A full walk through of the SGEMM implementation

Before getting started I'd like mention that most of this work was derived from detailed study of the cublas Kepler and Maxwell sgemm implementations. I've made modest improvements but most of hard problems have been solved by the excellent engineers at Nvidia and their expert knowledge of the hardware. The goal with this document is to disseminate that knowledge for others to leverage in their own code. I'd also like to link 2 excellent papers on the subject of sgemm: the original MAGMA paper and Junjie Lai's Kepler sgemm paper. This document is basically an extension of Junjie's work, but with the Maxwell architecture and additional assembly level optimizations. Some of the figures in these papers are complementary to the ones below, and may help if you find mine confusing. You should also have a basic understanding of the shared memory matrix multiply example in the Cuda C Programming Guide before you attempt to read any further. And finally you should also refer to the source code as the code examples in this document are over simplified. The comments also have some missing details that are difficult to describe here. The preprocessed and finalized sass are also helpful to look at as you can see how everything unrolls, is mapped to registers, and how the operand reuse flags are distributed.

SGEMM Overview

Using sgemm as an example, this document aims to describe maximizing the computational capabilities of the Maxwell architecture and beyond. Having thousands of computational cores does you no good unless you keep them fed with data. To do that you need to structure the computation to maximize reuse of the data that you pull through the various memory hierarchies. On the GPU these are: the device memory to L2 cache, the L2 to texture cache, the texture cache to registers, registers to shared memory, shared memory to registers, from registers to the instruction operand cache (new with Maxwell), and finally from registers back out to device memory. Each of these data paths has latencies that we'll need to hide with instruction and thread level parallelism (ILP & TLP). Additionally, there can also be banking and coalescing constraints. The sgemm code presented is able to navigate all these constraints and operate within 2% of the theoretical flops of the hardware.

This document will cover two different layouts: 64 threads per block and 256 threads per block. I'll mostly discuss the 64 thread version since the mappings are smaller and simpler. The 256 thread version is more or less the same thing just scaled up 4x. The two versions are optimized for small or large matrices respectively. The smaller 64 thread version can split a matrix up into 4 times as many blocks, useful when SM's are sparsely populated, but at cost of needing double the device memory bandwidth of the 256 thread version. On GM204 hardware this additional bandwidth actually exceeds what is available and so you only want to use it when having more blocks available to fill your SMs outweighs that cost (unless the L2 can hide it). Although, if you have enough parallelized work, using streams to fill your SM's is a better approach.

In both versions we'll use double buffered 8 register blocking to load each of A and B. The double buffering allows us to hide most of the latency of loading from shared memory: we can be calculating off one register block while at the same time loading the next. We choose 8 register blocking as it nicely aligns with using quad vector memory instructions and because we can keep the total register budget under 128. Crossing the 128 register barrier would cut our occupancy from 4 active warps per scheduler down to 3 for the 64 thread version and from 4 down to 2 for the 256 thread version. The 64 thread version isn't as susceptible to the drop and I've actually seen increased performance for some matrix sizes (less L2 and texture cache dilution with fewer blocks per SM), but the 256 thread version does work slightly better with more than 1 extra warp per scheduler to cover latencies. That our performance in either implementation isn't hugely effected by the drop in occupancy speaks to just how well this code hides latencies with ILP.

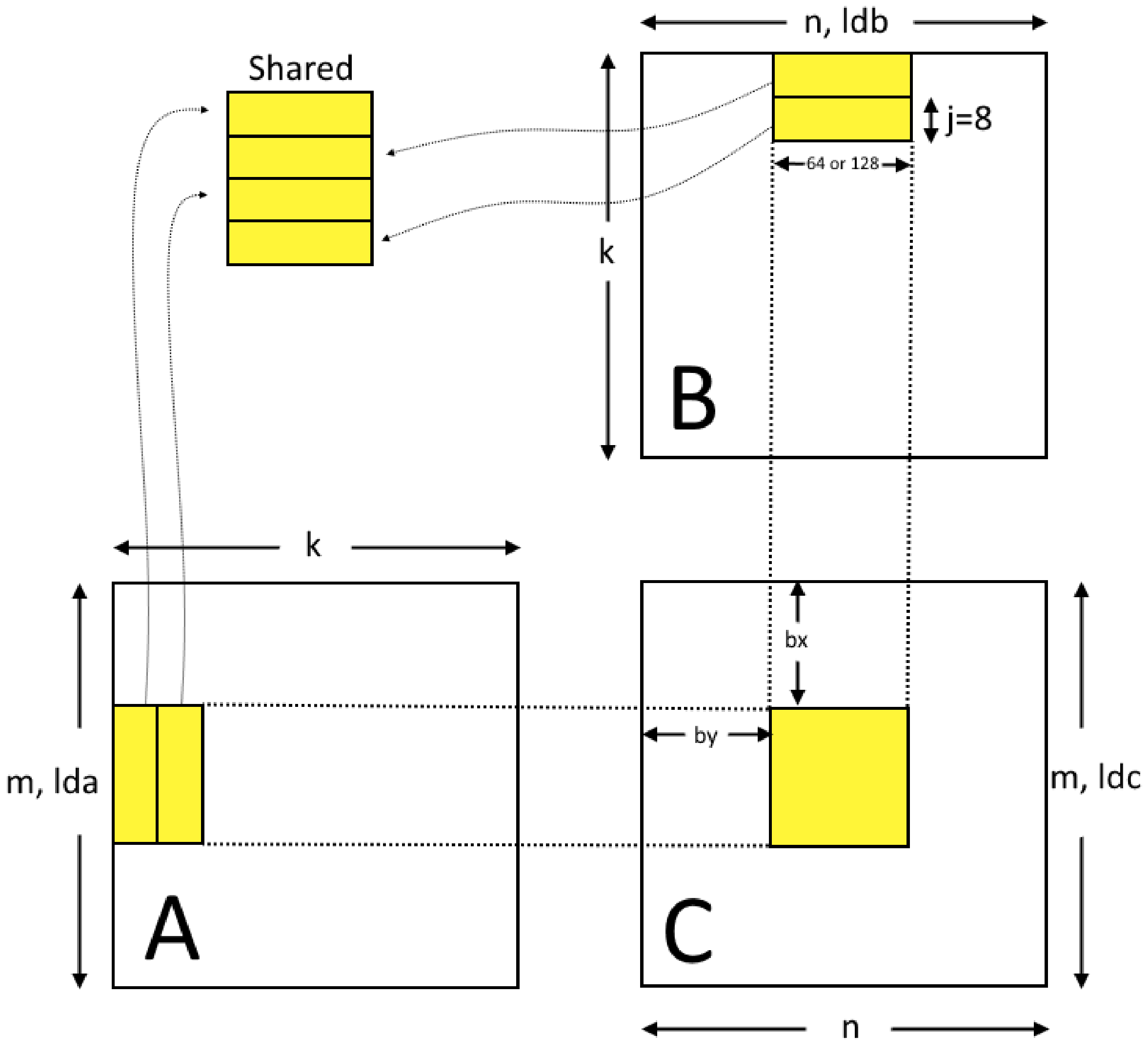
We'll also double buffer our shared memory to allow us to remove one of the bar.syncs we'd ordinarily need in the main loop: instead of waiting for all shared loads to complete before storing the next batch, we'll just start writing to a fresh area of shared while other threads can be reading from the previous area. You'll see below that this will add 3 XORs to the main loop but this is still less expensive than a bar.sync. As for the size of shared memory, this is defined by the width of memory each thread block loads times the main loop unroll factor. We have 64 (or 256) threads each of which will be calculating 8*8 or 64 points of C. All of these points together will be arranged in a square since we're pulling from A and B uniformly. So the width of that square is just the square root of the total number of points. For our two implementations we calculate:

```
64  Threads: sqrt(64  * 8*8) = 64  units wide
256 Threads: sqrt(256 * 8*8) = 128 units wide*

// * The code is not yet fully consistent and I sometimes refer to the 256 thread
version
// as the 128 unit wide version.  For example the kernel function name is
sgemm_kernel_128.
```

Our unroll factor is the number of lines we read at a time from A and B, store/read from shared, and compute. It will be constrained in several ways. We want to be able to hide the latencies of our texture loads by as much computational work as possible. However we don't want the size of the loop to exceed the size of the instruction cache. Doing so would add additional instruction fetch latencies that we'd then need to hide. On Maxwell I've measured this cache to be 8KB. So this means we don't want a loop size of more than 1024 8 byte instructions, where every 4th instruction is a control code. So 768 is the limit of useful instructions. Then there are also instruction alignment considerations so you'll want to be safely under that value as well. In short, using a loop unroll factor of 8 gives us 8 x 64=512 ffma instuctions plus the additional memory and integer arithmetic instructions needed for the loop (about 40). This puts us well under 768. Eight lines per loop also aligns well with the dimensions of our texture memory loads. Finally, 512 FFMAs should be enough to mostly hide the 200+ clock texture load latency.

