shown below:

Listing 25:

```
data SegOp lvl rep
  = SegMap lvl SegSpace [Type] (KernelBody rep)
  | SegRed lvl SegSpace [SegBinOp rep] [Type] (KernelBody rep)
  | SegScan lvl SegSpace [SegBinOp rep] [Type] (KernelBody rep)
  | SegHist lvl SegSpace [HistOp rep] [Type] (KernelBody rep)
  deriving (Eq, Ord, Show)
```

Each SegOp is a perfect nest of maps with some bottommost computation like a scalar computation or a reduction, scan or reduce by index. Consider again the matrix multiplication program below. The `matmul` function has two outer maps with a `reduce` using addition as the associative operator on a row of `A` and a column of `B`.

This will be represented in the IR as a `SegRed` where the `SegSpace` encodes the outer `maps`, the `[SegBinOp]` list will be addition, the `[Type]` will be the returned type of each thread (a single `f32` in this case) and the `KernelBody` corresponds to the `map2` computation and type conversion. The two levels of parallelism on the GPU (blocks and threads) is represented by the `lvl` of the `SegOp`. When the `matmul` function is an intragroup kernel, the `lvl` will be threads in a block. This `SegRed` will then be in the body of a block-level `SegMap` corresponding to the `map2` in the `matmul_intra` function.

Listing 26:

```
let matmul [m][n][k] (A: [m][k]f16) (B: [k][n]f16) =
    map (\ Arow ->
        map (\ Bcol ->
            -- dot product
            map2 (*) Arow Bcol |> map f32.f16 |> reduce (+) 0
        (transpose B)
    ) A
def matmul_intra [q] (A: [q][16][16]f16) (B: [q][16][16]f16) =
  #[incremental_flattening(only_intra)]map2 matmul A B
```

A pretty printed version of the GPU IR code is shown in Listing 27. We will use this in conjunction with the Futhark source program to show the transformation and pattern matching done by our extractTensorCores pass.

Listing 27:

```
let {mmm_intra_result : [q][16i64][16i64]f32} =
    #[incremental_flattening(only_intra)]
    segmap(block; ; grid=q; blocksize=4096i64)
    (gtid_block < q) (~phys_tblock_id) : {[16i64][16i64]f32} {
      let {intragroup_result : [16i64][16i64]f32} =
        -- SegRed
        segred(inblock; )
        (gtid_thrd_m < 16i64, gtid_thrd_n < 16i64, gtid_thrd_k < 16i64)
        (0.0f32, f32.add) -- binary operator and neutral element
        {
          -- Kernel body
          let {a : f16} =
            A[gtid_block, gtid_thrd_m, gtid_thrd_k]
          let {b : f16} =
            B[gtid_block, gtid_thrd_k, gtid_thrd_n]
          let {mul_res : f16} =
            fmul16(a, b)
          let {conversion : f32} =
            fpconv f16 mul_res to f32
          return {returns conversion}
        }
      return {returns intragroup_result}
    }
  in {mmm_intra_result}
```

We can reason that for some SegOp to match matrix multiplication, it must have a reduction over the k dimension of a $C_{mn} = \mathcal{A}_{mk}\mathcal{B}_{kn}$ matrix multiplication as the bottommost computation[8]. When pattern matching a SegOp we can therefore restrict our attention to all SegRed at the thread in block level with addition as the only binary operator. The implementation code for the pattern match is given below. Note that this is exactly the case for the IR code above. There is a SegRed running at the threads in blocks with an outer map with over the CUDA blocks.

[8]Recall that a SegOp is a perfect nest of maps with some bottommost computation (scalar, reduce, scan or histogram computation)

Listing 28:

```
innerOpMatch :: Scope GPU -> Op GPU -> Maybe InnerMMAMatch
innerOpMatch scope
  ( SegOp
     (SegRed (SegThreadInBlock _) space segBinOps _ts kernelbody)
   )
   -- checks this is addition with neutral element 0.0f16 or 0.0f32
   | Just ne <- segBinOpsMatch segBinOps =
     -- implementation body
```

Next, we need to look into the `KernelBody` of the `SegRed` to make sure that the reduced arrays were a result of multiplying the row and columns of two matrices. We do this through a backwards traversal of the statements in the `KernelBody` starting at the kernel result. For us to match matrix multiplication, we then require of the kernel body that:

1. the kernel body result is of type `f16`, or the result of a conversion from `f16` to `f32`.
2. the result comes from a product of the form `a * b`
3. both `a` and `b` are the result of indexing into some arrays `A` and `B` respectively
4. The indices into `A` and `B` are variant to exactly two dimensions of the `SegSpace` in the `SegRed`
5. The indices into `A` and `B` are variant to one common dimension (the reduction dimension) of the `SegRed`
6. The indices into `A` and `B` are each invariant to exactly one parallel dimension of the `SegSpace` in the `SegRed`.
7. Matrix `B` is transposed such that true matrix multiplication $C_{ij} = \sum_k A_{ik} B_{kj}$ is computed.

Since we are focusing on matrix multiplication in intragroup kernels, we allow the arrays to be indexed by potentially arbitrarily many *outer indices*. In the running Futhark example above in Listing 26, these outer indices would be used to index into the `q` dimension of $\mathcal{A}$ and $\mathcal{B}$ to slice out the $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ sub arrays used in matrix multiplication. We also see this is reflected in the IR code where both arrays are indexed by the `gtid_block` as the outermost index. The kernel body of the Listing 26 also meets all other requirements for our match since:

1. The result is a conversion from `f16` to `f32`
2. the result was a product of the two variables `a * b`
3. `a` and `b` are variables resulting by indexing arrays `A` and `B` respectively
4. `a` is variant to `tid_thrd_m, gtid_thrd_k` and `b` is variant to `gtid_thrd_k, tid_thrd_m`
5. both `A` and `B` are indexed by `gtid_thrd_k` and they are thereby variant to a common dimension of the `SegRed`.
6. `A` and `B` are both indexed by the outer index, `gtid_block`, making them variant to the outer most map over the `q` dimension.

7. We also see that `B` must be transposed due to the indexing into `B`. That is `gtid_thrd_k` appears as the second last indexing into `B`.

We further restrict the inner dimensions m, n, k of $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ to be specific, statically know small sizes. This restriction is made in order to make sure that all matrices can fit in shared memory, and can be perfectly partitioned to fit in the Tensor Cores. Finally, this all means that a `SegRed` running at the level of threads in a block has to be done using:

- mixed precision matrix multiplication of `f16` and `f32`.
- or purely `f16` matrix multiplication is performed.
- statically known dimensions ds.
- each dimension $d \in ds$ must be a multiple of 16 in the range $16 \leq d \leq 128$.

Note that in the IR example, all bounds of the `SegRed` are 16 and the intragroup matrix multiplication therefore matches all of our requirements.

In the implementation of our pattern matching function `innerOpMatch` the result is a `Maybe InnerMMAMatch`. In case our match succeedes, the function returns a `Just InnerMMAMatch` with information about how the `SegRed` can be transformed to utilize Tensor Cores, and `Nothing` if any of the above criteria did not match. For instance, if one of the dimensions is not statically known, we do no transformations to utilize the tensor cores. The `InnerMMAMatch` record has the following fields:

Listing 29:

```
data InnerMMAMatch = InnerMMAMatch
  {
    kernelBodyMatch :: KernelBodyMatch,
    ne :: SubExp,
    sizeM :: Int,
    sizeN :: Int,
    sizeK :: Int
  }
```

The `KernelBodyMatch` record is a helper type containing indexing information about how the $\mathcal{A}$ and $\mathcal{B}$ arrays were indexed. This together with the three size fields lets us reconstruct a semantically equivalent program using Tensor Cores.

After having successfully found matrix multiplication in an intragroup kernel, we need some way of transforming the high level `GPU` IR into code that uses the Tensor Cores. One possible way is to add a new type of GPU operation. Different IR representations of the compiler support different operations, and the new Tensor Core GPU operation would simply be another operation supported by the `GPU` representation of the IR.

One of our goals is however to find the simplest and most minimally invasive way to introduce Tensor Cores to the compiler. In cases where we have to add something to the compiler, we ideally want this to be isolated and not interact with other parts of the compiler. Adding compiler passes fits within these goals, since they are isolated code to code transformations. Introducing a new GPU operation seems to fall out of line with these goals as this entails potentially modifying many parts of the compiler. For this reason, we choose to add Tensor Core support

through special function calls. The compiler cannot optimize these functions calls away in later passes as they appear as opaque constructs. There is nothing inherently special about these new functions. They have a uniquely defined name that we rely on at code generation, and a signature that allows us to implement everything related to Tensor Cores in a separate CUDA header file. Generating these function calls happens immediately after successful pattern matching and in the same `extractTensorCores` pass.

Returning to the example in Listing 27, the transformation needs to replace the inner `SegRed` with semantically equivalent code that makes use of our special functions. Such a transformation could look like IR code in Listing 30. This transformation has replaced the `SegRed` with function calls that:

1. copies matrix $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ to shared memory.
2. perform the Tensor Core matrix multiplication
3. copy the result back into shared memory from registers.

The transformation is however not always as straightforward as the code in Listing 30. We go into more detail in the next subsections about how each of the three steps are implemented.

Listing 30:

```
let matmul_intra_result : [q][16i64][16i64]f32} =
    #[incremental_flattening(only_intra)]
    segmap(block; ; grid=q; blocksize=32i64)
    (gtid < q) (~phys_tblock_id) : {[16i64][16i64]f32} {
    --
    -- segmap that zero initializes registers here
    ---
      let {aCopied : [16i64][16i64]f16} =
        apply copyGlobalShared(...)
        : {*[16i64][16i64]f16}
      let {bCopied : [16i64][16i64]f16} =
        apply copyGlobalShared(...)
        : {*[16i64][16i64]f16}
      let {inBlockMMAres : [32i64][8i64]f32} =
        segmap(inblock; )
        (ltid < 32i64) (~ltid_flat) : {[8i64]f32} {
          let {threadMMAres : [8i64]f32} =
            apply tensorMMM(...)
            : {*[8i64]f32}
          return {returns (private) threadMMAres_6494}
        }
      let {cCopied : [16i64][16i64]f32} =
        apply copyRegistersShared(...) : {*[16i64][16i64]f32}
      return {returns cCopied}
    }
```

4.3.1 Copy global to shared

It might not in all cases be necessary, or even correct, to copy from global to shared. The `SegRed` might be preceded by some `code1` that contains more intrablock kernels with result $\mathcal{A}'_{mk}$ and $\mathcal{B}'_{kn}$. Consider this in the below example:

Listing 31: Intragroup matmul sorrounded by code1 and code2

```
map2 (\A B ->
    let A' = map f A -- code 1
    let C = matmul A' B -- SegRed matrix multiplication
    in map g C -- code 2
) As Bs
```

The matrix multiplication uses $\mathcal{A}'_{mk}$, and since code 1 will be an intragroup kernel, it is very likely that the compiler allocates $\mathcal{A}'_{mk}$ in shared memory. In such a case it would be inefficient and wasteful of shared memory to do a new copy of $\mathcal{A}'_{mk}$ into a new shared memory buffer. At the `GPU` representation of the IR, there is however no notion of memory and it is unknown if $\mathcal{A}'_{mk}$ is allocated in shared memory. For this reason, we always allocate a shared memory buffer for $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$, and then rely on a later pass on the `GPUMem` representation to fix and remove the copy to shared. We expand upon this in subsection 4.4.

Below we give the fully generated code for copying from global memory to shared for matrix $\mathcal{A}_{mk}$, including allocating the shared memory buffers of the running example:

Listing 32:

```
-- shared memory allocation
let {aScratch : [16i64][16i64]f16} = scratch(f16, 16i64, 16i64)
let {aCopied : [16i64][16i64]f16 } =
    copyGlobalShared(A, *aScratch, offsetA, 0.0f16, 16i64, 16i64, 1i64, 1i64
        )
    : {*[16i64][16i64]f16}
```

The arguments to the function call will match the correspondingly generated CUDA C++ template function that was shown earlier. In order the arguments are

1. The entire array $\mathcal{A}$
2. Allocated shared memory buffer,
3. Offset into $\mathcal{A}$ for each thread block to use
4. Place holder zero element to store the element type of the array.
5. Size m of the $\mathcal{A}_{mk}$ matrix
6. Size n
7. Warps in the m dimension of the output
8. Warps in the n dimension of the output.

The return type of the function is a `[16i64][16i64]f16` array corresponding to the copied result. We make sure that the returned result will use the same memory as `aScratch`, such that any usage of the result variable `aCopied` uses the copied shared memory buffer.

4.3.2 Setting the Block Size for Matrix Multiplication

For the element types supported by our transformation, the Tensor Cores operate on $16 \times 8 \times 16$ matrices using 32 threads. The IR code in Listing 27 has a block size of $16 \cdot 16 \cdot 16 = 4096$, since the `SegSpace` of the `SegRed` goes over 3 parallel dimensions each of size 16. This is many more threads than required to do the matrix multiplication with Tensor Cores, and also more than the maximum for a single CUDA block. Also note that for $16 \times 8 \times 16$ matrix multiplication, each thread computes 4 output elements. Therefore, even for the simple program in Listing 26, we have to handle the excess number of threads. Since we only need 32 threads for the matrix multiplication in this example, we can simply set the block size to 32. This is safe for the program in Listing 26 because there is only one nested `SegOp` corresponding to matrix multiplication. In other cases it might however not always be safe to do so. Consider the futhark program below. Some `code1` precedes the matrix multiplication and some `code2` follows it. If either `code1` or `code2` needs more than 32 threads, then shrinking the block size will result in wrong results. However, if both `code1` and `code2` needs 32 or fewer threads, then changing the block size is safe because the block size was bounded by the threads required for matrix multiplication.

Listing 33:

```
#[incremental_flattening(only_intra)]
map2 (\A B ->
    let A' = map f A -- code 1
    let C = matmul A' B -- SegRed matrix multiplication
    in map g C -- code 2
) As Bs
```

Depending on the size of the matrix multiplication, we might need more than 32 threads to efficiently use the Tensor Cores. When benchmarking our different prototypes, we found that each thread in general needs to do a relatively large amount of work. Typically, each thread needs to compute 64 or 128 output results to achieve the best performance. With this in mind, our pass first calculates the optimal block size for matrix multiplication as

$$blk^* = \left\lceil \frac{m \cdot n}{4096} \right\rceil \cdot 32 \tag{3}$$

Here $128 \cdot 32 = 4096$ corresponds to the optimal number of elements per warp. Note we know that m, n are bounded in $16 \leq m, n \leq 128$, due to the criteria for our match. As a result, the max block size needed for matrix multiplication is 128. We can use the `IntraMMAMatch` record to get the sizing information needed to calculate the above result.

After finding the optimal block size, blk^*, we adjust the block size of the outer `SegOp` as $\max(\text{blokSize(code1)}, blk^*, \text{blokSize(code2)})$. In case more than blk^* threads are need by `code1` or `code2` we attempt to also make use of these threads in the matrix multiplication, even though this may be more than the optimal number of threads for the matrix multiplication in isolation. There is, however, a limit to how many threads we can make use of, since the Tensor Cores require a minimum amount of sequentialization. As a consequence, in some cases, some threads may be idle during the matrix multiplication.

Ideally, both the matrix multiplication and the surrounding code should use the same number of threads, and this should also be the number of threads giving the best possible performance. However, the exact amount of threads that gives the best overall performance can be hard to figure out, and depends very much on the surrounding code. Additionally, while we can fairly

easily control the amount of sequentilization in our matrix multiplication code, it is not trivial to sequentialize arbitrary pieces of code in a general way. This means that in order to get the very best performance, the programmer may need to sequentialize `code1` and `code2` by hand, in order to get an optimal amount of parallelism.

Some work has already been put into doing this sort of general sequentialization automatically in the compiler [23], but including that into this work is out of scope.

In order to change the block size, we need all the intragroup kernels (meaning `code1` and `code2`) to also have parallel dimensions of statically known sizes, otherwise, we cannot statically change the block size. Therefore, with the presence of some `code1` and `code2`, our transformation only triggers if everything has static sizes.

Once we have decided on a block size, we must also decide how to divide the matrix multiplication between the threads, i.e. decide which threads compute what part of the resulting matrix. When there is no other code to take into account this can simply be done using a calculation like the one shown in Equation 3. However, it gets slightly more complicated if we want to try to use the same amount of threads as the surrounding code. In this case, we must find the best way to divide an $m \times n$ matrix into W submatrices, where W is the number of warps available for the calculation, since Tensor Core matrix multiplication is done at the warp level. We do this by choosing the pair of numbers (W_m, W_n) such that $W_m \cdot W_n = W$, for which the ratio $\frac{W_m}{W_n}$ is as close as possible to $\frac{m}{n}$. We then split the result matrix into W_m equal parts along the m dimension, and W_n equal parts along the n dimension, and assign a warp to each submatrix in this grid. If it is not possible to divide both dimensions evenly into parts that are multiples of 16, we decrement the number of active warps and try again.

4.3.3 Tensor Core Matrix Multiplication

Recall the usage of the `mma` ptx instruction: All threads in a warp calls the `mma` instruction supplying a slice of $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ in registers and a set of registers for the output $\mathcal{C}_{mn}$. Similarly to this, we want each thread in the block to call a Tensor Core matrix multiplication function, supplying a set of registers for the per thread output of $\mathcal{C}_{mn}$. We can implement this by a `SegMap`, with a parallel dimension equal to the chosen block size This is shown in Listing 30, where the block size is 32, for a $16 \times 16 \times 16$ matrix multiplication, and the `SegMap` has a matching parallel dimension. Also note that the per thread result of the map is `[8]f32`, because each thread computes $16 \cdot 16/32 = 8$ output elements.

Listing 34 shows what the fully generated IR function call for matrix multiplication would look like in Listing 26. Each thread supplies the full copy in shared memory of the slice $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ to use, the registers that the thread result should be stored in, `0.0f16` for the element type of the two matrices, and at last the matrix dimensions and required number of threads to use. The result of the function call is simply the supplied `threadCregs`. The CUDA implementation of `tensorMMM` is responsible for loading a slice $\mathcal{A}_{mk}$ and $\mathcal{B}_{kn}$ into registers. Our transformation allocates enough registers to hold the entire $\mathcal{C}_m$ matrix in registers, and we can therefore use these registers directly in our function call. In the future it might be possible to reuse these registers instead of having to write the register result to shared memory after each matrix multiplication. This is further discussed in section 6.

