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

Subscribe Si_Boehm Comments (16)

December 2022

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.[1] [2]

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[3] 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):[4]

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%
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%

–

Come work on kernels at Anthropic!

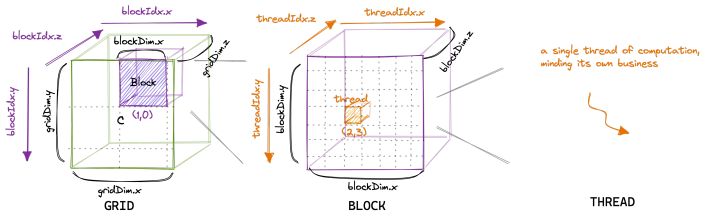
We're always hiring for capable performance & kernel engineers to optimize our models on TPUs, GPUs & Trainium. Apply here!

–

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.[5] 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 host[6], it creates a single grid, containing the blocks and threads as specified.[7] 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:

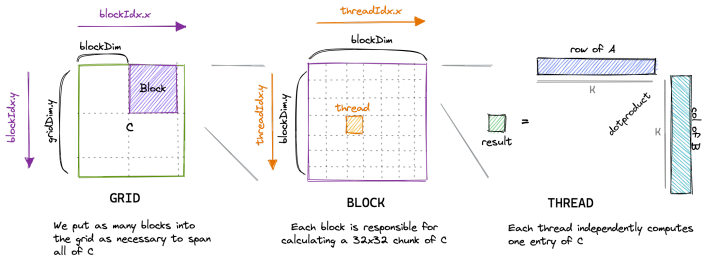
```
// create as many blocks as necessary to map all of C
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
```

```
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the
// The function call returns immediately on the host
```

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 return different values based on the thread that's accessing them.[8] [9]

```
__global__ void sgemm_naive(int M, int N, int K, float alp
                            const float *B, float beta, fl
  // compute position in C that this thread is responsible
  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 mu
  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 = α*(A@B)+β*C
    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
  }
}
```

To visualize this simple kernel:[10]



This kernel takes about 0.5s to process three 4092² 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 4092² matrices, followed by an addition of a 4092² matrix (to make the GEMM):

1. Total FLOPS:[11] `2*4092³ + 4092² = 137 GFLOPS`
2. Total data to read (minimum!): `3 * 4092² * 4B = 201MB`
3. Total data to store: `4092² * 4B = 67MB`

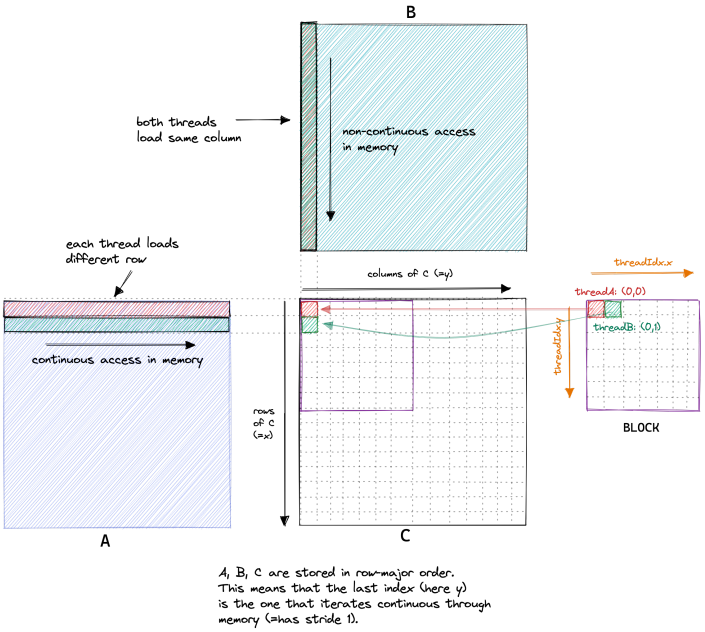
So 268MB is the absolute minimum of memory that any implementation would have to transfer from/to global GPU memory,[12] assuming it has a big enough cache.[13] Let's calculate some upper bounds on kernel performance. The GPU is advertised with 30TFLOPs/s of fp32 compute throughput and 768GB/s of global memory bandwidth. If we achieved those numbers,[14] [15] 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.[16]

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:



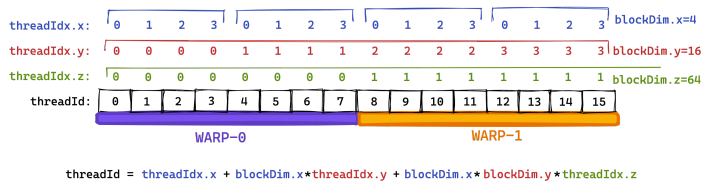
So to recap, when I run this kernel on an A6000 GPU it achieves ~300GFLOPs when multiplying two 4092x4092 float32 matrices. Pretty bad, considering that the A6000 is advertised as being able to achieve almost 30 TFLOPs.[17] So how can we start to make this faster? One way is to optimize the memory access 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.[18] There are four warp schedulers per multiprocessor. The grouping into warps happens based on a consecutive `threadId`. If we set the `blockDim` to be multi-dimension, then the threadId is calculated like so:

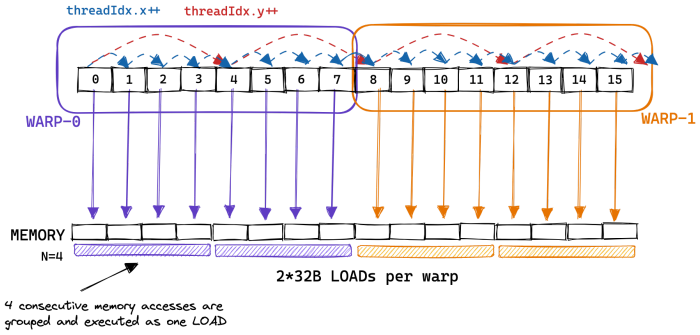
```
threadId = threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*
```

Then, threads with neighbouring `threadId` 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):[19]



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:

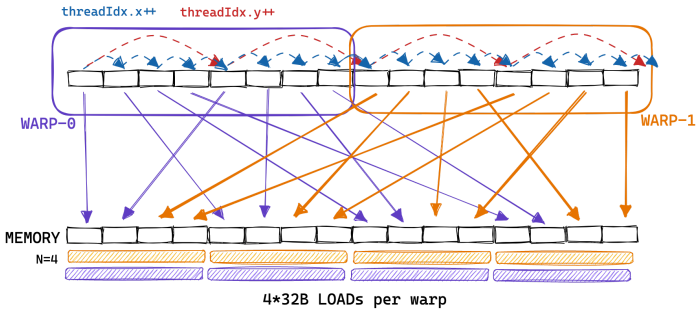


In reality, the GPU supports 32B, 64B and 128B memory accesses. So, if each thread is loading a 32bit float from global memory, the warp scheduler (probably the MIO) can coalesce this `32*4B=128B` load into a single transaction. This is only possible if the floats loaded are consecutive in memory, and if access is aligned.[20] 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.

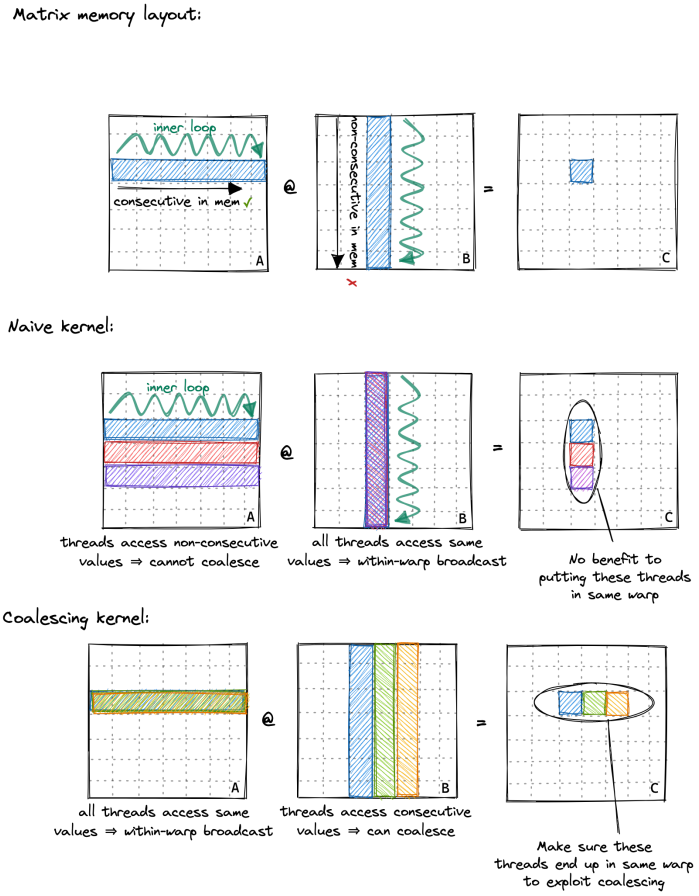
Looking back at the previous kernel, we assigned threads their entry of C like so:

```
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:



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

```
const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOC
const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOC

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:[21]

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

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.