So we now know what our total shared memory size needs to be: (8 lines per loop) x (block load width) x (word size) x (2 buffers for A) x (2 buffers for B). This is 8192 bytes for 64 threads and 16384 bytes for 256 threads. This sizing will not effect the occupancy, which is dominated by the register counts (which we will keep under 128).

Below is the basic memory layout shared by both implementations. Note that I equate the X dimension with loads from A and aligned along lda while the Y dimension equates with loads from B and aligned along ldb. These are the opposite of how x and y are typically spatially defined as drawn below. Also note that the images for A and C are laid out as the transpose. In retrospect I'd probably change this to have B as the transpose and swapped with A but this is how I originally worked it out. In the next section I'll start discussing the 64 thread version in detail.



SGEMM - 64 Thread Implementation

Loading A and B then Storing to Shared

To load the A and B matrices we use a technique that isn't efficiently possible in cuda c or ptx. We split the threads in half and have each half load one of the matrices. Since we have 64 threads this means each warp loads a Matrix. Conditional loads in cuda are not well optimized since the compiler makes no effort to determine if the loads are occurring uniformly over the warp. For texture loads this is a necessity as the instruction can only handle one texture at a time per warp. So the compiler will add in addition to the texture load a bunch of warp shuffles and branch and sync instructions to make sure this is enforced. It would be nice if Nvidia provided a way to hint that a conditional or predicate is warp uniform (and not just for branches, ie bra.uni).

The main advantage to using this technique is that we only need a single set of track registers to hold our texture load indexes. Inside of the main loop this is a big win as it cuts in half the number of integer addition instructions we need. We take any opportunity we can to improve the ratio of FFMA instructions to non-FFMA instructions.

We also maintain 4 separate track variables to avoid needing to use dependency barriers to increment a single one by ldx*2 after each texture load. A memory instruction does not make a copy of its operand registers when it's issued. This is done likely to save on transistors. Instead you use a barrier to protect against writes to those registers while the memory instruction is still in flight. Waiting at a barrier isn't necessarily bad, as TLP can kick in and cover the latency, but reducing the overall number of latencies that need to be covered can help performance as it increases the chances of having warps available to cover them. We don't have any additional loop IADDS for having 4 track variables, just 3 extra registers, which we can easily afford.

So we're going to be loading through the texture units. By using explicit texture loads instead of global loads with or without the non-coherent cache, we get a couple benefits. One is this makes the code much simpler as we don't need to worry about loading out of bounds. And two, with the same kernel code we can load 8 or 16 bit floats, significantly reducing bandwidth and storage needs. Some applications don't need full 32 bit precision and this is a big win in that case.

Furthermore, we're going to be loading quad vectors. This is the change to the cublas code that makes the biggest difference in terms of performance. Though I can understand why it isn't used in cublas, as it imposes a 4 word alignment constraint on your input data. And cublas has a fixed spec (which isn't to say it couldn't choose a different code path if quad alignment is detected). So, by using quad vectors we need to collapse lda/ldb indexes down by 4. This has the added benefit of allowing us to load matrices with indexes sized up to 31 bits instead of the normal texture load 27 bit limit. One other artifact of the quad loads is that our memory access patterns pull in and consume full cache lines with each fetch. This means we'll only be getting very limited use out of the texture cache (1-2%) and our memory cache performance will be dominated by the L2.

So below is some pseudo code that just shows the texture loads and shared stores from the main loop. You can see from the mapping that's it's pretty straight forward. You'll note that with the STS.128 instruction's we're going to hit bank conflicts, but these are unavoidable and turn out not to impact performance as batching loads and stores into vector instructions is a net win. Plus, I'm not even sure the instruction replay incurred during the bank conflict matters as I think these can probably be dual issued along with FFMA's. In fact all memory operations are dualed issued in our main loop and don't factor into the flops calculation at all (except in a subtle way described in the section on register bank conflicts).

From just this code and the knowledge of the number of clock consuming instructions in our main loop, we can make a rough estimate on an upper bound for the memory bandwidth this kernel will need. For GM204 here is the math:

- Each thread is making 4 vec4 4 byte loads per loop or 64 bytes per thread per loop.
- Down below we'll calculate that each loop consumes about 520 clocks.
- There are 128 threads executed per SM simultaneously.
- There are 16 SMs clocked at 1.216 GHz (boost).
- There are .931 GiB per GB:

64 x 128 x 16 x 1.216 x .931 / 520 = 285 GiB/sec

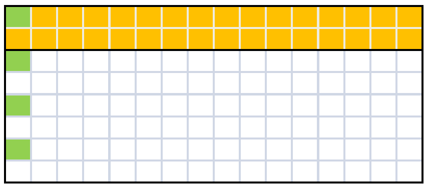
GM204 has 224 GiB/sec available. But some of this device bandwidth wont be needed as the L2 cache will serve some of it. But's it's always good to have headroom on your device bandwidth. Your loads aren't going to be executed in a perfectly uniform manner and the less headroom you have the more chance for stalls when they get bunched up. Though only code running close to theoretical throughput will likely notice these stalls, which our code happens to do.

So you can see that the 64 thread implementation isn't ideal for GM204. For GM107 it is however, and likely as well for the upcoming GM200 with the 384 bit memory bus. Compared to the 256 thread implementation this one uses double the bandwidth, and hence more power. So the bigger version is generally preferred when you have enough data to feed it.

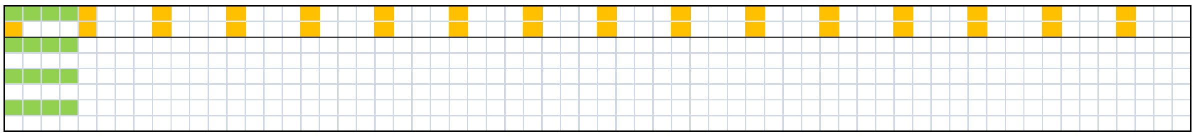
```
tid = threadId.x;
bx  = blockId.x;
by  = blockId.y;

blk = tid >= 32 ? by     : bx;
ldx = tid >= 32 ? ldb/4 : lda/4;
tex = tid >= 32 ? texB  : texA;
```

```
tid2  = (tid >> 4) & 1;
tid15 = tid & 15;

track0 = blk*64/4 + tid15 + (ldx * tid2);
track2 = track0 + ldx*2;
track4 = track0 + ldx*4;
track6 = track0 + ldx*6;

end = track0 + (k-8)*ldx;

writeS = tid15*4*4 + tid2*64*4;
writeS += tid >= 32 ? 2048 : 0;

while (track0 < end)
{
    tex.1d.v4.f32.s32 loadX0, [tex, track0];
    tex.1d.v4.f32.s32 loadX2, [tex, track2];
    tex.1d.v4.f32.s32 loadX4, [tex, track4];
    tex.1d.v4.f32.s32 loadX6, [tex, track6];

    st.shared.v4.f32 [writeS + 4*0*64], loadX0;
    st.shared.v4.f32 [writeS + 4*2*64], loadX2;
    st.shared.v4.f32 [writeS + 4*4*64], loadX4;
    st.shared.v4.f32 [writeS + 4*6*64], loadX6;

    // our loop needs one bar sync after share is loaded
    bar.sync 0;

    // Increment the track variables and swap shared buffers after the sync.
    // We know at this point that these registers are not tied up with any in
flight memory op.
    track0 += ldx*8;
    track2 += ldx*8;
    track4 += ldx*8;
    track6 += ldx*8;
    writeS ^= 4*16*64;

    // Additional loop code omitted for clarity.
}
```

Loading A and B by quad vector texture index:



Storing A and B into the shared address space with quad vectors:



Reading from Shared

Now that shared memory is loaded we switch from using half of our threads to process each of A and B. We need to start combining the values to calculate the dot products that make up the points of C (with each line processed we compute a partial product and sum for all values of C in the block). So each thread is going to read from both A's shared lines and B's shared lines.

Aside from the FFMAs, the shared loads are the real workhorse of this implementation. We double buffer them so the latencies are minimal. But we also want to ensure that we have no bank conflicts because we need this data as fast as it can be delivered. How do you load from shared using quad vectors without bank conflicts? Well, according to the documentation, so long as all the accesses are within 32 words (128 bytes), we're fine. In sgemm this works because we can arrange for different threads to load from the same shared memory location at the same time and use the shared broadcast mechanism. However, it turns out the documentation for Maxwell is incomplete and there are certain patterns that will lead to bank conflicts despite all threads in a warp being inside the same 128 bytes. This was likely done to save chip real estate. So we just need to find a pattern that works.

Inside of 128 bytes we can load 8 16 byte quads. We'll make this the pattern we load from A and B's shared memory. Our shared memory block is 4*64=256 bytes wide so to load the other half we'll unroll that load into an additional instruction strided 32 units apart. We don't need to worry about that one for the purpose of avoiding bank conflicts. These two loads of quad words per matrix form the 8 register blocking that we wanted. By combining these two 1D patterns of loads in 2D gives us the shared memory mapping illustrated below. This pattern also represents where in the C sub matrix each thread's 64 registers are placed (green squares).