Listing 34:

```
let {thrd_MMA_res: [8i64]f32} =
  tensorMMM(aCopied, bCopied, *threadCregs, 0.0f16, 0.0f16, 16i64, 16i64, 16
     i64, 32i64)
```

4.3.4 Storing the Result in shared memory

Finally, our transformation copies the per thread results in registers to shared memory. For the running example in Listing 27, the function call looks as follows:

Listing 35:

```
copyRegistersShared(inBlockMMAres, *cScratch, 0.0f16, 0.0f16, 16i64, 16i64,
    32i64)
```

The `inBlockMMARes` is the per thread result in registers, and `cScratch` is the shared memory buffer to copy into. This will also be the result of the function call. The remaining arguments are as usual used for type information, matrix dimensions and the number of threads to use.

One benefit of always copying to shared memory is that the compiler will automatically copy the shared memory buffer back to global memory as part of the outermost `SegMap` result. It can also be beneficial when some `code2` follows the matrix multiplication and uses its result, since the compiler would otherwise very likely copy the result to shared memory either way. In case the matrix multiplication result was needed in global memory, then an indirect copy to shared memory first has a relatively small performance cost. The disadvantage with this approach is that writing to shared memory is much slower than directly using the in-register matrix multiplication result. As an example, consider the attention like program in Listing 36. The result of the matrix multiplication is immediately afterwards used to update an accumulator. This `map2` code will need to load the matrix multiplication result back from shared memory to registers, even though we might have been able to avoid it.

Listing 36: Flash attention like program

```
def attention_like [q] (A: [m][k]f16) (B: [q][k][n]f16) : [m][n]f32 =
  let acc_init : *[m][n]f32 = replicate (m * n) 0.0f32 |> unflatten in
  loop (acc : *[m][n]f32) = (acc_init: *[m][n]f32) for i < q do
    let C' = matmul A B[i]
    in map2(map2 (+)) acc C'
def main [q][p] (A: [p][m][k]f16) (B: [p][q][k][n]f16) =
  #[incremental_flattening(only_intra)]map2 attention_like A B
```

Although the `map2` might have been possible to compute without loading the matrix multiplication result from shared memory, doing so is not straightforward. Consider the layout of values (in registers) and threads of the result matrix C in Figure 28. The matrix multiplication result in registers has a permuted layout between threads and values where the per thread result is scattered around. If the `map2` were to directly use this layout of values in registers, then the `acc` array should have the same layout of values in registers. For this reason, we always do a write back to shared memory.

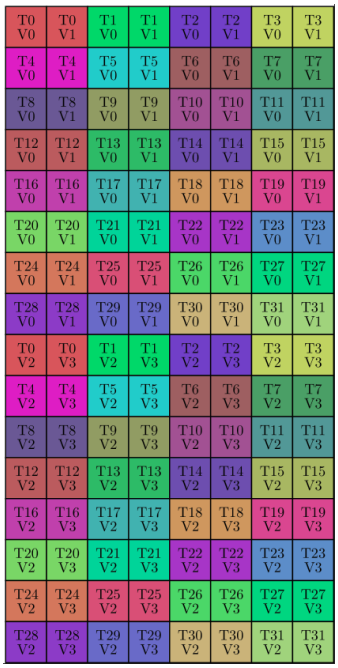

Figure 28: Layout of threads and values of the result matrix C from a $16 \times 8 \times 16$ matrix multiplication.

4.4 Transformations on IR With Memory

After the high level `GPU` IR representation, the Futhark compiler transforms the code into a new `GPUMem` IR representation that has a notion of memory. Figure 29 shows that the conversion from `GPU` IR to the more lower level `GPUMem` IR happens within the very first pass of the `GPUMem` pipeline that adds allocations. Since our special functions are opaque to the compiler, the allocated memory is often suboptimal either because it is allocated as global memory when it could have been shared memory or because a redundant copy takes place. We implement a new pass `tensorCoreMemFixup` that reduces the number of memory copies, see Figure 29, and also modify the "Add Allocations" pass to treat our functions differently from all other function calls. We describe these two modifications in the following subsections.

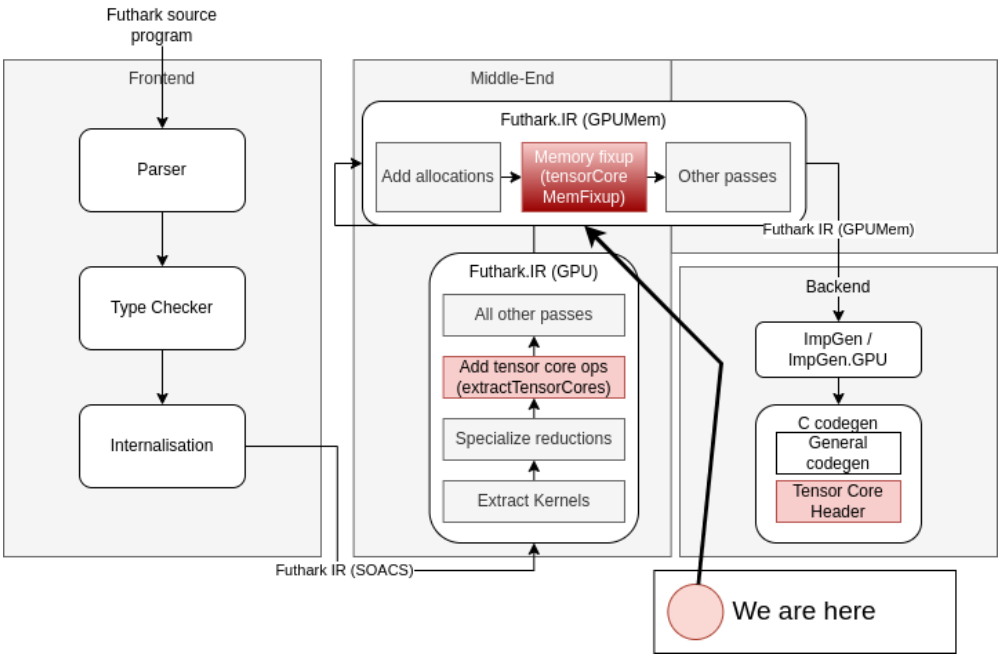

Figure 29: New compiler pass added for the `GPUMem` representation is highlighted in dark red. The add allocations pass will also be slightly modified.

4.4.1 Modifying the Default Allocation Space

The Futhark compiler does not do any form of analysis to determine where a function can be called from. As a consequence, all kernel functions must have their arguments in global memory such that they can be called either from host or device code. This is however problematic for our special functions. Consider the type signature for `copyGlobalShared` generated by the compiler when the program is transformed from IR on the `GPU` representation into IR on the `GPUMem` representation:

Listing 37:

```
1 fun copyGlobalShared (
2     global_mem : mem@device,
3     shared_mem : mem@device,
4     global : [16i64][16i64]f16 @ global_mem -> {lmad_info1},
5     shared : *[16i64][16i64]f16 @ shared_mem -> {lmad_info2},
6     offset: i64, elmTypeA: f16, Y : i64, X : i64, blockSize : i64)
7     -> {mem@device, lmad_info3}
```

The two first arguments `global_mem` and `shared_mem` represent the underlying memory for the arrays in argument `global` and `shared`. The special `mem@device` denotes where the memory block has been allocated, and the `[16i64][16i64]f16 @ global_mem` denotes that the backing memory for the array is found in the `global_mem` memory block. This shows that the compiler as expected has put all array argument into global memory.

For this particular function, this compiler limitation can be fixed by implementing a new pass on the `GPUMem` IR. The pass needs to both fix the type signature of the function definitions, and all function calls to `copyGlobalShared` such that the `shared_mem` argument is given a shared memory buffer. We did successfully implement such a pass that fixed all our special functions and function calls, but later found cases where compilation still fails due to mismatch in the expected

memory space and actual memory space. In particular, programs that have some intragroup kernel, `code2`, that uses the result of matrix multiplication will never work. The generated `GPUMem` IR code for `code2` was generated under the assumption that the matrix multiplication result is in global memory. If we only fix the memory space for our special functions, then there will be a type mismatch between the expected memory space of `code2` and the actual memory space after the memory fixup. For this reason, we found it necessary to slightly modify the allocation algorithm to handle our special functions differently. When memory has to be allocated for `code2`, the memory allocator can use that the matrix multiplication result is in shared memory to correctly allocate the memory needed to execute `code2`.

The allocation pass is modified in two places: function definitions, and function call expressions. Below we show how we modify the compiler to handle allocations differently for our special functions. Instead of always putting the arguments in some default space (global memory for the `GPU` representation) we match the function name against our special functions. In case the function name matches, we modify the memory space for the application appropriately. Specifically, we know that the first argument is a memory block that holds the entire $\mathcal{A}$ array, and it should therefore be forced to device space (line 5). The second argument is a memory block for the $\mathcal{A}_{mk}$ to be copied into shared. We force this argument to be in shared shared memory on line 6. The remaining arguments are unchanged, be making them `Nothing`. Line 7 also specifies that the function returned memory block must also be in shared memory. The helper function `funCallArgs`, checks whetther there is a `Just forcedSpace` for each argument, in which case it uses the forced memory space instead of the default space. The approach to fix function definitions is very similar.

Listing 38:

```
1  -- Allocations needed to handle function application
2  allocInExp (Apply fname args rettype loc) = do
3    space <- askDefaultSpace -- default space is global memory
4    (forced_arg_spaces, retSpace) <-
5      if MMM.copyGlobalSharedName `MMM.isPrefixOfName` fname then
6          pure ( [Just $ Space "device",
7                  Just $ Space "shared"]
8                 <> replicate (length args - 2) Nothing
9               , Space "shared")
10     else if -- any other special function match
11     else -- default case
12       pure (replicate (length args) Nothing, space)
13   -- Create new function arguments possibly using a forced memory space
14   args' <- funcallArgs args forced_arg_spaces
```

4.4.2 Reduce memory copies

Modifying the allocation spaces for our generated special functions does not solve all memory related problems. We frequently see that the allocation pass produces `A_desired_form = manifest(A)` operations that copy `A` into `A_desired_form`. We see memory manifestations in two cases:

1. Copy from global to shared
2. Copy from registers to shared

Recall in Listing 37, the input array in global memory had the type `[16i64][16i64]f16 @ global_mem -> lmad_info1`. The `lmad_info1` type describes a linear-memory accessor descriptor (LMAD) [20] with information about the array offset and for each dimension a stride and shape. When we generate the `copyGlobalShared` call, we do not supply the entire $\mathcal{A}$ and $\mathcal{B}$ arrays as we have otherwise shown. Instead we generate code that slices out the subarray corresponding to the $\mathcal{A}_{mk}$ and $\mathcal{B}_{nk}$ matrices:

Listing 39:

```
1 -- slice out a matrix from A
2 let {slicedA : [16i64][16i64]f16} = A[gtid_block * 256, :, :]
3 let {aCopied : [16i64][16i64]f16 } =
4     copyGlobalShared(slicedA, *aScratch, offsetA, 0.0f16, 16i64, 16i64, 32
         i64) : {*[16i64][16i64]f16}
```

The slicing notation `A[gtid_block * 256, :, :]` is not real IR syntax, but a much more convenient notation that is also used in Futhark the language and Python. Slicing an array can be a *free* operation only requiring re-indexing by the compiler. However, when given as a function argument, this requires that the sliced array is row major with zero offset. Our implementation slices out a matrix at an offset corresponding to `gtid_block * 256`, thereby forcing the compiler to generate a redundant copy.

A similar issue occurs when calling the `copyRegistersShared(inBlockMMAres, *cRegs,...)` function. Recall from subsection 4.3 that each thread in a parallel `map` calls a special function corresponding to Tensor Core matrix multiplication, storing the result in registers. This unfortunately also causes a redundant copy to another set of registers before calling `copyRegistersShared`.

All these redundant memory copies can luckily be reduced by our `tensorCoreMemFixup` pass. For reference, see Figure 29 for the placement of the pass in the compiler. The pass iterates over the statements of the program, and fore every `A_desired_form = manifest(A)` statement, the relation that array `A_desired_form` is a copy of array `A` is recorded in a hash map $\mathcal{H}$. Similarlly, the underlying memory block that holds `A_desired_form` and `A` is also recorded in $\mathcal{H}$.

The hash map is then used to change the arguments for all of our special function calls that uses the `A_desired_form` and its underlying memory block. Consider the below pretty printed `GPUMem` IR code generated by our compiler:

Listing 40:

```
1 let {slicedA : [16i64][16i64]f16 @ mem_slicedA -> lmad} = A[gtid_block *
     256, :, :]
2 let {slicedA_desired_form :  [16i64][16i64]f16 @ mem_slicedA_desired_form ->
      lmad } =
3     manifest(slicedA) -- copies slicedA
4 let {aCopied : [16i64][16i64]f16 } =  copyGlobalShared(
5   slicedA_desired_form, *aScratch, offsetA, 0.0f16, 16i64, 16i64, 32i64) :
       {*[16i64][16i64]f16}
```

The code shows that `slicedA` is copied into `slicedA_desired_form` with `mem_slicedA_desired_form` as the backing memory. When calling the `copyGlobalShared` function, the hash map $\mathcal{H}$ will therefore hold the following relation:

$$\mathcal{H} = \begin{cases} \texttt{mem\_slicedA\_desired\_form} & \rightarrow \texttt{mem\_slicedA} \\ \texttt{slicedA\_desired\_form} & \rightarrow \texttt{slicedA} \end{cases}$$

Using this hash map, the parameters for `copyGlobalShared` can now be replaced, and `slicedA` and `mem_slicedA` is used directly:

Listing 41:

```
let {slicedA : [16i64][16i64]f16 @ mem_slicedA -> lmad} =
  A[gtid_block * 256, :, :]
-- Downstream compiler pass has removed the manifest stement
let {aCopied : [16i64][16i64]f16 } =  copyGlobalShared(
  slicedA, *aScratch, offsetA, 0.0f16, 16i64, 16i64, 32i64) : {*[16i64][16i64]f16}
```

The pass works the same for all our special functions, and we rely on a downstream pass to recognize that `slicedA_desired_form` and its corresponding memory block `mem_slicedA_desired_form` are unused and can safely be removed.

With this small memory fixup, there is no wasted memory resulting from using the Tensor Cores through calls to special functions.

4.5 Using the Modified Compiler

The full modified compiler can be found at `https://github.com/caymand/futhark/tree/intragroup-mmm`. Additionally the core files for our added passes can be seen in Appendix A. However, it should be noted that these are not the only changes made to the compiler.

Compiling a Futhark program requires the user to specify a compilation *action*. Possible actions include `cuda`, `opencl`, `c` or even `python`. For Futhark utilties such as `futhark test` and `futhark bench`, these action are chosen using the `-backend` option. Instead of having our changes as part of the `cuda` action, we choose to make a new compilation action named `cudatc` that runs the `cuda` pipeline and our passes. This is the exact compilation pipeline we presented in Figure 24. Programs compiled compiled with the existing `cuda` action will thereby not use any of our new passes and the Tensor Cores are therefore not used in this case. Not all NVIDIA GPUs have Tensor Cores, and we therefore put the responsibility of checking the GPU requirements on the user. Compiling a Futhark program with Tensor Core support can be done with the below command:

```
futhark cudatc program.fut
```

This generates a compiled program called `program`. When a Futhark program is compiled with the `cuda` or `cudatc` actions, the CUDA kernels will at runtime be compiled through NVRTC, a runtime CUDA compilation library. Directly running this program with `./program` will fail because NVRTC cannot find the appropriate Cutlass header files. The user must pass a flag to the compiled program with the include path of Cutlass:

```
./program --nvrtc-option=-I<Cutlass include path>
```

We found this to be another good reason to make our changes part of a different compilation pipeline. Benchmarking or testing futhark program compiled with the `cudatc` action requires the user to supply the Cutlass include path. This can be done with:

```
futhark test --pass-option=--nvrtc-option=-I<Cutlass include path> \
  program.fut
```

5 Experimental Evaluation

This section sets out to describe how the modified compiler has been tested and benchmarked. We first describe the overall testing and benchmarking setups, and subsequently go into more detail and show results for a selection of Futhark programs used as case studies. We detail why the individual Futhark programs have special interest for our implementation and discuss the strengths and weaknesses of our compiler modifications for these programs.

5.1 Testing Methodology

Testing of the implementation has been done through blackbox testing of programs that have an intragroup kernel corresponding to matrix multiplication. The programs are compiled with the `cudatc` compilation action, and validation is performed by comparing the result to the result of the sequential `C` compilation action. The tests can be found in the futhark testing suite at `tests/tensor-cores` at the git repository `https://github.com/caymand/futhark/tree/intragroup-mmm`. The tests are straightforwardly run with the command:

```
futhark test --backend=cudatc \
    --pass-option=--nvrtc-option=-I<Cutlass include path> \
    tests/tensor-cores
```

All the programs that were used for benchmarking are included in our testing suite. In the next sections we describe the programs that are used as for benchmarking of our implementation. Testing showed that the *custom attention like* benchmark program does not validate. We investigated the issue and found that the error is not caused by our compiler modifications, but it is caused by a bug in the *memory block merging* pass that runs on the IR with memory information (`GPUMem`). Disabling this pass makes all tests pass. The details of this bug is discussed in subsubsection 5.4.2.

5.2 Benchmark Methodology and Hardware

All benchmarks are run on the NVIDIA A100 GPU. We use the `futhark bench` command to get the running time of our benchmark programs. All benchmark programs can be found at `https://github.com/caymand/mma/tree/main/src/fut_programs`.

For all the comparisons with the Futhark `cuda` action, we used autotuning beforehand to ensure the best possible scenario for the `cuda` action. We use the same compiler for the comparison of the `cuda` and `cudatc` actions and it can be found and compiled at `https://github.com/caymand/futhark/tree/intragroup-mmm`. The Tensor Cores does mixed precision matrix multiplication between single precision and half precision floating point values at no overhead. In normal CUDA code, the cast from half precision to single precision does not come for free since the two floating point formats have different sizes for the exponent, and cannot simply be truncated. For this reason, all our benchmarks of the `cuda` action use either only `f32` or `f16` such that the conversion between data types is not penalized.

5.3 Benchmark Limitations

Benchmarking our implementation is limited to programs that match our transformation criteria. In short, this means we can only benchmark programs that:

1. Have an intragroup kernel corresponding to matrix multiplication
2. Matrix multiplication uses `f16` or mixed precision floating point numbers
3. The matrix sizes are statically known with a size that fits in shared memory

We found that some programs that one might expect match our transformation not always do so, because of the flattening approach chosen by the compiler. Sometimes this can be fixed by using more Futhark attributes related to incremental flattening. In all cases, since there is a chance that the transformation has not run when we expect it to, we also manually verify that all compiled benchmark programs have indeed been instrumented with our special functions.

