Matrix Multiplication: Parallel Computation via GPGPU Programming

Teofilo Erin Zosa IV
Department of Computer Science and Engineering
The University of California, San Diego
La Jolla, CA 92092
PID: A53242430
tzosa@ucsd.edu

Abstract

In this assignment we were tasked with optimizing square matrix operations on a single GPU with respect to Gigaflops per second (Gflops/s), with the goal of achieving roughly 360 Gflops/s. This program achieved a peak of 480 Gflops/s via the use of shared memory, increased instruction-level parallelism, and taking advantage of various characteristics of the benchmarking GPU.

1 Terminology

In this paper:

1. A will refer to the first matrix
2. B will refer to the second matrix
3. AB will refer to the matrix that results as the product of A & B.
4. C will refer to the initially empty matrix that will contain AB at end of the matrix multiplication routine.

2 Method

2.1 Benchmarking Machine: UCSD Sorken Cluster

Performance metrics were obtained on the UCSD Sorken computing cluster, whose computing units are Dell PowerEdge 1950 servers with two octa-core Intel Xeon E5-2640 v3 processors and two Nvidia Tesla K80 Accelerators which in turn contain two Tesla GK210 GPUs. Aside from initial memory allocation, device setup, the call to the CUDA kernel, and host CPU verifications, all operations were run on a single GK210 GPU.

2.2 Benchmarking Machine Characteristics: CPU

Each Intel Xeon E5-2640 v3 processor runs at a clock speed of 2.60 GHz and has a shared 20 MB unified L3 cache (20-way set associative), eight 256KB unified L2 caches (8-way set associative) and eight 32KB L1i caches (8-way set associative) in addition to eight 32KB L1d caches (8-way set associative). All caches have a line size of 64 bytes.World [2017]

2.3 Benchmarking Machine Characteristics: GPU

Each Nvidia Tesla GK210 GPU has a compute capability of 3.7, contains 2496 processor cores (divided amongst 13 streaming multiprocessors, a.k.a., SMXs) with a base core clock speed of 560 MHz (boost clock speed of 562 MHz to 875 MHz) and 12 GB of 384-bit GDDR5 SDRAM with a memory clock speed of 2.5 GHz. (240 GB/s memory bandwidth)NVIDIA [2015], NVIDIA Corporation [2014].

2.4 Program Optimizations

2.4.1 Language-level Optimizations

The program was written in CUDA-C and all decomposable operations inside 'matMul' were inlined to optimize for performance. The local variables that were reused in the calculation of the Matrix AB (i.e. the element(s) of AB that were being updated) were register variables and their values were only written to C once they were fully computed. All loops were prefixed with the 'unroll' pragma (i.e. '#pragma unroll') so that the compiler could unroll the loops. In addition, a majority of the index calculation for the shared memory tiles of A and B was moved outside the loop that loaded the shared memory tiles so that, at each loop iteration, the index was incremented by a pre-calculated offset. In this way, the multiply-adds required to calculate elements of AB were the only floating point operations in the function.

2.5 Thread Block Size and Grid Size

Square thread block sizes of 8, 16, and 32 were tested, with the final program implementation using 32x32 thread blocks. Grid dimensions were determined by dividing the square matrix dimension by the corresponding thread block dimension and rounding that value up. When calculating multiple values of AB in parallel, the grid dimension was adjusted accordingly so that minimum number of thread blocks to compute the elements of AB were instantiated. To accommodate the case where thread blocks had smaller dimensions than shared memory tiles (explained in section 2.6.1), the maximum tile indexes were calculated by taking the minimum of the thread block size in that dimensions and the statically declared tile dimension. Though this makes the program more robust, for the sake of performance, the final version makes the assumption that the kernel will be called with thread blocks of the same dimensions as the tile size, so this min is removed. Of note is the fact that, even in the robust version, rectangular tile sizes that exceed the shared memory tiles in at least one dimension will result in undefined behavior.

2.6 Memory Accesses

As performance is a function of computation, which is in turn a function of operand availability, it is important to structure computation in a way that minimizes the accesses to farther components of the memory hierarchy. In addition to keeping oft-computed values as close to the processor as possible, we can also ensure that computational units have their set of operands fit into fast memory as much as possible. This will ensure that the computational unit causes the least amount of fetches from slow memory. On GPUs, this often means minimizing frequent, low-bandwidth host-to-device memory transfers, instead favoring infrequent, high-bandwidth memory transfers. Once on the GPU, accesses to slow and fast memory are also structured in a way that takes into account different types of memory access patterns.