Now that we have the basic pattern we need to divide it up into the two warps and then map the thread ids in those warps. The straight forward approach is to load in a simple scanning pattern either down or across. This turns out to produce the mysterious bank conflicts. But if we use a zigzag pattern illustrated by the thread numbers it works. I haven't done an exhaustive search of all load sizes and patterns to see what works and what doesn't, but it would be nice if Nvidia updated their documentation for Maxwell explaining this limitation.

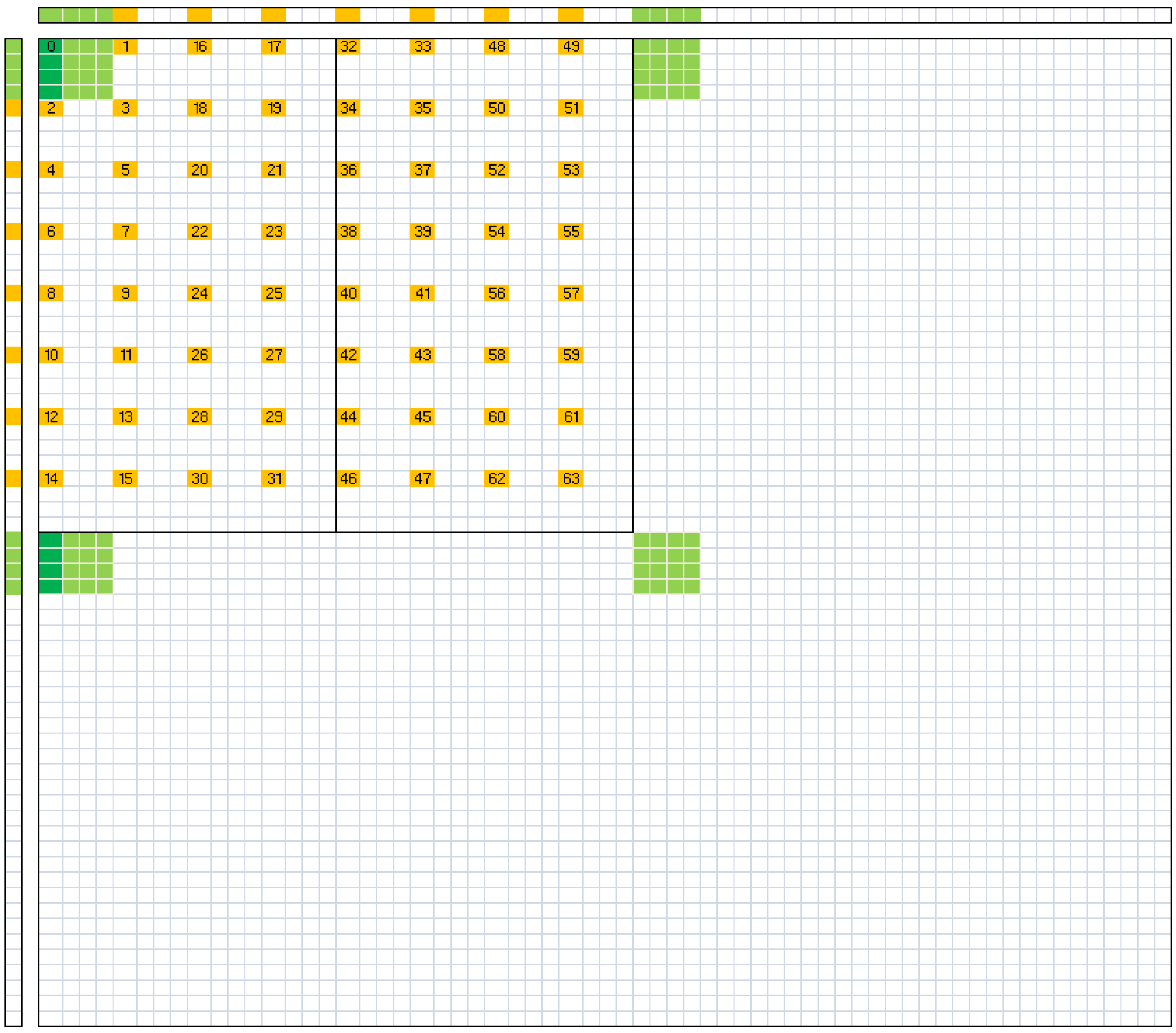
As far as figuring out the logic needed to map the threadId onto the pattern we want (readAs and readBs below) I have a simple technique. I just print out the binary representations of each threadId and the value I want it to map to. When you have the binary visualized this way it's pretty easy to work out which bits you need to keep or toss or shift around to make the mapping work (provided you chose a mapping that's possble).

I should also mention how I mean my illustrations to be interpreted. The yellow squares are threads (or TLP) and the green squares represent the ILP for the first thread. You should be able to imagine taking the pattern of green squares and shifting it on top of each of the yellow squares (preserving the relative position of greens to yellow). This should span the entire memory space, which is the goal of our shared mapping: every point from a line in A needs to be paired with each point from a line from B. The thin black lines represent how the threads are divided up into warps. The darker green squares below are meant to illustrate a step in the warp synchronous shuffling we'll be doing in a bit.

One other note is that cublas uses a more complicated readAs/readBs mapping here that achieves the same effect but does it at a cost of more instructions. This was one of the minor enhancements my code makes over cublas. The more complicated pattern even makes sense if you know ahead of time of this shared loading limitation. But it seems the dumb and straight forward approach turns out to find the simpler solution.

```
readAs = ((tid >> 1) & 7) << 4;
readBs = (((tid & 0x30) >> 3) | (tid & 1)) << 4 + 2048;

while (track0 < end)
{
    // Process each of our 8 lines from shared
    for (j = 0; j < 8; j++)
    {
        // We fetch one line ahead while calculating the current line.
        // Wrap the last line around to the first.
        prefetch = (j + 1) % 8;

        // Use even/odd rows to implement our double buffer.
        if (j & 1)
        {
            ld.shared.v4.f32 j0Ax00, [readAs + 4*(prefetch*64 + 0)];
            ld.shared.v4.f32 j0By00, [readBs + 4*(prefetch*64 + 0)];
            ld.shared.v4.f32 j0Ax32, [readAs + 4*(prefetch*64 + 32)];
            ld.shared.v4.f32 j0By32, [readBs + 4*(prefetch*64 + 32)];
        }
        else
        {
            ld.shared.v4.f32 j1Ax00, [readAs + 4*(prefetch*64 + 0)];
            ld.shared.v4.f32 j1By00, [readBs + 4*(prefetch*64 + 0)];
            ld.shared.v4.f32 j1Ax32, [readAs + 4*(prefetch*64 + 32)];
            ld.shared.v4.f32 j1By32, [readBs + 4*(prefetch*64 + 32)];
        }
    }
    // swap our shared memory buffers after reading out 8 lines
    readAs ^= 4*16*64;
    readBs ^= 4*16*64;

    // Additional loop code omitted for clarity.
}
```

1D readAs (left) and readBs (top) combine in 2D to form the C result sub matrix for this thread block:



Calculating C: Register Banks and Reuse

Now that we have 8 registers each of A and B populated for our threads we can perform the 64 FFMAs that makes up the core of useful work for which this kernel was designed. To be able to calculate this at full speed and at the lowest power we'll need to take several things into consideration. The primary ones being register banks and operand reuse.

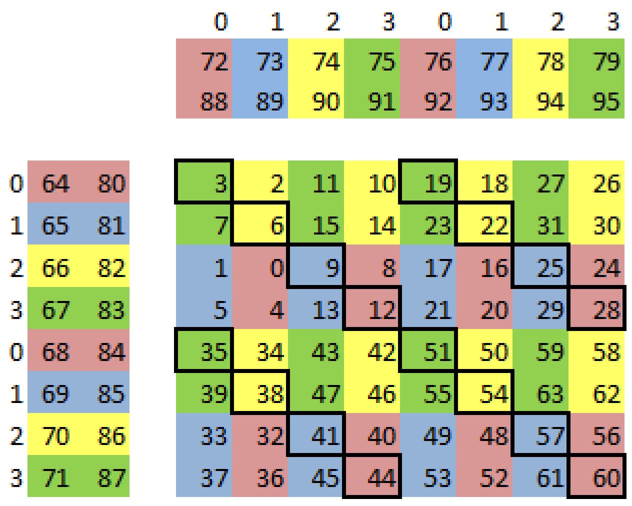
On Maxwell there are 4 register banks, but unlike on Kepler (also with 4 banks) the assignment of banks to numbers is very simple. The Maxwell assignment is just the register number modulo 4. On Kepler it is possible to arrange the 64 FFMA instructions to eliminate all bank conflicts. On Maxwell this is no longer possible. Maxwell, however, provides something to make up for this and at the same time offers the capability to significantly reduce register bank traffic and overall chip power draw. This is the operand reuse cache. The operand reuse cache has 8 bytes of data per source operand slot. An instuction like FFMA has 3 source operand slots. Each time you issue an instruction there is a flag you can use to specify if each of the operands is going to be used again. So the next instruction that uses the same register *in the same operand slot* will not have to go to the register bank to fetch its value.   And with this feature you can see how a register bank conflict can be averted.