For the existing Futhark `cuda` action we use the settings `-default-tile-size=16` and `-default-reg-tile-size=4`, since these gave better or similar results compared to the default values for all our benchmarks. Other than this, we did not manually tune these parameters for each problem and type. Similarly we also did not tune parameters for our modified action. It is therefore possible that the original `cuda` action could gain som additional performance through manual tuning of parameters not set by autotuning. On the other hand, this may also be the case for our modified backed, where especially the amount of requested threads for matrix multiplication was found to have a large influence on performance.

5.4 Benchmark Results

In this subsection we present different case studies of Futhark programs that contain matrix multiplication as an intragroup kernel. All these programs have been verified to trigger our transformation and therefore use Tensor Cores. We discuss the strengths or weaknesses for each program, and thereafter present the benchmarking result compared to the `f16` and `f32` performance of the `cuda` action.

5.4.1 Batched matrix multiplication

One of the simplest uses of our compiler modifications is batched matrix multiplication, as shown in the program below. This was also the program we used as the running example when demonstrating our transformations in section 4.

Listing 42: Batched matrix multiplication

```
def matmul [m][n][k] (A: [m][k]f16) (B: [k][n]f16) : [m][n]f32 =
    map (\Arow ->
        map (\Bcol ->
            map2 (*) Arow Bcol
            |> map f32.f16
            |> reduce (+) 0.0
        ) (transpose B)
    ) A

def batchedMMM [q][m][n][k] (A: [q][m][k]f16) (B: [q][k][n]f16) : [q][d][d]f32 =
  #[incremental_flattening(only_intra)]map2 matmul A B
```

The benchmarking shows essentially identical performance for the `cudatc` action and a hand-written implementation without pipelining, likely due to the fact that the generated code is essentially the same. It should, however, be noted, that a program like this could potentially benefit from pipelining global memory reads, as this is likely a bottleneck. Both results are very far from the peak performance of Tensor Cores, likely for this reason, but still far outperform the existing `cuda` action.

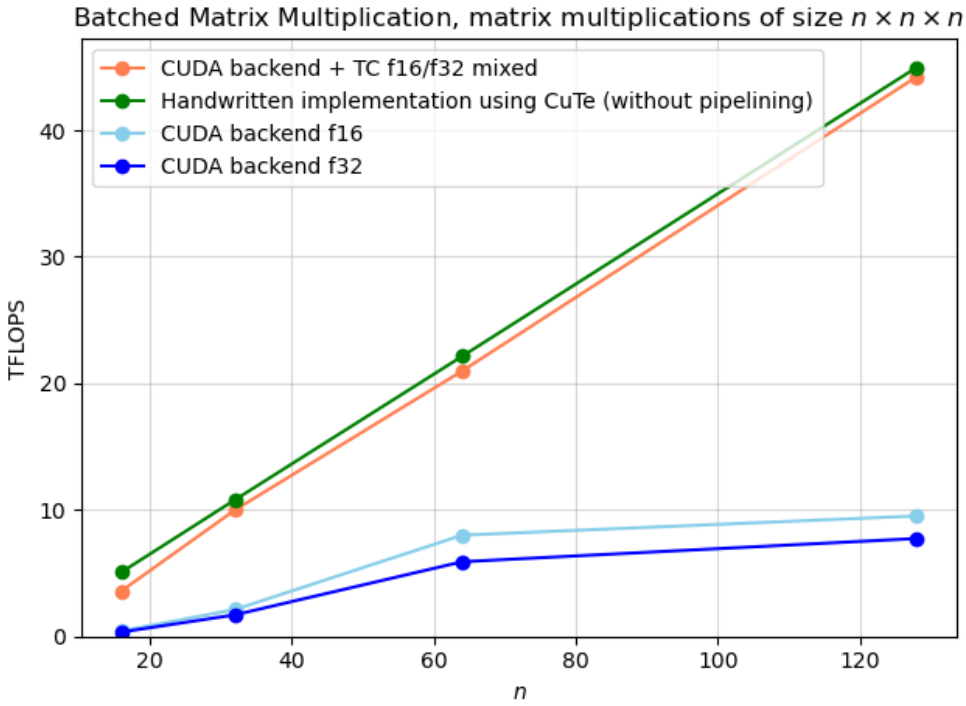

Figure 30: Performance in TFLOPS for batched matrix multiplications of size $n \times n \times n$, as a function of n.

5.4.2 Custom attention like

Below we show a *custom attention like* program. It closely resembles the custom attention program from the "Comparing Functional Array languages" benchmark found at (`https://github.com/diku-dk/CFAL-bench/blob/main/FlashAttention/futhark/custom-alg1-opt.fut`), but it misses the softmax calculation. The `matmul_f16` is a 16-bit matrix multiplication function equivalent to any of the previously shown matrix multiplication programs.

Listing 43: Custom attention like program

```
def oneIter [d] (K: [d][d]real) (V: [d][d]real) (Qi: [d][d]real) =
  let P_block = matmul_f16 Qi K
  -- To be real attention we would need to do: Let P_block = softmax P_block
  in matmul P_block V
def flashAttention [m][d]
    (Q: [m][d][d]real)
    (K: [d][d]real)
    (V: [d][d]real)  =
  map (oneIter K V) Q
entry run16 [m] (Q: [m][16][16]real) (K: [16][16]real) (V: [16][16]real) =
    #[incremental_flattening(only_intra)]flashAttention Q K V
```

As we previously mentioned, this program does not validate in our testing suite, and the incorrectness of the program is not caused by any of our compiler modifications, but is due to a bug in the *memory block merging* pass. Shared memory on the GPU is scarce, and memory block merging tries to reuse shared memory where possible. This optimization however seems to not track aliasing information for memory blocks across function calls, causing memory still in use to be overwritten by subsequent operations. For the above program, we saw that the matrix `V` is being stored in the same memory as the result of the first matrix multiplication, `P_block`. The compiler has previously aggressively inlined functions, which may explain why this problem has not been discovered. Turning off the pass makes the test pass.

The program has high computational throughput. One of the reasons for this may be that `P_block` is kept in shared memory, which means there is a relatively larger amount of computation per global memory read compared to e.g. batched matrix multiplication. Additionally, the first matrix multiplication loads all the data from global memory, and the layout will therefore be swizzled and no bank conflicts will occur. However, the second matrix multiplication will have bank conflicts when it reads `P_block` from shared to registers, since this layout is not swizzled. Different CUDA blocks will also need to load the same `K` and `V` matrices, likely leading to a high L2-cache hit-rate. We report below the result on our modified compiler with memory block merging disabled (no MBM) and with memory block merging enabled. Since enabling memory block merging produces the wrong results, we choose to show both cases.

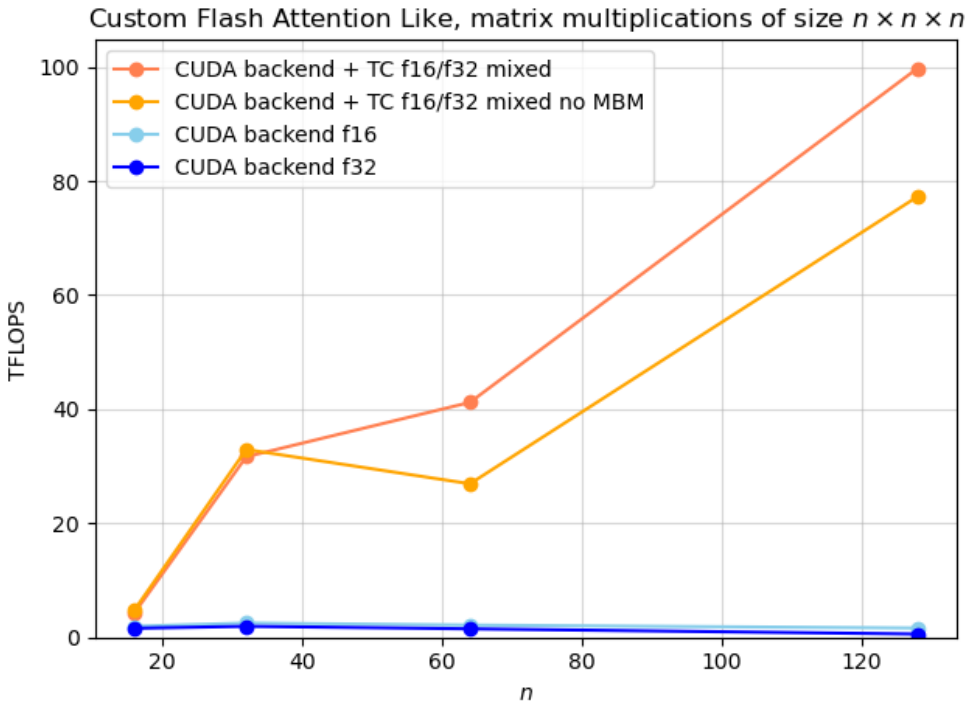

Figure 31: Attention like example similar to custom-alg1.fut from `https://github.com/diku-dk/CFAL-bench`. The results of turning memory block merging (MBM) off is also shown - indicated by the "no MBM". The plot shows performance in TFLOPS for the program with matrix multiplications of size $n \times n \times n$, as a function of n.

The program shows good performance, and it also beats the A100 `f16` peak performance without Tensor Cores of 78 TFLOPS. Without using the Tensor Cores, a hand written CUDA program could therefore never achieve this performance. The peak performance of the first flash attention algorithm was stated to be 124 TFLOPS on the A100 by the original author in [2]. Although our program is only similar to flash attention, this still shows promising results.

Disabling memory block merging does produce slightly worse results. This is likely caused by a lower degree of occupancy due to higher shared memory usage. We inspected the program generated with MBM enabled and observed that the bug can be fixed without allocating additional memory. The program should therefore be able to validate and give same performance if the MBM pass is fixed.

The performance of the stock `cuda` action seems unreasonably low. This may be caused by register spilling or excessive use of global memory in the place of shared memory or registers. Such issues could likely be mitigated by careful optimization of the input Futhark program by hand, in order to give performance more similar to what was seen for batched matrix multiplication and possibly even better.

5.4.3 Attention Like Program

Listing 44:

```
def attention_like [q][m][n][k] (A: [m][k]f16) (B: [q][k][n]f16) : [m][n]f32 =
  -- Below code makes A' be copied to shared in each iteration of the loop
  let A'= if q > 1 hen copy A else replicate (m * k) 0.0f16 |> unflatten
  -- Alternative variant results in no copy
  -- let A' = a
  let acc_init : *[m][n]f32 = replicate (m * n) 0.0f32 |> unflatten in
  loop (acc : *[m][n]f32) = (acc_init: *[m][n]f32) for i < q do
    let B': *[k][n]f16 = B[i]
    let C : *[m][n]f32 = matmulf32 A' B':
    in copy C
entry run_attention [q][p] (A: [p][16][16]f16) (B: [p][q][16][16]f16) =
  #[incremental_flattening(only_intra)]map2 attention_like A B
```

We benchmark another attention like program shown above. The if-statement is a "compiler-hack", which forces the compiler to allocate the result of the branch in shared memory, and in effect does a copy from global to shared memory. Matrix `A` is then reused multiple times to repeatedly do matrix multiplication with new matrices from `B`. Recall, that in a futhark loop, the last statement becomes the accumulator for the next iteration. We choose not to update the accumulator `acc` with the result `C` because it would require an efficient sequentialization of the accumulation code to become efficient. Instead the accumulator for the next iteration simply becomes the `C` that was computed in the current loop iteration. The `copy C` statement will not manifest a copy, but it is simply required for the program to type check, and avoid writing to global memory in each iteration.

Since `A'` will already be in shared memory, our pass will not copy `A'` from global and `A'` will therefore not have a swizzled memory layout. This means that although `A'` is reused, all the matrix multiplication computations will have shared memory bank conflicts. The matrices loaded from the `B` array on the other hand are loaded from global memory to shared memory. This will use our special global to shared memory copy and the layout of `B'` will therefore be swizzled. No bank conflicts therefore occur from using `B'`. We also tested a version where `A` is not copied to shared memory outside the loop. This is indicated by the reassignment of `A` to `A'` in a comment. This will test whether it can be beneficial to always load a matrix from global memory because it results in a swizzled memory layout.

The benchmarking results of this program are given below. For the `cuda` action, we only report the results when `A` was copied to shared using the if-statement. Without `A` being copied to shared outside the loop, the `cuda` action produces much worse results. Please note that the program that copies `A` outside the loop is not benchmarked for $n = 128$. This is because the non-statically known branch causes the compiler to allocate an additional shared memory buffer, causing the program to run out of shared memory. Surprisingly, we see that the version that repeatedly does a copy from global to shared memory inside the loop performs best.

After the first time that `A` is copied from global to shared it will be cached in the L2 cache. All subsequent copies of `A` in the loop will therefore only have to load from the L2 cache. When profiling the generated CUDA code, we also observe a near 50% L2 cache hit rate. This is expected because each loop iteration uses a new `B'` matrix, which will always result in a cache miss, and we therefore almost always find 1 out of 2 matrices in the L2 cache. This, in combination with the swizzled layout of `A` seems to beat copying `A` to shared memory once and reusing it `q` times.

Figure 32 also shows the performance of a hand written implementation using CuTe without pipelining. The implementation is similar to what our compiler generates except that the hand-written version keeps results in registers, rather than writing to shared memory in each iteration. Additionally `A` is also only loaded to shared memory once outside the loop using a swizzled layout. This difference seems to result in around a 2× speedup on the largest dataset. Again this program is likely bottlenecked by global memory reads, and would likely benefit greatly from pipelining.

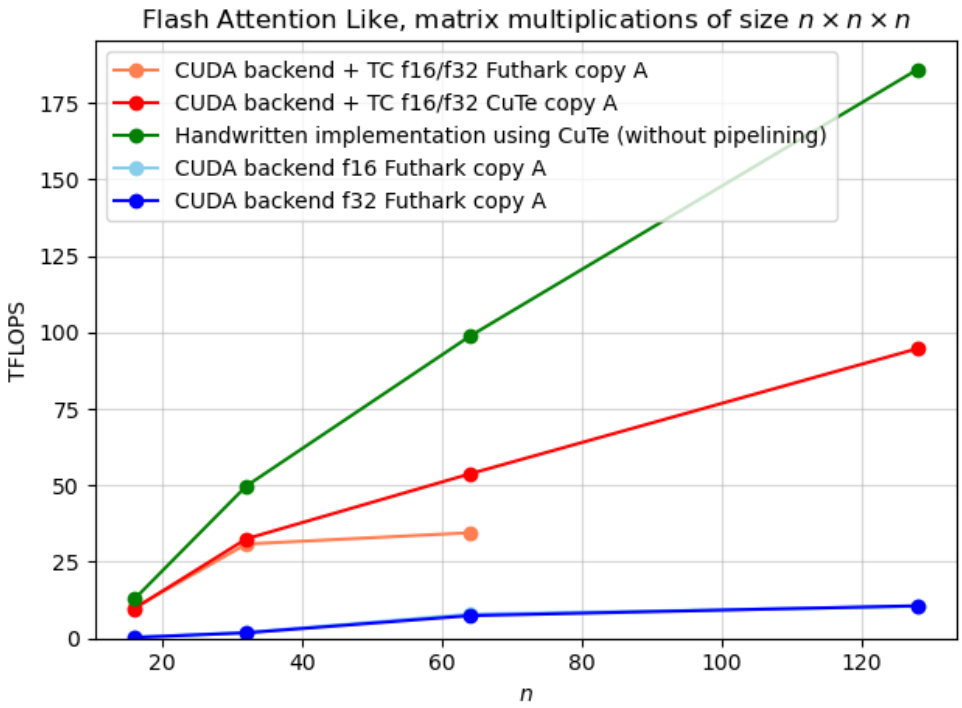

Figure 32: Program similar to flash attention that does matrix multiplication in a loop. The plot shows performance in TFLOPS for the program with matrix multiplications of size $n \times n \times n$, as a function of n.

5.4.4 Rodinia LUD matrix multiplication

As the final benchmarking program, we took inspiration from the LU-decomposition program from the Rodinia [1] benchmark suite. The program computes the matrix multiplication part of LUD from Rodinia. The `seq_acc` helper function subtracts the two matrices `mm` and `mat_blk` from each other using `r` as the sequentialization factor. The sequentialization code is somewhat complicated, but it allows almost arbitrary levels of per thread sequentialization. We however found that the only sequentialization factor that works in practice is whenever `r=b`, and the sequential accumulation from large matrix multiplication in subsubsection 5.4.5 could therefore directly have been used instead. We refer to this implementation for a simple way of sequentializing code in Futhark.

Listing 45: Matrix multiplication from LUD

```
def ludMult [m][b] (r: i64)
(top_per: [m][b][b]f16, lft_per: [m][b][b]f16, mat_slice: [m][m][b][b]f32) =
  #[incremental_flattening(only_inner)]
  map (\ (mat_arr: [m][b][b]f32, lft: [b][b]f16)  ->
         #[incremental_flattening(only_intra)]
         map (\ (mat_blk: [b][b]f32, top: [b][b]f16)  ->
                let mm = matmulf32 lft top
                in seq_acc r (-) (copy mat_blk) (copy mm)
             ) (zip mat_arr top_per)
      ) (zip mat_slice lft_per)
```

In the original Rodinia LUD, the matrix multiplication and subtraction kernels would be fused together. This has better locality, but our transformation would not trigger. In the above code, we have applied *loop fission* to get two intragroup kernels (matrix multiplication and subtraction). In the benchmarks below, the `cuda` results uses the original LUD program where subtraction is fused with the matrix multiplication kernel. Although we beat the `cuda` action, it is to a much lesser extent than the two attention like programs. This is likely due to the sequentialization approach used for the `seq_acc` kernel. It causes the CUDA block to have more threads than what is ideal for matrix multiplication. Otherwise the program is similar to the attention like program, since this program also does matrix multiplication in a loop (the second `map`), and the likely culprit for performance loss is therefore the `seq_acc` kernel. Similarly to the large matrix multiplication program, the ideal solution would be to keep results in registers, and do the subtraction on the data in registers. Again, this is not possible with the current implementation, as described in section 6.

