

How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog

In this post, I'll iteratively optimize an implementation of matrix multiplication written in CUDA. My goal is not to build a cuBLAS replacement, but to deeply understand the most important performance characteristics of the GPUs that are used for modern deep learning. This includes coalescing global memory accesses, shared memory caching and occupancy optimizations, among others. You can download the code for all kernels from [Github](#). Also checkout [wangzyon's repo](#) from which I copied the benchmarking setup. This post is less polished than my normal uploads, and includes many more sidenotes. I used it as notepad for ideas and scribbles while writing the kernels. That's why I called it a worklog :)

Matrix multiplication on GPUs may currently be the most important algorithm that exists, considering it makes up almost all the FLOPs during the training and inference of large deep-learning models. So how much work is it to write a performant CUDA SGEMM? GEMM performs $C = \alpha AB + \beta C$ at single (=32b) precision. from scratch? I'll start with a naive kernel and step-by-step apply optimizations until we get within 95% (on a good day) of the performance of cuBLAS (NVIDIA's official matrix library):cuBLAS at FP32 that is. In my setting, doing the matmul using TF32 or BF16 precision allows cuBLAS to use the tensor cores, which increases FLOPS by 2.5x or 3.5x. I may look into tensor cores / warp matrix functions in a future post.

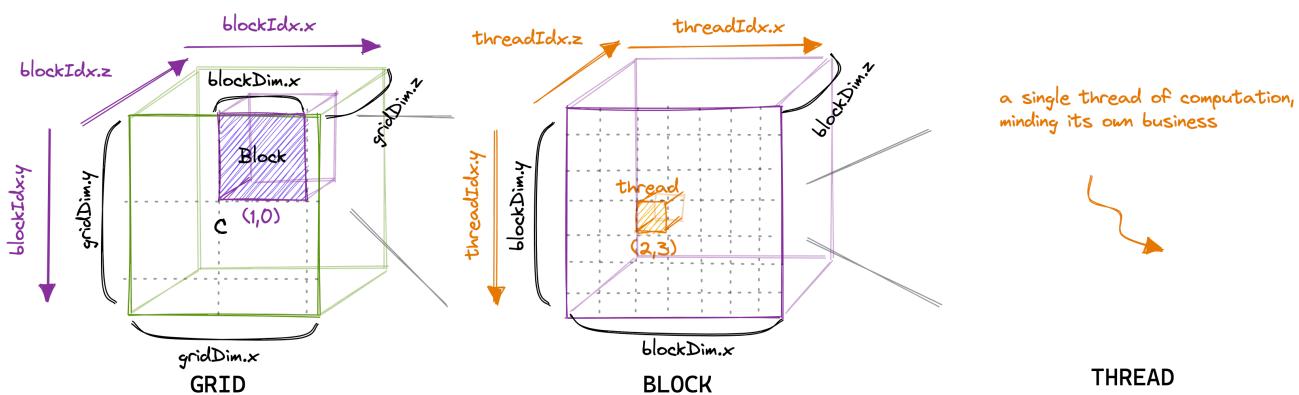
Kernel	GFLOPs/s	Performance relative to cuBLAS
1: Naive	309.0	1.3%
2: GMEM Coalescing	1986.5	8.5%
3: SMEM Caching	2980.3	12.8%

Kernel	Gflops/s	Performance relative to cuBLAS
4: 1D Blocktiling	8474.7	36.5%
5: 2D Blocktiling	15971.7	68.7%
6: Vectorized Mem Access	18237.3	78.4%
9: Autotuning	19721.0	84.8%
10: Warptiling	21779.3	93.7%
0: cuBLAS	23249.6	100.0%

Kernel 1: Naive Implementation

In the CUDA programming model, computation is ordered in a three-level hierarchy. Each invocation of a CUDA kernel creates a new grid, which consists of multiple blocks. Each block consists of up to 1024 individual threads. These constants can be looked-up in the [CUDA Programming guide](#). Threads that are in the same block have access to the same shared memory region (SMEM).

The number of threads in a block can be configured using a variable normally called `blockDim`, which is a vector consisting of three ints. The entries of that vector specify the sizes of `blockDim.x`, `blockDim.y` and `blockDim.z`, as visualized below:



Similarly, the number of blocks in a grid is configurable using the `gridDim` variable. When we launch a new kernel from the

hostIn accelerator lingo, *host* refers to the CPU and *device* is the accelerator, here the GPU., it creates a single grid, containing the blocks and threads as specified. From here on I'll only be talking about 2D grids and blocks, partly because the 3D-structure is seldom used and because drawing in 3D is too hard. It's important to keep in mind that the thread hierarchy we just talked about mostly concerns program correctness. For program performance, as we'll see later, it's not a good idea to treat all threads in the same block as equals.

For our first kernel, we'll use the grid, block and thread hierarchy to assign each thread a unique entry in the result matrix C. Then that thread will compute the dot product of the corresponding row of A and column of B, and write the result to C. Due to each location of C being written to by only one thread, we have to do no synchronization. We'll launch the kernel like so:

```
// create as many blocks as necessary to map all of C
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the device
// The function call returns immediately on the host
sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

CUDA code is written from a single-thread perspective. In the code of the kernel, we access the `blockIdx` and `threadIdx` built-in variables. These will contain return different values based on the thread that's accessing them. In our example, `threadIdx.x` and `threadIdx.y` will vary from 0 to 31 based on the position of the thread in the grid. Same for `blockIdx.x` and `blockIdx.y`, which will vary from 0 to `CEIL_DIV(N, 32)` or `CEIL_DIV(M, 32)` based on the position of the thread's block in the grid. We'll do a lot of indexing into strided in-memory representations of matrices. Edward Yang's post on [PyTorch Internals](#) contains a good explanation of strided tensors.

```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
                           const float *B, float beta, float *C) {
    // compute position in C that this thread is responsible for
    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
    const uint y = blockIdx.y * blockDim.y + threadIdx.y;

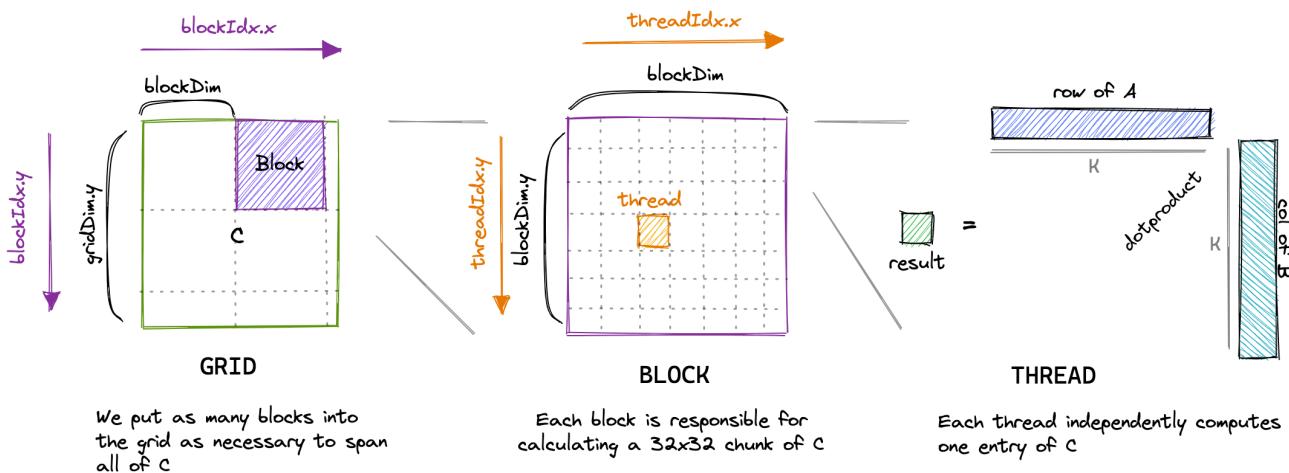
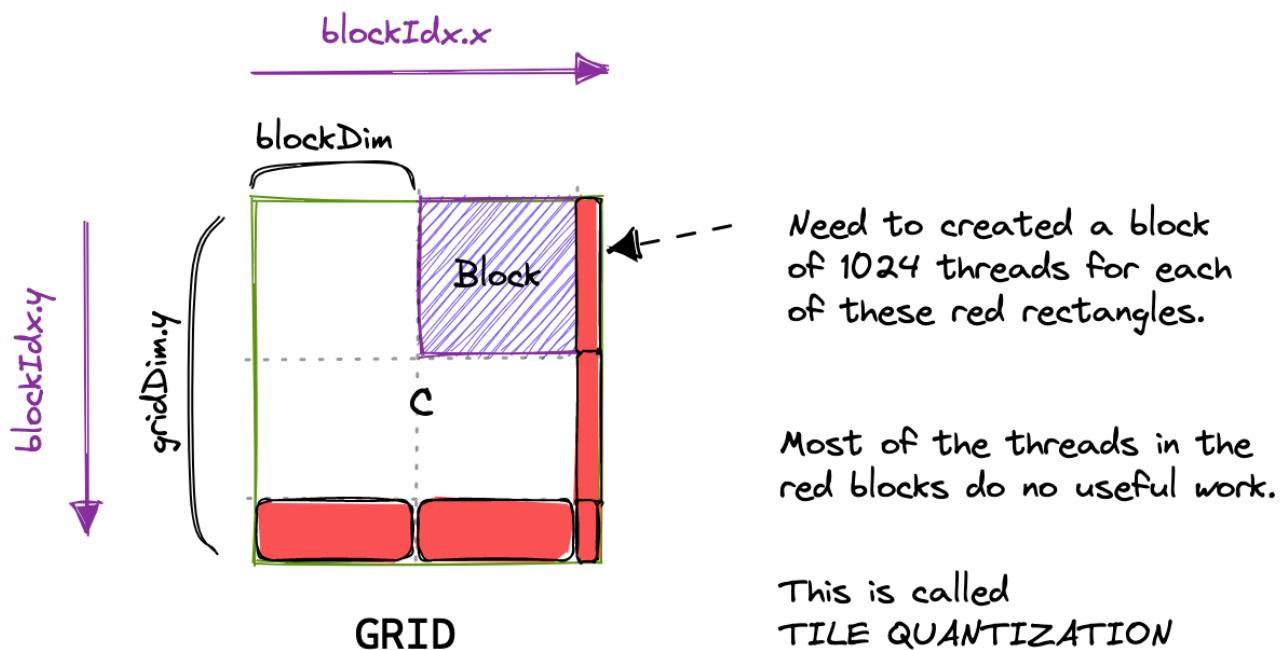
    // `if` condition is necessary for when M or N aren't multiples of 32.
    if (x < M && y < N) {
        float tmp = 0.0;
```

```

        for (int i = 0; i < K; ++i) {
            tmp += A[x * K + i] * B[i * N + y];
        }
        // C = alpha*(A*B) + beta*C
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}

```

To visualize this simple kernel: If the size of the matrix is not divisible by the size of the block, we'll have to launch extra blocks to process the remainder. For example, in the picture below, we'll create 9 blocks of equal threadsize, but only 4 of those fully utilize their 1024 threads. This artifact is called **tile quantization**, and appears whenever we try to map a fixed-sized volume across a variable-sized input.



This kernel takes about 0.5s to process three 4092^2 fp32 matrices on my A6000 GPU. Let's do some non-implementation-specific calculations:

Lower Bounding the Fastest Possible Runtime

For a matrix multiplication of two 4092^2 matrices, followed by an addition of a 4092^2 matrix (to make the [GEMM](#)):

1. Total FLOPS: For each of the 4092^2 entries of C, we have to perform a dot product of two vectors of size 4092, involving a multiply and an add at each step. "Multiply then add" is often mapped to a single assembly instruction called FMA (fused multiply-add), but still counts as two FLOPs. $2 * 4092^3 + 4092^2 = 137$ GFLOPS
2. Total data to read (minimum!): $3 * 4092^2 * 4B = 201MB$
3. Total data to store: $4092^2 * 4B = 67MB$

So 268MB is the absolute minimum of memory that any implementation would have to transfer from/to global GPU memory. Global memory is the GPU's main memory region. If Nvidia sells you a GPU advertised with 80GB of memory and 1TB/s of bandwidth, they're talking about the capacity and bandwidth of global memory. Later we'll talk about other memory regions on the GPU, like the shared memory, which is physically distinct and has very different performance characteristics. assuming it has a big enough cache. The cuBLAS kernel loads a total of 500MB of GMEM during the whole calculation. We'll see later how increasing arithmetic intensity allows us to achieve an access volume that low. Let's calculate some upper bounds on kernel performance. The GPU is advertised with 30TFLOPs/s of fp32 compute throughput and 768GB/s of global memory bandwidth. If we achieved those numbers, Reminder that peak FLOPs is a reductionist metric, since it depends on the instruction mix. There's no way you'd reach those 30TFLOPs/s if your FLOP of choice is DIV. However, since matmul uses mainly FMA instructions, which tends to be the fastest FLOPs, we have a good chance of actually getting close to that peak FLOP value. Similar story for the bandwidth: Peak bandwidth can only be reach if the access pattern suits the hardware. we'd need 4.5ms for the calculation and 0.34ms for the memory transfers. So in our napkin math, the calculation takes ~10x more time than the memory accesses. This means our final optimized kernel will be

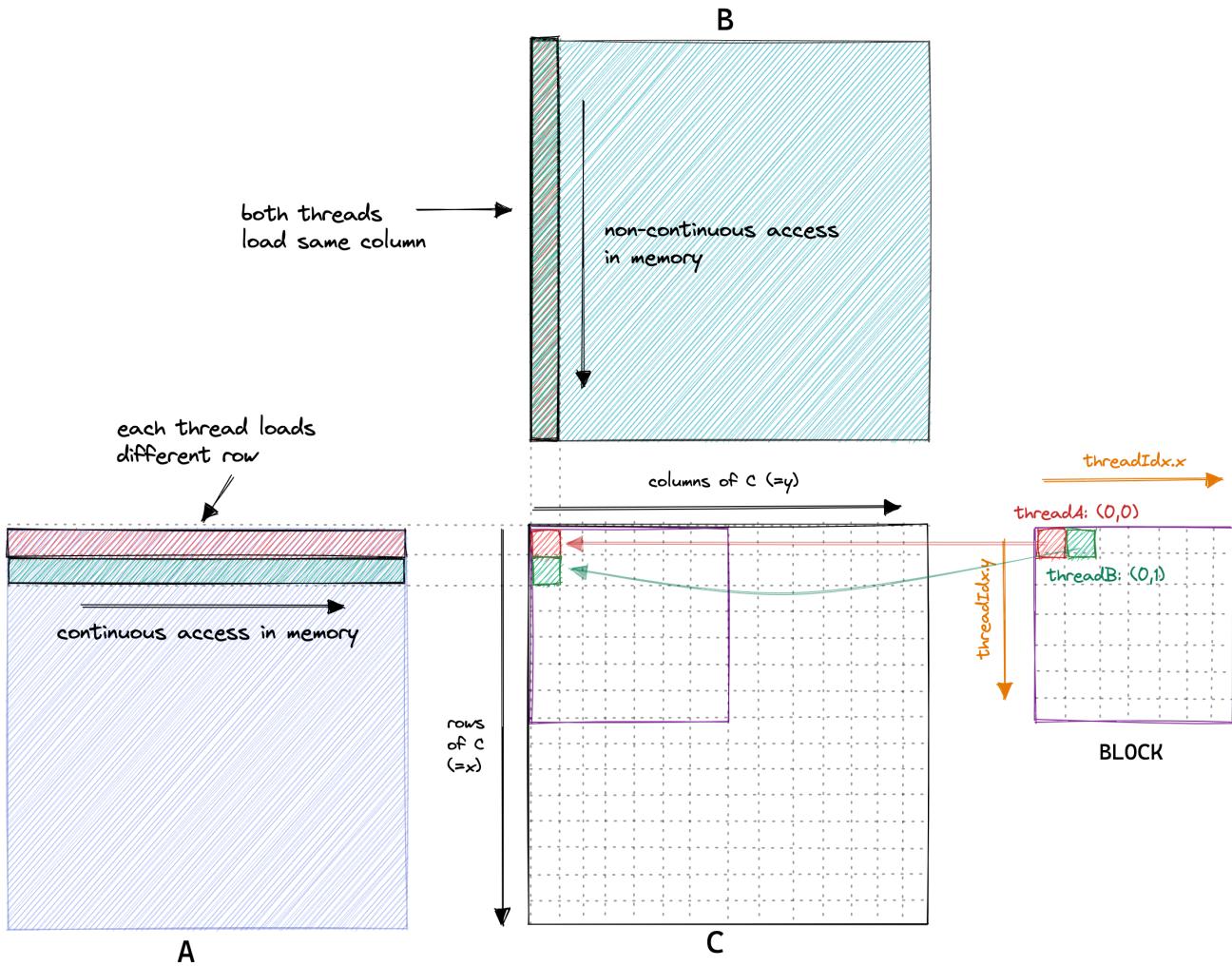
compute-bound, as long as we end up having to transfer <10x the absolute minimum memory volume of 278MB. The A6000 is advertised with 309TFLOPs/s of tensor core performance. If we could use tensor cores for our fp32 matmul, the calculation would only take 0.44ms, and an optimized kernel doing 4092^2 matrix mul would almost surely still be memory bound. This puts into perspective just how fast the tensor cores are.

Now that we've calculated some lower bounds for our fp32 GEMM calculation, let's get back to the kernel on hand, to figure out why it's so much slower than it could be.

Memory Access Pattern of the Naive Kernel

In our kernel, two threads in the same block with ThreadIds (0, 0) and (0, 1) will load the same column of B but different rows of A. If we assume the worst case of zero caching, then each thread has to load $2*4092+1$ floats from global memory. As we have 4092^2 threads total, this would result in 548GB of memory traffic.

Below is a visualization of the memory access pattern of our naive kernel, taking two threads A (red) and B (green) as an example:



A, B, C are stored in row-major order.
This means that the last index (here y)
is the one that iterates continuous through
memory (=has stride 1).

So to recap, when I run this kernel on an A6000 GPU it achieves ~300GFLOPs when multiplying two 4092x4092 float32 matrices. Pretty bad, considering that the A6000 is advertised as being able to achieve almost 30 TFLOPs. Just for comparison, 300 GFLOPs is also roughly the performance achieved by the optimized BLAS library on the 2015 Haswell CPU that I used in my [earlier post](#) on CPU matmul. So how can we start to make this faster? One way is to optimize the memory access pattern of our kernel such that global memory accesses can be coalesced (=combined) into fewer accesses.

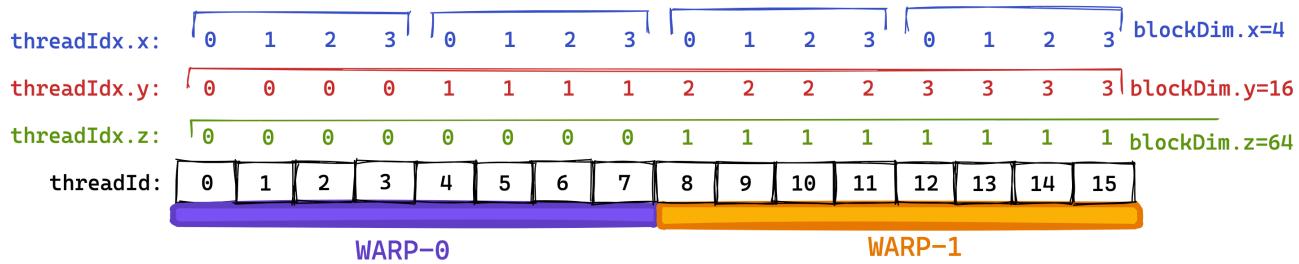
Kernel 2: Global Memory Coalescing

Before we get into global memory coalescing, we need to learn about the concept of a warp. For execution, the threads of a

block are grouped into so-called warps, consisting of 32 threads. A warp is then assigned to a warp scheduler, which is the physical core that executes the instructions. Before the Volta architecture, it used to be the case that all threads of a warp were fed from the same instruction stream. On a branch, the threads that didn't take the branch were inactivated using the so-called active mask. However, since Volta, it's no longer a good idea to rely on this 'warp-synchronous' behaviour, as instructions from different branches may be interleaved even for the same threads within a warp. There are four warp schedulers per multiprocessor. The grouping into warps happens based on a consecutive threadIdx. If we set the blockDim to be multi-dimension, then the threadIdx is calculated like so:

```
threadIdx = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z)
```

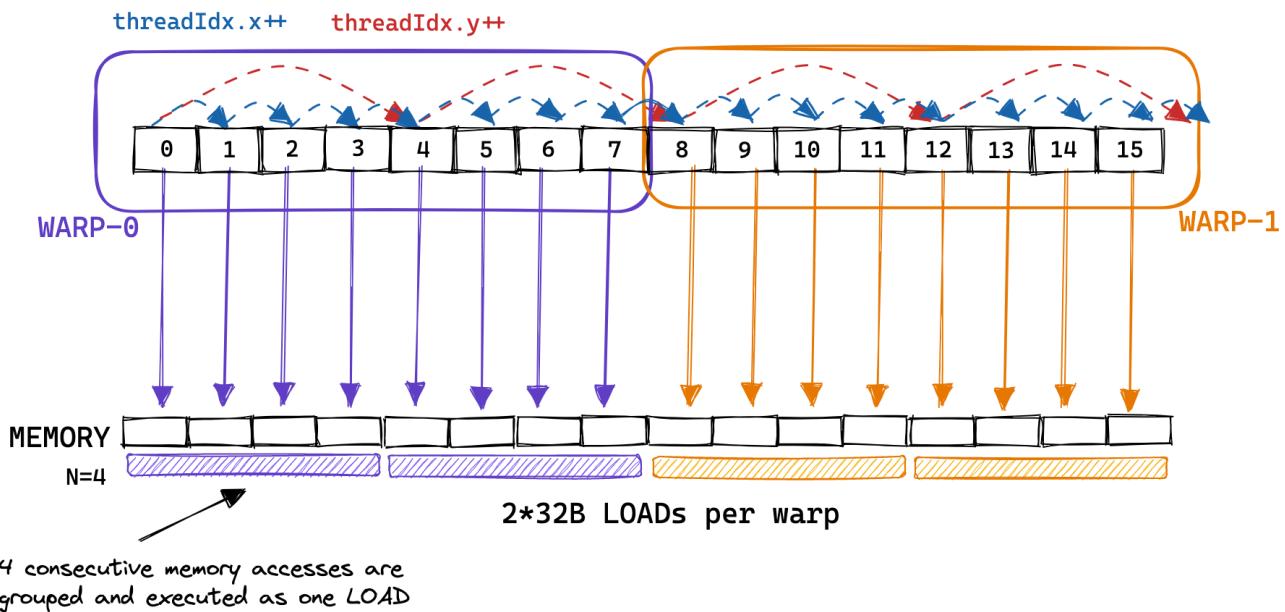
Then, threads with neighbouring threadIdx become part of the same warp. Below I tried to illustrate this, using a smaller "warpsize" of 8 threads (real warps always contain 32 threads): I like to think of the three dimensions x,y,z of threadIdx as being "column-major", due to the first dimension x being the one that's continuous in "warp space". I don't know if others use that term, but it makes the concept more clear to me.



```
threadId = threadIdx.x + blockDim.x*threadIdx.y + blockDim.x*blockDim.y*threadIdx.z
```

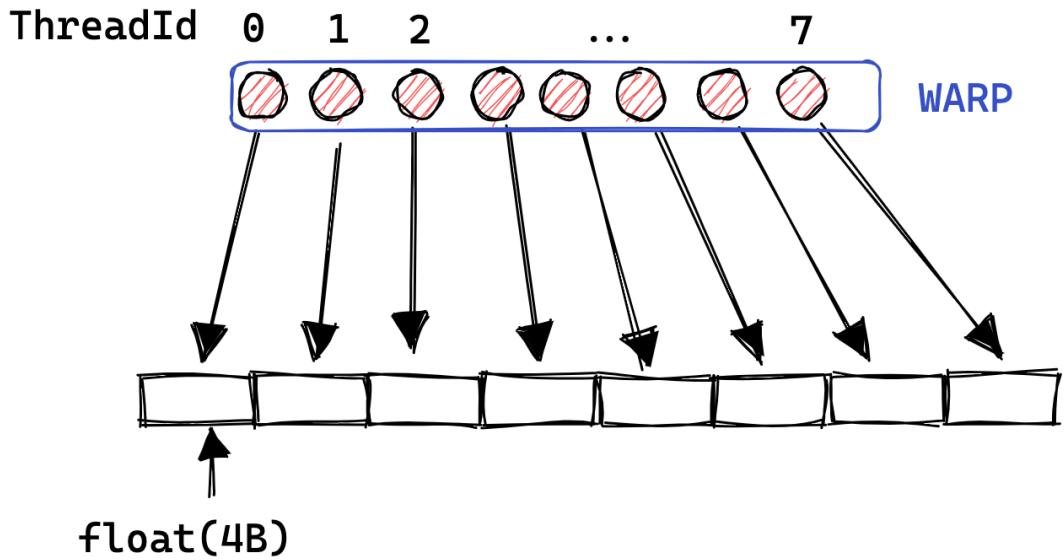
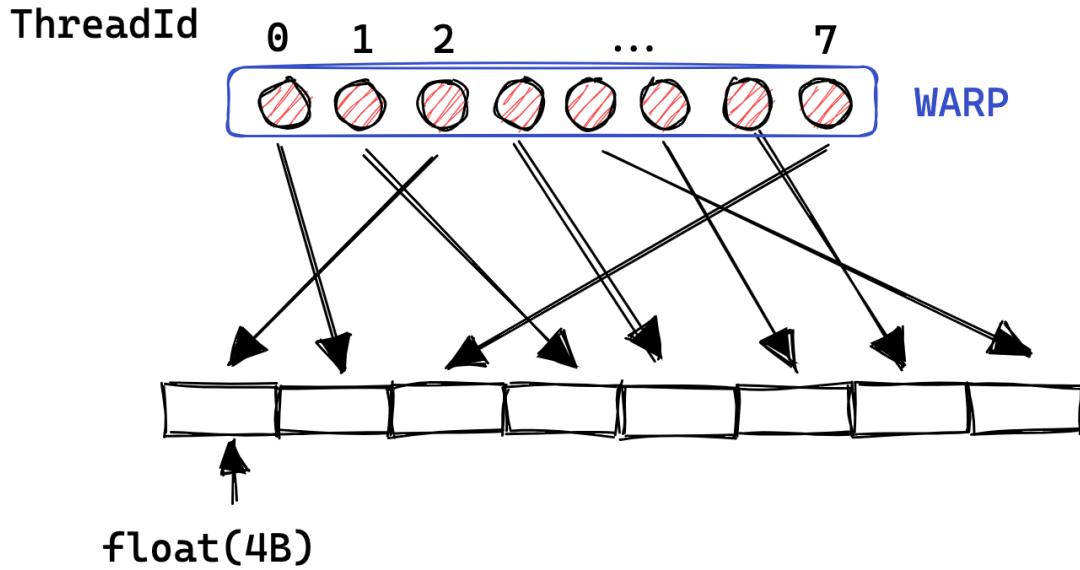
The concept of a warp is relevant for this second kernel, as sequential memory accesses by threads that are part of the same warp can be grouped and executed as one. This is referred to as **global memory coalescing**. It's the most important thing to keep in mind when optimizing a kernel's GMEM memory accesses toward achieving the peak bandwidth.

Below is an example, where consecutive memory accesses by threads in the same warp are grouped, allowing each warp to execute 8 memory accesses using only 2 32B loads:



In reality, the GPU supports 32B, 64B and 128B memory accesses. So, if each thread is loading a 32bit float from global memory, the warp scheduler (probably the MIO) can coalesce this $32 \times 4 = 128$ B load into a single transaction. This is only possible if the floats loaded are consecutive in memory, and if access is aligned. In that way, optimizing for global memory coalescing on GPU has a lot of similarities to optimizing for cache line utilization on CPU. Interestingly, to allow coalescing the threads within a warp have to access consecutive addresses, but the accesses don't have to be consecutive within-warp.

Illustrated below:



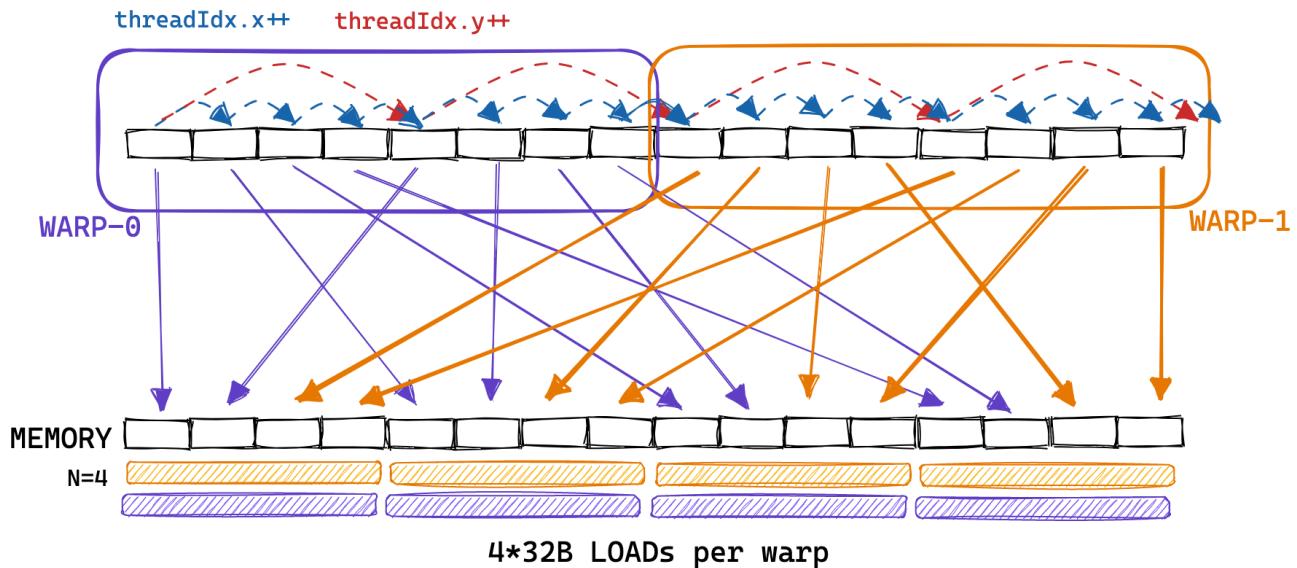
Both of these access patterns can be coalesced!

If they aren't, or if access cannot be coalesced for some other reason, then the GPU will execute as many 32B loads as necessary to fetch all floats, leading to a lot of wasted bandwidth. Profiling our naive kernel, we can observe the detrimental effect of non-coalesced access as we achieve only 15GB/s of GMEM throughput.

Looking back at the previous kernel, we assigned threads their entry of C like so:

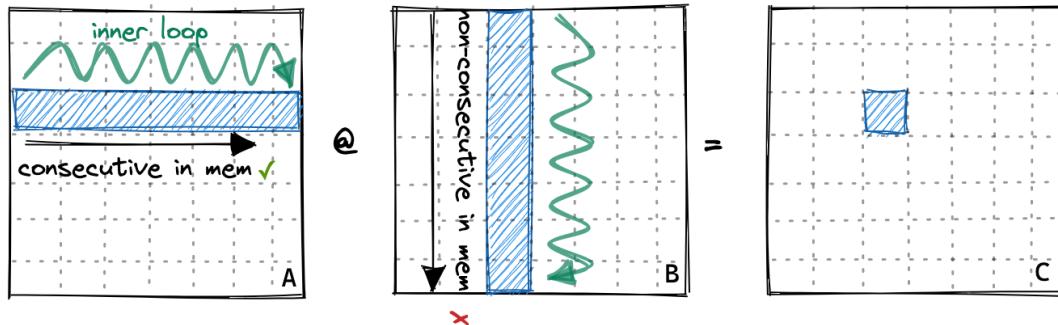
```
const uint x = blockIdx.x * blockDim.x + threadIdx.x;
const uint y = blockIdx.y * blockDim.y + threadIdx.y;
```

Hence, threads of the same warp (those with consecutive `threadIdx.x`) were loading the rows of A non-consecutively from memory. The naive kernel's pattern of accessing the memory of A looked more like so:

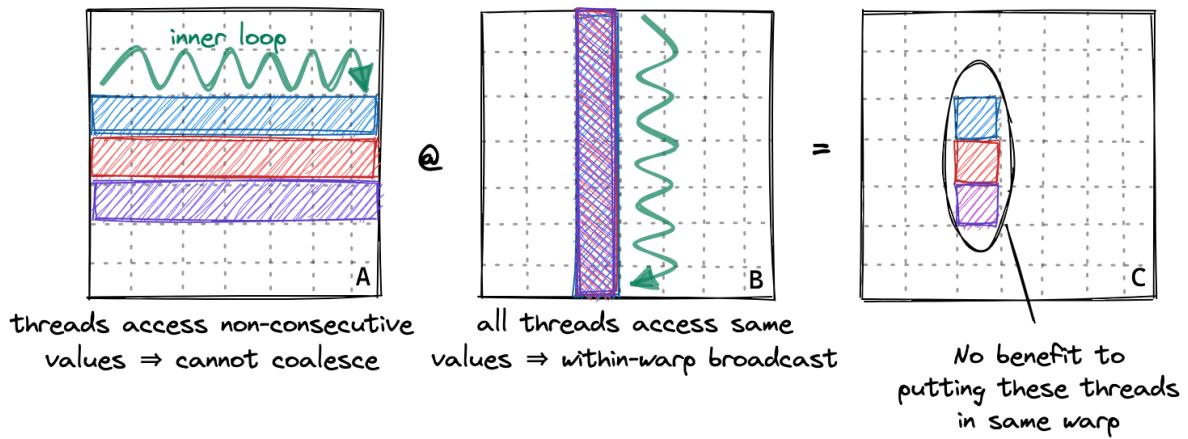


To enable coalescing, we can change how we assign positions of the result matrix C to threads. This change in the global memory access pattern is illustrated below:

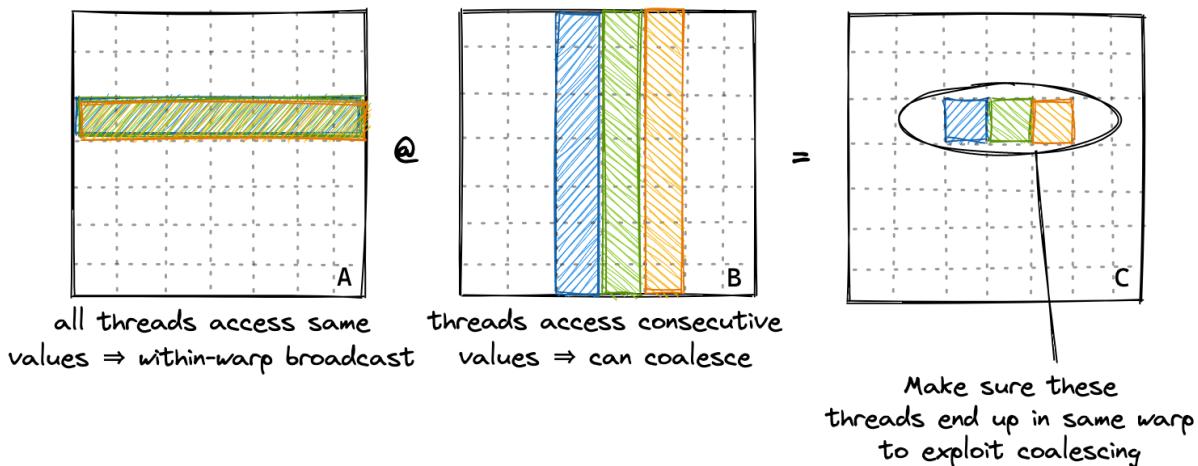
Matrix memory layout:



Naive kernel:



Coalescing kernel:



To implement this, we only need to change the first two lines:

```
const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
```

```
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);
```

```
if (x < M && y < N) {  
    float tmp = 0.0;
```

```

for (int i = 0; i < K; ++i) {
    tmp += A[x * K + i] * B[i * N + y];
}
C[x * N + y] = alpha * tmp + beta * C[x * N + y];
}

```

And we call it like so: This wasn't immediately obvious to me, but enabling GMEM coalescing changes nothing in the assembly, see the SASS output on [Godbolt](#). Access coalescing is done at kernel runtime by the hardware. This makes sense since coalescing requires aligned access, which cannot be guaranteed at compile time as we pass the matrix pointers as function arguments. Also: the assembly features partial unrolling of our inner loop even though the loop count K is not known at compile time. Exciting!

```

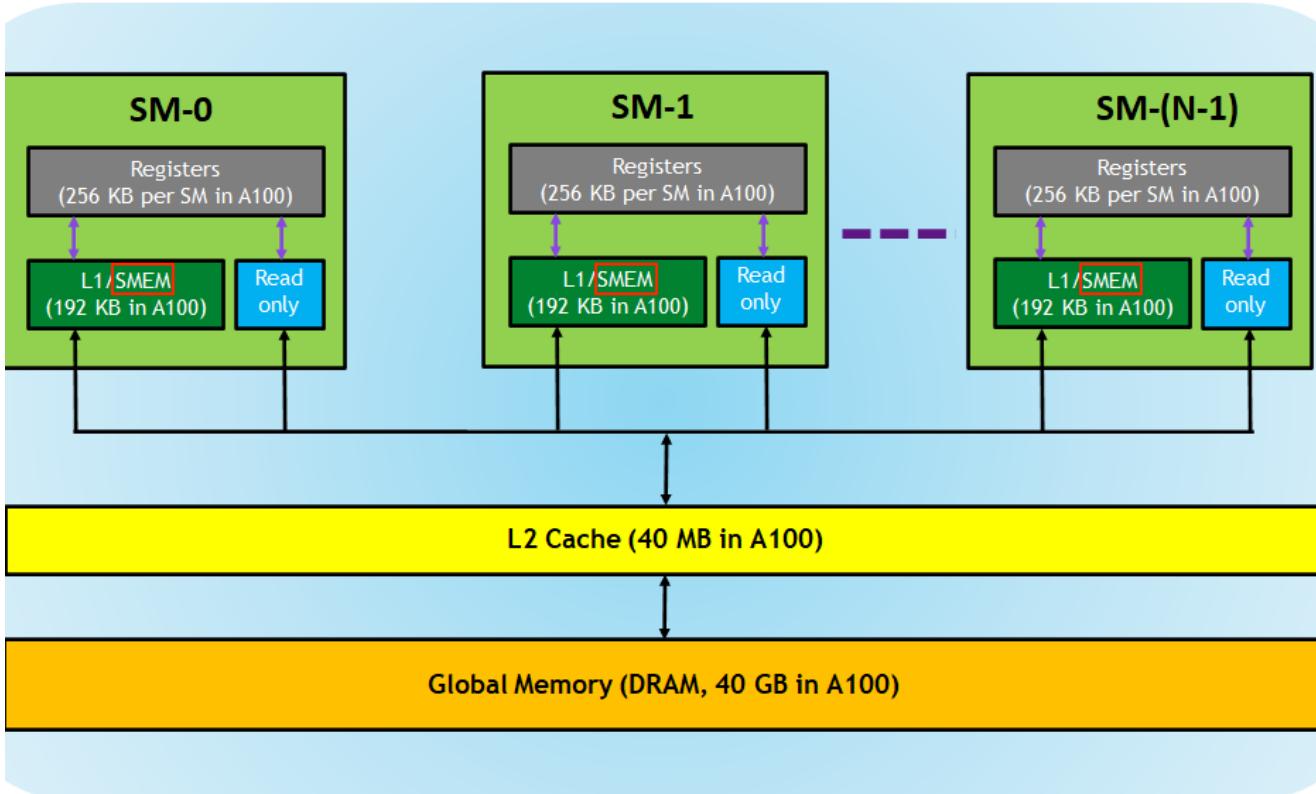
// gridDim stays the same
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
// make blockDim 1-dimensional, but don't change number of threads
dim3 blockDim(32 * 32);
sgemm_coalescing<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);

```

Global memory coalescing increases memory throughput from 15GB/s to 110GB/s. Performance reaches 2000 GFLOPS, a big improvement compared to the 300 GFLOPS of the first, naive kernel. For the next kernel, we'll use the GPU's fast on-chip memory, called shared memory, to cache data that will be re-used.

Kernel 3: Shared Memory Cache-Blocking

Next to the large global memory, a GPU has a much smaller region of memory that is physically located on the chip, called shared memory (SMEM). Physically, there's one shared memory per SM. Here's a helpful illustration of the memory hierarchy on an A100 GPU ([source](#)):

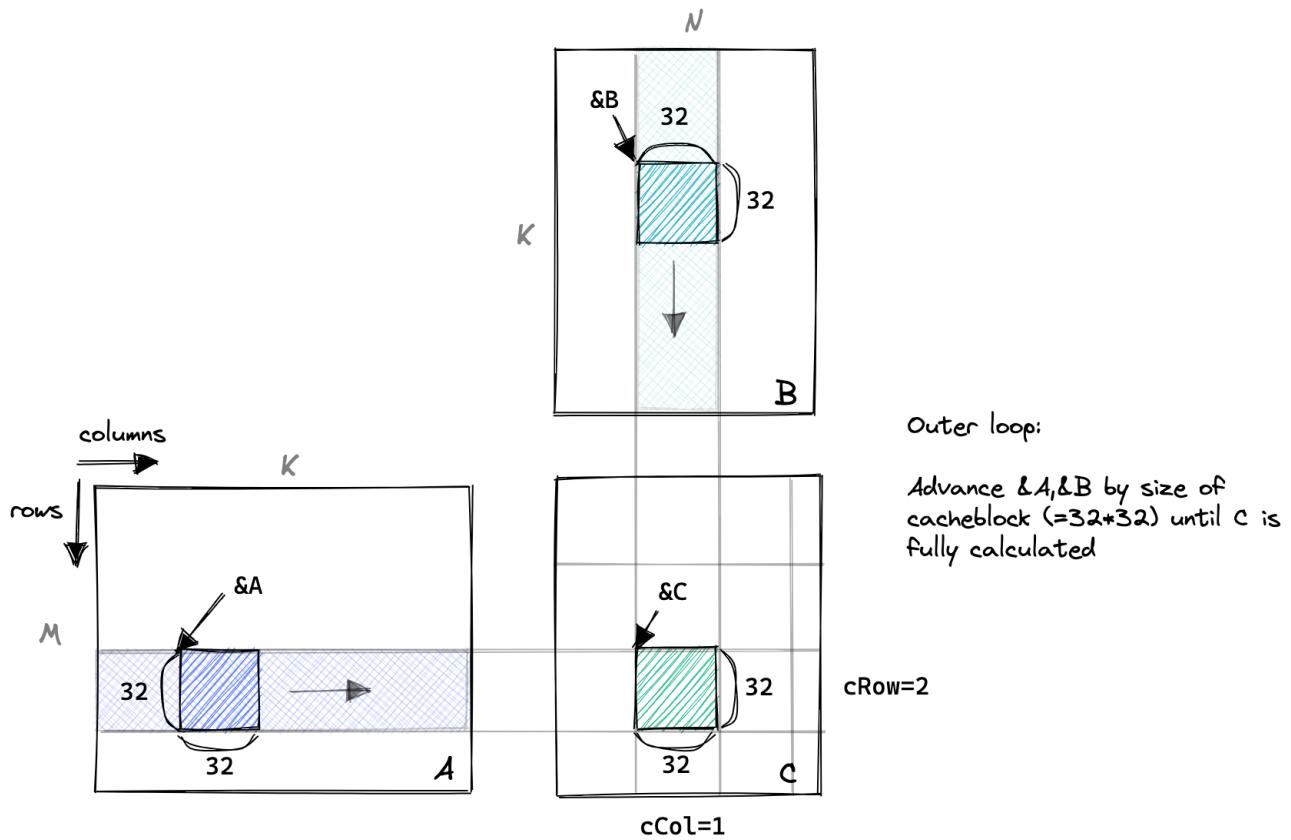


Logically, this shared memory is partitioned among the blocks. This means that a thread can communicate with the other threads in its block via the shared memory chunk. On my A6000 GPU, each block has access to a maximum of 48KB of shared memory. The amount of SMEM is configurable, by trading off a larger shared memory for a smaller L1 cache. For specifics, see the [compute capability documentation](#). Also, it's possible to use more than 48KB of SMEM per thread by utilizing dynamic shared memory.

As the shared memory is located on-chip, it has a much lower latency and higher bandwidth than global memory. I couldn't find good benchmark results for the Ampere architecture but for Volta (released in 2017) the benchmarks performed in [this paper](#) report 750GiB/s of global memory bandwidth, and 12,080GiB/s of shared memory bandwidth. It doesn't look like these numbers have changed much since Volta. Nvidia reports ~750GB of max GMEM bandwidth for my A6000 (Ampere).

So for this next kernel, we'll load a chunk of A and a chunk of B from global memory into shared memory. Then we'll perform as much work as possible on the two chunks, with each thread still being assigned one entry of C. We'll move the chunks along the columns of A and the rows of B performing partial sums on C until the result is computed.

This is illustrated below:



The important parts of the code are below, with variable names corresponding to the plot above: In general, I didn't write the code to work for arbitrary sizes of M , N and K , as the condition checking introduces a lot of clutter and isn't very interesting. To make sure the kernel works correctly, I test it with random data and a few different matrix sizes by comparing to cuBLAS.