2.6.1 Shared Memory

On GPUs, the memory hierarchy consists of (from slowest to fastest): a global memory residing in DRAM, an L2 cache shared between all SMXs, shared memory and a read-only data cache shared by all threads in a thread block, and a thread-local L1 cache, with the latter three memory subsets having roughly equal access latency. For GPU matrix multiplication, a common memory access optimization is to fetch operands from the GPU's global memory into a thread block's local shared memory in a process analogous to cache

blocking on a CPU. Since all threads in a thread block have access to this chunk of shared memory, an access pattern afforded in this paradigm is a one-to-one thread-to-operand fetch where each thread fetches one operand each. Since the calculation of each element in AB needs access to many of the same elements in A& B, the threads can load a constant portion of the data each. The approach taken in the program was to allocate a shared memory tile with the same cardinality as the number of threads in a thread block so that each thread could load a single element each and then proceed with computing their respective element (see algorithm 1).

In addition to reading data into fast memory, another important consideration is the memory access patterns behind these reads. In GPUs, threads are grouped in groups of 32 called 'warps', and these warps are grouped in thread blocks. When fetching data from global memory, intra-warp access patterns may influence performance. I.e., If adjacent threads in a warp access contiguous elements from global memory, the memory controller can detect this pattern and more efficiently load the elements achieving 100% bus utilization. As threads in a warp are arranged in row-major order, this was achieved in this program by incrementing a thread's offset by their 'x' index for their respective loads and stores.

2.6.2 Reduction of Bank Conflicts

Once data is in shared memory, intra-warp access patterns again influence performance, but this time due to the ways memory banks are accessed. GK210s have 32 unique memory banks. when the same bank is accessed by two or more threads for different data words, this results in a phenomenon known as a 'bank conflict' where the memory transfers from the same block must be serialized, increasing memory transfer time. When retrieving data from shared memory, threads in the same warp performed their computations in a way that minimized bank conflicts. With a a block size of 32x32, threads in the same warp (i.e. those with the same 'y' value in the thread block) accessed data from the same location in the A matrix at each step in the inner loop which accumulates intermediate values of AB. Since they are accessing the same word, this value is broadcast to all threads, effectively amounting to one memory access per warp. When accessing the B matrix, threads in the same warp all access data from separate banks, averting bank conflicts.

2.7 Increasing Instruction-Level Parallelism (ILP): Unrolling Loops

Loop unrolling has the benefit of calculating multiple values simultaneously which take advantage of the GPU's instruction pipeline. Additionally, this can reduce the fetches from slow to fast memory due to the spatial locality inherent in matrix multiplications. As memory transfer time penalties are a serious problem in GPGPU programming, increasing ILP is necessary to hide memory latency.

2.7.1 Unrolling the Element Calculation Loop

The first attempt at loop unrolling occurred during the innermost loop that updates the elements in C. Consecutive shared memory tiles were loaded simultaneously in order to halve the amount of loads from global memory. This loop was unrolled to calculate up to two values for a single element in C simultaneously.

2.7.2 Calculating Elements in Parallel

The second attempt at loop unrolling again occurred during the innermost loop that updates the elements in C. This time, shared memory tiles needed to calculate adjacent elements were loaded simultaneously in order to quarter the amount of loads from global memory. In this paradigm, each thread calculated four elements of AB that were a block size apart in either the x dimension, y dimension, or both (i.e. if thread blocks are 32x32 and the matrix is greater than 32x32, thread 0 of block 0 would calculate $AB_{0,0}, AB_{0,32}, AB_{32,0}, \& AB_{32,32}$. As the elements shared common operands (i.e. elements in the same row of AB derive from the same row in A), this reduced the total loads for the calculation of each element of AB from $2n$ to n, though at the cost of doubling the amount of shared memory compared to a single shared memory tile for A and B respectively.

2.8 Interleaving Loads and Computation

In order to reduce the overhead of syncing threads, an attempt was made to keep a second pair of shared memory tiles for the next computation. When threads were finished with their computations, they could then load the operands for the next iteration without having to wait for other threads to finish computing. This cut the number of calls to '___syncthreads' by about half, at the cost of doubling the required amount of shared memory.