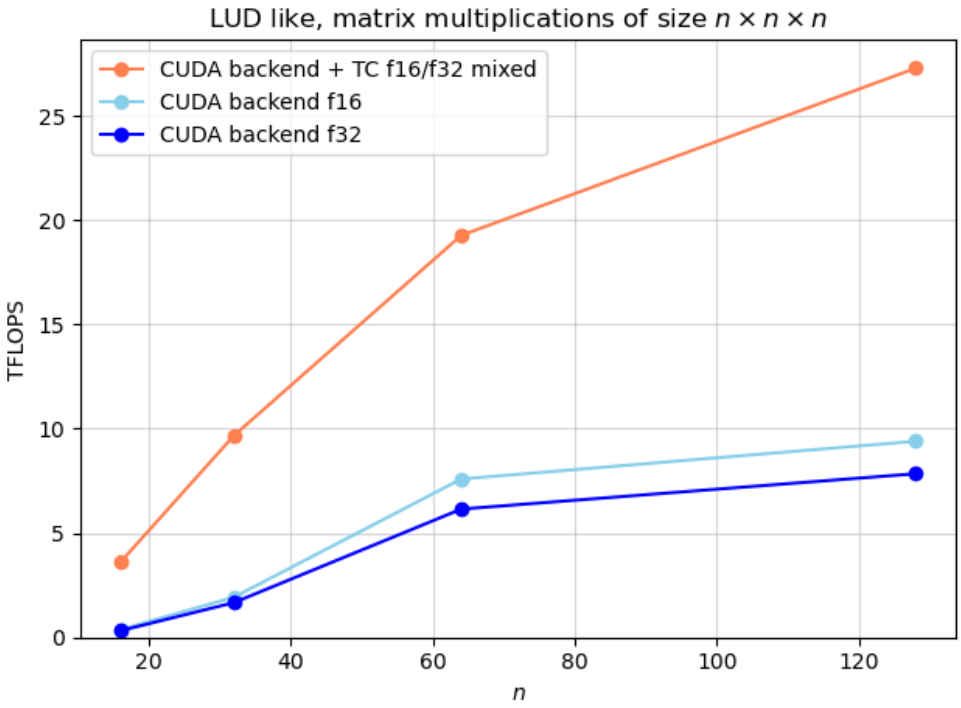

Figure 33: Matrix multiplication kernel from the Rodinia LUD benchmark. The plot shows performance in TFLOPS for the program with matrix multiplications of size $n \times n \times n$, as a function of n.

5.4.5 Large matrix multiplication

The program shown below is an attempt at demonstrating the use of our compiler modifications for large matrix multiplication, similar to the prototypes of section 3. The program takes as input 2 large matrices which have already been partitioned into tiles fitting in shared memory. The reduction over tiles in the outer K dimension has been fully sequentialized, while the addition of C with the accumulated result has been partly sequentialized. This is not a natural way of expressing matrix multiplication in Futhark, but it ensures that the pattern matching of our pass will succeed, and that a not far from optimal block size will be used.

Listing 46: Large matrix multiplication program

```
def matAdd [m][n] (X: *[m][n]f32) (Y: [m][n]f32): [m][n]f32 =
  loop acc: *[m][n]f32 = (X : *[m][n]f32) for i < m do
        acc with [i, :] = map2 (+) acc[i] Y[i]

def handleKBlocks[K][m][n][k] (Arow: [K][m][k]f16) (Bcol: [K][k][n]f16)
  : [m][n]f32 =
    let acc_init : *[m][n]f32 = replicate (m * n) 0.0f32 |> unflatten in
    loop (acc: *[m][n]f32) = acc_init for K_i < K do
        let C = matmul Arow[K_i] Bcol[K_i]
        in matAdd acc C

def run [M][K][N][m][n][k] (A: [M][K][m][k]f16) (B: [K][N][k][n]f16)
  : [M][N][m][n]f32 =
    #[incremental_flattening(only_inner)]map (\Arow ->
        #[incremental_flattening(only_intra)]map (\Bcol ->
            handleKBlocks Arow Bcol
        ) (transpose B)
    ) A
```

For a more fair comparison with the unmodified Futhark compiler we use the the program shown below with $m' = M \cdot m$, $n' = N \cdot n$, and $k' = K \cdot k$. This program is much simpler, and a much more natural way to express matrix multiplication in Futhark. Additionally, it should allow the unmodified compiler to apply incremental flattening and block-register tiling more freely.

Listing 47: Simplified large matrix multiplication program

```
def matmulf16 [m'][n'][k'] (A: [m'][k']f16) (B: [k'][n']f16) : [m'][n']f16 =
  map (\Arow ->
        map (\Bcol ->
               map2 (*) Arow Bcol
               |> reduce (+) 0.0
            ) (transpose B)
      ) A
```

The benchmarking results for large matrix multiplication are a bit disappointing. We do beat the `cuda` action, but only by around a factor of 2×. The results are very far from what our prototype implementations could achieve. There are multiple of reasons for why the performance is worse, the main one likely being that results are not accumulated in registers, but instead loaded in and out of shared memory. We discuss this further in section 6. As shown for our prototypes, this program could also benefit from pipelining global memory reads.

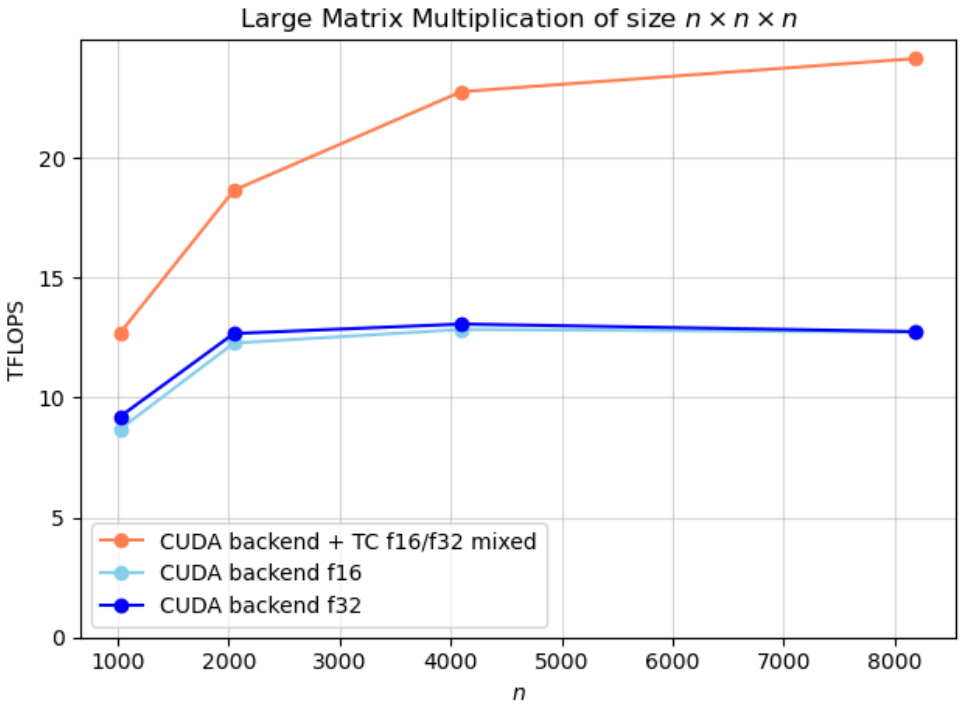

Figure 34: Large matrix multiplication. The `cuda` results are from a standard idiomatic Futhark matrix multiplication program, while the `cudatc` results use "pre-tiled" inputs, with tiles of size $128 \times 128 \times 64$. The plot shows performance in TFLOPS for matrix multiplications of size $n \times n \times n$, as a function of n.

6 Future Work

Much of the future work is related to better utilizing the Tensor Cores. This is in terms of supporting more data types and architectures, but also in more optimizations such as better register usage to unlock more performance. Using the Tensor Cores should also be even more accessible to Futhark programmers by better integrating our changes with the incremental flattening strategy of the compiler. We finish by discussing all this in the next subsections.

6.1 Accumulation in registers

One of the main reasons why the large matrix multiplication benchmark of our compiler modifications severely underperforms compared to our prototypes, is that results are repeatedly loaded in and out of registers. The compiler output for this program is shown in Listing 48. For brevity, some parts of the code has been replaced by pseudocode in comments surrounded by <>.

Listing 48: Compressed compiler output

```
FUTHARK_KERNEL_SIZED(run_square_xlzisegmap_intrablock_9369_dim1, 1, 1)
void run_square_xlzisegmap_intrablock_9369(
    // <Args>
)
{
    // <Variable declarations and initializations>
    // <Zero initialization of global memory>
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int64_t K_i_9376 = 0; K_i_9376 < (int64_t) 128; K_i_9376++) {
        // <Variable declarations and initializations>
        // <Zero initialization of registers>
        barrier(CLK_LOCAL_MEM_FENCE);
        futrts_copyGlobalShared(
            &ext_mem_10678, A_mem_10652, color_10738,
            offsetA_10267, (f16) 0.0F, Int<(int64_t) 128>{},
            Int<(int64_t) 64>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
        );
        futrts_copyGlobalShared(
            &ext_mem_10681, B_mem_10653, color_10737,
            offsetB_10270, (f16) 0.0F, Int<(int64_t) 64>{},
            Int<(int64_t) 128>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
        );
        futrts_tensorMMM(
            &ext_mem_10682, ext_mem_10678, ext_mem_10681,
            mem_10673, (f16) 0.0F, (f16) 0.0F, Int<(int64_t) 128>{},
            Int<(int64_t) 128>{}, Int<(int64_t) 64>{}, Int<(int64_t) 2>{},
            Int<(int64_t) 2>{}, Int<(int64_t) 1>{}, Int<(int64_t) 1>{}
        );
        for (int64_t i_0 = 0; i_0 < (int64_t) 128; i_0++) {
            mem_10689[i_0] = ext_mem_10682[i_0];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        futrts_copyRegistersShared(
            &ext_mem_10693, mem_10689, color_10737,
            (f16) 0.0F, (f16) 0.0F, Int<(int64_t) 128>{},
            Int<(int64_t) 128>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
        );
        barrier(CLK_LOCAL_MEM_FENCE);
        // <Addition of C result in shared mem with acc in shared mem>
    }
    // <Copy of accumulated result from shared memory to global memory>
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

In each iteration of the sequential loop over the outer K dimension, the generated code first zero initializes the C registers, and uses them to hold the result of matrix multiplication. Then they are used to store the result in shared memory, only to load this result back to registers in order to add the current iterations result to the accumulated values. A much more efficient solutions would be to initialize the registers to zero before entering the sequential loop, and then accumulating the result in registers, which is done ”for free” as a part of the `mma` operation. This is also what is done in our prototypes.

As a simple proof of concept for this, we tried performing these optimizations by hand on

the compiler output from our modified version of the Futhark compiler. This was done by simply moving some code outside the sequential loop, and exchanging some variables. These modifications are illustrated in Listing 49. Changes are highlighted using comments with `**`.

Listing 49: Modified compiler output

```
FUTHARK_KERNEL_SIZED(run_square_xlzisegmap_intrablock_9369_dim1, 1, 1)
void run_square_xlzisegmap_intrablock_9369(
    // <Args>
)
{
    // * Added declarations for variables now used outside loop: *
    // <Variable declarations and initializations>
    // * Zero initialization of global memory removed *
    // * Moved outside loop: *
    // <Zero initialization of registers>
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int64_t K_i_9376 = 0; K_i_9376 < (int64_t) 128; K_i_9376++) {
        // * Removed declarations for variables now used outside loop: *
        // <Variable declarations and initializations>
        futrts_copyGlobalShared(
            &ext_mem_10678, A_mem_10652, color_10738,
            offsetA_10267, (f16) 0.0F, Int<(int64_t) 128>{},
            Int<(int64_t) 64>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
        );
        futrts_copyGlobalShared(
            &ext_mem_10681, B_mem_10653, color_10737,
            offsetB_10270, (f16) 0.0F, Int<(int64_t) 64>{},
            Int<(int64_t) 128>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
        );
        futrts_tensorMMM(
            &ext_mem_10682, ext_mem_10678, ext_mem_10681,
            mem_10673, (f16) 0.0F, (f16) 0.0F, Int<(int64_t) 128>{},
            Int<(int64_t) 128>{}, Int<(int64_t) 64>{}, Int<(int64_t) 2>{},
            Int<(int64_t) 2>{}, Int<(int64_t) 1>{}, Int<(int64_t) 1>{}
        );
        for (int64_t i_0 = 0; i_0 < (int64_t) 128; i_0++) {
            mem_10689[i_0] = ext_mem_10682[i_0];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        // * Addition of C result and accumulator in shared memory removed *
    }
    // * Moved outside loop: *
    futrts_copyRegistersShared(
        &ext_mem_10693, mem_10689, color_10737,
        (f16) 0.0F, (f16) 0.0F, Int<(int64_t) 128>{},
        Int<(int64_t) 128>{}, Int<(int64_t) 2>{}, Int<(int64_t) 2>{}
    );
    barrier(CLK_LOCAL_MEM_FENCE);
    // * Changed to copy from result of copyRegistersShared *
    // <Copy of accumulated result from shared memory to global memory>
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Making these changed alone increased the performance in the benchmark on $8192 \times 8192 \times 8192$

matrix multiplication from around 24 TFLOPS to 105 TFLOPS. This is, however, still quite far from the expected performance according to our experiments with the prototype. The main reason for this is that a very large amount of shared memory is requested for the kernel, since the generated code originally used 2 separate shared memory buffers for the iteration result and the accumulator. With the hand modifications, only one of these buffers is used.

If we modify the amount of requested shared memory in the generated C code by hand, and change the shared memory buffers used in the generated CUDA code accordingly, we can reach 162 TFLOPS on the $8192 \times 8192 \times 8192$ benchmark. Our prototype with similar optimizations applied, i.e. without double-buffering or pipelining reaches about 160 TFLOPS, so is actually narrowly beat by the modified Futhark program. This result was surprising, but profiling revealed that it is likely due to the fact that the different data layout of the Futhark implementation, caused by the input being "pre-tiled", results in a greater L2-cache hit-rate.

Making the compiler produce code that accumulates in registers when possible would require some additional analysis of the input programs, in order to determine if values can actually be kept in registers and are not accessed by other threads.

Perhaps a simple first implementation of this could simply extend our pattern match of matrix multiplication between matrices of statically known sizes that fit in shared memory to matrix multiplication between matrices of all sizes, and then perform block-tiling in order to get tiles that fit in shared memory. The sequential loop over block tiles would then be generated by the compiler, and results could then easily be accumulated in registers throughout this loop.

However, the ideal solution should be more general, and cover more cases than just large matrix multiplication. Ideally it could perhaps even allow general fusing with other kernels over the input or output of matrix multiplication. This could then be used to efficiently implement GEMM of the form $\mathcal{D} = \alpha\mathcal{A}\mathcal{B} + \beta\mathcal{C}$, or linear transformations followed by activation functions, as commonly used in neural networks.

6.2 Double-buffering or pipelining

Another reason why the code generated by the `cudatc` action is not able to achieve as good performance as our handwritten large matrix multiplication prototypes is the lack of pipelining. As described in subsection 2.2, this is an optimization that could benefit a wide range of programs, since this allows the interleaving of data movement and computation. A pass performing double buffering already exists in the compiler, so this could potentially be modified to also work with our modifications. Additionally, it may be useful to generalize this pass to n-stage pipelining.

6.3 Interoperability with incremental flattening and autotuning

Currently, making use of our compiler modifications generally requires using expression attributes to ensure that kernels of the right form are produced by the initial "extract kernels" stage of the compiler middle-end. Additionally, since our passes do not update the conditionals generated by incremental flattening to choose different code versions, these conditionals often end up being nonsensical if they are present, potentially leading to the branch containing the code generated by our pass not being taken at runtime, even if it would give better performance. For this reason, the programs we have tested and benchmarked generally use the expression attributes to ensure that only a single version of the code is generated by incremental flattening, and that this version matches the format expected by our passes.

An obvious first step would be to update these conditionals whenever block sizes are changed. Additionally, it should ideally be ensured that code matched by our passes is produced whenever possible, without the user needing to use expression attributes. However, this would likely break the monotonicity assumption of the autotuning[22] used to select the optimal version of the code produced by incremental flattening, since different versions of the code would then not only exploit different amounts of parallelism, but also use different hardware.

Finally, being able to use some form of autotuning to find the optimal amount of threads to use for the tensor matrix multiplication would also be a very nice feature, but would likely also not conform to the monotonicity requirement, similarly to tile sizes.

6.4 Better integration of swizzling

With the current implementation, we do not use our special function for copying to shared in cases where an argument to matrix multiplication is loaded into shared memory outside of the code pattern matched by our compiler pass. Instead we simply use the data in shared memory as is, which leads to bank conflicts when the data is loaded into registers, as no swizzling is then applied. Ideally, this should be avoided. If the data has been loaded from global memory to shared memory at another point in the program, and is not used anywhere else, we could simply find this load and replace it by our function. However, if the data is used elsewhere, we cannot just apply swizzling, as the other code would not be expecting a swizzled layout. Similarly, we would also not be able to easily apply swizzling if the argument in shared memory was the result of another computation.

This issue could potentially be solved by making a swizzled copy of the data, before it is used in our function for matrix multiplication. However, this would double the shared memory usage of this data, and also require additional reads and writes of shared memory. We did some simple experiments with our prototype implementation in an attempt to figure out if these trade-offs would be worth it, but this did not seem to be the case.

A better, but much more involved solution, would be to have support for swizzled layouts in the memory annotated Futhark IR, similarly to the `ComposedLayout` of CuTe. Such a capability could be useful for more than just matrix multiplication, since compiler passes could then potentially in general detect access patterns leading to bank conflicts and apply swizzling as needed in these cases.

6.5 Avoiding bank conflicts when writing matrix multiplication results (C arrays)

Currently, the `copyRegistersShared` function, used to write matrix multiplication results (C arrays) to shared memory, causes 4-way bank conflicts in the worst case when the result is of type `f32`. This is due to the fact that the rows of the result in the `mma` operation, which we recall in Figure 35, contain 8 consecutive elements that can be written in parallel using vectorized stores. This means only 8 distinct banks are accessed in each row, which is only $\frac{1}{4}$ of the 32 availible banks.

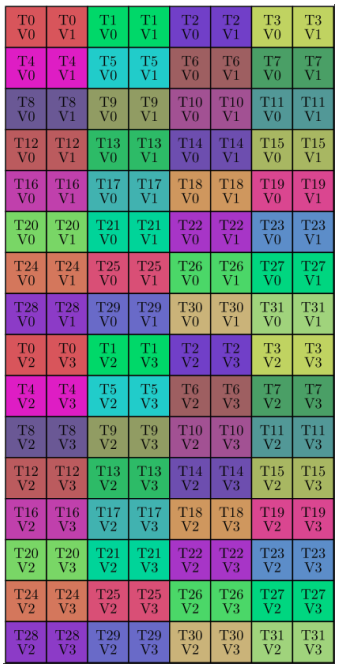

Figure 35: Layout of threads and values in registers after mma computation.