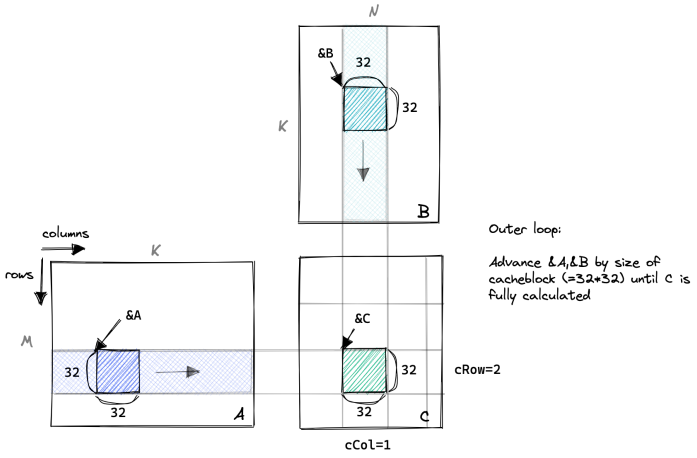
Kernel 3: Shared Memory Cache-Blocking

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.[22] 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.[23]

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.[24]

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.

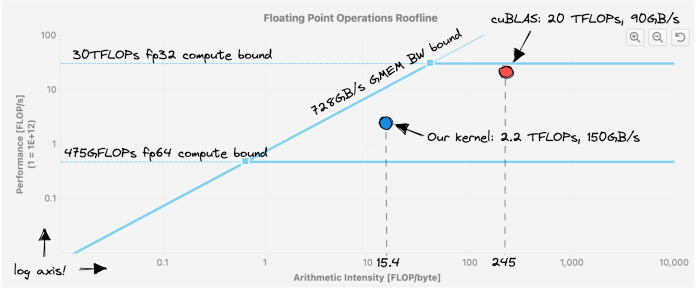
This is illustrated below:



The important parts of the code are below, with variable names corresponding to the plot above:[25]

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

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 fr
  // global memory into shared memory.
  // Make the threadCol (=threadIdx.x) the consecutive ind
  // to allow global memory access coalescing
  As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K
  Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N

  // block threads in this block until cache is fully popu
  __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 thread
  // fetching the next block into the cache before slower
  __syncthreads();
}
C[threadRow * N + threadCol] =
    alpha * tmp + beta * C[threadRow * N + threadCol];
```

This kernel achieves ~2200 GFLOPS, a 50% improvement over the previous version.[26] We're still far away from hitting the ~30 TFLOPs that the GPU can provide. This is obvious from the roofline plot below:[27]



At a CHUNKSIZE of 32, this uses `2*32*32*4B=8KB` of shared memory space.[28] 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 issue-able instructions available.[29] 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):[30]

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
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

Metric	Value
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:

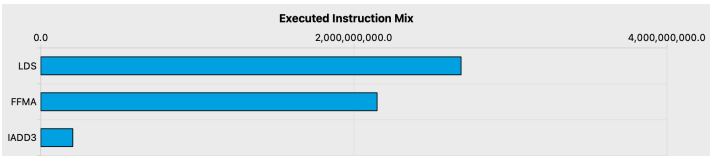
Registers per Thread	37
SMEM per Block	8192 B
Threads per Block	1024

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:[31]

- **Shared memory**: 8192B/Block + 1024B/Block for CUDA runtime usage = 9216B/Block. (102400B per SM) / (9216B per Block) = 11.11 ⇒ 11 Blocks upper limit.
- **Threads**: 1024 Threads per Block, max 1536 threads per SM ⇒ 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 threads / 32) = 32 warps per block, hence 1280 regs per warp * 32 warps per block = 40960 regs per block. Max 65536 regs per SM ⇒ upper limit 1 block. [32]

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 active warps / 48 max active warps = 66%.

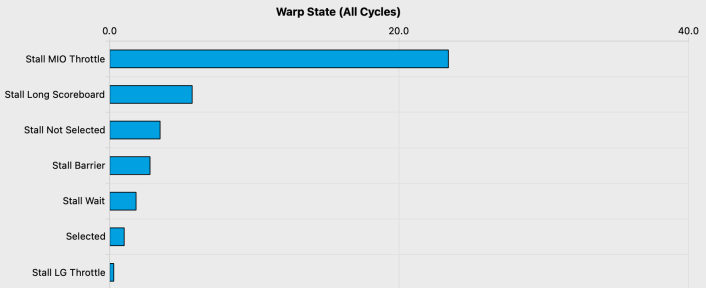
A 66% occupancy is not too bad, so this doesn't explain why our kernel runs so slow.[33] Looking at the profiler gives us some hints. First, if we look at the mix of executed instructions, most of them are memory loads:[34]



Our inner loop looks like this in PTX (Godbolt link):