2.9 Block Size and Occupancy

From the output of ptaxs, the most optimal version used 55 registers and 32 bytes of shared memory per thread. As the GK210 has 128K registers and 112KB shared memory per SMX (64K registers per thread block), this implies that the threads have not hit the register or shared memory resource limit, and consequently do not limit our occupancy. Combined with the fact that the GK210 can accommodate 2048 threads and 16 thread blocks per SMX, we see that the final tile size of 32x32 maximizes our occupancy (i.e. occupancy of 1).

2.10 Experimental Setup

To compare the effect of varying block sizes, tests were carried out on matrices with dimensions 256x256, 512x512, 1024x1024, and 2048x2048. Next, tests were conducted on matrices with dimensions that were multiples of 32, plus and minus one, from 63x63 to 2049x2049 (i.e. 63x63, 64x64, 65x65, 95x95, 96x96, 97x97, and so on)[1] for the following versions:

1. CUDA-Best (Block Size = 32x32; Constant Block Size)
2. CUDA-Best (Block Size = 32x32)
3. CUDA-Best (Block Size = 16x16)
4. CUDA-Best (Block Size = 8x8)
5. CUDA-1IL (Block Size = 32x32; Interleaved Loads and Computation)
6. CUDA-C_Unroll (Block Size = 32x32; C Unrolled by two)
7. CUDA-1 (Block Size = 32x32; Coalesced Shared Memory)
8. CUDA-Naive (Block Size = 32x32; Naive Implementation)
9. CUDA-1u (Block Size = 32x32; Uncoalesced Shared Memory)

. Performance values were taken as the average of 10 tests for each matrix dimension.

3 Results

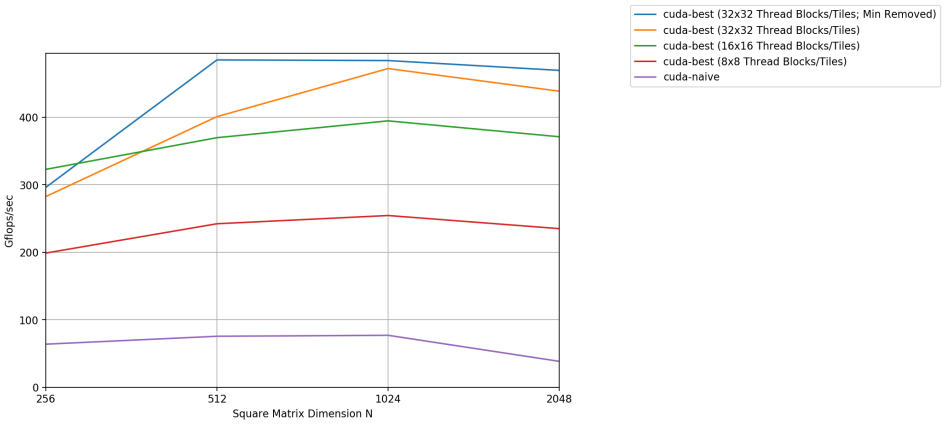

Figure 1: Performance As a Function of Select Matrix Dimension

[1] a total of 189 matrix sizes

From figure 1, we can see increasing block sizes increased the matrix multiplication performance up to the thread block limit of 1024 (32x32) threads.

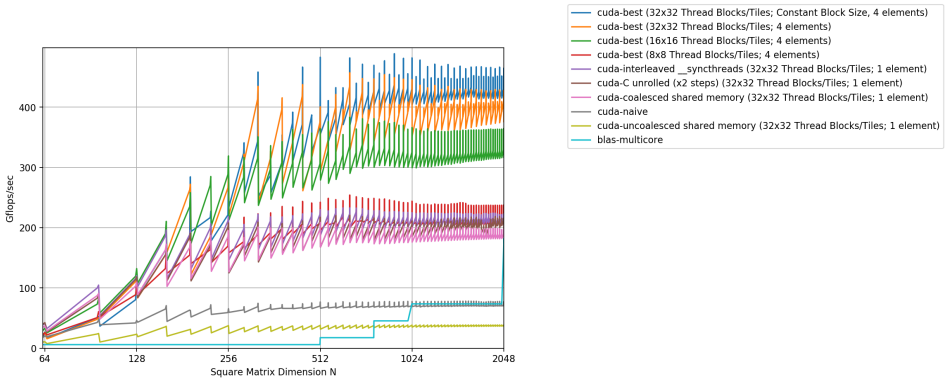

Figure 2: Performance As a Function of Matrix Dimension (n=189)

As we can see from figure 2, calculating four elements of AB in parallel yielded significant performance benefits, with all tile sizes outperforming all versions that calculated elements one at a time. Interleaving memory accesses (see section 2.8 resulted in the greatest performance of all single-element-calculating versions.