One possible solution to this would be to swizzle the layout of C matrices in shared memory. However, due to the challenges of integrating swizzled layouts with the rest of Futhark as mentioned above, this may not be the best solution.

A better solution, at least in some cases, may be to write directly to global memory whenever possible, i.e. when the result in shared would be written directly back to global memory anyway. Writing directly to global memory can be done fully coalesced with `f32` results, since 8 such elements, i.e. a row from the `mma` result, corresponds to 32 B which is the minimum global memory transaction size. This could be achieved by implementing a `copyRegistersGlobal` function, very similar to `copyRegistersShared`, and inserting this as a part of the high-level IR transformations when possible, and modifying the memory annotated IR accordingly.

A final, more complex, solution may be to use the fact that each warp typically performs multiple `mma` operations with a layout as the one shown in Figure 35. This means that for each of these operations, the values residing in the registers of a thread could potentially reside in different banks. Therefore it may be possible to modify layouts and the order of memory accesses in a way that ensures that each time a warp stores data, all threads access distinct banks, even though the shared memory layout is not swizzled. This is, however, far from trivial to implement, especially due to the intrinsic architecture constraints that all threads in a warp must execute the exact same code, and that data in registers can only be accessed using static indexing. In fact, these restrictions may well mean that this solution is impossible to implement.

6.6 Generalizing the Solution

There are a number of ways in which our solution could be generalized. Some of these are detailed below.

6.6.1 Supporting other matrix multiplication-like patterns

Our compiler modifications currently only targets matrix multiplication where each resulting element is calculated as shown below:

$$c_{mn} = \sum_{k=0}^{K} a_{mk} b_{kn}$$

The 3 patterns shown below are very similar, except for the order of indexes:

$$c_{mn} = \sum_{k=0}^{K} a_{mk} b_{nk}, \quad c_{mn} = \sum_{k=0}^{K} a_{km} b_{kn}, \quad c_{mn} = \sum_{k=0}^{K} a_{km} b_{nk}$$

In Futhark, these 4 distinct patterns correspond to different combinations of transposing or not transposing the A and B arrays shown in Listing 50, with the one shown corresponding to the pattern we currently match.

Listing 50: Matrix multiplication in Futhark

```
map (\a ->
    map (\b ->
        map2 (*) a b
        |> reduce (+) 0.0
    ) (transpose B)
) A
```

Supporting the other patterns would be fairly straightforward. The pattern matching would simply have to be extended, and the instructions used for loading data into registers in the `tensorMMM`[9] function would have to be changed depending on the pattern.

6.6.2 Supporting more data types

In our work we have focused on `f16` and `f16/f32` mixed precision tensor core operations. There are, however, versions of these operations that operate on other data types. The most relevant for Futhark is probably the `f64` operations and the `i8/i32` and `u8/i32` mixed precision operations, since the other operations use types that are not yet available in futhark.

Adding support for these would require generalizing the pattern matching, finding optimal configurations in terms of swizzling and elements calculated per thread for these data types, and adding these to the CUDA prelude.

6.6.3 Supporting and optimizing for more architectures

Our solution is optimized for the NVIDIA Ampere architecture, and only works for this and newer architectures. The preceding Volta and Turing architectures also have Tensor Cores, but require the use of different PTX instructions. Similarly, the newer Hopper architecture also adds new Tensor Core operation that are not available on older architectures[14]. Therefore, even though

[9]The function name of the special function that performs matrix multiplication

our solution is still compatible with the Hopper architecture, it is likely that these new operations would have to be used in order to achieve the best possible performance on this newer architecture.

Making optimal use of these other tensor core operations in Futhark would mostly require changing the CUDA and CuTe prelude, and finding optimal configurations for these new operations. Modifying the CuTe configuration code should be fairly simple and not require a major rewrite of the CUDA code.

Additionally a way of selecting target architecture would have to be added. This could be done either automatically at runtime based on available hardware, or manually during compilation.

7 Conclusion

In this work, we have shown how use of the NVIDIA Tensor Cores can be integrated into the Futhark compiler. We started out by writing and testing prototype implementations of matrix multiplication using the Tensor Cores, achieving around the same performance as cuBLAS. This led us to identify 3 functions that the Futhark compiler could emit to use the Tensor Cores and abstract away most implementation details related to using the Tensor Cores. Two passes were then implemented in the compiler: (1) for recognizing code suitable for execution using the Tensor Cores and emitting the three Tensor Core related functions and (2) a pass that fixes memory related issues caused by calling special functions.

The modified compiler is implemented as its own `cudatc` action, and the `cuda` compiler action has been left unmodified. Futhark programmers must therefore opt in to using the Tensor Cores by using the new `cudatc` action. Programs with an intragroup kernel corresponding to matrix multiplication can now take advantage of the Tensor Cores. Compared to hand writing CUDA programs with PTX instructions, this makes the Tensor Cores more accessible to high performance computing.

The work presented is mostly a proof of concept, and there are many opportunities for generalizing and optimizing the Tensor Core usage and leverage more performance. The benchmarking results are however promising, and in all the cases tested, the implementation using Tensor Cores beats the existing compiler implementation. The speedup compared to the stock Futhark compiler is given in Table 3.

	Batched	Custom Attention	Attention	Rodinia LUD	Large
Relative to stock `f16`	4.6×	60.1×	9.0×	2.9×	1.9×
Relative to stock `f32`	5.7×	167.5×	8.9×	3.5×	1.9×

Table 3: Speedup for the largest array size for all benchmarks, relative to the stock `cuda` action on `f16` and `f32` programs.

This speedup is, however, not only caused by using the Tensor Cores, but also the other optimizations that are hidden in the implementation of our special functions, such as vectorized loads and swizzling. There is also more work left in making the Tensor Cores more accessible in Futhark, including making incremental flattening work with the Tensor Cores. We leave these optimizations as future work, and conclude that this thesis has succeeded in showing that new and more heterogeneous hardware can be utilized in Futhark.

References

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[2] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. 2023. Accessed: 12-12-2024.

[3] Cosmin E. Oancea. Lecture notes for the software track of the pmph course. 2018.

[4] Aman Kansal, Christos Kozyrakis, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, July 2010.

[5] Mary Hall, Cosmin Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. Scheduling languages: A past, present, and future taxonomy. *arXiv preprint arXiv:2410.19927*, 2024.

[6] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998.

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519530, June 2013.

[8] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302313, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 1019, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Philippe Tillet. triton.language.dot. Link: https://triton-lang.org/main/python-api/generated/triton.language.dot.html, visited 28/11/2024.

[11] Anders Lietzen Holst and Æmilie Cholewa-Madsen. Teaching the futhark compiler block and register tiled matrix multiplication. Master's thesis, University of Copenhagen, 2020.

[12] NVIDIA Corporation. *CUDA C Programming Guide*, 2024. Accessed: 2024-11-01.

[13] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*, 2020. Accessed: 2024-11-01.

[14] NVIDIA Corporation. *PTX ISA*, 2024. Accessed: 2024-11-01.

[15] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*, 2023. Accessed: 2024-11-01.

[16] NVIDIA Corporation. Achieved occupancy.

[17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556571, June 2017.

[18] Morten Clausen. Regular segmented single-pass scan in futhark. Master's thesis, University of Copenhagen, 2021.

[19] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. Compiling generalized histograms for gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[20] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. Memory optimizations in an array language. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 424–438, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[21] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67, New York, NY, USA, 2019. ACM.

[22] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. Dataset sensitive autotuning of multi-versioned code based on monotonic properties. TFP 2021, 2021.

[23] Christian Marslev and Jonas Grønborg. Efficient sequentialization of parallelism. Master's thesis, University of Copenhagen, 2024.

A Source Code

A.1 `preludeTensorCores.cu`

```
1  using namespace cute;
2
3  template<class TypeIn>
4  struct convert_type {
5      using TypeOut = TypeIn;
6  };
7
8  template<>
9  struct convert_type<f16> {
10     using TypeOut = half_t;
11 };
12
13 template<class ElmTypeAIn, class ElmTypeBIn, class ElmTypeCIn, class SizeM, class
   ↪  SizeN, class WarpsM, class WarpsN>
14 struct get_mma_config {};
15
16 // TODO: use FMA when Tensor Cores not available?
17
```

```
template<class SizeM, class SizeN, class WarpsM, class WarpsN>
struct get_mma_config<half_t, half_t, half_t, SizeM, SizeN, WarpsM, WarpsN> {
    // TODO: should depend on architecture available
    using MMATraits = MMA_Traits<SM80_16x8x16_F16F16F16F16_TN>;
    using ACopyOpSharedRegisters = SM75_U32x4_LDSM_N;
    using BCopyOpSharedRegisters = SM75_U16x8_LDSM_T;

    using MMATile = Tile<Int<16 * WarpsM{}>, Int<16 * WarpsN{}>, _16>;
    using TiledMMA = TiledMMA<
        MMA_Atom<MMATraits>,
        Layout<Shape<WarpsM,WarpsN,_1>,
        MMATile
    >;
};

template<class SizeM, class SizeN, class WarpsM, class WarpsN>
struct get_mma_config<half_t, half_t, float, SizeM, SizeN, WarpsM, WarpsN>{
    // TODO: should depend on architecture available
    using MMATraits = MMA_Traits<SM80_16x8x16_F32F16F16F32_TN>;
    using ACopyOpSharedRegisters = SM75_U32x4_LDSM_N;
    using BCopyOpSharedRegisters = SM75_U16x8_LDSM_T;

    using MMATile = Tile<Int<16 * WarpsM{}>, Int<16 * WarpsN{}>, _16>;
    using TiledMMA = TiledMMA<
        MMA_Atom<MMATraits>,
        Layout<Shape<WarpsM,WarpsN,_1>,
        MMATile
    >;
};

template<class SizeY, class SizeX, class Swizzle, class Majorness, int shift_len>
struct get_layout_config {};

template<class SizeY, class SizeX, int shift_len>
struct get_layout_config<SizeY, SizeX, _1, LayoutRight, shift_len>{
    using SharedLayout = ComposedLayout<Swizzle<3, 3, shift_len>, _0,
    ↪  Layout<Shape<SizeY, SizeX>, Stride<SizeX, _1>>>;
};

template<class SizeY, class SizeX, int shift_len>
struct get_layout_config<SizeY, SizeX, _0, LayoutRight, shift_len>{
    using SharedLayout = Layout<Shape<SizeY, SizeX>, Stride<SizeX, _1>>;
};

template<class SizeY, class SizeX, int shift_len>
struct get_layout_config<SizeY, SizeX, _1, LayoutLeft, shift_len>{
    using SharedLayout = ComposedLayout<Swizzle<3, 3, shift_len>, _0,
    ↪  Layout<Shape<SizeY, SizeX>, Stride<_1, SizeY>>>;
};

template<class SizeY, class SizeX, int shift_len>
struct get_layout_config<SizeY, SizeX, _0, LayoutLeft, shift_len>{
    using SharedLayout = Layout<Shape<SizeY, SizeX>, Stride<_1, SizeY>>;
};

template<class ElmTypeIn, class SizeY, class SizeX, class WarpsM, class WarpsN>
```

```
72 FUTHARK_FUN_ATTR void futrts_copyGlobalShared(unsigned char **mem_out_p, unsigned
   ↪  char *global_mem, unsigned char *shared_mem, int64_t offset, ElmTypeIn,
   ↪  SizeY, SizeX, WarpsM, WarpsN)
73 {
74     *mem_out_p = shared_mem;
75 
76     int flatThreadIdx = threadIdx.z * blockDim.y * blockDim.x + threadIdx.y *
       ↪  blockDim.x + threadIdx.x;
77 
78     if (flatThreadIdx < WarpsM{} * WarpsN{} * 32) {
79       using ElmType = typename convert_type<ElmTypeIn>::TypeOut;
80 
81       using CopyOpGlobalShared = SM80_CP_ASYNC_CACHEGLOBAL<uint128_t>;
82 
83       constexpr int elmsPerLoad = 16 / sizeof(ElmType);
84       constexpr int threadsX = SizeX{} / elmsPerLoad;
85       constexpr int threadsY = (WarpsM{} * WarpsN{} * 32) / threadsX;
86 
87       constexpr unsigned int sizeXunsigned = SizeX{};
88       constexpr unsigned int shift_len = max(bit_width(sizeXunsigned) - 4, _3{});
89 
90       using LayoutConfig = get_layout_config<SizeY, SizeX, _1, LayoutRight,
         ↪  shift_len>;
91       typename LayoutConfig::SharedLayout s_layout;
92 
93       auto g_layout = make_layout(Shape<SizeY, SizeX>{}, LayoutRight{});
94 
95       TiledCopy copy_global_shared =
         ↪  make_tiled_copy(Copy_Atom<CopyOpGlobalShared, ElmType>{},
96           make_layout(Shape<Int<threadsY>, Int<threadsX>>{}, LayoutRight{}),
97           Layout<Shape<_1,Int<elmsPerLoad>>>{}
98       );
99 
100      Tensor s = make_tensor(make_smem_ptr(reinterpret_cast<ElmType
         ↪  *>(shared_mem)), s_layout);
101      Tensor g = make_tensor(make_gmem_ptr(&reinterpret_cast<ElmType
         ↪  *>(global_mem)[offset]), g_layout);
102 
103      ThrCopy thr_copy_global_shared =
         ↪  copy_global_shared.get_slice(flatThreadIdx);
104      Tensor tAgA = thr_copy_global_shared.partition_S(g);
105      Tensor tAsA = thr_copy_global_shared.partition_D(s);
106 
107      copy(copy_global_shared, tAgA, tAsA);
108 
109      cp_async_fence();
110    }
111 
112 //     Assuming the copied data is only used in futrts_tensorMMM, we do not need
   ↪  to wait for it here
113 //     cp_async_wait<0>();
114 //     __syncthreads();
115 }
116 
117 template<class ElmTypeAIn, class ElmTypeBIn, class ElmTypeCIn, class SizeM, class
   ↪  SizeN, class WarpsM, class WarpsN, int numRegs>
```

```
118 FUTHARK_FUN_ATTR void futrts_copyRegistersShared(unsigned char **mem_out_p,
    ↪ ElmTypeCIn (&registers_mem)[numRegs], unsigned char *shared_mem, ElmTypeAIn,
    ↪ ElmTypeBIn, SizeM, SizeN, WarpsM, WarpsN)
119 {
120     *mem_out_p = shared_mem;
121 
122     int flatThreadIdx = threadIdx.z * blockDim.y * blockDim.x + threadIdx.y *
        ↪ blockDim.x + threadIdx.x;
123 
124     if (flatThreadIdx < WarpsM{} * WarpsN{} * 32) {
125         using ElmTypeA = typename convert_type<ElmTypeAIn>::TypeOut;
126         using ElmTypeB = typename convert_type<ElmTypeBIn>::TypeOut;
127         using ElmTypeC = typename convert_type<ElmTypeCIn>::TypeOut;
128 
129         using MMAConfig = get_mma_config<ElmTypeA, ElmTypeB, ElmTypeC, SizeM,
            ↪ SizeN, WarpsM, WarpsN>;
130         typename MMAConfig::TiledMMA tiled_mma;
131 
132         auto s_layout = make_layout(Shape<SizeM, SizeN>{}, LayoutRight{});
133 
134         ThrMMA thr_mma = tiled_mma.get_slice(flatThreadIdx);
135 
136         auto r_layout = partition_shape_C(thr_mma, s_layout.shape());
137         Tensor tCrC = make_tensor(make_rmem_ptr(reinterpret_cast<ElmTypeC
            ↪ *>(registers_mem)), r_layout);
138 
139         Tensor s = make_tensor(make_smem_ptr(reinterpret_cast<ElmTypeC
            ↪ *>(shared_mem)), s_layout);
140         Tensor tCsC = thr_mma.partition_C(s);
141 
142         copy(AutoVectorizingCopy{}, tCrC, tCsC);
143     }
144     __syncthreads();
145 }
146 
147 template<class ElmTypeAIn, class ElmTypeBIn, class ElmTypeCIn, class SizeM, class
    ↪ SizeN, class SizeK, class WarpsM, class WarpsN, class ASwizzled, class
    ↪ BSwizzled, int numRegs>
148 FUTHARK_FUN_ATTR void futrts_tensorMMM(ElmTypeCIn (*mem_out_p)[numRegs], unsigned
    ↪ char *A_mem, unsigned char *B_mem, ElmTypeCIn (&C_mem)[numRegs], ElmTypeAIn,
    ↪ ElmTypeBIn, SizeM, SizeN, SizeK, WarpsM, WarpsN, ASwizzled, BSwizzled)
149 {
150     int flatThreadIdx = threadIdx.z * blockDim.y * blockDim.x + threadIdx.y *
        ↪ blockDim.x + threadIdx.x;
151 
152     using ElmTypeA = typename convert_type<ElmTypeAIn>::TypeOut;
153     using ElmTypeB = typename convert_type<ElmTypeBIn>::TypeOut;
154     using ElmTypeC = typename convert_type<ElmTypeCIn>::TypeOut;
155 
156     using MMAConfig = get_mma_config<ElmTypeA, ElmTypeB, ElmTypeC, SizeM, SizeN,
        ↪ WarpsM, WarpsN>;
157     typename MMAConfig::TiledMMA tiled_mma;
158 
159     constexpr unsigned int sizeKunsigned = SizeK{};
160     constexpr unsigned int shift_lenK = max(bit_width(sizeKunsigned) - 4, _3{});
161 
```

```
162     constexpr unsigned int sizeNunsigned = SizeN{};
163     constexpr unsigned int shift_lenN = max(bit_width(sizeNunsigned) - 4, _3{});
164
165     using ALayoutConfig = get_layout_config<SizeM, SizeK, ASwizzled, LayoutRight,
        ↪  shift_lenK>;
166     using BLayoutConfig = get_layout_config<SizeN, SizeK, BSwizzled, LayoutLeft,
        ↪  shift_lenN>;
167     typename ALayoutConfig::SharedLayout sA_layout;
168     typename BLayoutConfig::SharedLayout sB_layout;
169
170     auto sC_layout = make_layout(Shape<SizeM, SizeN>{}, LayoutRight{});
171
172     ThrMMA thr_mma = tiled_mma.get_slice(flatThreadIdx);
173
174     auto rC_layout = partition_shape_C(thr_mma, sC_layout.shape());
175     Tensor tCrC = make_tensor(make_rmem_ptr(reinterpret_cast<ElmTypeC *>(C_mem)),
        ↪  rC_layout);
176
177     Tensor sA = make_tensor(make_smem_ptr(reinterpret_cast<ElmTypeA *>(A_mem)),
        ↪  sA_layout);
178     Tensor sB = make_tensor(make_smem_ptr(reinterpret_cast<ElmTypeB *>(B_mem)),
        ↪  sB_layout);
179
180     TiledCopy copyA_shared_registers = make_tiled_copy_A(Copy_Atom<typename
        ↪  MMAConfig::ACopyOpSharedRegisters, ElmTypeA>{}, tiled_mma);
181     TiledCopy copyB_shared_registers = make_tiled_copy_B(Copy_Atom<typename
        ↪  MMAConfig::BCopyOpSharedRegisters, ElmTypeB>{}, tiled_mma);
182
183     Tensor tCrA  = thr_mma.partition_fragment_A(sA);
184     Tensor tCrB  = thr_mma.partition_fragment_B(sB);
185
186     auto smem_thr_copy_A   =
        ↪  copyA_shared_registers.get_thread_slice(threadIdx.x);
187     Tensor tCsA            = smem_thr_copy_A.partition_S(sA);
188     Tensor tCrA_copy_view  = smem_thr_copy_A.retile_D(tCrA);
189
190     auto smem_thr_copy_B   =
        ↪  copyB_shared_registers.get_thread_slice(threadIdx.x);
191     Tensor tCsB            = smem_thr_copy_B.partition_S(sB);
192     Tensor tCrB_copy_view  = smem_thr_copy_B.retile_D(tCrB);
193
194     // Wait for data copied asynchronously by futrts_copyGlobalShared
195     cp_async_wait<0>();
196     __syncthreads();
197
198     constexpr int K_BLOCK_MAX = size<2>(tCrA);
199     CUTE_UNROLL
200     for (int k_block = 0; k_block < K_BLOCK_MAX; ++k_block)
201     {
202         // Copy shared->registers
203         copy(copyA_shared_registers, tCsA(_,_,k_block),
            ↪  tCrA_copy_view(_,_,k_block));
204         copy(copyB_shared_registers, tCsB(_,_,k_block),
            ↪  tCrB_copy_view(_,_,k_block));
205
206         // Perform mma on k_block in registers
```

```
207             gemm(tiled_mma, tCrA(_,_,k_block), tCrB(_,_,k_block), tCrC);
208         }
209
210         for (int32_t i = 0; i < numRegs; i++)
211             (*mem_out_p)[i] = C_mem[i];
212     }
```