```
// advance pointers to the starting positions
A += cRow * BLOCKSIZE * K; // row=cRow, col=0
B += cCol * BLOCKSIZE; // row=0, col=cCol
C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol

float tmp = 0.0;
// the outer loop advances A along the columns and B along
// the rows until we have fully calculated the result in C.
for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
    // Have each thread load one of the elements in A & B from
    // global memory into shared memory.
    // Make the threadCol (=threadIdx.x) the consecutive index
    // to allow global memory access coalescing
    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];

    // block threads in this block until cache is fully populated
```

```

__syncthreads();

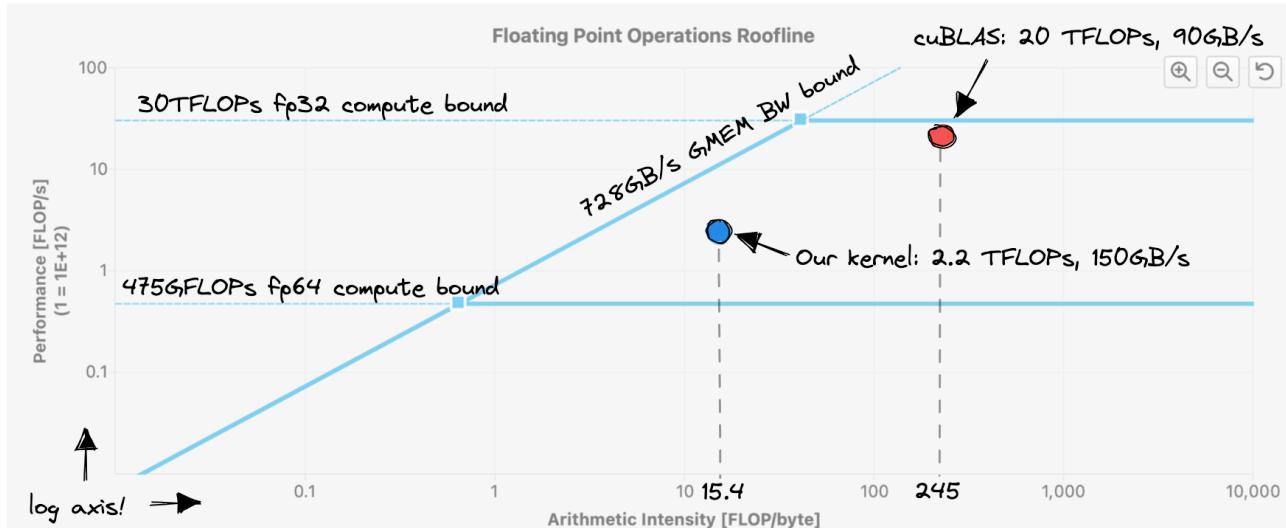
// advance pointers onto next chunk
A += BLOCKSIZE;
B += BLOCKSIZE * N;

// execute the dotproduct on the currently cached block
for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
    tmp += As[threadRow * BLOCKSIZE + dotIdx] *
           Bs[dotIdx * BLOCKSIZE + threadCol];
}
// need to sync again at the end, to avoid faster threads
// fetching the next block into the cache before slower threads are done
__syncthreads();
}

C[threadRow * N + threadCol] =
    alpha * tmp + beta * C[threadRow * N + threadCol];

```

This kernel achieves ~2200 GFLOPS, a 50% improvement over the previous version. There's only a 50% improvement partly because our previous kernel already had pretty good L1 cache hit rates. We're still far away from hitting the ~30 TFLOPs that the GPU can provide. This is obvious from the roofline plot below: Notice how we're achieving a higher memory bandwidth than cuBLAS. But because we're doing much less work per byte loaded from memory (=lower arithmetic intensity), overall performance is worse.



At a CHUNSIZE of 32, this uses $2 \times 32 \times 32 \times 4B = 8KB$ of shared memory space. This info can also be obtained by compiling with --ptxas-options=-v, which outputs: Used 37 registers, 8192 bytes smem, 400 bytes cmem[0]. My A6000 GPU has a maximum of 48KB of shared memory space available for each block, so we're far away from hitting

that limit. This is not necessarily a problem, as there are downsides to increasing per-block shared-memory usage. Each multiprocessor (SM) has a maximum of 100KB of SMEM available. This means that if we'd modify our kernel to use the full 48KB of SMEM available, each SM could only keep two blocks loaded at the same time. In CUDA parlance, increasing per-block SMEM utilization can decrease **occupancy**. Occupancy is defined as the ratio between the number of active warps per SM and the maximum possible number of active warps per SM.

High occupancy is useful because it allows us to hide the high latency of our operations, by having a bigger pool of issueable instructions available. On GPUs, math operations like FMA have a latency of 4 cycles which is equal to 2.6ns at a 1.5GHz clock. Compare this to a recent x86 CPU, where FMA has a 6 cycle latency or 1.8ns at a 3.5GHz clock. There are three main limits to keeping more active blocks loaded on an SM: register count, warp count and SMEM capacity. Let's do an example calculation for our current kernel.

Occupancy Calculation for Kernel 3

Here are the relevant hardware stats for my GPU, obtained from the `cudaGetDeviceProperties` API (Multiprocessors are the SMs we talked about earlier): The amount of shared memory is configurable by using a feature called `SharedMemoryCarveout`. The so-called unified data cache is partitioned into L1 cache and shared memory, so we can trade-off less shared-memory for more L1 cache.

Metric	Value
Name	NVIDIA RTX A6000
Compute Capability	8.6
max threads per block	1024
max threads per multiprocessor	1536
threads per warp	32
warp allocation granularity	4

Metric	Value
max regs per block	65536
max regs per multiprocessor	65536
reg allocation unit size	256
reg allocation granularity	warp
total global mem	48685 MB
max shared mem per block	48 KB
CUDA runtime shared mem overhead per block	1024 B
shared mem per multiprocessor	102400 B
multiprocessor count	84
max warps per multiprocessor	48

And here are the resource demands for our kernel:

Registers per Thread	37
SMEM per Block	8192 B
Threads per Block	1024

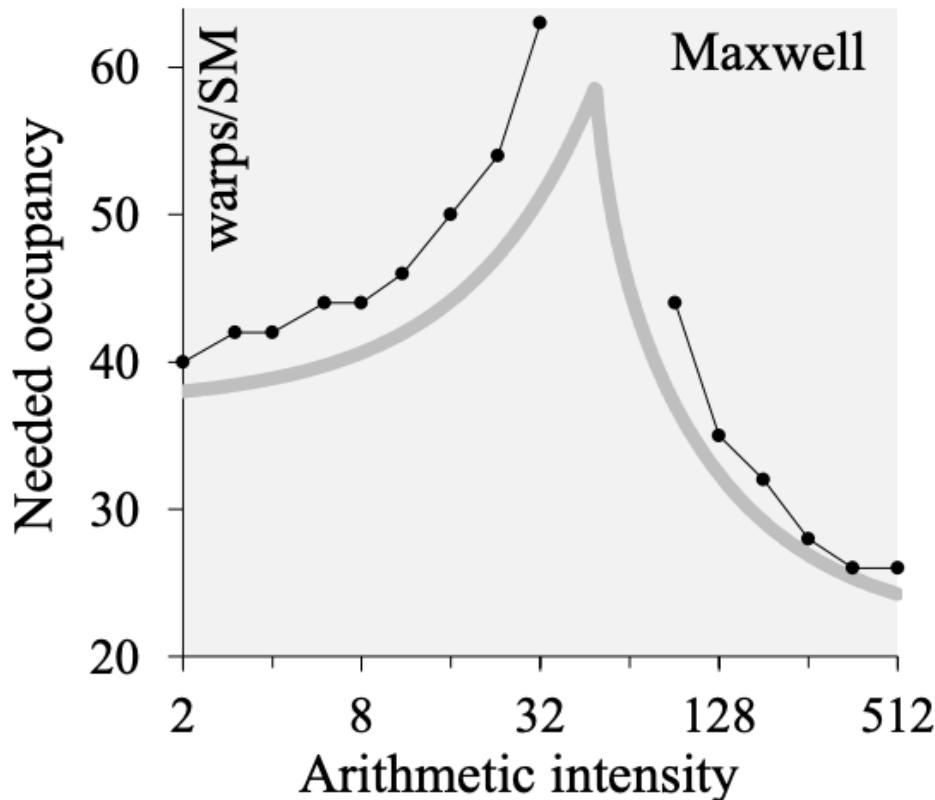
Work is scheduled onto the SMs on a block granularity. Each SM will load more blocks, as long as it has enough resources to accommodate them. Calculation: I found lots of official and unofficial occupancy calculators, but no official formulae as how to calculate the occupancy. The results are correct (I checked using NVIDIA's official tools), but there may be small errors eg in the application of rounding.

- **Shared memory:** 8192B/Block + 1024B/Block for CUDA runtime usage = 9216B/Block. $(102400\text{B per SM}) / (9216\text{B per Block}) = 11.11 \Rightarrow 11$ Blocks upper limit.
- **Threads:** 1024 Threads per Block, max 1536 threads per SM \Rightarrow Upper limit 1 block.
- **Registers:** 37 regs per thread * 32 threads per warp = 1184 regs per warp. Register allocation granularity is

256 regs on a warp level, hence rounding up to 1280 regs per warp. We have $(1024 \text{ threads} / 32) = 32$ warps per block, hence $1280 \text{ regs per warp} * 32 \text{ warps per block} = 40960 \text{ regs per block}$. Max 65536 regs per SM \Rightarrow upper limit 1 block. Surprisingly, there's more space in the register file than in shared memory! Each block can use max 48 KB of SMEM, but $65536 * 4B = 262 \text{ KB}$ of register space.

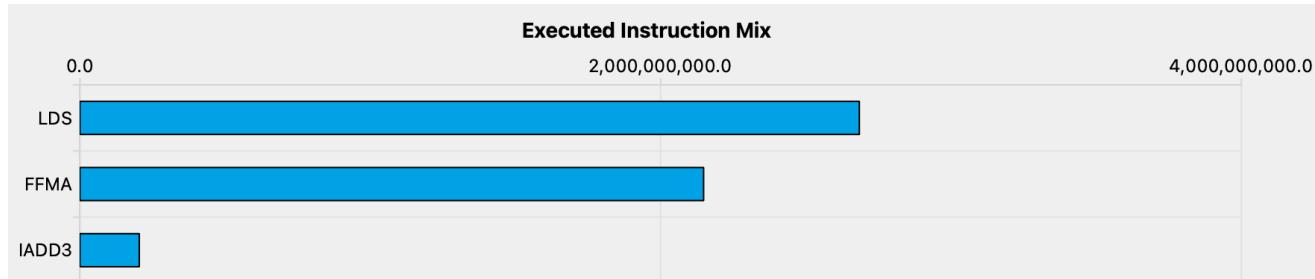
So this kernel is limited by the number of threads per block, and the number of registers per thread. We cannot load more than one block per SM, giving us a final occupancy of $32 \text{ active warps} / 48 \text{ max active warps} = 66\%$.

A 66% occupancy is not too bad, so this doesn't explain why our kernel runs so slow. We know that it's possible to optimize our kernel towards high arithmetic intensity (AI) by observing that cuBLAS achieves $\sim 245 \text{ FLOPs/Byte}$. Both at very high and very low AI, high occupancy is not needed to achieve peak throughput. For more details on this, see [V. Volkov's PhD thesis](#) and its coverage of "cusp behaviour":



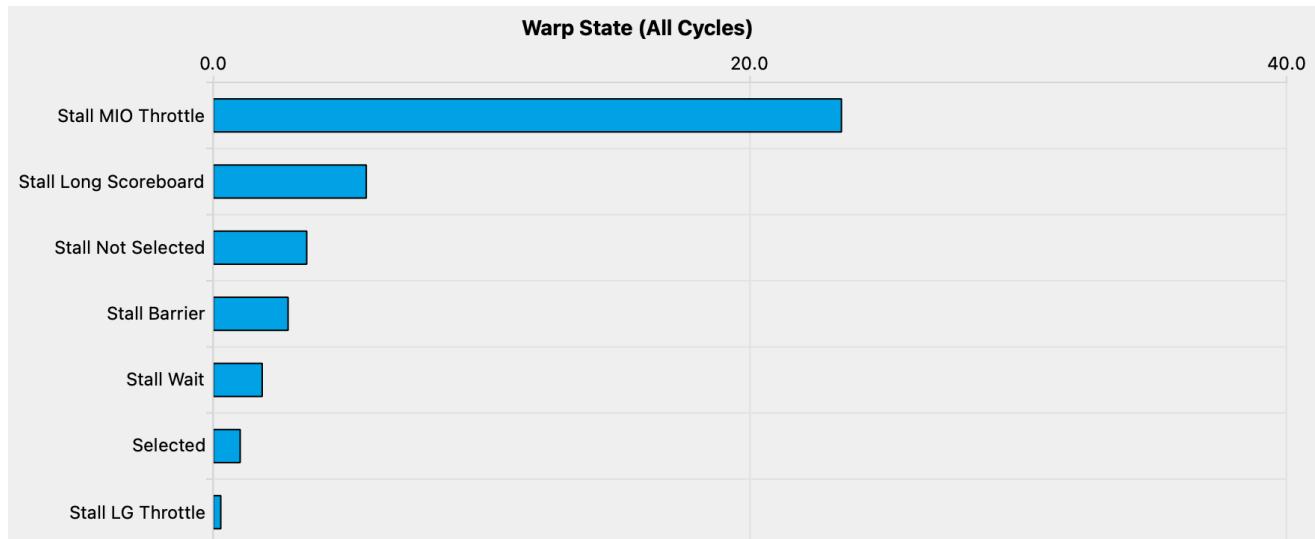
Looking at the profiler gives us some hints. First, if we look at the mix of executed instructions, most of them are memory loads:LDS are shared memory loads. FMA is our fused multiply

add. IADD3 is a “3 input integer addition”, which we need for moving the pointers along the K dimension.



```
ld.shared.f32    %f91, [%r8+3456];  
ld.shared.f32    %f92, [%r7+108];  
fma.rn.f32      %f93, %f92, %f91, %f90;
```

That's not good, given that a memory load is bound to have a higher latency than a simple FMA, and given that we know our kernel should be compute bound. We see this effect when looking at the profiler's sampling of warp states. This quantifies how many cycles were spent in each state per executed instruction: Stall Not Selected means that the warp was eligible to be scheduled, but the scheduler selected another eligible warp instead. This adds evidence to our earlier hypothesis that occupancy is currently not a problem.



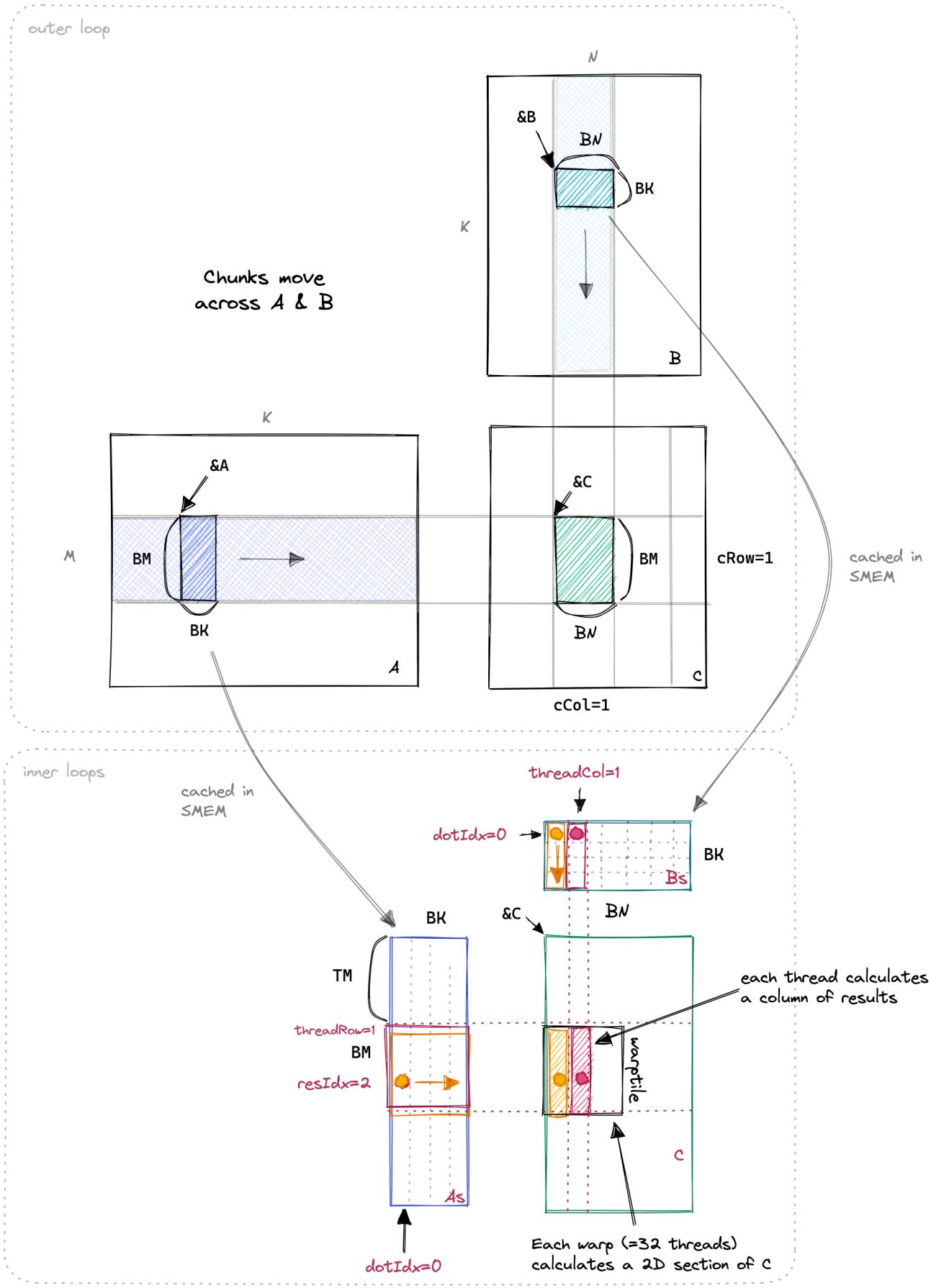
The meaning of the states is documented in the [Kernel Profiling Guide](#). For Stall MIO Throttle it reads:

Warp was stalled waiting for the MIO (memory input/output) instruction queue to be not full. This stall reason is high in cases of extreme utilization of the MIO pipelines, which include special math instructions, dynamic branches, as well as shared memory instructions

We're not using special math instructions, nor dynamic branches, so it's clear that we're stalling waiting for our SMEM accesses to return. So how do we make our kernel issue less SMEM instructions? One way is to have each thread compute more than one output element, which allows us to perform more of the work in registers and relying less on SMEM.

Kernel 4: 1D Blocktiling for Calculating Multiple Results per Thread

So this next kernel works like our last kernel, but adds a new inner loop, for calculating multiple C entries per thread. We now use a SMEM cache size of $BM \times BK + BN \times BK = 64 \times 8 + 64 \times 8 = 1024$ floats, for a total of 4KB per block. Below a visualization. I have highlighted two of the threads and the values they access in the inner loop in orange and red.