So the first step we'll want to take is to minimize the number of bank conflicts that we'll have to hide with operand reuse. To do this we'll need to explicitly choose the registers that we'll be using. This is one of the primary advantages of using maxas as your assembler. ptxas does an ok job of avoiding bank conflicts but it's not perfect and it does an especially bad job when vector instructions are involved (as is heavily the case in this instance). So we'll pick:

- 0-63 to be the C registers
- 64-71 and 80-87 to be the double buffered blocking registers for matrix A

- 72-79 and 88-95 to be the double buffered blocking registers for matrix B

If we arrange this in an 8 by 8 matrix as shown below, we can color each register with its bank index. For the C registers we choose colors different from the corresponding blocking registers. In this way you can see that we can eliminate all the bank conflicts with the C registers and the blocking registers. This leaves the unavoidable 16 bank conflicts with the blocking registers themselves. These are outlined in black:



			0	1	2	3	0	1	2	3
			72	73	74	75	76	77	78	79
			88	89	90	91	92	93	94	95
0	64	80	3	2	11	10	19	18	27	26
1	65	81	7	6	15	14	23	22	31	30
2	66	82	1	0	9	8	17	16	25	24
3	67	83	5	4	13	12	21	20	29	28
0	68	84	35	34	43	42	51	50	59	58
1	69	85	39	38	47	46	55	54	63	62
2	70	86	33	32	41	40	49	48	57	56
3	71	87	37	36	45	44	53	52	61	60

Without the reuse cache, each of these 16 bank conflicts would cause a 1 clock stall in our computation. This should slow our computation down by about 20% (128 clocks added to a 520 clock loop). But if you assemble the sgemm code with the --noreuse flag you'll see performance only drops by a couple hundred Gflops or so. This mystery is solved if you read through Nvidia's patent on operand collectors and particularly if you search for sections involving bank conflicts. It describes a number of ways in which some bank conflicts can be mitigated. It's hard to say how Maxwell is handling it but it probably involves being able to leverage TLP to hide the bank conflict latency. So the operand collectors have some limited ability to mask bank conflicts on their own but they can probably be quickly overwhelmed. By having a longer lasting cache instead of just temporary operand buffers the hardware is far more capable of averting bank conflict stalls. It just needs the assembler to direct it with the reuse flags so it knows ahead of time what registers are worth caching and what to discard as registers are written to.

The tedious task of optimally setting the reuse flags is handled for you by maxas. What's left up to us is to order our instructions in such as way as to maximize the amount of reuse that can be achieved. The simplest ordering would be a basic double nested "for loop" that would traverse across the matrix line by line. This only makes effective use of 4 of our 8 bytes per operand of reuse cache and it doesn't hide all the bank conflicts. If instead your scans go back and forth you can hide all the conflicts and improve register reuse (39% overall). But the pattern that works best is if while going back and forth, you apply a swirl (47% overall reuse). Below is the FFMA instruction ordering listed by the C register number:

```
 1,  0,  2,  3,  5,  4,  6,  7, 33, 32, 34, 35, 37, 36, 38, 39,
45, 44, 46, 47, 41, 40, 42, 43, 13, 12, 14, 15,  9,  8, 10, 11,
17, 16, 18, 19, 21, 20, 22, 23, 49, 48, 50, 51, 53, 52, 54, 55,
61, 60, 62, 63, 57, 56, 58, 59, 29, 28, 30, 31, 25, 24, 26, 27
```

You'll note that the swirl dimension chosen is is spaced by 2 in one of the directions. This spacing has the effect of making the the C register appear in alternating banks. My best guess for the reason for doing this is extremely subtle. Because our memory instructions are interleaved with the FFMAs, and because these instructions don't have a copy of their operand registers, there is a period of about 20 clocks where they can be accessing the register bank. Our C registers are being constantly dirtied and hence cannot be reused, so we're always pulling them from the register bank. So it's primarily these registers that can hit delayed bank conflicts with our memory load and store instructions. It's probably not possible to completely design around these bank conflicts but you can lessen their impact. By alternating the C register bank on each instruction we ensure a bank conflict can only last at most one clock. I've run several benchmarks to test this hypothesis and it seems to bear out. And one last note: the other advantage of using all quad vector loads and stores (besides they're being more efficient) is that these reduce the number of memory instructions needed and hence the chances for delayed register bank conflicts.

Given that we know there can be delayed bank conflicts with the memory operand registers, it's worth experimenting with choosing different banks for those operands. With maxas we have full control over the register mappings and you'll note in the source code that we chose very specific banks for track0-3, tex, readAs, readBs and writeS. Each of these bank selections was tested to maximize the flops performance of the kernel. This is a level of optimization that I'm not sure the cublas implementation makes. One mistake I know it makes is that for the first FFMA it chooses a C register (3) with a blocking register bank conflict. This prevents the ability for that conflict to be hidden by the reuse cache because there is no prior instruction to load at least one of the operands into the cache. On GM204 this bug is costing cublas sgemm 28 Gflops of performance.

The final consideration to make with our FFMA's is exactly how to interleave them with all the memory operations described above. To see what I'm talking about here check out the preprocessed versions of the source code. The double buffered shared loads we want as early as possible in order to cover their latency so we'll start dual issuing them with the very first FFMA. We'll space them by two instructions as the memory unit seems to work optimally at half throughput. The texture loads we'll place in the middle of two sets of shared loads. This is done to not overwhelm the memory units with instructions. For the 64 thread implementation we even split the four loads up into two sets and put them in different FFMA blocks. The shared store instructions we place as far down as possible to give the texure loads a chance to load their operands. We cant place them in the last FFMA block, as that one is where we start loading the blocking registers for the next loop iteration. We place the BAR.SYNC right between the 7th and 8th FFMA blocks.

All of these position decisions were heavily tested to be optimal. I should note that making these fine grained ordering and positioning choices is not possible with ptxas. In fact, ptxas has a tendency to optimize away our shared double buffered loading scheme. Between choosing our register banks, optimizing instruction ordering for operand reuse, and placing our memory instructions exactly where we want them is what can take an implementation that runs at 70% of theoretical performance and turns it into one that runs at 98%.

Speaking of theoretical performance we can now calculate that value. In the main loop, we need to take the total number of FFMA instructions (512) and divide that by the total sum of all the clock consuming instructions (dual issues are free). Here's the breakdown:

Op	Count
FFMA	512
LDS	32 dual issued
STS	4 dual issued
TLD	4 dual issued
IADD	4
XOR	3
SETP	1
BAR	1 dual issued
BRA	1 dual issued

That's a total of 520 clock consuming instructions. So the upper bound on the kernel efficiency is 512/520 or 98.5%. For large matrices our actual performance is very close to that value. It's 98% on GM107, but 96% on GM204. I'm not sure where we lose our performance on GM204 but looking at nvprof stats from synthetic benchmarks with no memory operations seems to imply that it's not within the kernel. Maybe a driver update will fix this, or maybe it's just some internal cost of having many more SMs.

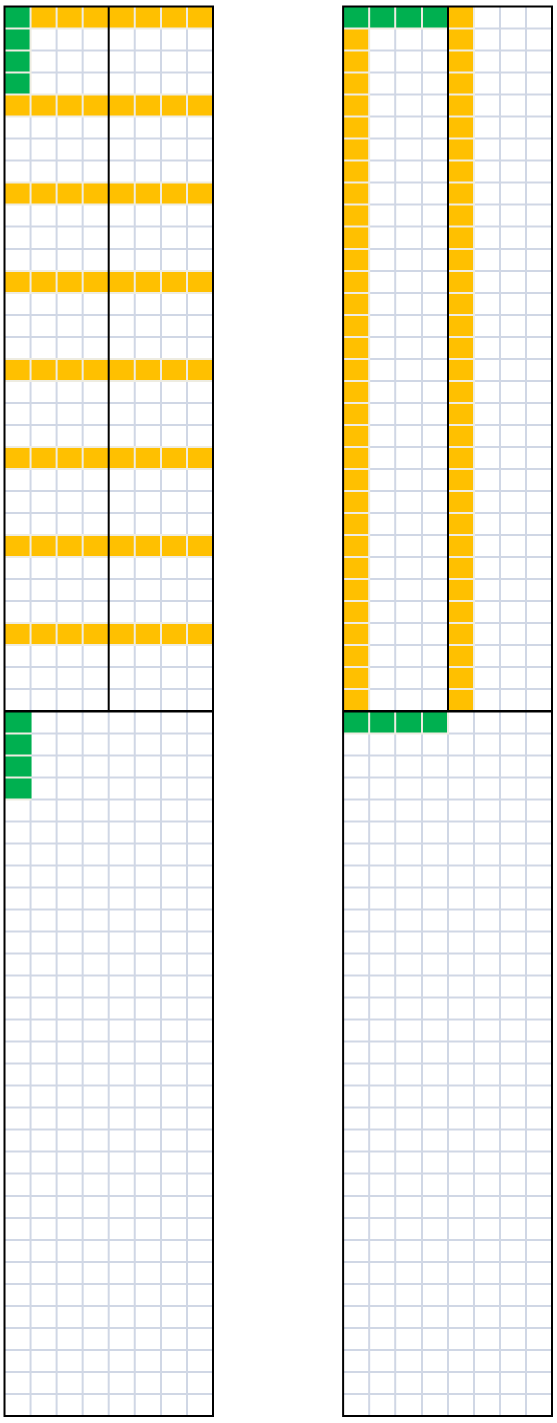
The calculation for the 256 thread version is only changed by needing 2 fewer IADDs (it also has 2 fewer TLD and STS instructions but those are dual issued), so 512/518 or 98.8%.