```
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:[35]



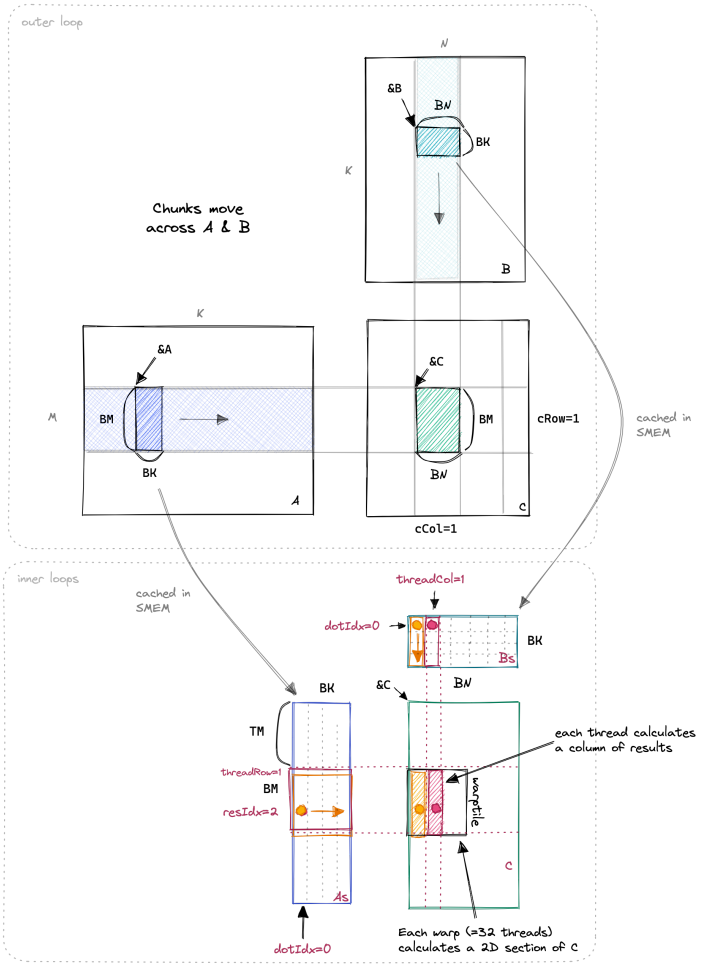
The meaning of the states is documented in the Kernel Profiling Guide. For `Stall MIO Throttle` it reads:

> 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*BK + BN*BK = 64*8 + 64*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.



All of the important changes for this kernel happen in the inner loop. The loading for GMEM to SMEM stays largely the same as before. Let's have a look:[36]

```
// allocate thread-local cache for results in registerfile
float threadResults[TM] = {0.0};

// outer loop over block tiles
for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
  // populate the SMEM caches (same as before)
  As[innerRowA * BK + innerColA] = A[innerRowA * K + inner
  Bs[innerRowB * BN + innerColB] = B[innerRowB * N + inner
  __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
    // reuse of the Bs entry, which we can cache in a tmp
    float Btmp = Bs[dotIdx * BN + threadCol];
    for (uint resIdx = 0; resIdx < TM; ++resIdx) {
      threadResults[resIdx] +=
          As[(threadRow * TM + resIdx) * BK + dotIdx] * Bt
```

```
    }
  }
  __syncthreads();
`
```

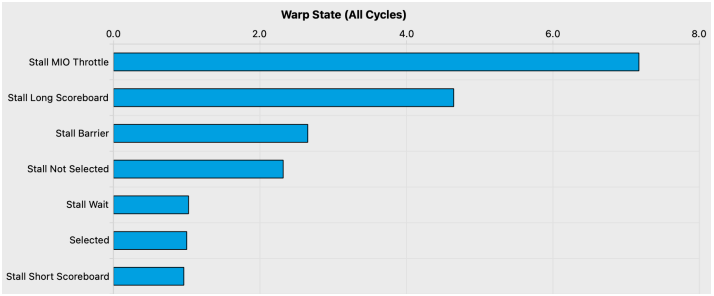
This kernel achieves ~8600 GFLOPs, 2.2x faster than our previous kernel. Let's calculate how many memory accesses each thread performed in our previous kernel, where each thread calculated one result:

- 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:[37]



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[dot
  }
}
```

Interestingly, this has no adverse effect on performance. This is surprising since our inner two loops now incur BK (=8) * TM (=8) * 2 = 128 SMEM accesses, instead of the previous 72. Looking at the assembly (Godbolt link) has the answer:

```
// 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 ...
```

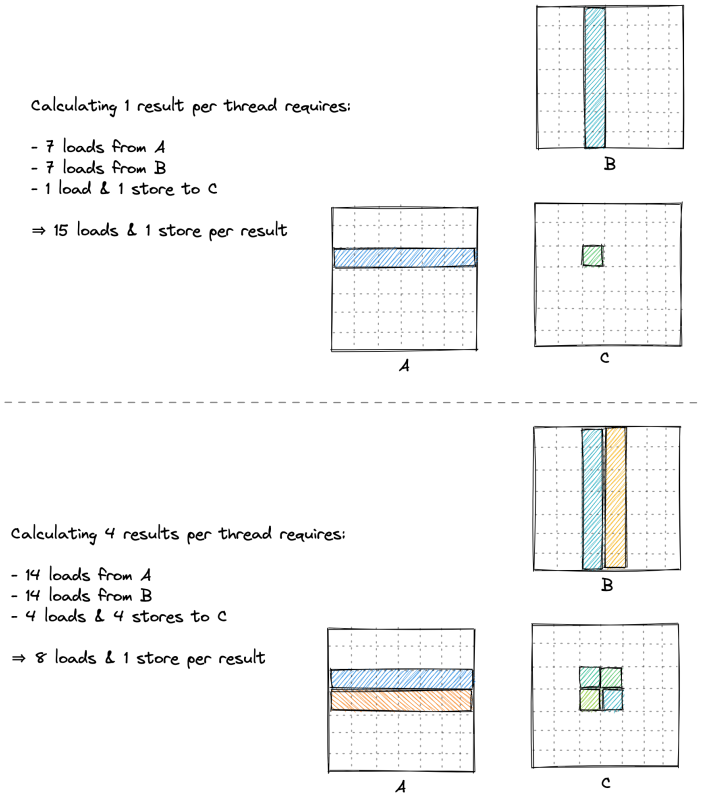
The compiler unrolls both loops[38] and then eliminates the repeated SMEM loads of the Bs entries, so we end up with the same amount of SMEM accesses as our optimized CUDA code.

When the PTX is compiled to SASS, the SMEM loads from Bs are vectorized:[39]

```
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.[40] Below I tried to make it more immediately obvious why calculating more results per thread raises arithmetic intensity:[41]



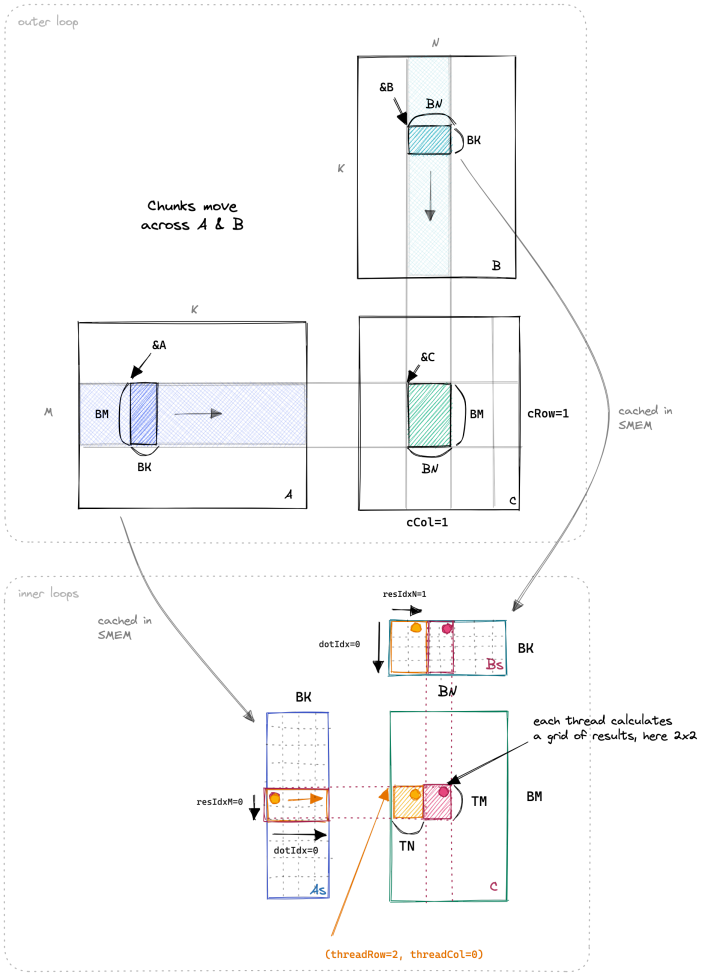
In conclusion, all our kernels perform the same number of FLOPs, but we can reduce the number of GMEM accesses by calculating more results per thread. We'll continue optimizing arithmetic intensity for as long as we're still memory bound.