```

for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches (same as before)
    As[innerRowA * BK + innerColA] = A[innerRowA * K + innerColA];
    Bs[innerRowB * BN + innerColB] = B[innerRowB * N + innerColB];
    __syncthreads();

    // advance blocktile for outer loop
    A += BK;
    B += BK * N;

    // calculate per-thread results
    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
        // we make the dotproduct loop the outside loop, which facilitates
        // reuse of the Bs entry, which we can cache in a tmp var.
        float Btmp = Bs[dotIdx * BN + threadCol];
        for (uint resIdx = 0; resIdx < TM; ++resIdx) {
            threadResults[resIdx] +=
                As[(threadRow * TM + resIdx) * BK + dotIdx] * Btmp;
        }
    }
    __syncthreads();
}

```

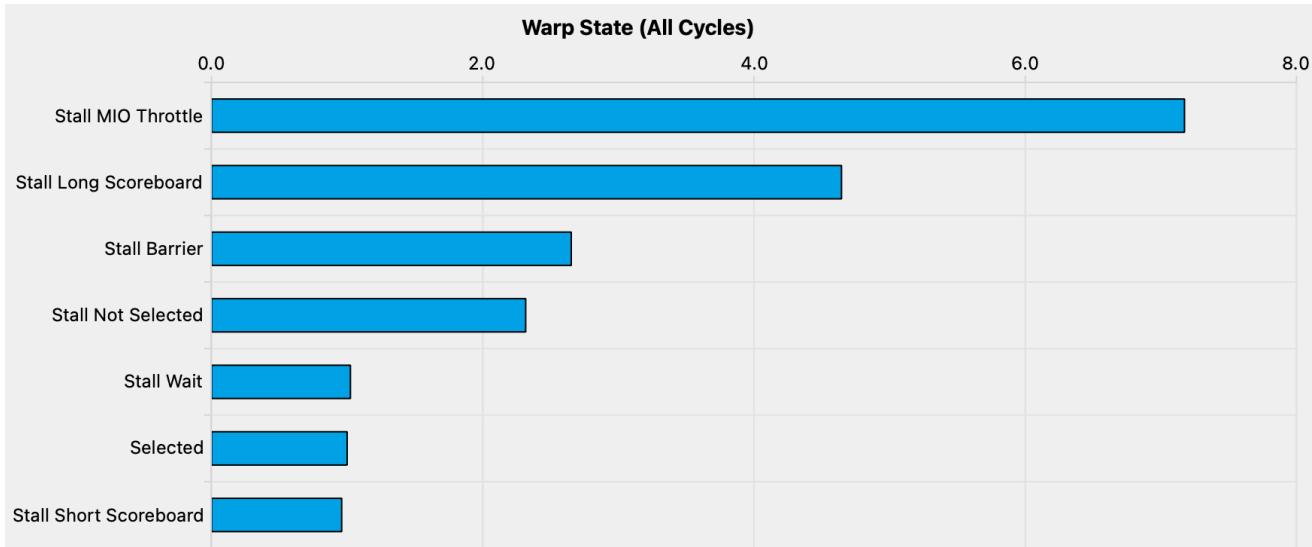
This kernel achieves ~8600 GFLOPs, 2.2x faster than our previous kernel. Let's calculate how many memory accesses each thread performed in our previous kernel, where each thread calculated one result:

- GMEM: $K/32$ iterations of outer loop * 2 loads
- SMEM: $K/32$ iterations of outer loop * BLOCKSIZE (=32) * 2 loads
- Memory accesses per result: $K/16$ GMEM, $K*2$ SMEM

And for our new kernel, where each thread calculates eight results:

- GMEM: $K/8$ iterations of outer loop * 2 loads
- SMEM: $K/8$ iterations of outer loop * $BK(=8)$ * $(1 + TM(=8))$
- Memory accesses per result: $K/32$ GMEM, $K*9/8$ SMEM

As expected, we now spend much fewer cycles per instruction stalling due to memory pressure: Careful: The axis has changed compared to the previous plot.



Sidenote on Compiler Optimizations

Above we explicitly cached the entry of B into Btmp and reordered the two inner loops for efficiency. If we don't do that, then the code looks like this:

```
for (uint resIdx = 0; resIdx < TM; ++resIdx) {
    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
        threadResults[resIdx] +=
            As[(threadRow * TM + resIdx) * BK + dotIdx] * Bs[dotIdx * BN + threadCol];
    }
}
// first inner-most loop
ld.shared.f32    %f45, [%r9];
ld.shared.f32    %f46, [%r8];
fma.rn.f32      %f47, %f46, %f45, %f212;
ld.shared.f32    %f48, [%r9+256];
ld.shared.f32    %f49, [%r8+4];
fma.rn.f32      %f50, %f49, %f48, %f47;
ld.shared.f32    %f51, [%r9+512];
ld.shared.f32    %f52, [%r8+8];
fma.rn.f32      %f53, %f52, %f51, %f50;
ld.shared.f32    %f54, [%r9+768];
ld.shared.f32    %f55, [%r8+12];
fma.rn.f32      %f56, %f55, %f54, %f53;
ld.shared.f32    %f57, [%r9+1024];
ld.shared.f32    %f58, [%r8+16];
fma.rn.f32      %f59, %f58, %f57, %f56;
ld.shared.f32    %f60, [%r9+1280];
ld.shared.f32    %f61, [%r8+20];
fma.rn.f32      %f62, %f61, %f60, %f59;
ld.shared.f32    %f63, [%r9+1536];
ld.shared.f32    %f64, [%r8+24];
fma.rn.f32      %f65, %f64, %f63, %f62;
ld.shared.f32    %f66, [%r9+1792];
```

```

ld.shared.f32    %f67, [%r8+28];
fma.rn.f32      %f212, %f67, %f66, %f65;
// second inner-most loop
ld.shared.f32    %f68, [%r8+32];
fma.rn.f32      %f69, %f68, %f45, %f211;
ld.shared.f32    %f70, [%r8+36];
fma.rn.f32      %f71, %f70, %f48, %f69;
ld.shared.f32    %f72, [%r8+40];
fma.rn.f32      %f73, %f72, %f51, %f71;
ld.shared.f32    %f74, [%r8+44];
fma.rn.f32      %f75, %f74, %f54, %f73;
ld.shared.f32    %f76, [%r8+48];
fma.rn.f32      %f77, %f76, %f57, %f75;
ld.shared.f32    %f78, [%r8+52];
fma.rn.f32      %f79, %f78, %f60, %f77;
ld.shared.f32    %f80, [%r8+56];
fma.rn.f32      %f81, %f80, %f63, %f79;
ld.shared.f32    %f82, [%r8+60];
fma.rn.f32      %f211, %f82, %f66, %f81;
// ... continues like this for inner-loops 3-8 ...

```

The compiler unrolls both loopsThe compiler can unroll them since the loop count is known at compile time. and then eliminates the repeated SMEM loads of the Bs entries, so we end up with the same amount of SMEM accesses as our optimized CUDA code.

When the PTX is compiled to SASS, the SMEM loads from Bs are vectorized:This already hints at an optimization we'll perform later: Transposing As such that we can also vectorize those loads.

```

LDS      R26, [R35.X4+0x800] // a 32b load from As
LDS.128 R8,  [R2]           // a 128b load from Bs
LDS.128 R12, [R2+0x20]
LDS      R24, [R35.X4+0x900]
LDS.128 R20, [R2+0x60]
LDS      R36, [R35.X4+0xb00]
LDS.128 R16, [R2+0x40]
LDS.128 R4,  [R2+0x80]
LDS      R38, [R35.X4+0xd00]

```

Areas of Improvement: Arithmetic Intensity

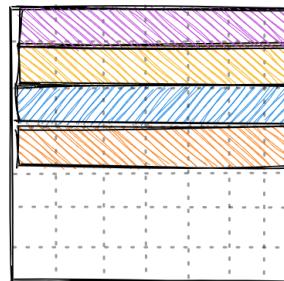
Our current kernel still suffers from the same stalling-for-memory problem as kernel 3, just to a lesser extent. So we'll just apply the same optimization again: computing even more results per thread. The main reason this makes our kernel run faster is that it increases arithmetic intensity. Defined as the

number of FLOPs executed per byte transferred (load + store!) between GMEM and SMEM. Below I tried to make it more immediately obvious why calculating more results per thread raises arithmetic intensity: It's more efficient to calculate a square of results per thread than a column of results because we can share more of the inputs:

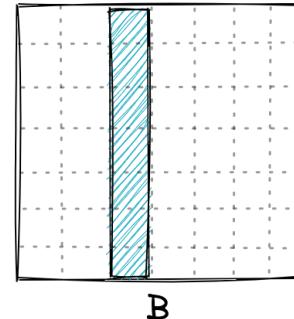
calculating 4 results per thread requires:

- 28 loads from A
- 7 loads from B
- 4 loads & 4 stores to C

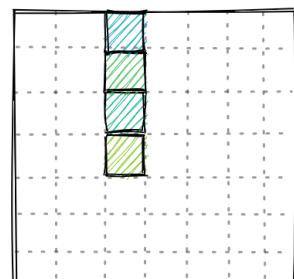
$\Rightarrow 9.75$ loads & 1 store per result



A



B

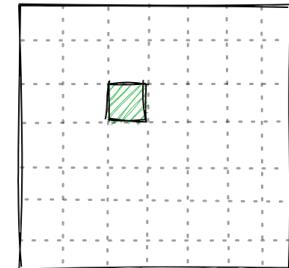
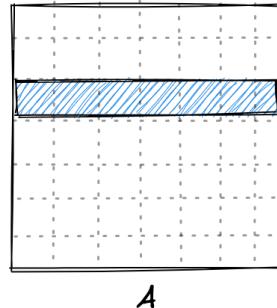
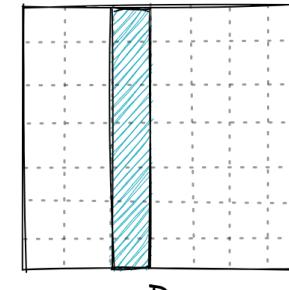


C

Calculating 1 result per thread requires:

- 7 loads from A
- 7 loads from B
- 1 load & 1 store to C

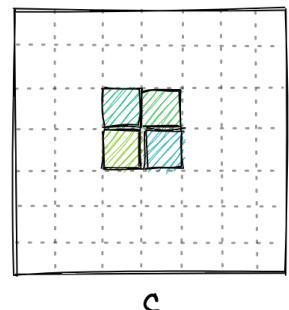
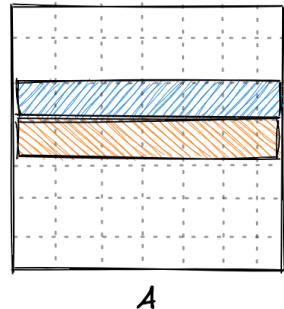
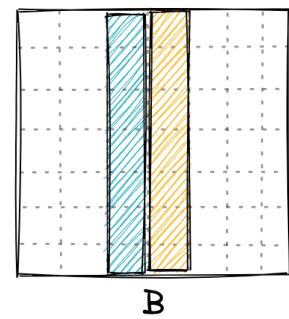
⇒ 15 loads & 1 store per result



Calculating 4 results per thread requires:

- 14 loads from A
- 14 loads from B
- 4 loads & 4 stores to C

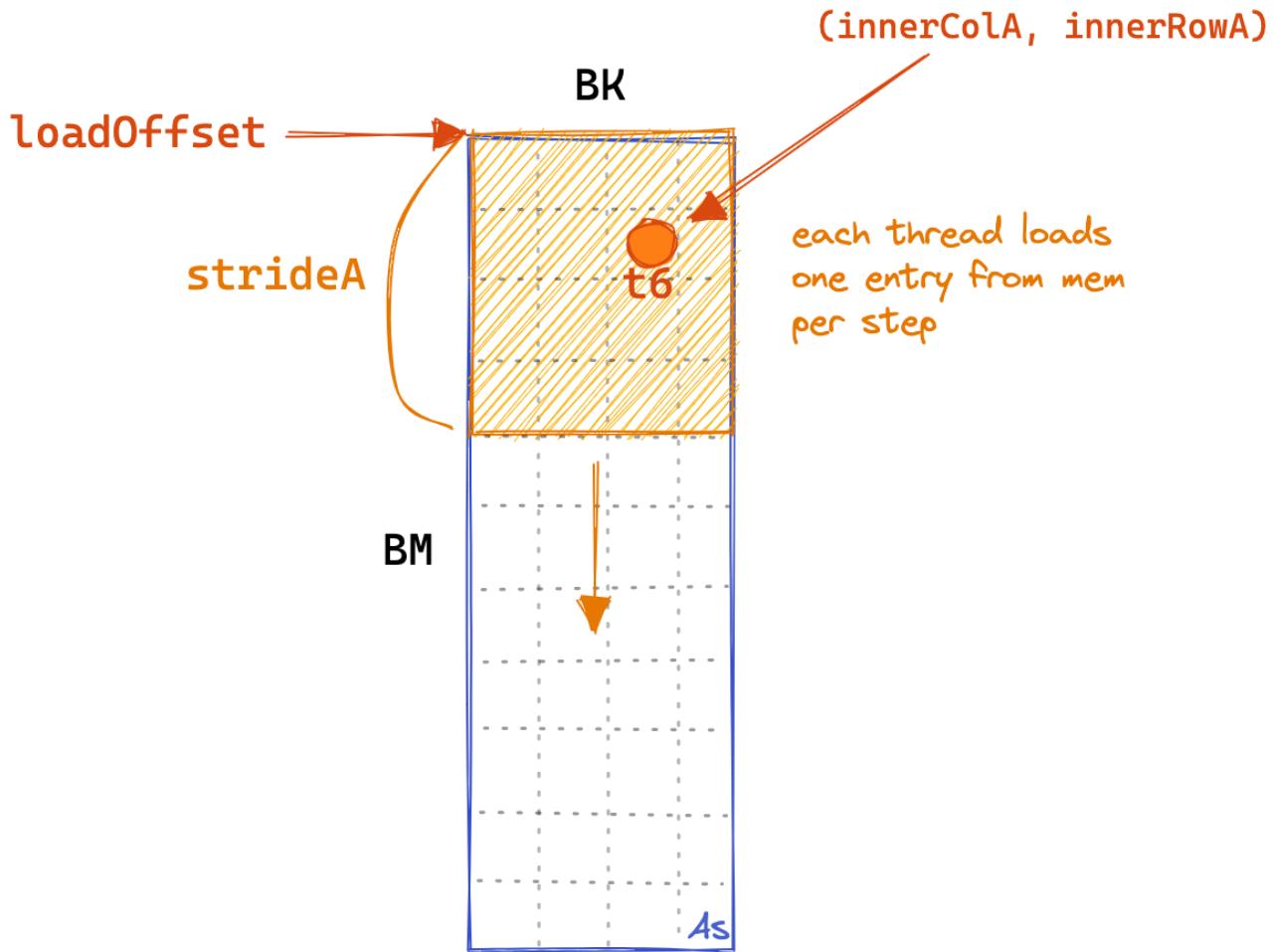
⇒ 8 loads & 1 store per result



In conclusion, all our kernels perform the same number of FLOPs, but we can reduce the number of GMEM accesses by calculating more results per thread. We'll continue optimizing arithmetic intensity for as long as we're still memory bound.

Kernel 5: Increasing Arithmetic Intensity via 2D Blocktiling

The basic idea for kernel 5 will be to compute a grid of 8*8 elements of C per thread. The first stage of the kernel is for all threads to work together to populate the SMEM cache. We'll have each thread load multiple elements. This code looks like so: Here's a graphical representation of the GMEM loading:

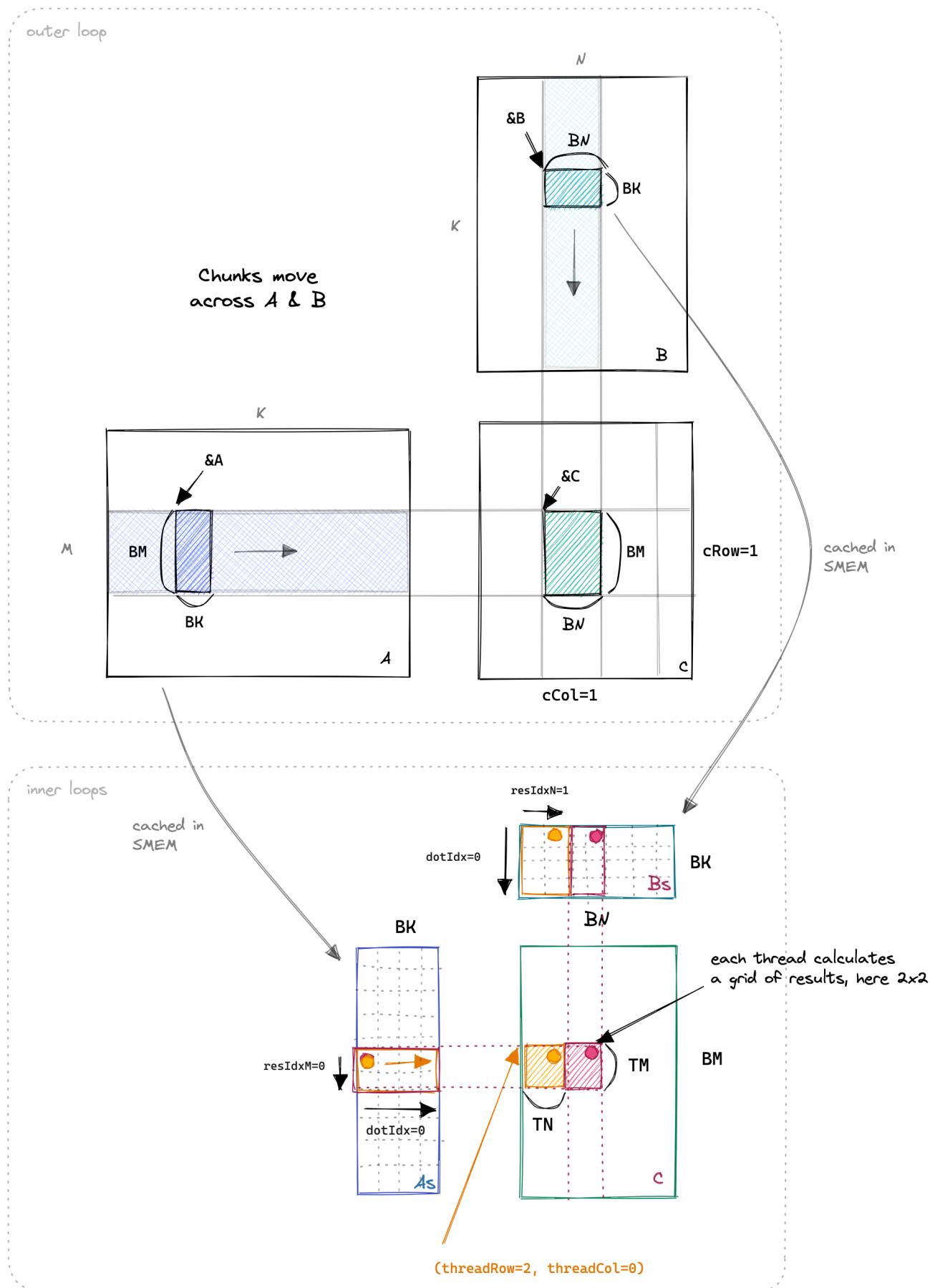


Load-Tile spans full width of As
for optimal GMEM coalescing