Warp Synchronous Shuffle Mapping

At the end of the loop our C sub matrix for the thread block is now calculated. So it's now time to store the result back to global memory. Because we used quad vector loads from shared our C values are somewhat bunched up and not at all optimal for coalesced writes to global. We could just write the data out as it is, but we can do a little better. By shifting our C registers around between threads of the same warp using shared memory, we can reorganize it for coalesced writes. You might think the warp shuffle instruction would work best here, but we need to swap different registers from different threads and hence it's not suitable for this purpose.

We're going to split the shuffle up into 8 chunks. The darker green line on the above shared memory mapping represents the first chunk. The other 7 will be the subsequent vertical selections of the C registers. So each thread stores 8 registers at a time in shared memory, then immediately reads back out 8 more registers but arranged in a way such that a remapping of our thread ids can store that data to global in a coalesced pattern. So to store to shared we need to reuse our original shared memory mapping and in one of the dimensions collapse it down from 4 strided units to one. The thread id mapping for reading it out will be 32 values with a stride of 1 unit.

```
tid31 = tid & 31;
tid32 = tid & 32;

// Remove the high bits if present from the last loop's xor.
// Also remove the 2048 added onto readBs.
readAs &= 0x7ff;
readBs &= 0x7ff;

// Write to shared using almost the same shared mapping as before but collapse
readBs down to stride one.
writeCs = (readBs / 4) * 64 + readAs;

// Read out with a mapping amenable to coalesced global writes
readCs = ((tid32 << 3) + tid31) << 2;
```

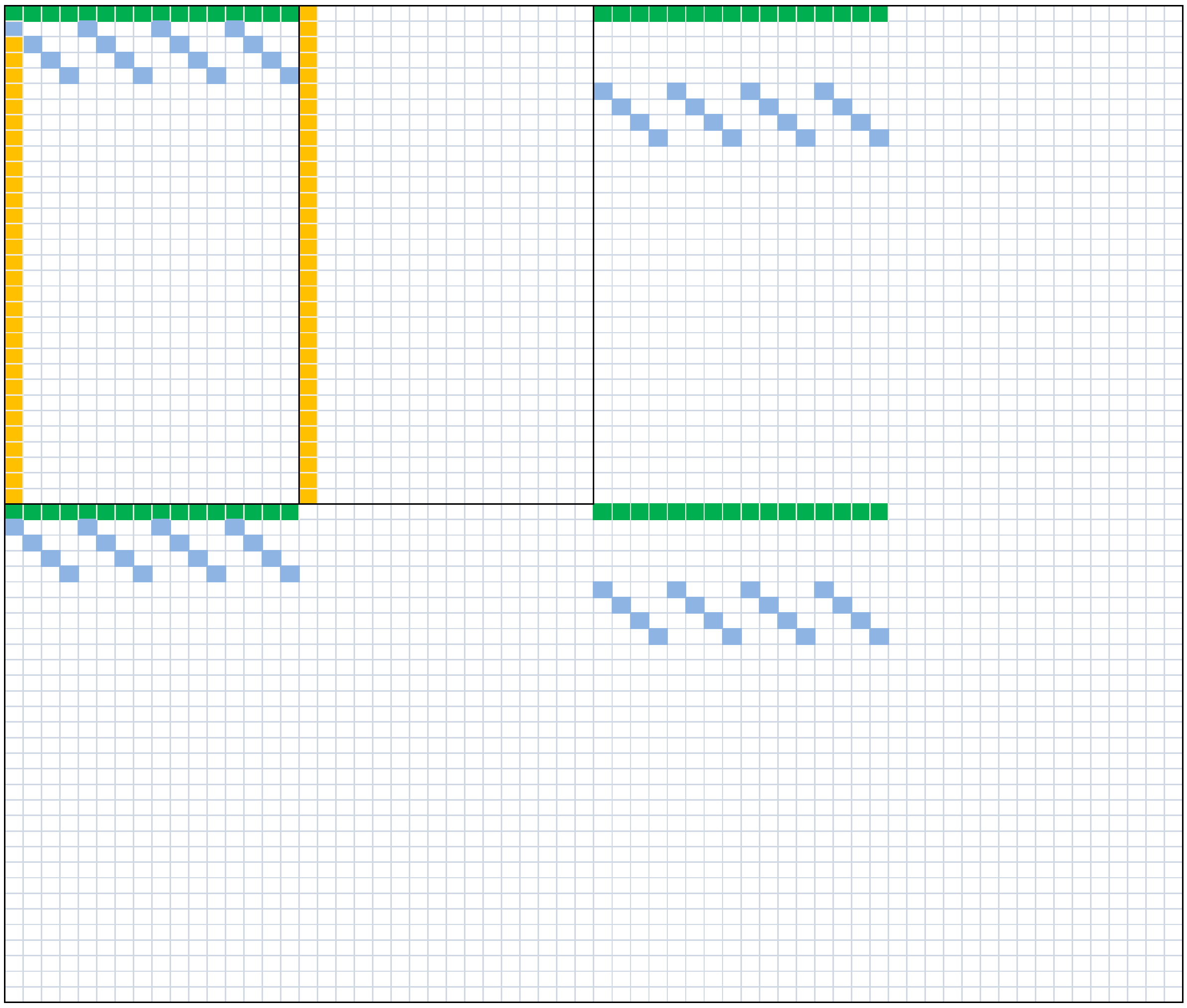


Warp Shuffling and Coalesced Storing to Global

Armed with the above mapping we can now output the C values. Note that we don't need a bar.sync prior to writing to shared memory since that's already been done on our final loop. Also note that since we're not sharing data between warps we don't need to sync between writing and reading in our shared memory shuffle. The read from readCs will only happen after the store to writeCs is complete. Note that the added shared memory latencies here can mostly be hidden with TLP and the net clocks added for having the warp synchronous shuffle is only a dozen or so. This is far less than would be required if using uncoalesced stores to global.

```
ldc4 = ldc * 4;

cx = bx*64 + tid31;
cy = by*64 + (tid32 >> 1);

Cy00 = (cy*ldc + cx) * 4 + C;
Cy04 = Cy00 + ldc4 * 4;
Cy08 = Cy00 + ldc4 * 8;
Cy12 = Cy00 + ldc4 * 12;

foreach copy vertical line of 8 registers from C into .v4.f32 cs0 and cs4
{
    // Feed the 8 registers through the warp shuffle before storing to global
    st.shared.v4.f32 [writeCs + 4*00], cs0;
    st.shared.v4.f32 [writeCs + 4*32], cs4;

    ld.shared.f32 cs0, [readCs + 4*(0*64 + 00)];
    ld.shared.f32 cs1, [readCs + 4*(0*64 + 32)];
    ld.shared.f32 cs2, [readCs + 4*(1*64 + 00)];
    ld.shared.f32 cs3, [readCs + 4*(1*64 + 32)];
    ld.shared.f32 cs4, [readCs + 4*(2*64 + 00)];
    ld.shared.f32 cs5, [readCs + 4*(2*64 + 32)];
    ld.shared.f32 cs6, [readCs + 4*(3*64 + 00)];
    ld.shared.f32 cs7, [readCs + 4*(3*64 + 32)];

    st.global.f32 [Cy00 + 4*00], cs0;
    st.global.f32 [Cy00 + 4*32], cs1;
    st.global.f32 [Cy04 + 4*00], cs2;
    st.global.f32 [Cy04 + 4*32], cs3;
    st.global.f32 [Cy08 + 4*00], cs4;
    st.global.f32 [Cy08 + 4*32], cs5;
    st.global.f32 [Cy12 + 4*00], cs6;
    st.global.f32 [Cy12 + 4*32], cs7;

    Cy00 += ldc4;
    Cy04 += ldc4;
    Cy08 += ldc4;
    Cy12 += ldc4;

    // After processing forth set shift over to the stride 32 registers
    if (4th iteration)
    {
        Cy00 += ldc4 * 28;
        Cy04 += ldc4 * 28;
        Cy08 += ldc4 * 28;
        Cy12 += ldc4 * 28;
    }
}
```

In the diagram below, the blue squares represent how the green lines are constructed from the 8 states of the Cy00, Cy04, Cy08 and Cy12 matrix C offsets. The portion of their vertical placement that isn't stride 32 is a mapping into the loop iteration and not a spatial position.