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:[42]

```
for (uint loadOffset = 0; loadOffset < BM; loadOffset += s
  As[(innerRowA + loadOffset) * BK + innerColA] =
      A[(innerRowA + loadOffset) * K + innerColA];
}
for (uint loadOffset = 0; loadOffset < BK; loadOffset += s
  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:

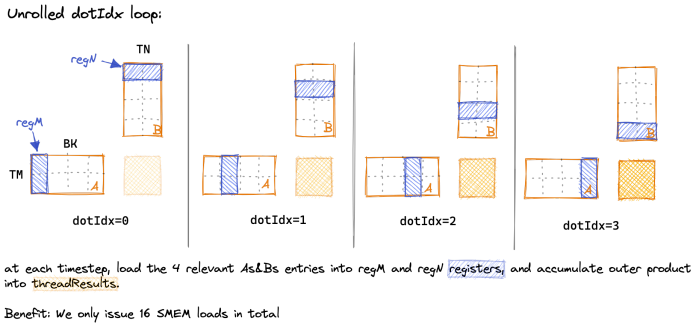


The interesting parts of the code look like this:[43]

```
// 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 +=
    As[(innerRowA + loadOffset) * BK + innerColA] =
        A[(innerRowA + loadOffset) * K + innerColA];
  }
  for (uint loadOffset = 0; loadOffset < BK; loadOffset +=
    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:[44]



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

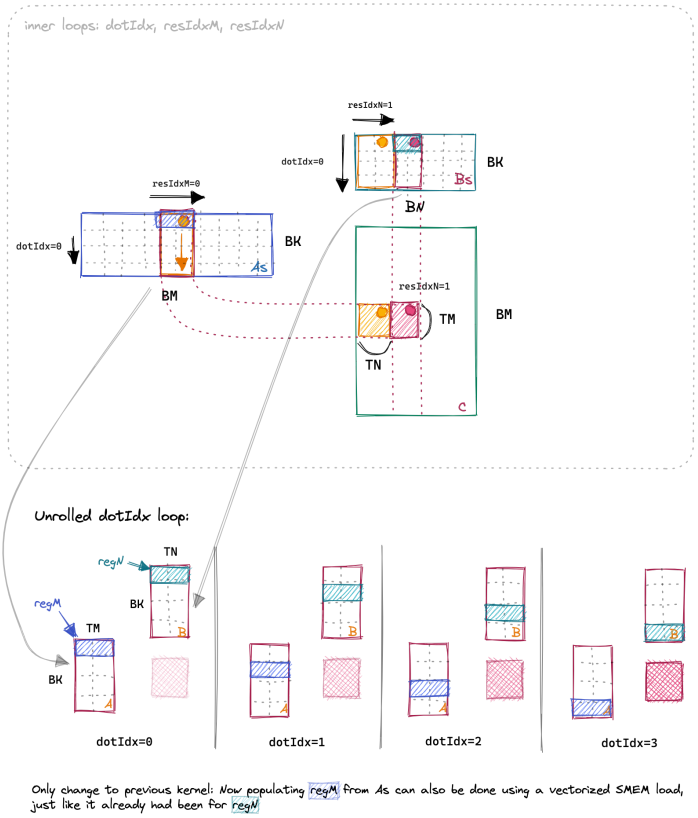
- GMEM: K/8 (outer loop iters) * 2 (A+B) * 1024/256 (sizeSMEM/numThreads) loads
- SMEM: K/8 (outer loop iters) * 8 (dotIdx) * 2 (A+B) * 8 loads
- Memory accesses per result: K/64 GMEM, K/4 SMEM

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:



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

Looking at the assembly[45] we see that loading `As` into the registers, which used to be a 32b `LDS` load, is now also a 128b `LDS.128` load, just like it had already been for `Bs`. This gives us a 500GFLOPs speedup, or ~3%.

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

The code looks like this:[46]

```
float4 tmp =
    reinterpret_cast<float4 *>(&A[innerRowA * K + innerCol
// 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
    reinterpret_cast<float4 *>(&B[innerRowB * N + innerCol
  __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
    reinterpret_cast<float4 *>(&B[innerRowB * N + innerCol
```

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

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

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.[47]

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[48]

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.[49]
2. The kernel implementation was correct for the ~400 different hyperparameter settings that remained.