```
for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
    As[(innerRowA + loadOffset) * BK + innerColA] =
        A[(innerRowA + loadOffset) * K + innerColA];
}
for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
    Bs[(innerRowB + loadOffset) * BN + innerColB] =
        B[(innerRowB + loadOffset) * N + innerColB];
}
__syncthreads();
```

Now that the SMEM cache is populated, we have each thread multiply its relevant SMEM entries and accumulate the result into local registers. Below I illustrated the (unchanged) outer

loop along the input matrices, and the three inner loops for the dot product and the TN and TM dimension:



```

// allocate thread-local cache for results in registerfile
float threadResults[TM * TN] = {0.0};
// register caches for As and Bs
float regM[TM] = {0.0};
float regN[TN] = {0.0};

// outer-most loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
    // populate the SMEM caches
    for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
        As[(innerRowA + loadOffset) * BK + innerColA] =
            A[(innerRowA + loadOffset) * K + innerColA];
    }
    for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
        Bs[(innerRowB + loadOffset) * BN + innerColB] =
            B[(innerRowB + loadOffset) * N + innerColB];
    }
    __syncthreads();

    // advance blocktile
    A += BK;      // move BK columns to right
    B += BK * N; // move BK rows down

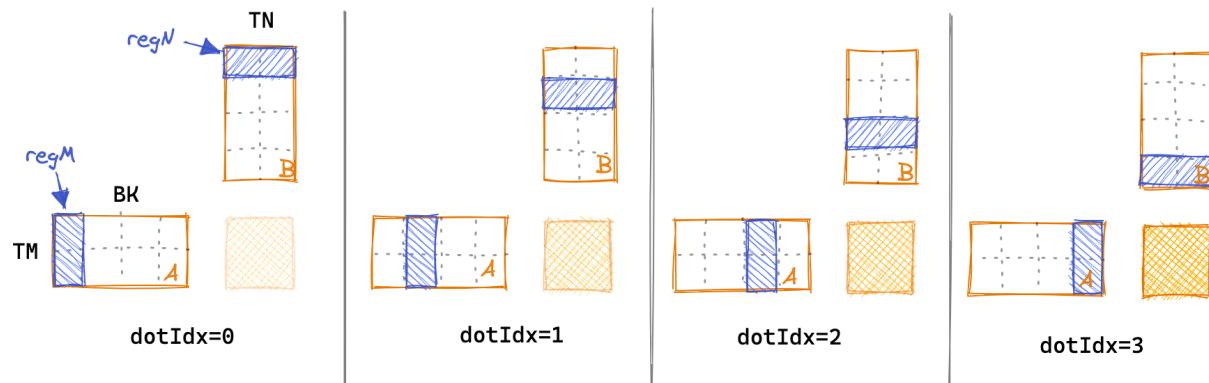
    // calculate per-thread results
    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
        // load relevant As & Bs entries into registers
        for (uint i = 0; i < TM; ++i) {
            regM[i] = As[(threadRow * TM + i) * BK + dotIdx];
        }
        for (uint i = 0; i < TN; ++i) {
            regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
        }
        // perform outer product on register cache, accumulate
        // into threadResults
        for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
            for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
                threadResults[resIdxM * TN + resIdxN] +=
                    regM[resIdxM] * regN[resIdxN];
            }
        }
    }
    __syncthreads();
}

```

In the inner loop, we can reduce the number of SMEM accesses by making `dotIdx` the outer loop, and explicitly loading the values we need for the two inner loops into registers. Below is a drawing of the `dotIdx` loop across time, to visualize which SMEM entries get loaded into thread-local registers at each step:

had to reduce some dimensions to make it easier to draw. In the kernel: $BK=TM=TN=8$.

Unrolled dotIdx loop:



at each timestep, load the 4 relevant As&Bs entries into regM and regN registers, and accumulate outer product into threadResults.

Benefit: We only issue 16 SMEM loads in total

Resulting performance: 16TFLOPs, another 2x improvement. Let's repeat the memory access calculation. We're now calculating $TM \times TN = 8 \times 8 = 64$ results per thread.

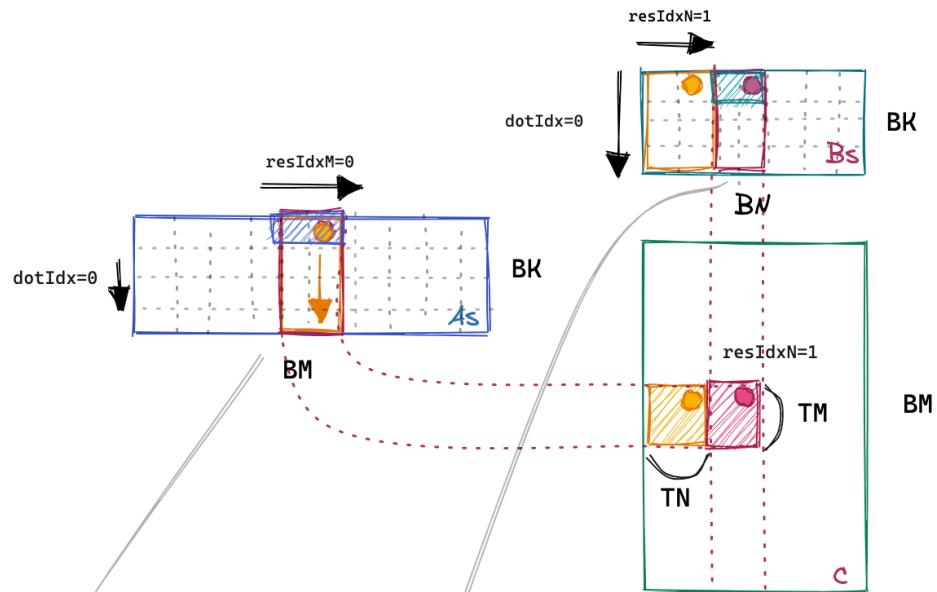
- GMEM: $K/8$ (outer loop iters) * 2 (A+B) * 1024/256 (sizeSMEM/numThreads) loads
- SMEM: $K/8$ (outer loop iters) * 8 (dotIdx) * 2 (A+B) * 8 loads
- Memory accesses per result: K/64 GMEM, K/4 SMEM

Slowly performance is reaching acceptable levels, however, warp stalls due to memory pipeline congestion are still too frequent. For kernel 6 we'll take two measures to try to improve that: Transposing As to enable auto-vectorization of SMEM loads, and promising the compiler alignment on the GMEM accesses.

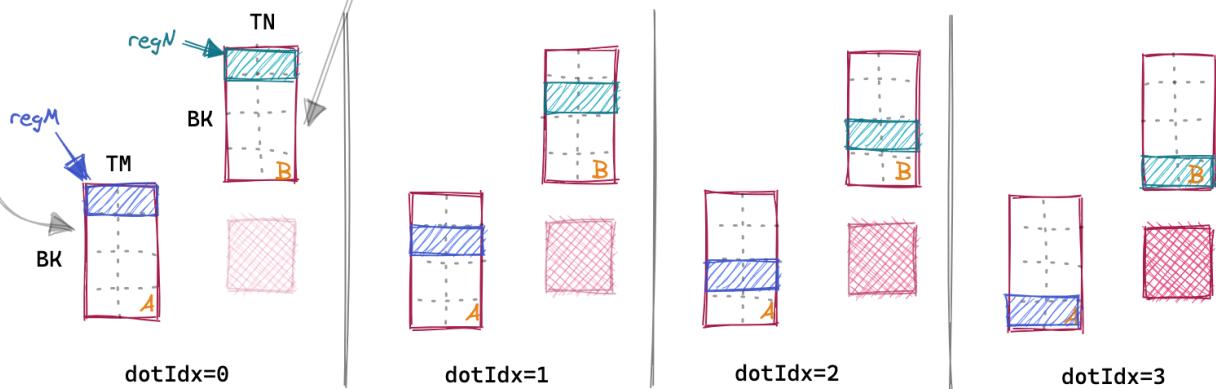
Kernel 6: Vectorize SMEM and GMEM Accesses

The first optimization that I already hinted at earlier is to transpose As. This will allow us to load from As using vectorized SMEM loads (LDS.128 in SASS). Below the same visualization of the three inner loops as for kernel 5, but now with As transposed in memory:

inner loops: dotIdx, resIdxM, resIdxN



Unrolled dotIdx loop:



Only change to previous kernel: Now populating `regM` from `As` can also be done using a vectorized SMEM load, just like it already had been for `regN`.

Next, we'll vectorize all loads and stores from/to GMEM using **vector datatypes**, namely float4.