Unrolling C resulted in only slightly more throughput (about 20 Gflops/s) than only coalescing accesses to shared memory. Uncoalesced access to shared memory resulted in worse performance than the naive version, being overtaken by multi-core BLAS for matrices greater than 512x512.

Of interest is the performance plateau of the best-performing CUDA version for matrices greater than 1024x1024, whereas the performance of multicore BLAS begins to rise sharply.

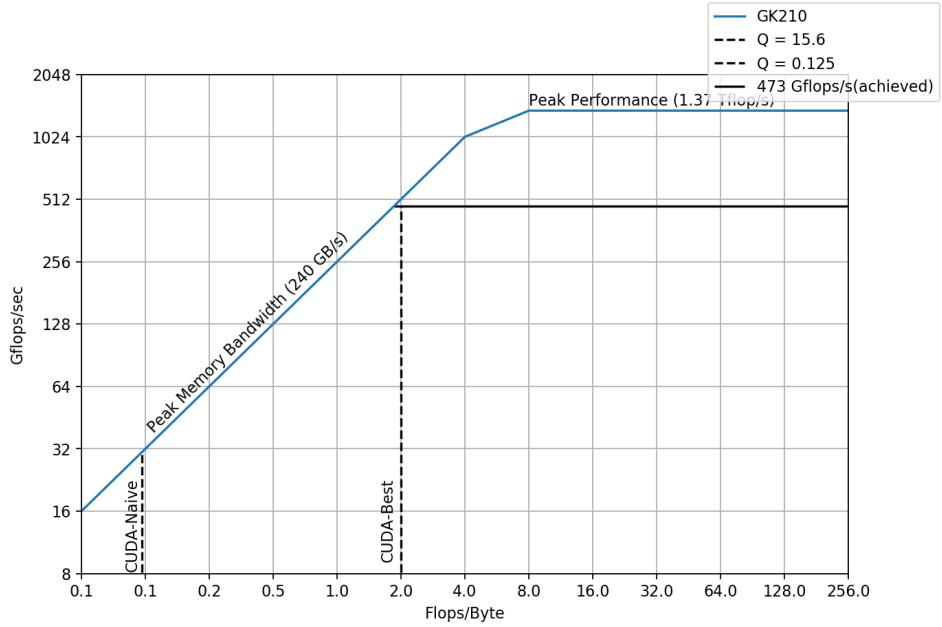

Figure 3: Roofline Model (240 GB/s Memory Bandwidth)

From figure 3 we see that we perform an estimated 2 flops/byte if our peak memory bandwidth is 240 GB/s.

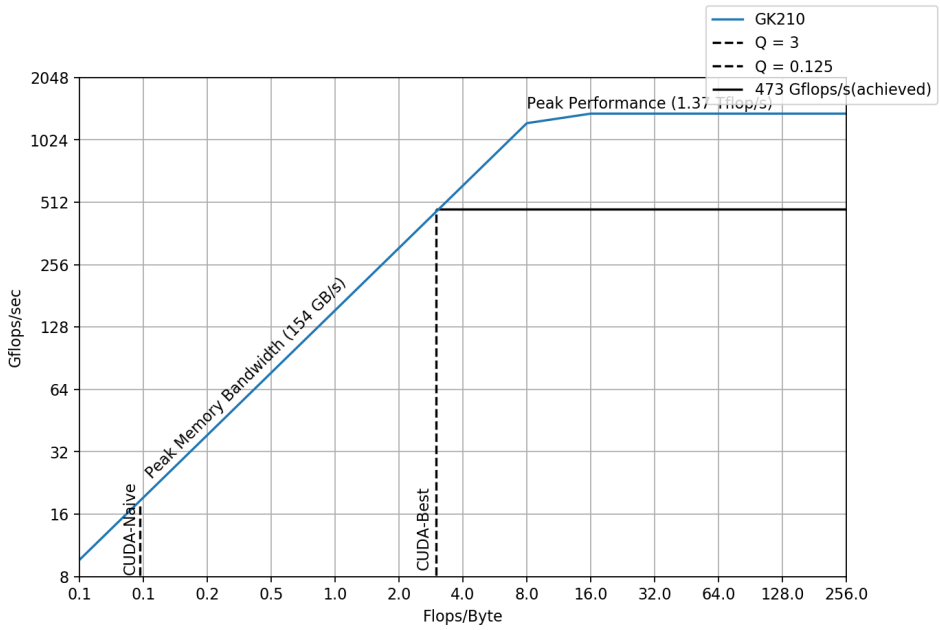

Figure 4: Roofline Model (154 GB/s Memory Bandwidth)