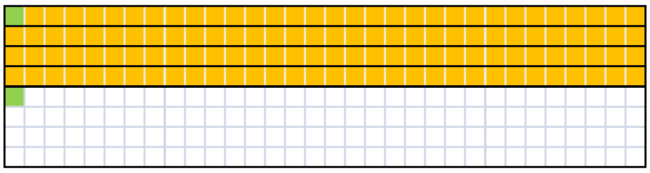
Whew... so that's it at a high level. There are even lower level details covered in the code comments, particularly details on how to synchronize the memory accesses with computation. The comments are only found in the 256 thread version. Speaking of which, below I show how having four times as many threads alters the mappings.

SGEMM - 256 Thread Implementation

Loading A and B

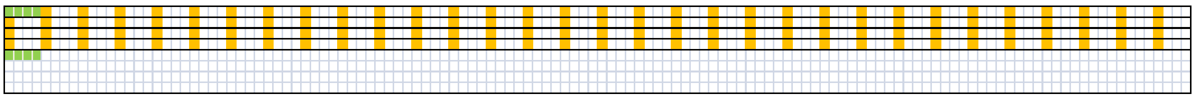
```
tid = threadId.x;
blk = tid >= 128 ? blockId.y : blockId.x;
ldx = tid >= 128 ? ldb/4     : lda/4;
tex = tid >= 128 ? texB      : texA;
```

```
tid4   = (tid >> 5) & 3
tid31  = tid & 31
tid96  = tid & 96
tid128 = tid & 128

track0 = blk*128/4 + tid31 + (ldx * tid4)
track4 = track0 + ldx*4;

end = track0 + (k-8)*ldx;
```



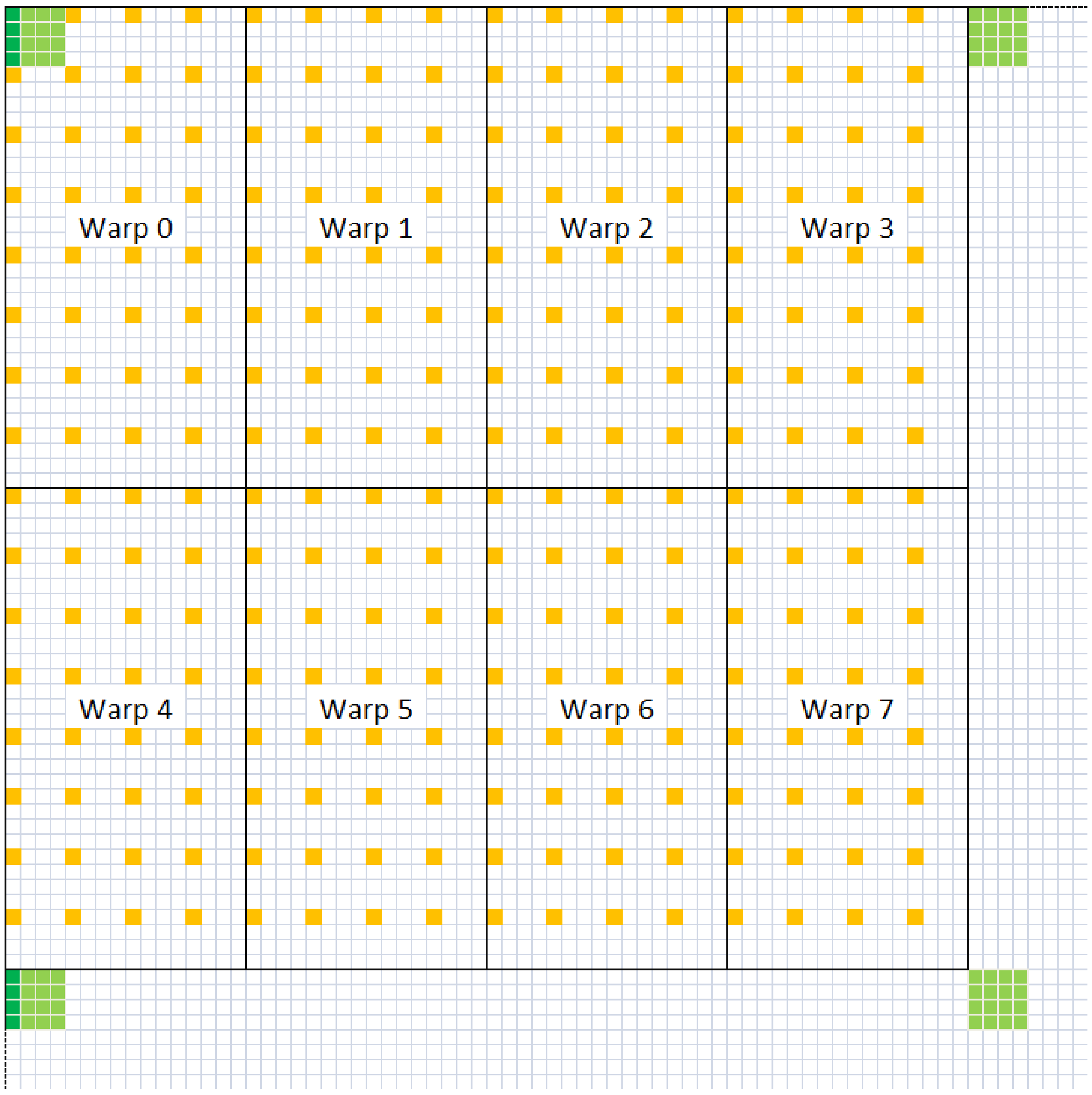
Storing to Shared

```
writeS  = tid31*4*4 + tid4*128*4;
writeS += tid >= 128 ? 4096 : 0;
```



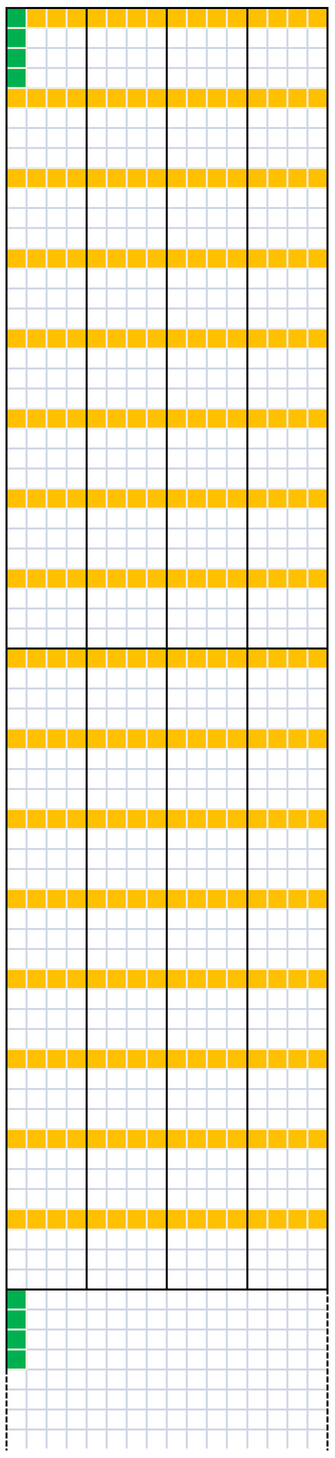
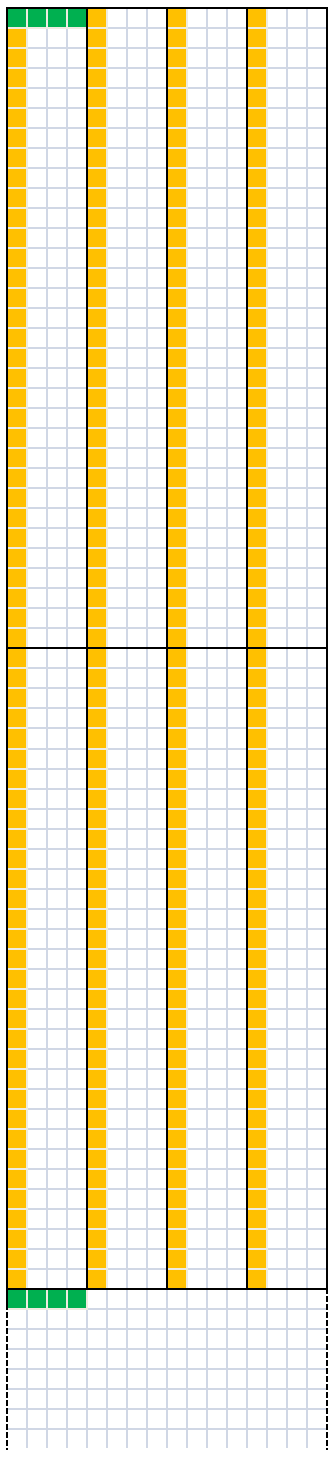
Reading from Shared

```
readAs = ((tid128 >> 4) | ((tid >> 1) & 7)) << 4;
readBs  = (((tid & 0x70) >> 3) | (tid & 1)) << 4 + 4096;
```