```
float4 tmp =
    reinterpret_cast<float4*>(&A[innerRowA * K + innerColA * 4])[0];
// transpose A during the GMEM to SMEM transfer
As[(innerColA * 4 + 0) * BM + innerRowA] = tmp.x;
As[(innerColA * 4 + 1) * BM + innerRowA] = tmp.y;
As[(innerColA * 4 + 2) * BM + innerRowA] = tmp.z;
As[(innerColA * 4 + 3) * BM + innerRowA] = tmp.w;
```

```
reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =  
    reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];  
__syncthreads();
```

This leads to the 32b GMEM load instructions (LDG.E and STG.E) being replaced with 128b counterparts (LDG.E.128 and STG.E.128). Initially, I was confused as to why running this:

```
reinterpret_cast<float4 *>(&Bs[innerRowB * BN + innerColB * 4])[0] =  
    reinterpret_cast<float4 *>(&B[innerRowB * N + innerColB * 4])[0];
```

would be any faster than just manually unrolling the access (or using pragma unroll):

```
Bs[innerRowB * BN + innerColB * 4 + 0] = B[innerRowB * N + innerColB * 4 + 0];  
Bs[innerRowB * BN + innerColB * 4 + 1] = B[innerRowB * N + innerColB * 4 + 1];  
Bs[innerRowB * BN + innerColB * 4 + 2] = B[innerRowB * N + innerColB * 4 + 2];  
Bs[innerRowB * BN + innerColB * 4 + 3] = B[innerRowB * N + innerColB * 4 + 3];
```

Shouldn't the compiler just be able to coalesce the 2nd version and also generate 128b loads? I think the reason is that the compiler has no way to verify that the float* B pointer that is passed to the kernel is 128b aligned, which would be a requirement for using LDG.E.128. So the reinterpret_cast's only purpose is to promise the compiler that the float* B pointer will be aligned. Compare this to SMEM loads, where the compiler automatically generates vectorized loads because that memory is not user-managed.

Kernel 6 achieves 19TFLOPs. The profiler still shows a bunch of problem areas and optimization opportunities: We're running into shared-memory bank conflicts (which cuBLAS avoids), our occupancy is higher than necessary, and we haven't implemented any double buffering (which the [CUTLASS docs](#) seem to suggest is pretty useful).

But before we get to those, let's cover some more low-hanging fruit: Autotuning the kernel's parameters.

Kernel 9: Autotuning I skipped kernels 7 and 8, which I wrote while figuring out how to best eliminate shared memory bank conflicts. They eliminate the conflicts

but were overall still slower, so I won't cover them here.

We've accumulated a total of five template parameters:

- BM, BN and BK, which specify how much data we cache from GMEM into SMEM.
- TM and TN, which specify how much data we cache from SMEM into the registers.

For kernel 6, these were set to BM=BN=128 and BK=TM=TN=8. I wrote a bash script that searches through all sensible combinations and benchmarks their runtime. This required me to make sure that:

1. I knew which parameter combinations were sensible, and skip those that weren't. Example of a non-sensible configuration: I wanted to vectorize all SMEM loads, hence $BM * BK$ (the size of As) needed to be divisible by $4 * NUM_THREADS$, as each thread in the block issues a 4-wide load during each iteration of the GMEM to SMEM loading loop.
2. The kernel implementation was correct for the ~400 different hyperparameter settings that remained.

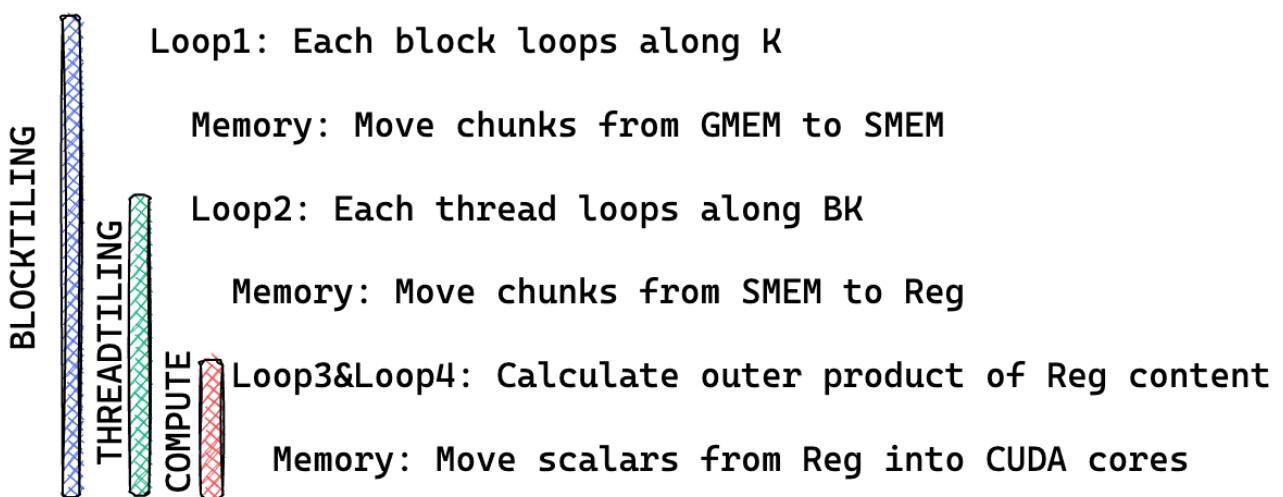
The necessary modifications to the code ended up taking quite some time to implement.

It turns out that the optimal parameters vary quite a bit depending on the GPU model. I guess that's why compilers like [Triton](#) provide routines for autotuning. I wonder how this works for cuBLAS, they probably store a precomputed mapping from {GPU type, matrix size, dtype, ...} to the optimal GEMM implementation inside the cuBLAS binary. On my A6000, BM=BN=128 BK=16 TM=TN=8 increased performance by 5%, from 19 to 20 TFLOPs. On an A100 SMX4 40GB, that same configuration reached 12 TFLOPs, 6% worse than the optimal setting found by the autotuner (BM=BN=64 BK=16 TM=TN=4), which reached 12.6 TFLOPs. The A100 has worse fp32 performance than the A6000, which is why the FLOPs numbers are lower (cuBLAS reaches 14.7 TFLOPs on the A100). Nvidia rates the A100 at 19.5 TFLOPs and the A6000 at 38.7 TFLOPs.

I can't explain why these specific parameters end up producing the optimal performance. Autotuning works, every high-performance library uses it, but it also feels very unsatisfying. I'm sure that with enough time, enough access to low-level performance counters and some facetime with Nvidia engineers, I'd eventually figure it out. It's good have a strong belief that [computers can be understood](#).

Kernel 10: Warptiling

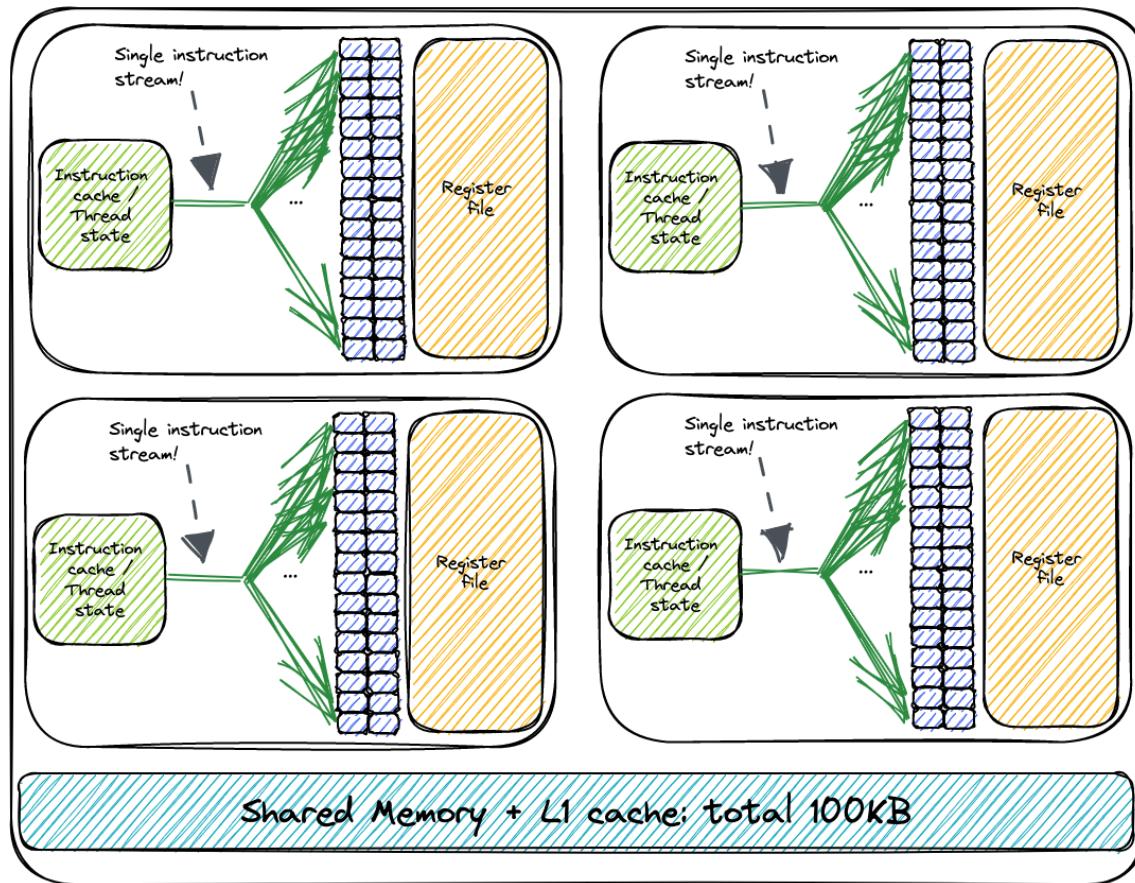
Currently, our loop structure looks like this:



We'll now add another hierarchy of tiling, in between our blocktiling and threadtiling loops: warptiling. Warptiling is somewhat confusing initially since unlike blocks and threads, warps don't show up anywhere in the CUDA code explicitly. They are a hardware feature that has no direct analog in the scalar CUDA-software world. We can calculate a given thread's warpId as `warpId=threadIdx.x % warpSize`, where warpSize is a built-in variable that is equal to 32 on any CUDA GPU I've ever worked with.

Warps are relevant for performance since (among other reasons):

- Warps are the unit of scheduling that is mapped to the warp-schedulers that are part of the SM. On my A6000, there are four warp schedulers in each SM. This is how I imagine this looks:



- Shared-memory bank conflicts (I'll cover those in a future post) happen only between threads that are in the same warp.
- There's a register cache on recent GPUs, and tighter threadtiling gives us more register cache locality.

Warptiling is elegant since we now make explicit all levels of parallelism:

- Blocktiling: Different blocks can execute in parallel on different SMs.
- Warptiling: Different warps can execute in parallel on different warp schedulers, and concurrently on the same warp scheduler.
- Threadtiling: (a very limited amount of) instructions can execute in parallel on the same CUDA cores (= instruction-level parallelism aka ILP).

```
// dotIdx loops over contents of SMEM
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
    // populate registers for this thread's part of the warptile
    for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRowIdx) {
        for (uint i = 0; i < TM; ++i) {
            regM[wSubRowIdx * TM + i] =

```

```

        As[(dotIdx * BM) + warpRow * WM + wSubRowIndex * WSUBM +
            threadRowInWarp * TM + i];
    }
}

for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
    for (uint i = 0; i < TN; ++i) {
        regN[wSubColIdx * TN + i] =
            Bs[(dotIdx * BN) + warpCol * WN + wSubColIdx * WSUBN +
                threadColInWarp * TN + i];
    }
}