If empirical memory bandwidth is 154 Gb/s, from figure 4 we see we are actually performing an estimated 3 flops/byte.

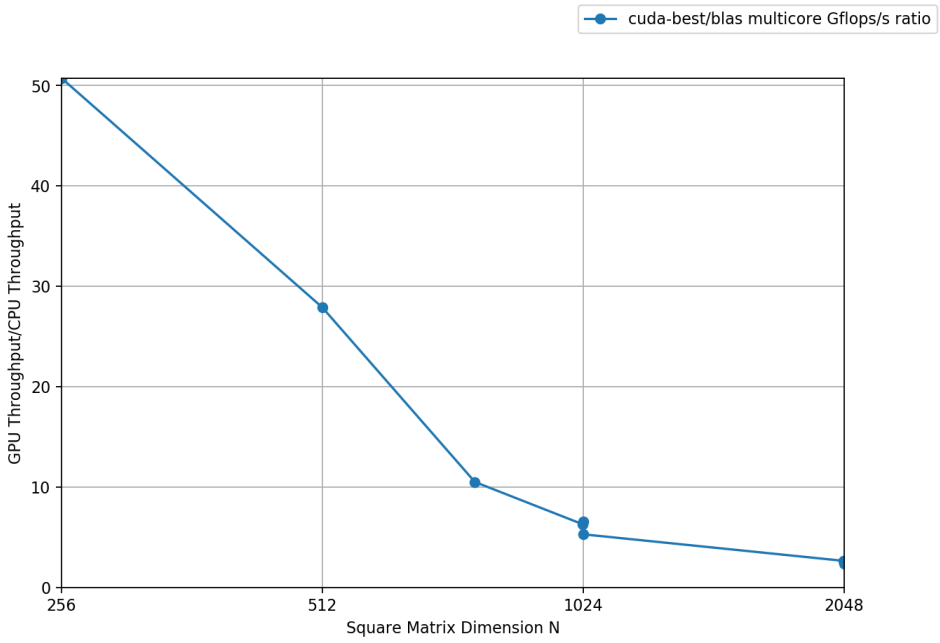

Figure 5: Speed-up Ratio (cuda-best vs. BLAS multicore)

We find from figure 5 that the speed ratio between the CUDA matrix multiplication and the BLAS multicore version is quite high for smaller matrix sizes (50x for 256x256 matrices, 30x for 512x512 matrices) but drops sharply for matrices greater than 768x768. From table 1, we see that this is due to the fact that the average Gflops for BLAS multicore continue to increase as we increase the size of the matrices, whereas the average Gflops for the CUDA

Table 1: BLAS (multi-core) vs. CUDA-Best(Gflops/s)

n	BLAS (Gflops)	CUDA-Best (Gflops)
256	5.84	294.98
512	17.4	482.92
768	45.3	469.59
1023	73.7	430.39
1024	73.6	481.88
1025	73.5	404.51
2047	171	430.24
2048	182	464.98
2049	175	415.34

matrix multiplication start high but plateau fairly early (at about 482 Gflops for 512x512 matrices).

Algorithm 1 Calculating an Element in AB

$A\_idx{=}row * N + threadIdx.x$
$B\_idx{=}col + threadIdx.y * N$
$\_c{=}0$
for $stride = 0; stride < gridDim.x; {+}{+}stride, A\_idx{+}{=}incr_A, B\_idx{+}{=}incr_B$ **do**

$$\text{A\_tile} \leftarrow \underbrace{\begin{Bmatrix} A\_idx_{ty_0,tx_0} & \dots & A\_idx_{ty_0,tx_{n-1}} \\ \vdots & \ddots & \vdots \\ A\_idx_{ty_{n-1},tx_0} & \dots & A\_idx_{ty_{n-1},tx_{n-1}} \end{Bmatrix}_{|n=32|}}_{|n=32|}$$

$$\text{B\_tile} \leftarrow \underbrace{\begin{Bmatrix} B\_idx_{ty_0,tx_0} & \dots & B\_idx_{ty_0,tx_{n-1}} \\ \vdots & \ddots & \vdots \\ B\_idx_{ty_{n-1},tx_0} & \dots & B\_idx_{ty_{n-1},tx_{n-1}} \end{Bmatrix}_{|n=32|}}_{|n=32|}$$