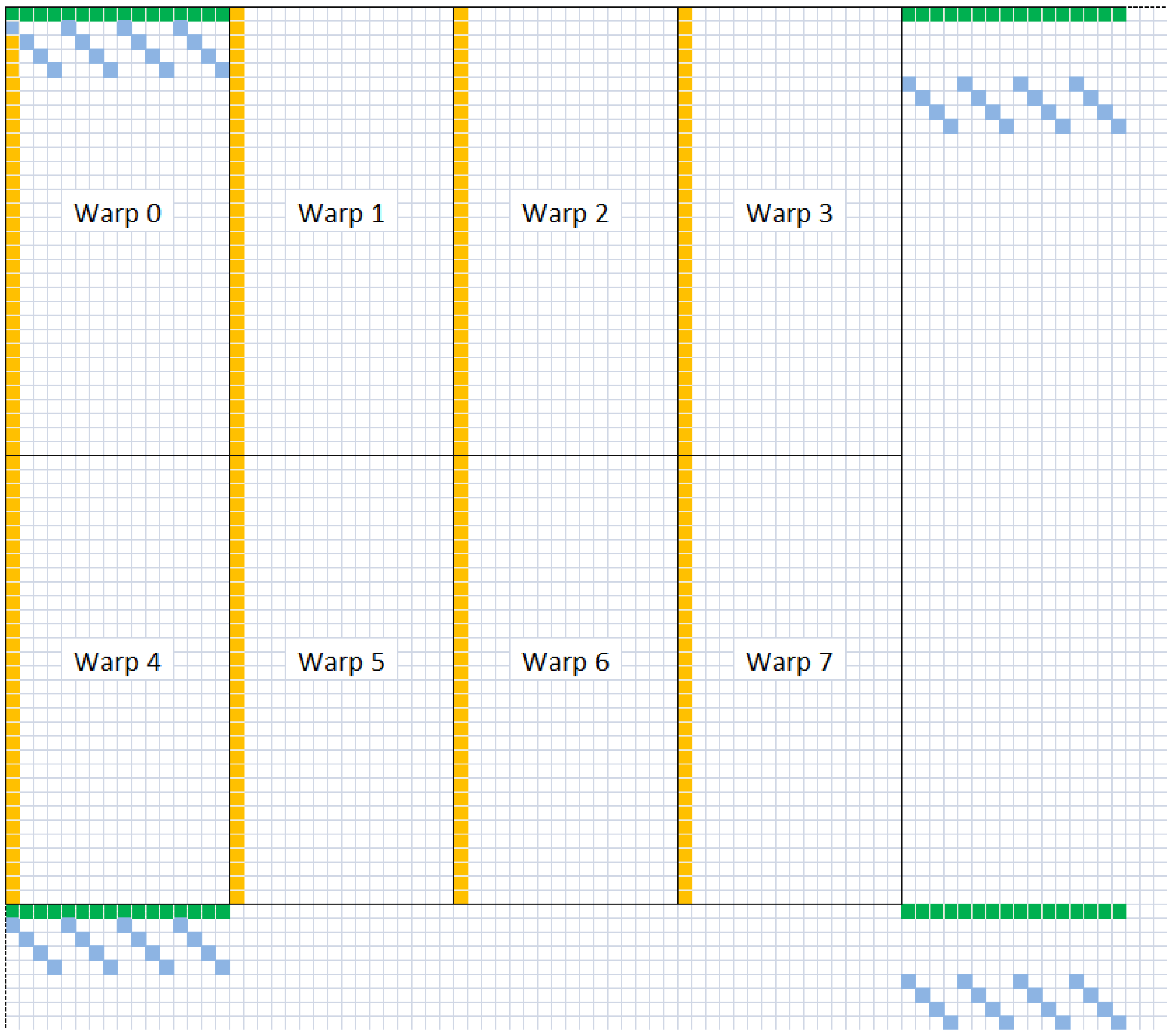
Warp Synchronous Shuffle

```
readAs &= 0xfff;
readBs &= 0xfff;

writeCs = (readBs / 4) * 128 + readAs;

readCs = ((tid96 << 4) | tid31 | (tid128 >> 2)) << 2;
```

Storing to Global

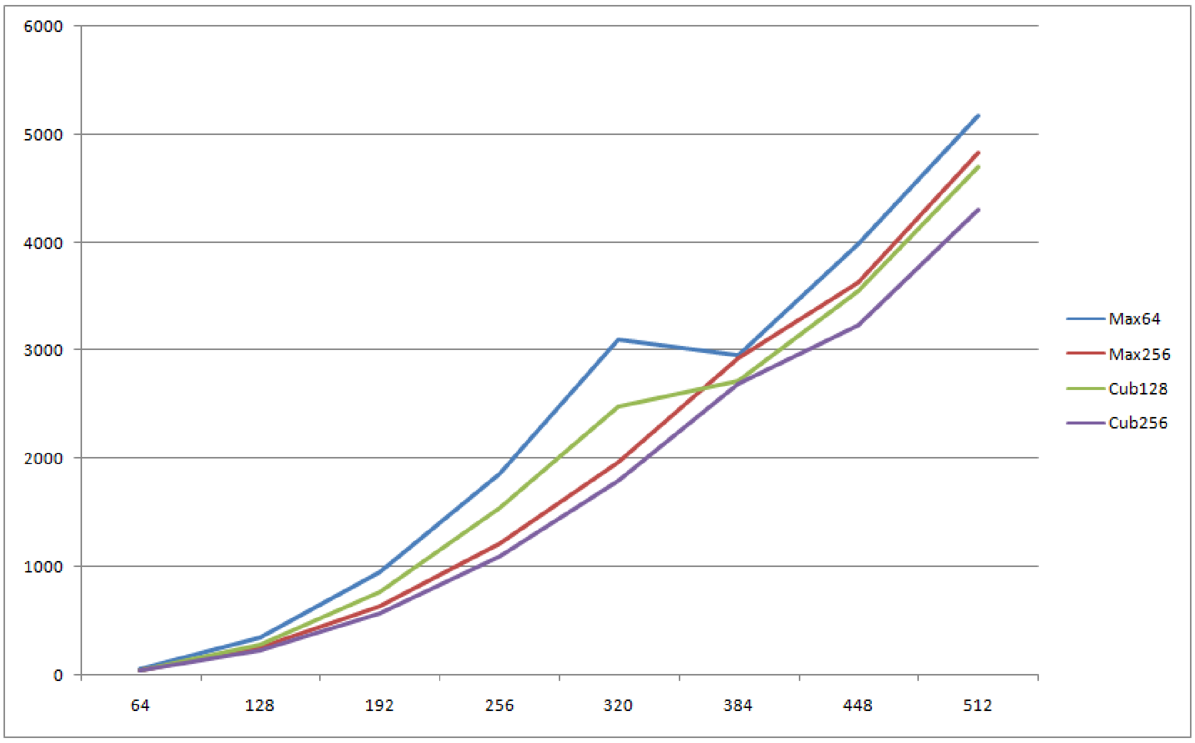
```
ldc4 = ldc * 4;

cx = bx*128 + tid31 | (tid128 >> 2);
cy = by*128 + (tid96 >> 1);

cy00 = (cy*ldc + cx) * 4 + C;
cy04 = cy00 + ldc4*4;
cy08 = cy00 + ldc4*8;
cy12 = cy00 + ldc4*12;
```