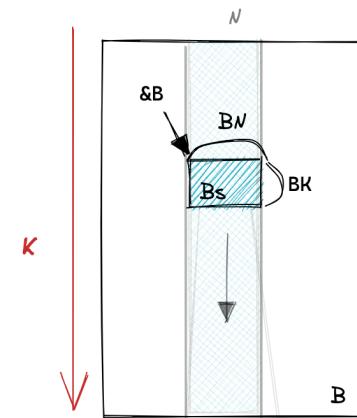
// execute warptile matmul. Later this will map well to
// warp-wide matrix instructions, executed on tensor cores.
for (uint wSubRowIndex = 0; wSubRowIndex < WMITER; ++wSubRowIndex) {
    for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubColIdx) {
        // calculate per-thread results with register-cache locality
        for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
            for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
                threadResults[(wSubRowIndex * TM + resIdxM) * (WNITER * TN) +
                    (wSubColIdx * TN) + resIdxN] +=
                    regM[wSubRowIndex * TM + resIdxM] *
                    regN[wSubColIdx * TN + resIdxN];
            }
        }
    }
}
}

```

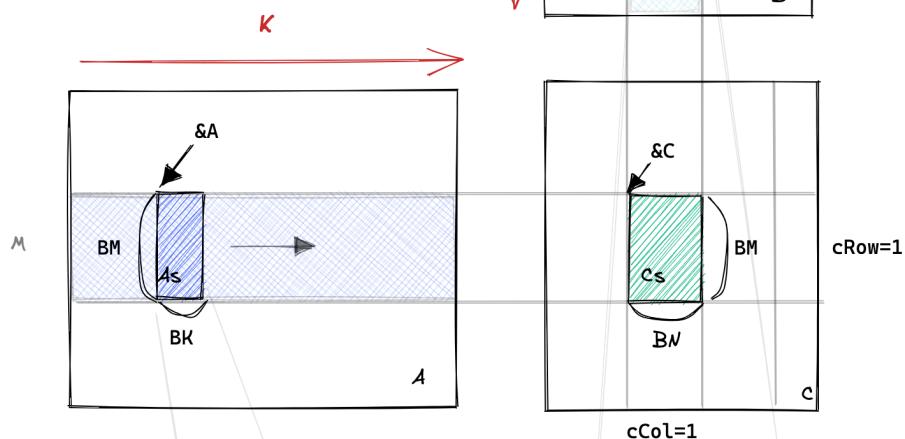
I tried my best to visualize all three levels of tiling below, although the structure is getting quite complex. The [CUTLASS docs about efficient GEMMs](#) go even more in-depth into warptiling, and their visualizations are illuminating. Each warp will compute a chunk of size $(WSUBN * WNITER) \times (WSUBM * WMITER)$. Each thread computes $WNITER * WMITER$ many chunks of size $TM \times TN$.

Outermost loop over K

This loop determines which slices As & Bs of A & B get loaded from GMEM into SMEM



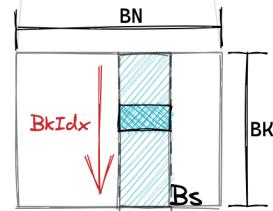
Whole GPU



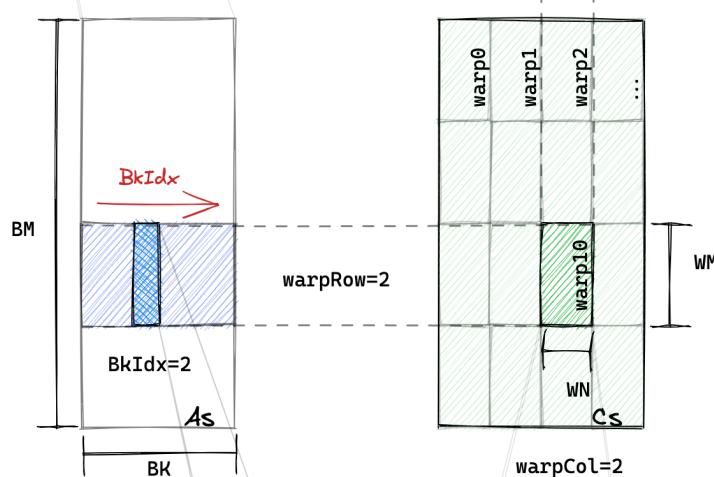
cache from GMEM into SM-local SMEM

Loop over BK

This loop determines which parts of As & Bs get loaded from SM-wide SMEM into Warpscheduler-local registerfile

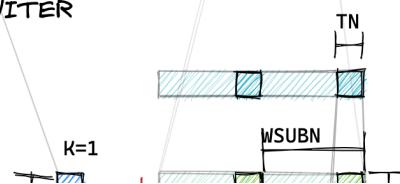


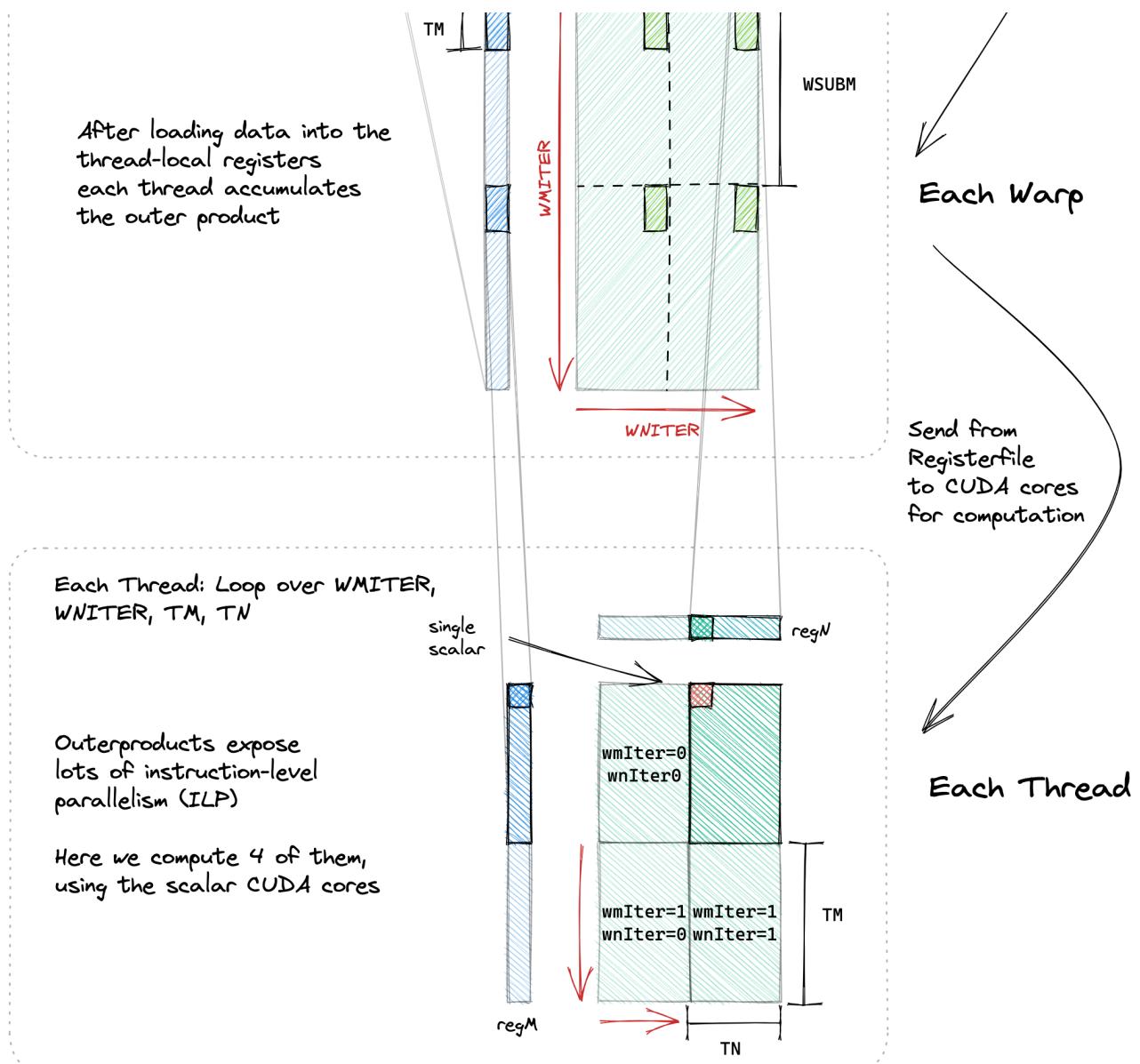
Each SM



cache from SMEM into Registerfile

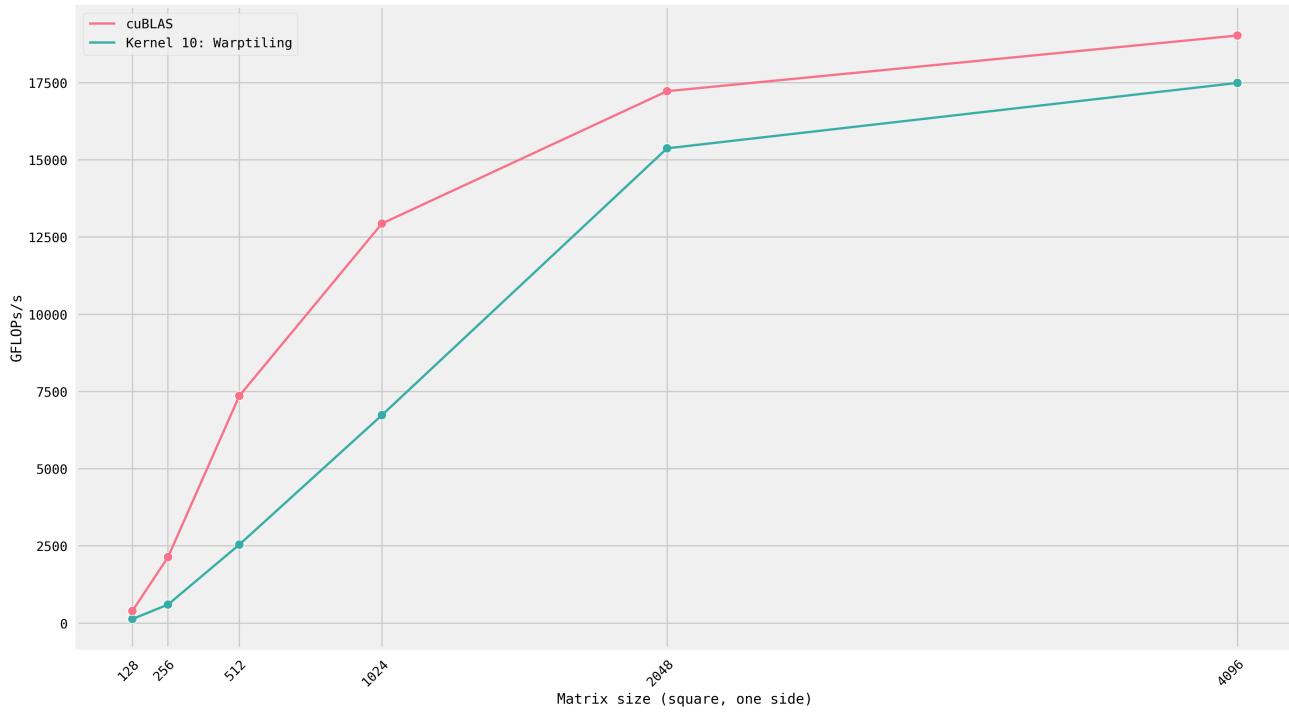
Warptiling: Loop over WMITER, WNITER





After autotuning the parameters, performance improves from 19.7 TFLOPs to 21.7 TFLOPs on an A100.

Here's a plot that compares our warptiling kernel against cuBLAS across increasing matrix sizes: I generated this plot on an A100, which is why the absolute FLOPs numbers are different.

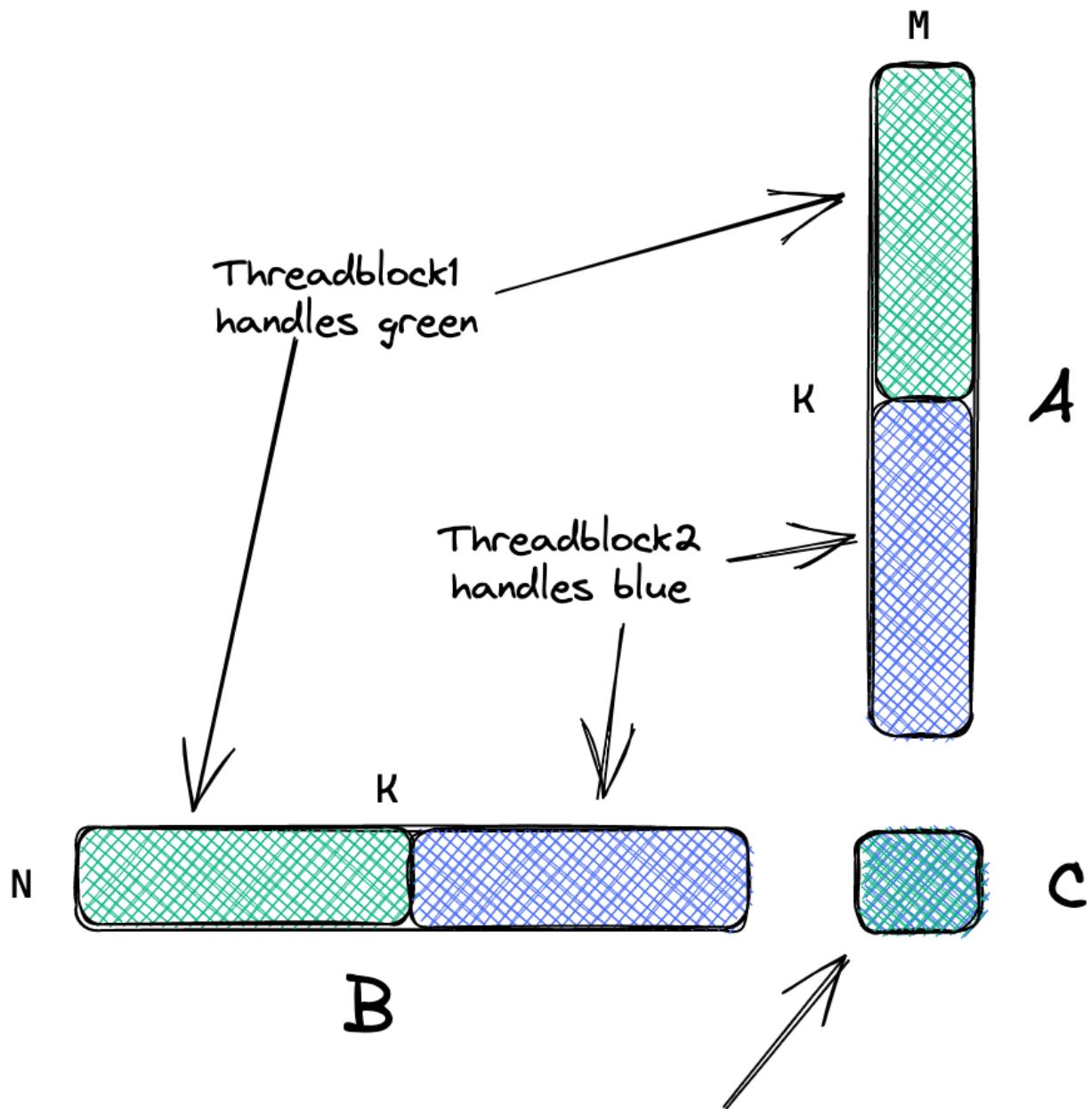


At dimensions 2048 and 4096, our measured FLOPs are only a few percentage points slower than cuBLAS. However, for smaller matrices, we're doing poorly in comparison to Nvidia's library! This happens because cuBLAS contains not one single implementation of SGEMM, but hundreds of them. There's a reason I guess for why the library is 500MB of compiled code. To print all the kernels: `cuobjdump --list-text <cublas location>`. At runtime, based on the dimensions, cuBLAS will pick which kernel to run. I launched matmuls for square matrices on all dimensions up to 4096 and found 16 different SGEMM kernels. Here's a [script](#) for finding the kernel that was launched by cuBLAS (h/t [Horace He](#)). I traced the cuBLAS call and these are the kernels it's calling at each size: I used the [Nsight Systems](#) CLI for this.

Matrix size	Name	Duration
128	ampere_sgemm_32x32_sliced1x4_nn	15.295 μ s
256	ampere_sgemm_64x32_sliced1x4_nn followed by splitKreduce_kernel	12.416 μ s + 6.912 μ s
512	ampere_sgemm_32x32_sliced1x4_nn	41.728 μ s
1024	ampere_sgemm_128x64_nn	165.953 μ s

Matrix size	Name	Duration
2048	ampere_sgemm_128x64_nn	1.247 ms
4096	ampere_sgemm_128x64_nn	9.290 ms

At dimension 256 it calls two kernels: a matmul kernel followed by a reduction kernel. Split-K refers to partitioning the K-dimension across multiple threadblocks. This means that each block will only compute part of the chunk of C, and cuBLAS follows up with a reduce kernel to accumulate the final result. This requires some extra memory space to store the intermediate results before the reduction. I imagine this looks like so (but I'm uncertain here):



The two blocks need to synchronize when writing into the C result matrix (presumably using atomics)

So if we were trying to write a high-performance library that works for all shapes and sizes we would have specializations for different shapes, and at runtime dispatch to the one that's the best fit.

I also want to report a negative results: For this kernel, I additionally implemented an optimization called *thread swizzling*. This technique assumes that threadblocks are launched in order of increasing `blockIdx`, and optimizes the mapping of `blockIdx` to C chunks in a way that should increase L2

locality. Remember that L2 is a cache for global memory that exists once for the whole GPU. This [Nvidia post](#) has more info and visualizations. It didn't increase performance, presumably because L2 hit rate is already fairly high at 80%, so I ended up removing the swizzling code. The commit is [here](#) if anyone is interested.

It makes sense to move the loop over BK towards the outside, since it follows our maxim of "load some data, then do as much work on that data as possible". It further means that all *computation* that happens inside the BK loop will be independent and can be parallelized (for example using ILP).

We can now also start prefetching the data necessary for the next loop iteration already, a technique called double buffering.

Work in Progress: Kernel 11

If I get back to working on this post, here's what I'll look at next:

1. Double buffering, for better interleaving of computation and memory loading. For now, see [CUTLASS Pipelining](#). In CUTLASS, double buffering is done on two levels: GMEM \Rightarrow SMEM, and SMEM \Rightarrow Registerfile.
 - In Hopper, new instructions were introduced for warp specialization, for example for having some warp use fewer registers than others. This, in combination with special instructions to load directly from GMEM into SMEM without first going through the registers, can be used to reduce register pressure.
2. Getting rid of SMEM bank conflicts. This can be done by [optimizing the data layout in SMEM](#).
3. Better understanding the GEMM kernels that are implemented in [Triton](#), by looking at the generated PTX.

Conclusion

Writing this post was a similar experience to my previous post on [optimizing SGEMM on CPU](#): Optimizing SGEMM iteratively is one of the best ways to deeply understand the performance characteristics of the hardware. For writing the CUDA programs I was surprised by how easy it was to implement the code once I had made a good visualization of how I wanted the kernel to work.

Also: Powerlaws are everywhere. It took me two weekends to write the first 6 kernels which reach 80% of peak FLOPs, and then 4 more weekends to do autotuning and warptiling to get to 94%. How much I'm learning while writing this code has also seen diminishing results, hence I'm putting off hunting the last 6% until some future time.

All my code is available on [Github](#).

Lastly, a big thanks to the creators of [Godbolt.org](#) (for looking at PTX and SASS assembly) and [Excalidraw](#) (for drawing the kernels)! Both of these tools are a joy to use and have helped me learn much faster.

Further Resources and References

- I started writing this post because I stumbled over [wangzyon's Github repository](#), first experimenting with his kernels and then rewriting everything from scratch. Also relevant is this [Nvidia Blogpost about the CUTLASS library](#).
- Mandatory references: the official [CUDA Toolkit Programming Guide](#) and the [CUDA Best Practices Guide](#). The [Kernel Profiling Guide](#) contains even more info on low-level hardware details like caches and pipelines, and on the various metrics that can be collected.
- Onur Mutlu is a professor at ETH who uploads his lectures to Youtube. Particularly relevant for this post are [Computer Architecture](#) and [Acceleration on Heterogeneous Systems](#).
- [Understanding Latency Hiding on GPUs](#), a Ph.D. thesis that goes in-depth on how to design workloads such that they fully utilize memory bandwidth and computation. It's from 2016 and hence only covers older GPU architectures. The

chapter about warp-synchronous programming is outdated, see [using CUDA warp-level primitives](#).

- Lei Mao (an engineer at Nvidia) has good CUDA content on his [blog](#), including about [proper CUDA error handling](#).
- It seems like there aren't any good official resources for understanding SASS. There is [Nvidia's Docs on CUDA binary utilities](#). More useful might be looking at Open Source SASS assemblers, like Da Yan's [turingas](#).
- I'm collecting examples of readable, yet optimized CUDA code to learn from:
 - ONNX Runtime's [CUDA provider](#), e.g. their [implementation of softmax](#).
 - NVIDIA's Open Source [CUTLASS](#) library, e.g. their [GEMM implementation](#) which uses double-buffering to prefetch the innermost dimension, which is still missing in my kernels.