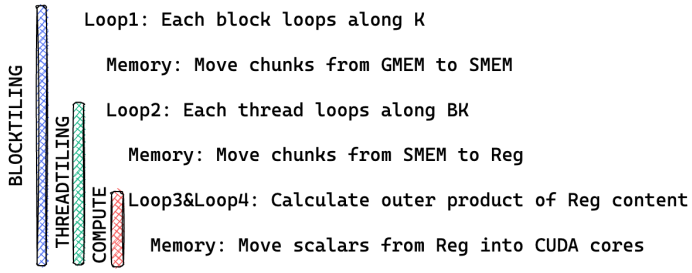
The necessary modifications to the code ended up taking quite some time to implement.

It turns out that the optimal parameters vary quite a bit depending on the GPU model.[50] 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.[51]

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.[52]

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.[53]
- 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:

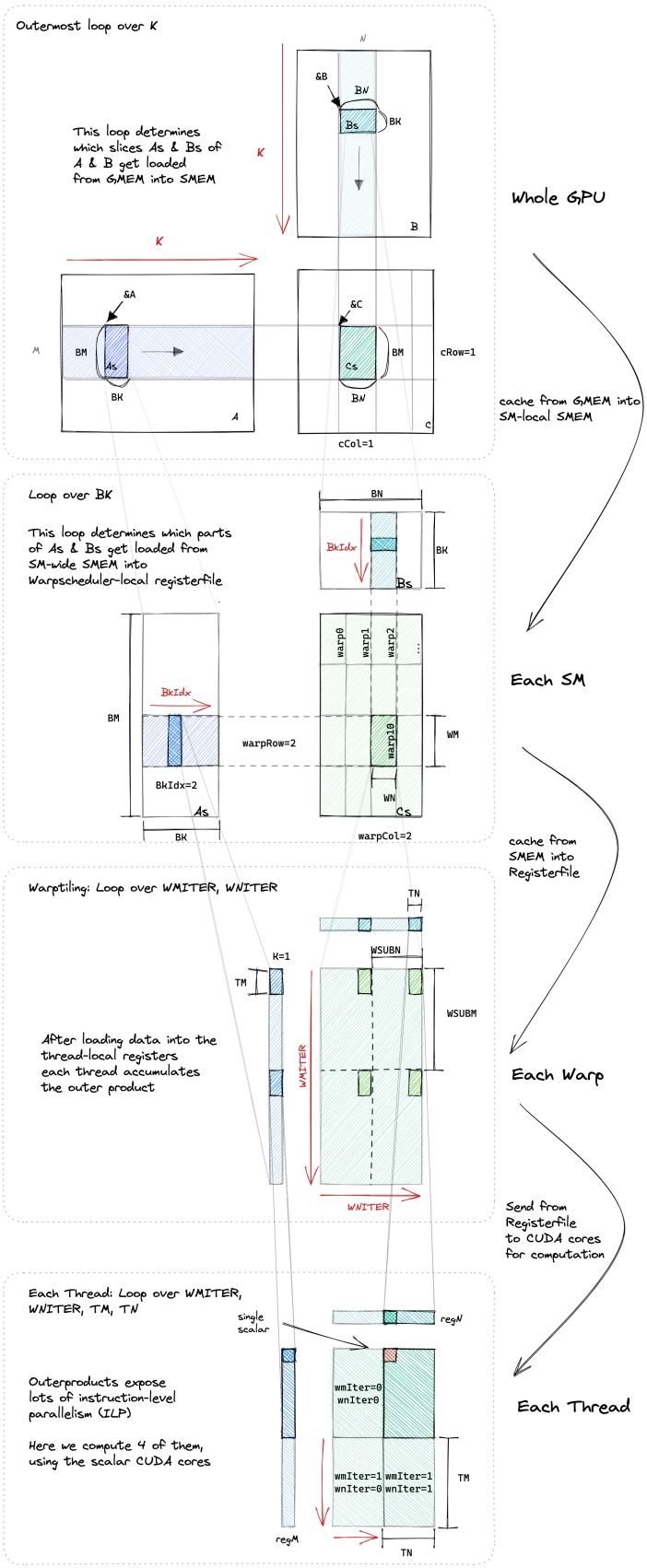
- Blocktiling: Different blocks can execute in parallel on different SMs.
- Warptiling: Different warps can execute in parallel on different warp schedulers, and concurrently on the same warp scheduler.
- Threadtiling: (a very limited amount of) instructions can execute in parallel on the same CUDA cores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