A.2 TensorCores.hs

```
1  module Futhark.Optimise.TensorCores
2    (tensorCoreMemFixup, extractTensorCores)
3  where
4
5  import Control.Monad
6  import Futhark.Pass
7    ( Pass (..),
8      intraproceduralTransformationWithConsts,
9    )
10 import Futhark.IR.GPU
11 import Futhark.IR.GPUMem
12 import Futhark.Pass.Simplify
13 import Futhark.Optimise.TensorCores.ExtractTensorCores (transformProg)
14 import Futhark.Optimise.TensorCores.TensorCoreMemFixup (fixFuns)
15
16
17 -- | Transforms intragroup kernels corresponding to matrix multiplication into
18 -- function calls that use the Tensor Cores.
19 extractTensorCores :: Pass GPU GPU
20 extractTensorCores =
21   Pass
22     "tensor-mma"
23     "Extracts NVIDIA tensor core MMA operations"
24     transformProg
25
26 -- | Fixes up the memory allocation caused by inserting function calls for
27 -- tensor core operations.
28 tensorCoreMemFixup :: Pass GPUMem GPUMem
29 tensorCoreMemFixup =
30   Pass
31     "mma-fixup"
32     "Extracts NVIDIA tensor core MMA operations"
33     $ intraproceduralTransformationWithConsts pure fixFuns
34       >=> passFunction simplifyGPUMem
```