BenchMarks

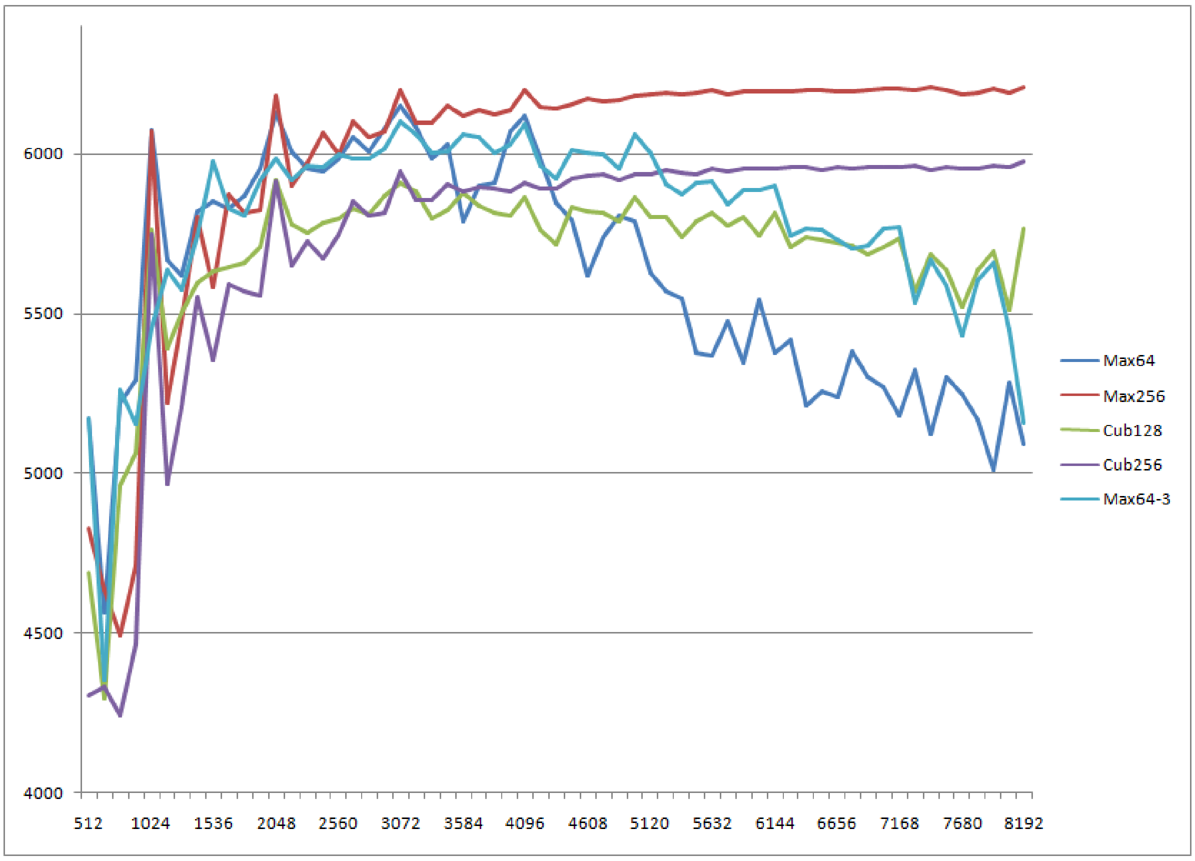
GPU specs: GM204 clocked at 1607 sustained. I've eliminated the 1620 boost clock scores. Memory clocked at 8000MHz.

Below we have the GFlops values for small matrices. Here is where the 64 thread version shines. It's able to break up the work into 4 times as many blocks as the 256 thread version and can balance the load better over 16 SMs. At these sizes the L2 is more than able to cover for the increased bandwidth needs of this implementation. Though it does run about 5-10% higher TDP. Our implementations are also performing as much as 25% over cublas. One interesting note about the cublas 128 version: it is actually a hybrid between the 256 and 64 thread versions. It loads matrix A at 128 units wide and matrix B at 64 units wide. It has all threads loading both matrices and so has double the IADDs in the main loop. It is also memory bound but much less so than our 64 thread version.

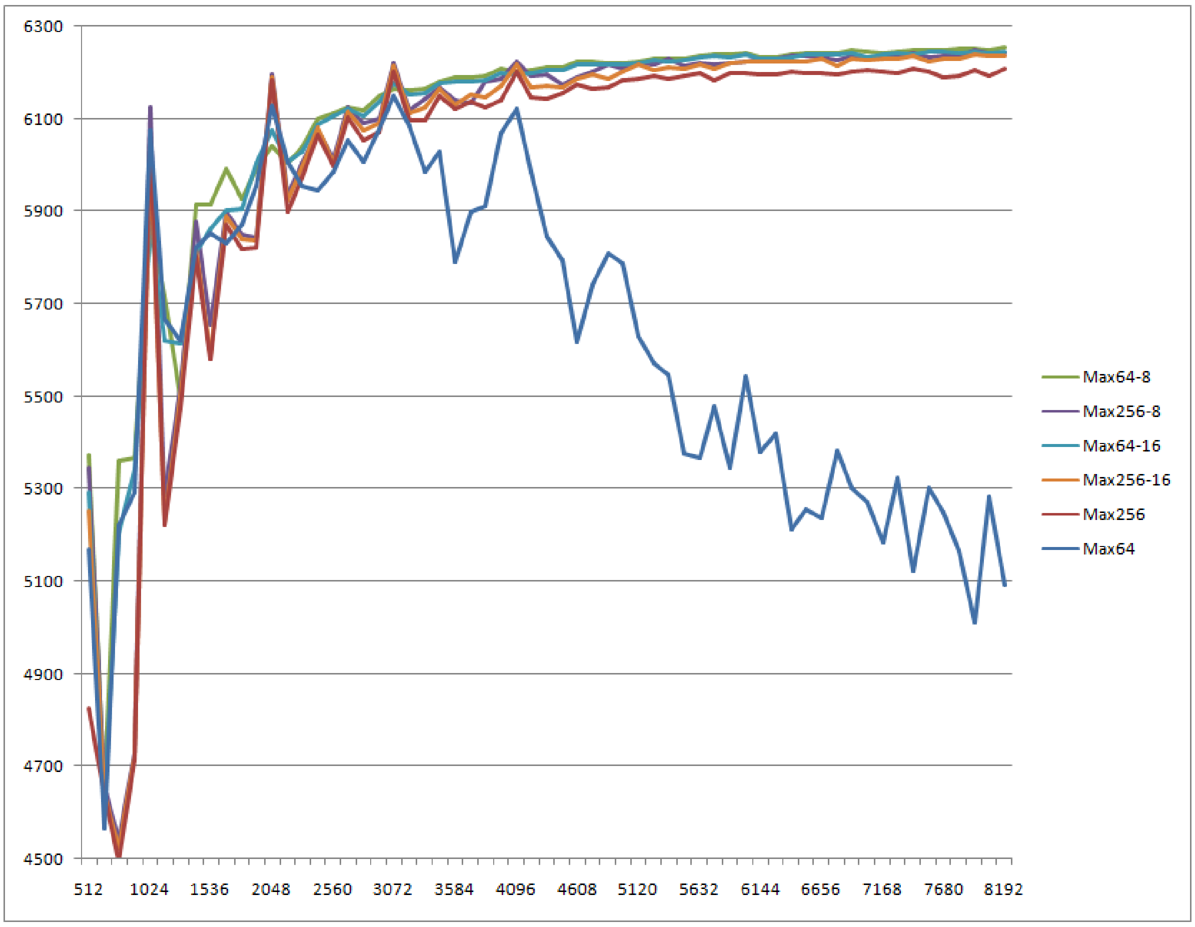


Next we look at large Matrices. Here is where the 256 thread versions overtake the 64 and 128 thread versions. You can see that as the matrices get larger and L2 cache gets more diluted they're able to hold steady. This is because the total bandwidth required is less than device memory has available. This is not the case with the 64 and 128 thread versions. At the 4096 size we're beating cublas by 4.8%.

I've also added one additional plot of the 64 thread version but with an occupancy of 3 warps per scheduler instead of 4 (or 25% down to 18.75%). This was done by bumping the register count past 128. You can see the performance actually increases in the bandwidth starved regimes. This is explained by significantly better L2 cache performance (+6% hit rate) when processing 2 fewer blocks per SM. So if you have a kernel that has enough ILP and is bandwidth bound, you may want to experiment with *lower* occupancy. Unmasked latencies start appearing and performance drops if we lower occupancy any more for this or the 256 thread version (which can't run at 3 warps per scheduler).



Finally we compare performance just within our own implementations and play with different size normalized floats. This shows that even our 256 thread version is a little bit memory bound. Surprisingly, the 64 thread version is the overall speed winner here. The extra 1% in speed is explained by the 1% better L2 cache performance. The memory accesses overlap more for the 64 thread version and the L2 has a better time caching those. Although, the 256 version wins overall for flops/watt. Power levels drop by at least 10% for these smaller precision floats.



Failed Experiments

Lastly, I include a section on things I tried but ended up not panning out.

Double Buffered Texture Loads: The 256 thread version has enough spare registers (10) to double buffer the texture loads. I set an additional predicate in the main loop that tracks even/odd rows. Then I can load and store separate batches on alternating rows, doubling the effective size of the loop. This slightly lowered the performance because of the additional PSETP instruction. It turns out that we're 100% covering the latencies of the texture loads and increasing the gap between loads and stores had no effect. However, this technique could work well for kernels with a smaller unroll factor.

Block Remapping: The block scheduler doles out the available blocks to each SM in block id order. However, in each block, we don't have to use the block id we're given. We can remap them if we like. We'd do this because processing the blocks in a different order might have an effect on L2 performance. For sgemm, block id ordering gives us long rectangles of matrix C that are processed concurrently on all the SMs. If instead we process more even square sections we can can decrease the overall amount of A and B we need to load from memory for each batch of blocks. So I gave this a try and it turned out to hurt L2 performance. It seems L2 likes having that concentrated narrow side of the rectangle which is heavily accessed, coupled with the longer side witch is more dilutely accessed. Trying to even things out confused the L2 into trying to cache too many things because nothing stood out as being more frequently accessed than anything else. So it didn't work out here, but it's a good bit of information to have if you're trying to tweak better L2 cache performance for your kernel, and block remapping might something to try.

-- Scott