The warptiling looks like this in the CUDA code:[54]

```
// dotIdx loops over contents of SMEM
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
  // populate registers for this thread's part of the warp
  for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRow
    for (uint i = 0; i < TM; ++i) {
      regM[wSubRowIdx * TM + i] =
          As[(dotIdx * BM) + warpRow * WM + wSubRowIdx * W
             threadRowInWarp * TM + i];
    }
  }
  for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubCol
    for (uint i = 0; i < TN; ++i) {
      regN[wSubColIdx * TN + i] =
          Bs[(dotIdx * BN) + warpCol * WN + wSubColIdx * W
             threadColInWarp * TN + i];
    }
  }

  // execute warptile matmul. Later this will map well to
  // warp-wide matrix instructions, executed on tensor cor
  for (uint wSubRowIdx = 0; wSubRowIdx < WMITER; ++wSubRow
```

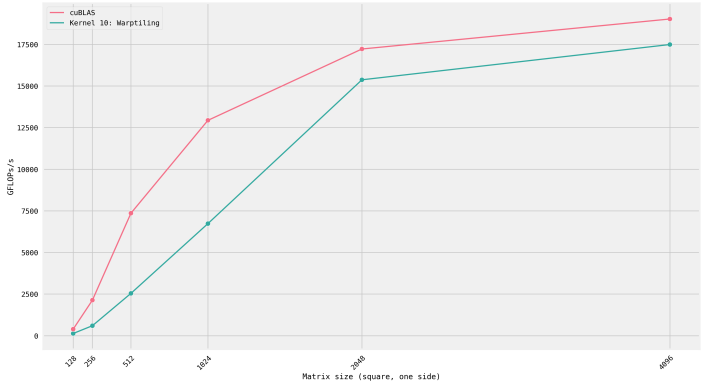
```
    for (uint wSubColIdx = 0; wSubColIdx < WNITER; ++wSubC
      // calculate per-thread results with register-cache
      for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
        for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
          threadResults[(wSubRowIdx * TM + resIdxM) * (WNI
                        (wSubColIdx * TN) + resIdxN] +=
              regM[wSubRowIdx * 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.[55] Each warp will compute a chunk of size `(WSUBN * WNITER) x (WSUBM * WMITER)`. Each thread computes `WNITER * WMITER` many chunks of size `TM*TN`.


Outermost loop over K
This loop determines which slices As & Bs of A & B get loaded from GMEM into SMEM
N
&B
BN
Bs
BK
K
B
K
&A
M
BM
As
BK
A
&C
Cs
BM
cRow=1
BN
C
cCol=1
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
BN
BkIdx
BK
Bs
BkIdx
BM
warpRow=2
BkIdx=2
As
BK
warp0
warp1
warp2
...
warp18
WM
WN
Cs
warpCol=2
Each SM
cache from SMEM into Registerfile
Warptiling: Loop over WMITER, WNITER
After loading data into the thread-local registers each thread accumulates the outer product
TN
WSUBN
K=1
TM
WSUBM
WMITER
WNITER
Each Warp
Send from Registerfile to CUDA cores for computation
Each Thread: Loop over WMITER, WNITER, TM, TN
Outerproducts expose lots of instruction-level parallelism (ILP)
Here we compute 4 of them, using the scalar CUDA cores
single scalar
regN
wmIter=0 wnIter0
wmIter=1 wnIter=0
wmIter=1 wmIter=1 wnIter=1
TM
regM
TN
Each Thread

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: [56]



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. [57] At runtime, based on the dimensions, cuBLAS will pick which kernel to run.[58] I traced the cuBLAS call and these are the kernels it's calling at each size:[59]

Matrix size	Name	Duration
128	`ampere_sgemm_32x32_sliced1x4_nn`	15.295 µs
256	`ampere_sgemm_64x32_sliced1x4_nn` *followed by* `splitKreduce_kernel`	12.416 µs + 6.912 µs
512	`ampere_sgemm_32x32_sliced1x4_nn`	41.728 µs
1024	`ampere_sgemm_128x64_nn`	165.953 µs
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.[60] 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.[61] 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.[62]

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 ⇒ SMEM, and SMEM ⇒ 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.

If you enjoy kernel work like this you're likely a good fit for the Performance team at Anthropic. Come work with me! The team is headed by Tristan Hume who is the most capable & thoughtful manager I've ever had. We optimize Anthropic's model for GPUs, TPUs and AWS Trainium. Feel free to reach out!

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 Heterogeneuous 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.