A.3 ExtractTensorCores.hs

```
1  module Futhark.Optimise.TensorCores.ExtractTensorCores
2    (transformProg)
3  where
4
5  import Control.Monad
6  import Control.Monad.RWS.Strict
7  import Control.Monad.Reader
8  import Control.Monad.State.Strict
9  import Control.Monad.Writer
```

```haskell
import Data.Bits
import Data.Foldable (toList)
import Data.List (elemIndex, intersect, partition)
import Data.Loc (Loc (NoLoc), SrcLoc (SrcLoc))
import Data.Map.Strict qualified as M
import Data.Semigroup
import Data.Set (difference, fromList)
import Futhark.Analysis.SymbolTable qualified as ST
import Futhark.Builder
import Futhark.Construct
import Futhark.IR.GPU
import Futhark.IR.GPUMem
import Futhark.Optimise.Simplify.Rep
import Futhark.Optimise.TensorCores.Utils
import Futhark.Optimise.TileLoops.Shared
import Futhark.Pass (PassM)
import Prelude hiding (lookup)


-- | Divide and round up.
divUp :: Int -> Int -> Int
divUp x y = (x + y - 1) `div` y




-- Tensor core functions emitted (gemm, copy global shared, copy registers
↪  shared)
type TcFunDef = (MMMSignature, FunDef GPU)

type TcFuns = [TcFunDef]

data ExtractTcEnv = ExtractTcEnv
  { envScope :: Scope GPU,
    envBlockSize :: Maybe Int
  }

-- | Monad that the tensor core match and GPU IR code transformations runs
↪  within.
type TensorCoreM = RWS ExtractTcEnv TcFuns VNameSource

instance HasScope GPU TensorCoreM where
  askScope = asks envScope

instance LocalScope GPU TensorCoreM where
  localScope extension = local $
    \env -> env {envScope = M.union extension $ envScope env}

askBlockSize :: TensorCoreM (Maybe Int)
askBlockSize = asks envBlockSize

localBlockSize :: (Maybe Int -> Maybe Int) -> TensorCoreM a -> TensorCoreM a
localBlockSize f = local $ \env -> env {envBlockSize = f $ envBlockSize env}

runBuilderMMM :: Builder GPU a -> Scope GPU -> TensorCoreM (a, Stms GPU)
runBuilderMMM m s =
```

```
64    modifyNameSource $ runState $ runBuilderT m s
65
66
67  -- | Create the gemm function defintion.
68  mkGemmFun ::
69    (MonadFreshNames m) =>
70    PrimType ->
71    PrimType ->
72    PrimType ->
73    Int ->
74    Int ->
75    Int ->
76    Int ->
77    m TcFunDef
78  mkGemmFun elmTypeA elmTypeB elmTypeC sizeM sizeN sizeK sizeRegs = do
79    let typeA =
80          Array
81            elmTypeA
82            (Shape [mkInt64Const sizeM, mkInt64Const sizeK])
83            Nonunique
84        typeB =
85          Array
86            elmTypeB
87            (Shape [mkInt64Const sizeK, mkInt64Const sizeN])
88            Nonunique
89        typeCin =
90          Array
91            elmTypeC
92            (Shape [mkInt64Const sizeRegs])
93            Unique
94        typeCout =
95          [ ( Array
96                elmTypeC
97                (Shape [Free $ mkInt64Const sizeRegs])
98                Unique,
99              RetAls [] []
100           )
101         ]
102
103   aParam <- newParam "A" typeA
104   bParam <- newParam "B" typeB
105   cParam <- newParam "C" typeCin
106   aElmTypeParam <- newParam "elmTypeA" $ Prim elmTypeA
107   bElmTypeParam <- newParam "elmTypeB" $ Prim elmTypeB
108   mParam <- newParam "M" $ Prim int64
109   nParam <- newParam "N" $ Prim int64
110   kParam <- newParam "K" $ Prim int64
111   mWarpsParam <- newParam "mWarps" $ Prim int64
112   nWarpsParam <- newParam "nWarps" $ Prim int64
113   aSwizzledParam <- newParam "aSwizzledParam" $ Prim int64
114   bSwizzledParam <- newParam "bSwizzledParam" $ Prim int64
115
116   fName <- fmap (nameFromString . prettyString) $ newName $ VName gemmName 0
117   let funParams =
118         [ aParam,
119           bParam,
```

```
120             cParam,
121             aElmTypeParam,
122             bElmTypeParam,
123             mParam,
124             nParam,
125             kParam,
126             mWarpsParam,
127             nWarpsParam,
128             aSwizzledParam,
129             bSwizzledParam
130           ]
131     pure
132       ( GemmSignature elmTypeA elmTypeB elmTypeC sizeM sizeN sizeK sizeRegs,
133         FunDef Nothing mempty fName typeCout funParams $
134           resultBody [Var $ paramName cParam]
135       )
136
137 -- | Create the copy global shared function definition.
138 mkCopyGlobalShared :: (MonadFreshNames m) => PrimType -> Int -> Int -> m TcFunDef
139 mkCopyGlobalShared elmType sizeY sizeX = do
140   let arrShape =
141         Shape
142           [ mkInt64Const sizeY,
143             mkInt64Const sizeX
144           ]
145
146   globalParam <- newParam "global" $ Array elmType arrShape Nonunique
147   sharedParam <- newParam "shared" $ Array elmType arrShape Unique
148   offsetParam <- newParam "offset" $ Prim int64
149   elmTypeParam <- newParam "elmTypeA" $ Prim elmType
150   yParam <- newParam "Y" $ Prim int64
151   xParam <- newParam "X" $ Prim int64
152   mWarpsParam <- newParam "mWarps" $ Prim int64
153   nWarpsParam <- newParam "nWarps" $ Prim int64
154
155   fName <-
156     fmap (nameFromString . prettyString) $ newName $ VName copyGlobalSharedName 0
157
158   let sharedOut =
159         [ ( Array elmType (fmap Free arrShape) Unique,
160             RetAls [] []
161           )
162         ]
163   let funParams =
164         [ globalParam,
165           sharedParam,
166           offsetParam,
167           elmTypeParam,
168           yParam,
169           xParam,
170           mWarpsParam,
171           nWarpsParam
172         ]
173   pure
174     ( CopyGlobalSharedSignature elmType sizeY sizeX,
175       FunDef Nothing mempty fName sharedOut funParams $
```

```
176           resultBody [Var $ paramName sharedParam]
177       )
178
179   -- | Create the copy from registers to shared memory definition.
180   mkCopyRegistersShared ::
181     (MonadFreshNames m) =>
182     PrimType ->
183     PrimType ->
184     PrimType ->
185     Int ->
186     Int ->
187     Int ->
188     Int ->
189     m TcFunDef
190   mkCopyRegistersShared elmTypeA elmTypeB elmTypeC sizeM sizeN sizeRegs blockSize =
      ↪  do
191     registersParam <-
192       newParam "registers" $
193         Array
194           elmTypeC
195           (Shape [mkInt64Const blockSize, mkInt64Const sizeRegs])
196           Nonunique
197     sharedParam <-
198       newParam "shared" $
199         Array
200           elmTypeC
201           (Shape [mkInt64Const sizeM, mkInt64Const sizeN])
202           Unique
203     aElmTypeParam <- newParam "elmTypeA" $ Prim elmTypeA
204     bElmTypeParam <- newParam "elmTypeB" $ Prim elmTypeB
205     mParam <- newParam "M" $ Prim int64
206     nParam <- newParam "N" $ Prim int64
207     mWarpsParam <- newParam "mWarps" $ Prim int64
208     nWarpsParam <- newParam "nWarps" $ Prim int64
209
210     fName <-
211       nameFromString . prettyString
212         <$> newName (VName copyRegistersSharedName 0)
213
214     let sharedOut =
215           [ ( Array
216                 elmTypeC
217                 (Shape [Free $ mkInt64Const sizeM, Free $ mkInt64Const sizeN])
218                 Unique,
219               RetAls [] []
220             )
221           ]
222     let funParams =
223           [ registersParam,
224             sharedParam,
225             aElmTypeParam,
226             bElmTypeParam,
227             mParam,
228             nParam,
229             mWarpsParam,
230             nWarpsParam
```

```
231           ]
232     pure
233       ( CopyRegistersSharedSignature elmTypeC sizeM sizeN sizeRegs blockSize,
234         FunDef Nothing mempty fName sharedOut funParams $
235           resultBody [Var $ paramName sharedParam]
236       )
237
238 -- | Create some  shared (scratch) memory of the specified type and shape.
239 scratchMem :: PrimType -> [Int] -> Exp GPU
240 scratchMem elmType dims = BasicOp $ Scratch elmType $ map mkInt64Const dims
241
242 -- | Rebuild the matrix multiplication computation.  The SegRed will be
243 -- replae by calls to three tensor core related functions.
244 buildMMM :: VName -> Int -> TensorCoreMatch -> Builder GPU TcFuns
245 buildMMM
246   resName
247   actualBlockSize
248   match@(TensorCoreMatch kernelBodyMatch ne sizeM sizeN sizeK) = do
249     -- Get the best tile of warps given the block size.
250     let (warpsM, warpsN) = getOptimalWarps actualBlockSize match
251     let blockSize = warpsM * warpsN * 32
252     let cValsPerThread = sizeM * sizeN `div` blockSize
253
254     let elmTypeC = typeC kernelBodyMatch
255         elmTypeB = typeB kernelBodyMatch
256         elmTypeA = typeA kernelBodyMatch
257     gemmFun <- mkGemmFun elmTypeA elmTypeB elmTypeC sizeM sizeN sizeK
      ↪  cValsPerThread
258     copyGlobalSharedFunA <- mkCopyGlobalShared (typeA kernelBodyMatch) sizeM
      ↪  sizeK
259     copyGlobalSharedFunB <- mkCopyGlobalShared (typeB kernelBodyMatch) sizeK
      ↪  sizeN
260     copyRegistersSharedFun <-
261       mkCopyRegistersShared -- Maybe too much formatting
262         elmTypeA
263         elmTypeB
264         elmTypeC
265         sizeM
266         sizeN
267         cValsPerThread
268         blockSize
269     let addedFuns =
270           [ gemmFun,
271             copyGlobalSharedFunA,
272             copyGlobalSharedFunB,
273             copyRegistersSharedFun
274           ]
275
276     let thrdInBlock = SegThreadInBlock SegNoVirt
277     cRegs_list <-
278       segMap1D "cRegs" thrdInBlock ResultPrivate (mkInt64Const blockSize) $ \_ ->
        ↪  do
279         cScratch <- letExp "cScratch" $ scratchMem elmTypeC [cValsPerThread]
280         cLoop <- forLoop (mkInt64Const cValsPerThread) [cScratch] $ \i [cMerge]
          ↪  -> do
281           cZeroed <- update "cZeroed" cMerge [i] ne
```

```
282           resultBodyM [Var cZeroed]
283         pure [varRes cLoop]
284     let [cRegs] = cRegs_list
285     aScratch <- letExp "aScratch" $ scratchMem elmTypeA [sizeM, sizeK]
286     bScratch <- letExp "bScratch" $ scratchMem elmTypeB [sizeK, sizeN]
287
288     let innerIndecesASlice =
289           [ DimSlice (mkInt64Const 0) (mkInt64Const sizeM) (mkInt64Const 1),
290             DimSlice (mkInt64Const 0) (mkInt64Const sizeK) (mkInt64Const 1)
291           ]
292         innerIndecesBSlice =
293           [ DimSlice (mkInt64Const 0) (mkInt64Const sizeK) (mkInt64Const 1),
294             DimSlice (mkInt64Const 0) (mkInt64Const sizeN) (mkInt64Const 1)
295           ]
296     slicedA <-
297       letExp "slicedA" $
298         BasicOp $
299           Index (arrA kernelBodyMatch) $
300             Slice $
301               fmap DimFix (outerIndecesA kernelBodyMatch) <> innerIndecesASlice
302     slicedB <-
303       letExp "slicedB" $
304         BasicOp $
305           Index (arrB kernelBodyMatch) $
306             Slice $
307               fmap DimFix (outerIndecesB kernelBodyMatch) <> innerIndecesBSlice
308
309     --  Need to pass this explicitly as LMAD info is lost on function call
310     let pe64DimsA =
311           fmap pe64 $
312             outerDimsA kernelBodyMatch
313               <> [mkInt64Const sizeM, mkInt64Const sizeK]
314         pe64IndiciesA =
315           fmap pe64 $
316             outerIndecesA kernelBodyMatch
317               <> [mkInt64Const 0, mkInt64Const 0]
318         pe64DimsB =
319           fmap pe64 $
320             outerDimsB kernelBodyMatch
321               <> [mkInt64Const sizeK, mkInt64Const sizeN]
322         pe64IndiciesB =
323           fmap pe64 $
324             outerIndecesB kernelBodyMatch
325               <> [mkInt64Const 0, mkInt64Const 0]
326
327     flatIndexAExp <- toExp $ flattenIndex pe64DimsA pe64IndiciesA
328     offsetA <- letExp "offsetA" flatIndexAExp
329     flatIndexBExp <- toExp $ flattenIndex pe64DimsB pe64IndiciesB
330     offsetB <- letExp "offsetB" flatIndexBExp
331
332     let copyArgsA =
333           [ (Var slicedA, ObservePrim),
334             (Var aScratch, Consume),
335             (Var offsetA, ObservePrim),
336             (Constant $ blankPrimValue $ typeA kernelBodyMatch, ObservePrim),
337             (mkInt64Const sizeM, ObservePrim),
```

```
338             (mkInt64Const sizeK, ObservePrim),
339             (mkInt64Const warpsM, ObservePrim),
340             (mkInt64Const warpsN, ObservePrim)
341           ]
342         copyRetsA =
343           [ ( Array
344                 (typeA kernelBodyMatch)
345                 (Shape [Free $ mkInt64Const sizeM, Free $ mkInt64Const sizeK])
346                 Unique,
347               RetAls [] []
348             )
349           ]
350
351         copyArgsB =
352           [ (Var slicedB, ObservePrim),
353             (Var bScratch, Consume),
354             (Var offsetB, ObservePrim),
355             (Constant $ blankPrimValue $ typeB kernelBodyMatch, ObservePrim),
356             (mkInt64Const sizeK, ObservePrim),
357             (mkInt64Const sizeN, ObservePrim),
358             (mkInt64Const warpsM, ObservePrim),
359             (mkInt64Const warpsN, ObservePrim)
360           ]
361         copyRetsB =
362           [ ( Array
363                 (typeB kernelBodyMatch)
364                 (Shape [Free $ mkInt64Const sizeK, Free $ mkInt64Const sizeN])
365                 Unique,
366               RetAls [] []
367             )
368           ]
369
370       aCopied <-
371         letExp "aCopied" $
372           Apply
373             (funDefName $ snd copyGlobalSharedFunA)
374             copyArgsA
375             copyRetsA
376             (Safe, SrcLoc NoLoc, [])
377
378       bCopied <-
379         letExp "bCopied" $
380           Apply
381             (funDefName $ snd copyGlobalSharedFunB)
382             copyArgsB
383             copyRetsB
384             (Safe, SrcLoc NoLoc, [])
385
386       let blksize = mkInt64Const blockSize
387       inBlockMMAres_list <-
388         segMap1D "inBlockMMAres" thrdInBlock ResultPrivate blksize $ \thread_idx ->
          ↪  do
389           threadCregs <- index "threadCregs" cRegs [thread_idx]
390           let mmmArgs =
391                 [ (Var aCopied, ObservePrim),
392                   (Var bCopied, ObservePrim),
```

```
393                 (Var threadCregs, Consume),
394                 (Constant $ blankPrimValue $ typeA kernelBodyMatch, ObservePrim),
395                 (Constant $ blankPrimValue $ typeB kernelBodyMatch, ObservePrim),
396                 (mkInt64Const sizeM, ObservePrim),
397                 (mkInt64Const sizeN, ObservePrim),
398                 (mkInt64Const sizeK, ObservePrim),
399                 (mkInt64Const warpsM, ObservePrim),
400                 (mkInt64Const warpsN, ObservePrim),
401                 (mkInt64Const 1, ObservePrim),
402                 (mkInt64Const 1, ObservePrim)
403               ]
404         let mmmRets =
405               [ ( Array
406                     (typeC kernelBodyMatch)
407                     (Shape [Free $ mkInt64Const cValsPerThread])
408                     Unique,
409                   RetAls [] []
410                 )
411               ]
412         threadMMAres <-
413           letExp "threadMMAres" $
414             Apply
415               (funDefName $ snd gemmFun)
416               mmmArgs
417               mmmRets
418               (Safe, SrcLoc NoLoc, [])
419         pure [varRes threadMMAres]
420       let [inBlockMMAres] = inBlockMMAres_list
421
422       cScratch <- letExp "cScratch" $ scratchMem elmTypeC [sizeM, sizeN]
423       let copyArgsC =
424             [ (Var inBlockMMAres, ObservePrim),
425               (Var cScratch, Consume),
426               (Constant $ blankPrimValue $ typeA kernelBodyMatch, ObservePrim),
427               (Constant $ blankPrimValue $ typeB kernelBodyMatch, ObservePrim),
428               (mkInt64Const sizeM, ObservePrim),
429               (mkInt64Const sizeN, ObservePrim),
430               (mkInt64Const warpsM, ObservePrim),
431               (mkInt64Const warpsN, ObservePrim)
432             ]
433       let copyRetsC =
434             [ ( Array
435                   (typeC kernelBodyMatch)
436                   (Shape [Free $ mkInt64Const sizeM, Free $ mkInt64Const sizeN])
437                   Unique,
438                 RetAls [] []
439               )
440             ]
441       cCopied <-
442         letExp "cCopied" $
443           Apply
444             (funDefName $ snd copyRegistersSharedFun)
445             copyArgsC
446             copyRetsC
447             (Safe, SrcLoc NoLoc, [])
448
```

```
449      letBindNames [resName] $ BasicOp $ SubExp $ Var cCopied
450      pure addedFuns
451
452  -- Functions for traversing the input program and transforming the relevant
453  -- statement to use tensor cores.
454  transformProg :: Prog GPU -> PassM (Prog GPU)
455  transformProg (Prog opaqueTypes consts funs) = do
456    (transformedFuns, mmmFuns) <-
457      modifyNameSource $
458        (\(a, s, w) -> ((a, w), s)) . runRWS (mapM transformFunDef funs) init_env
459    let (_, addedFuns) = unzip mmmFuns
460    pure $ Prog opaqueTypes consts (addedFuns <> transformedFuns)
461    where
462      init_env = ExtractTcEnv (scopeOf consts) Nothing
463
464  transformFunDef :: FunDef GPU -> TensorCoreM (FunDef GPU)
465  transformFunDef funDef@(FunDef entry attrs name retType params body) =
466    FunDef entry attrs name retType params <$> inScopeOf funDef (transformBody
       ↪  body)
467
468  transformStms :: Stms GPU -> TensorCoreM (Stms GPU)
469  transformStms = mapStmsWithScope transformStm
470
471  transformStm :: Stm GPU -> TensorCoreM (Stms GPU)
472  transformStm stm@(Let (Pat [PatElem resName _]) _ e) = do
473    scope <- askScope
474    maybeBlockSize <- askBlockSize
475    case (innerSegOpExpMatch scope e, maybeBlockSize) of
476      (Just match, Just blockSize) -> do
477        (mmmFuns, stms) <- runBuilderMMM (buildMMM resName blockSize match) scope
478        tell mmmFuns
479        pure stms
480      _ -> transformStmDefault stm
481  transformStm stm = transformStmDefault stm
482
483  transformStmDefault :: Stm GPU -> TensorCoreM (Stms GPU)
484  transformStmDefault (Let pat aux e) = do
485    e' <- transformExp e
486    pure $ oneStm $ Let pat aux e'
487
488  -- TODO: match WithAcc?
489  transformExp :: Exp GPU -> TensorCoreM (Exp GPU)
490  transformExp (Match subExps cases body matchDec) =
491    Match subExps
492      <$> mapM transformCase cases
493      <*> transformBody body
494      <*> pure matchDec
495  transformExp (Loop params form body) =
496    localScope (scopeOfFParams (map fst params) <> scopeOfLoopForm form) $ do
497      newBody <- transformBody body
498      pure $ Loop params form newBody
499  transformExp (Op op) = Op <$> transformOp op
500  transformExp e = pure e
501
502  transformCase :: Case (Body GPU) -> TensorCoreM (Case (Body GPU))
503  transformCase (Case pat body) = Case pat <$> transformBody body
```

```
504
505 transformBody :: Body GPU -> TensorCoreM (Body GPU)
506 transformBody (Body dec stms res) =
507   Body dec
508     <$> transformStms stms
509     <*> pure res
510
511 transformOp :: Op GPU -> TensorCoreM (Op GPU)
512 transformOp (SegOp sOp) = SegOp <$> transformSegOp sOp
513 transformOp op = pure op
514
515 -- | First we match an out SegMap to get the block size.  Later we
516 -- try to set the block size dependent on the matrix multiplication dims.
517 transformSegOp :: SegOp SegLevel GPU -> TensorCoreM (SegOp SegLevel GPU)
518 transformSegOp
519   sOp@( SegMap
520           ( SegBlock
521               SegNoVirt
522               (Just (KernelGrid (Count numBlocks) (Count _blockSize)))
523             )
524           space@(SegSpace _ _)
525           ts
526           body@(KernelBody _ stms _)
527         ) = do
528     scope <- askScope
529     case execWriter $ runReaderT (maxBlockSizeStms stms) scope of
530       Known (Max maxBlockSize) -> do
531         transformedBody <-
532           localBlockSize (const $ Just maxBlockSize) $
533             transformKernelBody body
534         let blocks = Count numBlocks
535         let blocksize = Count $ mkInt64Const maxBlockSize
536         let grid = KernelGrid blocks blocksize
537         pure $ SegMap (SegBlock SegNoVirt (Just grid)) space ts transformedBody
538       Unknown ->
539         transformSegOpDefault sOp
540 transformSegOp sOp = transformSegOpDefault sOp
541
542 transformSegOpDefault :: SegOp SegLevel GPU -> TensorCoreM (SegOp SegLevel GPU)
543 transformSegOpDefault (SegMap level space ts body) =
544   SegMap level space ts
545     <$> transformKernelBody body
546 transformSegOpDefault (SegRed level space ops ts body) =
547   SegRed level space ops ts
548     <$> transformKernelBody body
549 transformSegOpDefault (SegScan level space ops ts body) =
550   SegScan level space ops ts
551     <$> transformKernelBody body
552 transformSegOpDefault (SegHist level space ops hist body) =
553   SegHist level space ops hist
554     <$> transformKernelBody body
555
556 transformKernelBody :: KernelBody GPU -> TensorCoreM (KernelBody GPU)
557 transformKernelBody (KernelBody desc stms res) =
558   KernelBody desc
559     <$> transformStms stms
```

```
560       <*> pure res
561
562   -- | Do we know the block size or not?
563   data KnownUnknown a = Known a | Unknown
564     deriving (Show, Eq, Ord)
565
566   instance (Monoid a) => Monoid (KnownUnknown a) where
567     mempty = Known mempty
568
569   instance (Semigroup a) => Semigroup (KnownUnknown a) where
570     Known a <> Known b = Known $ a <> b
571     _ <> _ = Unknown
572
573   type MaxBlockSizeM = ReaderT (Scope GPU) (Writer (KnownUnknown (Max Int)))
574
575   -- | Find the max block size required for the intragroup kernels.
576   -- This is needed in case more threads are needed than what is ideal for
577   -- matmul using the tensor cores.
578   maxBlockSizeWalker :: Walker GPU MaxBlockSizeM
579   maxBlockSizeWalker =
580     (identityWalker @GPU)
581       { walkOnOp = maxBlockSizeOp,
582         walkOnBody = maxBlockSizeBody
583       }
584
585   maxBlockSizeStms :: Stms GPU -> MaxBlockSizeM ()
586   maxBlockSizeStms = mapStmsWithScope maxBlockSizeStm
587
588   maxBlockSizeStm :: Stm GPU -> MaxBlockSizeM ()
589   maxBlockSizeStm (Let _ _ e) = maxBlockSizeExp e
590
591   maxBlockSizeExp :: Exp GPU -> MaxBlockSizeM ()
592   maxBlockSizeExp = walkExpM maxBlockSizeWalker
593
594   maxBlockSizeOp :: Op GPU -> MaxBlockSizeM ()
595   maxBlockSizeOp op = do
596     scope <- askScope
597     case (innerOpMatch scope op, op) of
598       (Just match, _) -> do
599         tell $ Known $ Max $ getOptimalBlockSize match
600       (_, SegOp sOp)
601         | (SegThreadInBlock _) <- segLevel sOp ->
602             tell $ foldl prodKnownSegDim (Known 1) $ unSegSpace $ segSpace sOp
603       _ -> pure ()
604
605   maxBlockSizeBody :: Scope GPU -> Body GPU -> MaxBlockSizeM ()
606   maxBlockSizeBody scope (Body _ stms _) = localScope scope $ maxBlockSizeStms stms
607
608   prodKnownSegDim ::
609     KnownUnknown (Max Int) ->
610     (VName, SubExp) ->
611     KnownUnknown (Max Int)
612   prodKnownSegDim (Known (Max acc)) (_, Constant (IntValue n)) =
613     Known $ Max $ acc * valueIntegral n
614   -- TODO: should lookup if variable dimension?
615   -- Currently we do not match if any dimensions are not statically known integers.
```

```haskell
-- The only reason a lookup might be relevant is if the variable is bound to a
-- statically known variable. This might already be handled by constant folding.
prodKnownSegDim _ _ = Unknown

getFactorPairs :: Int -> [(Int, Int)]
getFactorPairs n = [(x, n `div` x) | x <- [1 .. n], n `mod` x == 0]

getRatio :: Int -> Int -> Float
getRatio m n = fromIntegral m / fromIntegral n

-- Note: This should in the future depend on the type.
-- F64 might require a different warp layout than f16 for best performance.
getOptimalWarps :: Int -> TensorCoreMatch -> (Int, Int)
getOptimalWarps blockSize (TensorCoreMatch _ _ sizeM sizeN _) =
  -- warp tiles will be as close to square as possible, which should maximize
  ↪  register reuse
  let Arg _ (warpsM, warpsN) = minimum $ map ratioDifference usedfactorPairs
   in (warpsM, warpsN)
  where
    minValsPerThread = 8
    maxBlockSize = (sizeM * sizeN) `div` minValsPerThread
    usedBlockSize = min blockSize maxBlockSize
    targetRatio = getRatio sizeM sizeN
    -- Minimum values per thread used for f16 MMA Atom
    usedfactorPairs = helper $ usedBlockSize `divUp` 32
    ratioDifference (x, y) = Arg (abs $ targetRatio - getRatio x y) (x, y)
    helper 0 = error "Could not find appropriate number of warps"
    helper numwarps =
      let factorPairs = getFactorPairs numwarps
       in if not $ any (\(x, y) -> (sizeM `div` 16) `mod` x == 0 && (sizeN `div`
       ↪  16) `mod` y == 0) factorPairs
            then
              helper $ numwarps - 1
            else
              factorPairs

-- NOTE: In the future the optimal block size should depend on:
--  1. The array element type (f16 or f64)
--  2. Architecture
--  3. Type of program. Sometimes more threads can be good.
-- The current estimate might not be optimal in all cases.
getOptimalBlockSize :: TensorCoreMatch -> Int
getOptimalBlockSize (TensorCoreMatch _ _ sizeM sizeN _sizeK) =
  let optimalElmsPerWarp = 4096
   in ((sizeM * sizeN) `divUp` optimalElmsPerWarp) * 32

-- Pattern matching

data TensorCoreMatch = TensorCoreMatch
  { kernelBodyMatch :: KernelBodyMatch,
    ne :: SubExp,
    sizeM :: Int,
    sizeN :: Int,
    sizeK :: Int
  }
  deriving (Show, Eq, Ord)
```

```haskell

-- NOTE: The only type currently supported is f16 for A and B and f16/f32 for C.
-- Future work should also support f64 or maybe even mixed u8/s32.
data KernelBodyMatch = KernelBodyMatch
  { innerIndecesA :: [VName],
    innerIndecesB :: [VName],
    outerIndecesA :: [SubExp],
    outerIndecesB :: [SubExp],
    outerDimsA :: [SubExp],
    outerDimsB :: [SubExp],
    arrA :: VName,
    arrB :: VName,
    m :: VName,
    n :: VName,
    k :: VName,
    typeA :: PrimType,
    typeB :: PrimType,
    typeC :: PrimType
  }
  deriving (Show, Eq, Ord)

innerSegOpExpMatch :: Scope GPU -> Exp GPU -> Maybe TensorCoreMatch
innerSegOpExpMatch scope (Op op) = innerOpMatch scope op
innerSegOpExpMatch _ _ = Nothing

innerOpMatch :: Scope GPU -> Op GPU -> Maybe TensorCoreMatch
innerOpMatch
  scope
  ( SegOp
      segRed@(SegRed (SegThreadInBlock _) space segBinOps _ts body)
    )
    | Just ne <- segBinOpsMatch segBinOps =
        do
          let (dimVars, segDims) = unzip $ unSegSpace space
          let freeVars = freeIn segRed
          bodyMatch <- inBlockKernelBodyMatch dimVars freeVars body scope
          constSegDims <- mapM constantValueMatch segDims
          case constSegDims of
            [m, n, k]
              | all sizeMatches constSegDims ->
                  Just (TensorCoreMatch bodyMatch ne m n k)
            _ -> Nothing
innerOpMatch _ _ = Nothing

-- NOTE: This should definitely depend on the element type when more types are
-- supported. The current bounds fit within the shared memory when using f16
-- on the A100.
sizeMatches :: Int -> Bool
sizeMatches x =
  x `mod` 16 == 0
    && 0 < x
    && x <= 128
    --  Check if x is power of 2
    && popCount x == 1

constantValueMatch :: SubExp -> Maybe Int
```

```
726 constantValueMatch (Constant (IntValue v)) = Just $ valueIntegral v
727 constantValueMatch _ = Nothing
728
729 -- Does the list of indexing variables only have a single dimension?
730 singleDim :: [d] -> Maybe d
731 singleDim [v] = Just v
732 singleDim _ = Nothing
733
734 hasCorrectMatMulOperandType :: Type -> PrimType -> Bool
735 hasCorrectMatMulOperandType (Array typ _ _) pt = typ == pt
736 hasCorrectMatMulOperandType _ _ = False
737
738 -- The indexVars corresponds to the variables from the SegSpace
739 inBlockKernelBodyMatch ::
740   [VName] ->
741   Names ->
742   KernelBody GPU ->
743   Scope GPU ->
744   Maybe KernelBodyMatch
745 inBlockKernelBodyMatch
746   indexVars@[_, _, indexVar3]
747   freeVars
748   (KernelBody _ stms [Returns _ _ (Var res)])
749   scope = do
750     let f16_type = FloatType Float16
751     let sTable = ST.insertStms (informStms stms) $ ST.fromScope $ addScopeWisdom
      ↪  scope
752     (resExp, _) <- ST.lookupExp res sTable
753     -- Check that the result is optionally converted from f16->f32.
754     -- In case it is not, then since the program type checks and the
755     -- arrays are asserted to be f16, then all operators must be f16
756     resWithoutConversion <- case resExp of
757       BasicOp (ConvOp (FPConv Float16 Float32) (Var converted)) -> do
758         (convertedExp, _) <- ST.lookupExp converted sTable
759         pure convertedExp
760       notConvertedExp ->
761         pure notConvertedExp
762     (mulArg1, mulArg2) <- matchesMul $ removeExpWisdom resWithoutConversion
763     (mulArg1Exp, _) <- ST.lookupExp mulArg1 sTable
764     (mulArg2Exp, _) <- ST.lookupExp mulArg2 sTable
765     (arr1, slice1) <- matchesMulArg $ removeExpWisdom mulArg1Exp
766     (arr2, slice2) <- matchesMulArg $ removeExpWisdom mulArg2Exp
767
768     -- For now we only support mixed f16/f32 tensor core operations.
769     -- Therefore the array operands A and B in C = A @ B must be f16.
770     arr1Type <- ST.lookupType arr1 sTable
771     arr2Type <- ST.lookupType arr2 sTable
772     guard $ hasCorrectMatMulOperandType arr1Type f16_type
773     guard $ hasCorrectMatMulOperandType arr2Type f16_type
774
775     resType <- ST.lookupType res sTable
776     slice1' <- mapM dimFix $ unSlice slice1
777     slice2' <- mapM dimFix $ unSlice slice2
778     let seIndexVars = map Var indexVars
779     let (seInnerIndeces1, outerIndeces1) = partition (`elem` seIndexVars) slice1'
780     let (seInnerIndeces2, outerIndeces2) = partition (`elem` seIndexVars) slice2'
```

```haskell
    let outerIndeces = outerIndeces1 <> outerIndeces2
    innerIndeces1 <- mapM getIndexVar seInnerIndeces1
    innerIndeces2 <- mapM getIndexVar seInnerIndeces2
    -- Check that each array has one unique (n or m) and one commen (k) dimension
    -- as the inner dimensions of the intragroup kernel
    k <- singleDim $ innerIndeces1 `intersect` innerIndeces2
    n <- singleDim $ toList $ fromList innerIndeces1 `difference` fromList
     ↪  innerIndeces2
    m <- singleDim $ toList $ fromList innerIndeces2 `difference` fromList
     ↪  innerIndeces1
    -- TODO: Do we maybe want to allow something that is not matrix
     ↪  multiplication?
    -- It would just require us to "not" transpose B in CuTe
    -- In the meantime, this checks where in the indexing slice k appears.
    -- For B it must be [n, k] and for A it must be [k, n]
    elemIndex k innerIndeces1 >>= guard . (== 1) -- [m, k] matrix
    elemIndex k innerIndeces2 >>= guard . (== 0) -- [k, n] matrix
    case (arr1Type, arr2Type, resType) of
      (Array type1 (Shape arr1Dims) _, Array type2 (Shape arr2Dims) _, Prim
       ↪  resTypePrim)
        | k == indexVar3 && all (`subExpFreeIn` freeVars) outerIndeces ->
            let arr1OuterDims = take (length arr1Dims - 2) arr1Dims
             in let arr2OuterDims = take (length arr2Dims - 2) arr2Dims
                 in Just
                      ( KernelBodyMatch
                          innerIndeces1
                          innerIndeces2
                          outerIndeces1
                          outerIndeces2
                          arr1OuterDims
                          arr2OuterDims
                          arr1
                          arr2
                          m
                          n
                          k
                          type1
                          type2
                          resTypePrim
                      )
      _ -> Nothing
inBlockKernelBodyMatch _ _ _ _ = Nothing

getIndexVar :: SubExp -> Maybe VName
getIndexVar (Var v) = Just v
getIndexVar _ = Nothing

-- A bit weird, but we also count constants as free
subExpFreeIn :: SubExp -> Names -> Bool
subExpFreeIn (Constant _) _ = True
subExpFreeIn (Var v) names = v `nameIn` names

matchesMul :: Exp GPU -> Maybe (VName, VName)
matchesMul (BasicOp (BinOp (FMul _) (Var arg1) (Var arg2))) = Just (arg1, arg2)
matchesMul _ = Nothing
```

```haskell
matchesMulArg :: Exp GPU -> Maybe (VName, Slice SubExp)
matchesMulArg (BasicOp (Index v s)) = Just (v, s)
matchesMulArg _ = Nothing

segBinOpsMatch :: [SegBinOp GPU] -> Maybe SubExp
segBinOpsMatch [SegBinOp Commutative lambda nes _]
  | lambdaMatch lambda = nesMatch nes
segBinOpsMatch _ = Nothing

lambdaMatch :: Lambda GPU -> Bool
lambdaMatch (Lambda [Param _ arg1 _, Param _ arg2 _] _ body) =
  lambdaBodyMatch arg1 arg2 body
lambdaMatch _ = False

lambdaBodyMatch :: VName -> VName -> Body GPU -> Bool
lambdaBodyMatch arg1 arg2 (Body _ stms [SubExpRes _ (Var v)]) =
  any (lambdaStmMatch arg1 arg2 v) stms
lambdaBodyMatch _ _ _ = False

lambdaStmMatch :: VName -> VName -> VName -> Stm GPU -> Bool
lambdaStmMatch
  arg1
  arg2
  v
  ( Let
      (Pat [PatElem v' _])
      _
      ( BasicOp
          (BinOp (FAdd _) (Var arg1') (Var arg2'))
      )
  ) =
    v == v' && arg1 == arg1' && arg2 == arg2'
lambdaStmMatch
  arg1
  arg2
  v
  ( Let
      (Pat [PatElem v' _])
      _
      (BasicOp (BinOp (Add _ _) (Var arg1') (Var arg2')))
  ) =
    v == v' && arg1 == arg1' && arg2 == arg2'
lambdaStmMatch _ _ _ _ = False

nesMatch :: [SubExp] -> Maybe SubExp
nesMatch [s@(Constant v)] | zeroIsh v = Just s
nesMatch _ = Nothing
```

A.4 TensorCoreMemFixup.hs

```haskell
module Futhark.Optimise.TensorCores.TensorCoreMemFixup
  (fixFuns)
where

import Control.Monad
import Control.Monad.RWS.Strict
```

```haskell
import Data.List (lookup)
import Data.Semigroup
import Futhark.IR.GPU
import Futhark.IR.GPUMem
import Futhark.Optimise.TensorCores.Utils
import Futhark.Pass (PassM)
import Prelude hiding (lookup)


type FixEnv = Scope GPUMem

data SpaceType = Device | Shared | Scalar

type FixState = [(VName, (VName, Bool))]

-- The monad the memory fixup runs within.
type FixM a = RWST FixEnv () FixState PassM a

fixFuns :: Stms GPUMem -> FunDef GPUMem -> PassM (FunDef GPUMem)
fixFuns consts fun
  | gemmName `isPrefixOfName` funDefName fun =
      pure $
        fun
          { funDefParams = fixParamsGemmFun $ funDefParams fun,
            funDefRetType = fixRetType Scalar $ funDefRetType fun
          }
  | copyGlobalSharedName `isPrefixOfName` funDefName fun =
      pure $
        fun
          { funDefParams = fixParamsCopyGlobalShared $ funDefParams fun,
            funDefRetType = fixRetType Shared $ funDefRetType fun
          }
  | copyRegistersSharedName `isPrefixOfName` funDefName fun =
      pure $
        fun
          { funDefParams = fixParamsCopyRegistersShared $ funDefParams fun,
            funDefRetType = fixRetType Shared $ funDefRetType fun
          }
  | otherwise = do
      let initScope = scopeOf consts <> scopeOfFParams (funDefParams fun)
      let body = funDefBody fun
      stms' <- fixStmtsWithScope initScope . bodyStms $ body
      pure $ fun {funDefBody = body {bodyStms = stms'}}

fixParamsCopyGlobalShared :: [FParam GPUMem] -> [FParam GPUMem]
fixParamsCopyGlobalShared
  ( Param attrs1 vName1 (MemMem (Space "device"))
      : Param attrs2 vName2 (MemMem (Space "device"))
      : rest
    ) =
    Param attrs1 vName1 (MemMem (Space "device"))
      : Param attrs2 vName2 (MemMem (Space "shared"))
      : rest
fixParamsCopyGlobalShared params = params

fixParamsCopyRegistersShared :: [FParam GPUMem] -> [FParam GPUMem]
```

```haskell
fixParamsCopyRegistersShared
  ( Param attrs1 vName1 (MemMem (Space "device"))
      : Param attrs2 vName2 (MemMem (Space "device"))
      : p3@(Param _ _ (MemArray t shp _ (ArrayIn _ _)))
      : rest
    ) =
    Param attrs1 vName1 (MemMem space)
      : Param attrs2 vName2 (MemMem (Space "shared"))
      : p3
      : rest
    where
      space = ScalarSpace (drop 1 $ shapeDims shp) t
fixParamsCopyRegistersShared params = params

fixParamsGemmFun :: [FParam GPUMem] -> [FParam GPUMem]
fixParamsGemmFun
  ( Param attrs1 vName1 (MemMem (Space "device"))
      : Param attrs2 vName2 (MemMem (Space "device"))
      : Param attrs3 vName3 (MemMem (Space "device"))
      : p4
      : p5
      : p6@(Param _ _ (MemArray t shp _ (ArrayIn _ _)))
      : rest
    ) =
    Param attrs1 vName1 (MemMem (Space "shared"))
      : Param attrs2 vName2 (MemMem (Space "shared"))
      : Param attrs3 vName3 (MemMem space)
      : p4
      : p5
      : p6
      : rest
    where
      space = ScalarSpace (shapeDims shp) t
fixParamsGemmFun params = params

fixRetType ::
  SpaceType ->
  [(RetType GPUMem, RetAls)] ->
  [(RetType GPUMem, RetAls)]
fixRetType
  spaceType
  [ (MemMem (Space "device"), als1),
    (MemArray t shp u (ReturnsNewBlock (Space "device") n lmad), als2)
    ] =
    --  TODO: check if ReturnsInBlock is preferred
    [ (MemMem newSpace, als1),
      (MemArray t shp u (ReturnsNewBlock newSpace n lmad), als2)
    ]
    where
      getNewSpace Device = Space "device"
      getNewSpace Shared = Space "shared"
      getNewSpace Scalar = ScalarSpace (fmap extToSubExp (shapeDims shp)) t
      newSpace = getNewSpace spaceType
fixRetType _ rets = rets

extToSubExp :: ExtSize -> SubExp
```

```haskell
extToSubExp (Ext n) = mkInt64Const n
extToSubExp (Free se) = se

fixStmtsWithScope :: Scope GPUMem -> Stms GPUMem -> PassM (Stms GPUMem)
fixStmtsWithScope scope stms = do
  (res, _, _) <- runRWST (fixStmts stms) scope []
  pure res

fixStmts :: Stms GPUMem -> FixM (Stms GPUMem)
fixStmts = mapStmsWithScope fixStmt

fixStmt :: Stm GPUMem -> FixM (Stms GPUMem)
fixStmt
  stm@( Let
          (Pat [PatElem resName (MemArray _ _ _ (ArrayIn resMem _))])
          _
          (BasicOp (Manifest _ inputName))
        ) = do
    info <- lookupInfo inputName
    case info of
      LetName (MemArray _ _ _ (ArrayIn inputMem _)) -> do
        modify ([(resName, (inputName, False)), (resMem, (inputMem, False))] <>)
        defaultFixStm stm
      _ -> defaultFixStm stm
fixStmt
  ( Let
      ( Pat
          [ PatElem vName1 _,
            PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
            ]
      )
      aux
      (Apply fName args rets info)
  ) | gemmName `isPrefixOfName` fName = do
    let space = ScalarSpace (shapeDims shp2) t2
    let newRets = fixRetType Scalar rets
    -- For each argument we
    (replacedArgs, removedCopy) <- mapAndUnzipM replaceArg args
    let (removedAcopy : removedBcopy : _) = removedCopy
    -- If these are true, and a manifest copy was removed, we can do swizzling,
    -- otherwise the arrays are already in shared and we cannot swizzle them.
    let removedAorB =
          [ (mkInt64Const $ boolToInt $ not removedAcopy, ObservePrim),
            (mkInt64Const $ boolToInt $ not removedBcopy, ObservePrim)
          ]
    let newArgs = take (length replacedArgs - 2) replacedArgs <> removedAorB
    pure $
      oneStm $
        Let
          ( Pat
              [ PatElem vName1 (MemMem space),
                PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
              ]
          )
          aux
          (Apply fName newArgs newRets info)
```

```
fixStmt
  ( Let
      ( Pat
          [ PatElem vName1 _,
            PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
            ]
        )
      aux
      (Apply fName args rets info)
    ) | copyGlobalSharedName `isPrefixOfName` fName = do
    let space = Space "shared"
    --  TODO: check if need to handle uniqueness/consumption
    let newRets = fixRetType Shared rets
    (newArgs, _removedCopy) <- mapAndUnzipM replaceArg args
    let ((Var srcMemMem, _) : _ : (Var srcArray, _) : _restArgs) = newArgs
    srcMemInfo <- lookupInfo srcMemMem
    case srcMemInfo of
      LetName (MemMem srcMemSpace)
        | srcMemSpace == space ->
            -- Array is already in shared. Therefore, the copyGlobalShared call
            -- should be removed and we return removedCopy=True.
            modify ([(vName2, (srcArray, True)), (vName1, (srcMemMem, True))] <>)
      _ ->
        pure ()
    pure
      $ oneStm
      $ Let
        ( Pat
            [ PatElem vName1 (MemMem space),
              PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
            ]
        )
        aux
      $ Apply fName newArgs newRets info
fixStmt
  ( Let
      ( Pat
          [ PatElem vName1 _,
            PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
            ]
        )
      aux
      (Apply fName args rets info)
    ) | copyRegistersSharedName `isPrefixOfName` fName = do
    let space = Space "shared"
    let newRets = fixRetType Shared rets
    (newArgs, _removedCopy) <- mapAndUnzipM replaceArg args
    pure $
      oneStm $
        Let
          ( Pat
              [ PatElem vName1 (MemMem space),
                PatElem vName2 (MemArray t2 shp2 u2 (ArrayIn mName2 lmad2))
              ]
          )
          aux
```

```
231           (Apply fName newArgs newRets info)
232 fixStmt stm = defaultFixStm stm
233
234 defaultFixStm :: Stm GPUMem -> FixM (Stms GPUMem)
235 defaultFixStm (Let pat aux e) = do
236   e' <- fixExp e
237   pure $ oneStm $ Let pat aux e'
238
239 boolToInt :: Bool -> Int
240 boolToInt True = 1
241 boolToInt False = 0
242
243 -- TODO: this may be too aggressive
244 -- For each argument to a tensor core function call, replace the argument
245 -- if it comes from a manifest statement
246 -- Consider the generated call to tensorMMM that performs the GEMM on Tensor
    ↪  Cores:
247 --   let A' = manifest(A, 0)
248 --   let C = tensorMMM A' ...
249 -- We would insted pass:
250 --   let C = tensorMMM A ...
251 -- We do this because A' is manifested in global memory, but we want the
    ↪  arguments
252 -- to be in shared, because we know this function can only be called fromExp
253 -- kernel code!!!.
254 -- In case the argument was not caused by a manifest statement, it might already
255 -- be in shared memory. The removedCopy indicates if a manifest was removed.
256 replaceArg :: (SubExp, Diet) -> FixM ((SubExp, Diet), Bool)
257 replaceArg (Var v, d) = do
258   manifestMap <- get
259   case lookup v manifestMap of
260     Just (v', removedCopy) ->
261       pure ((Var v', d), removedCopy)
262     Nothing ->
263       pure ((Var v, d), False)
264 replaceArg a = pure (a, False)
265
266 fixExp :: Exp GPUMem -> FixM (Exp GPUMem)
267 fixExp (Match subExps cases body matchDec) =
268   Match subExps
269     <$> mapM fixCase cases
270     <*> fixBody body
271     <*> pure matchDec
272 fixExp (Loop params form body) =
273   localScope (scopeOfFParams (map fst params) <> scopeOfLoopForm form) $ do
274     newBody <- fixBody body
275     pure $ Loop params form newBody
276 fixExp (Op op) = Op <$> fixOp op
277 fixExp e = pure e
278
279 fixCase :: Case (Body GPUMem) -> FixM (Case (Body GPUMem))
280 fixCase (Case pat body) = Case pat <$> fixBody body
281
282 fixBody :: Body GPUMem -> FixM (Body GPUMem)
283 fixBody (Body dec stms res) = Body dec <$> fixStmts stms <*> pure res
284
```

```haskell
fixOp :: Op GPUMem -> FixM (Op GPUMem)
fixOp (Inner hostOp) = Inner <$> fixHostOp hostOp
fixOp op = pure op

fixHostOp :: HostOp NoOp GPUMem -> FixM (HostOp NoOp GPUMem)
fixHostOp (SegOp op) = SegOp <$> fixSegOp op
fixHostOp op = pure op

fixSegOp :: SegOp SegLevel GPUMem -> FixM (SegOp SegLevel GPUMem)
fixSegOp (SegMap level space ts body) =
  SegMap level space ts <$> fixKernelBody body
fixSegOp (SegRed level space ops ts body) =
  SegRed level space ops ts <$> fixKernelBody body
fixSegOp (SegScan level space ops ts body) =
  SegScan level space ops ts <$> fixKernelBody body
fixSegOp (SegHist level space ops hist body) =
  SegHist level space ops hist <$> fixKernelBody body

fixKernelBody :: KernelBody GPUMem -> FixM (KernelBody GPUMem)
fixKernelBody (KernelBody desc stms res) =
  KernelBody desc <$> fixStmts stms <*> pure res
```

A.5 Utils.hs

```haskell
module Futhark.Optimise.TensorCores.Utils
  ( gemmName,
    copyGlobalSharedName,
    copyRegistersSharedName,
    isTCName,
    isPrefixOfName,
    getTCName,
    MMMSignature (..),
    mkInt64Const,
    mapStmsWithScope
  )
where

import Data.List (find, isPrefixOf)
import Futhark.IR

data MMMSignature
  = GemmSignature
      { elmTypeAGemm :: PrimType,
        elmTypeBGemm :: PrimType,
        elmTypeCGemm :: PrimType,
        sizeMGemm :: Int,
        sizeNGemm :: Int,
        sizeKGemm :: Int,
        sizeRegsGemm :: Int
      }
  | CopyGlobalSharedSignature
      { elmTypeCPGS :: PrimType,
        sizeYCPGS :: Int,
        sizeXCPGS :: Int
      }
  | CopyRegistersSharedSignature
```

```
33        { elmTypeCPRS :: PrimType,
34          sizeMCPRS :: Int,
35          sizeNCPRS :: Int,
36          sizeRegsCPRS :: Int,
37          blockSizeCPRS :: Int
38        }
39    deriving (Show, Eq, Ord)
40
41  gemmName :: Name
42  gemmName = "tensorMMM"
43
44  copyGlobalSharedName :: Name
45  copyGlobalSharedName = "copyGlobalShared"
46
47  copyRegistersSharedName :: Name
48  copyRegistersSharedName = "copyRegistersShared"
49
50  isPrefixOfName :: Name -> Name -> Bool
51  isPrefixOfName prefix name = nameToString prefix `isPrefixOf` nameToString name
52
53  funNames :: [Name]
54  funNames = [gemmName, copyGlobalSharedName, copyRegistersSharedName]
55
56  isTCName :: Name -> Bool
57  isTCName name = any (`isPrefixOfName` name) funNames
58
59  getTCName :: Name -> Maybe Name
60  getTCName name = find (`isPrefixOfName` name) funNames
61
62
63  -- Helper functions
64
65  -- | Creates an i64 SubExp
66  mkInt64Const :: Int -> SubExp
67  mkInt64Const = Constant . IntValue . intValue Int64
68
69  -- | Map a function over stmts and update the scope for each stmt.
70  mapStmsWithScope ::
71    (Monoid a, LocalScope rep f) =>
72    (Stm rep -> f a) ->
73    Stms rep ->
74    f a
75  mapStmsWithScope f stms =
76    case stmsHead stms of
77      Nothing -> pure mempty
78      Just (stm, stms') -> do
79        stm' <- f stm
80        stms'' <- inScopeOf stm $ mapStmsWithScope f stms'
81        pure $ stm' <> stms''
```