`__syncthreads()`
$\_c{+}{=}A\_tile[threadIdx.y][:] * B\_tile[:][threadIdx.x]$
`__syncthreads()`
$C[row][col]{=}\_c$

3.1 Analysis

3.1.1 Uncoalesced Shared Memory Loads Vs. No Shared Memory

An interesting observation is that uncoalesced shared memory loads resulted in worse performance than the naive version which retrieved operands directly from global memory. This may be due to the fact that the uncoalesced accesses to global memory resulted in values being loaded in piece by piece as opposed to coalesced memory accesses where contiguous units are loaded into shared memory. Though the naive version doesn't make use of shared memory, some of its global memory accesses are coalesced and some of these operands are stored in the L2, L1, and read-only data caches, which undoubtedly allow the algorithm to capitalize on the spatial localilty inherent in matrix multiplications.

3.2 Interleaved Shared Memory Loads

While interleaving memory accesses did result in the best performance for versions where each thread calculated a single element, this approach could not feasibly be combined with the version that calculates four values simultaneously. Interleaving memory accesses requires double the amount of shared memory as compared to non-interleaving versions. Since the

Algorithm 2 Calculating One Element of AB with Interleaving

$A\_idx = row * N + threadIdx.x$
$B\_idx = col + threadIdx.y * N$
$curr = 0$
$\_c = 0$

$$\text{A\_tile[curr]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|}\hline A\_idx_{ty_0,tx_0} & \dots & A\_idx_{ty_0,tx_{n-1}} \\ \hline \vdots & \ddots & \vdots \\ \hline A\_idx_{ty_{n-1},tx_0} & \dots & A\_idx_{ty_{n-1},tx_{n-1}} \\ \hline \end{array}\right\}|n{=}32|}_{|n=32|}$$

$$\text{B\_tile[curr]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|}\hline B\_idx_{ty_0,tx_0} & \dots & B\_idx_{ty_0,tx_{n-1}} \\ \hline \vdots & \ddots & \vdots \\ \hline B\_idx_{ty_{n-1},tx_0} & \dots & B\_idx_{ty_{n-1},tx_{n-1}} \\ \hline \end{array}\right\}|n{=}32|}_{|n=32|}$$

`__syncthreads()`
$A\_idx_0 += incr_A$
$B\_idx_0 += incr_B$
for $stride = 0; stride < gridDim.x; ++stride, A\_idx_{0\&1} += incr_A, B\_idx_{0\&1} += incr_B$ **do**
  $\_c += A\_tile[curr][threadIdx.y][:] * B\_tile[curr][:][threadIdx.x]$
  $curr = !curr$

$$\text{A\_tile[curr]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|}\hline A\_idx_{ty_0,tx_0} & \dots & A\_idx_{ty_0,tx_{n-1}} \\ \hline \vdots & \ddots & \vdots \\ \hline A\_idx_{ty_{n-1},tx_0} & \dots & A\_idx_{ty_{n-1},tx_{n-1}} \\ \hline \end{array}\right\}|n{=}32|}_{|n=32|}$$

$$\text{B\_tile[curr]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|}\hline B\_idx_{ty_0,tx_0} & \dots & B\_idx_{ty_0,tx_{n-1}} \\ \hline \vdots & \ddots & \vdots \\ \hline B\_idx_{ty_{n-1},tx_0} & \dots & B\_idx_{ty_{n-1},tx_{n-1}} \\ \hline \end{array}\right\}|n{=}32|}_{|n=32|}$$

  `__syncthreads()`
$\_c += A\_tile[threadIdx.y][:] * B\_tile[:][threadIdx.x]$
`__syncthreads()`
$C[row][col] = \_c$

calculation of four values itself requires twice the amount of shared memory, a combined approach would exceed the maximum default shared memory limit for thread blocks.

3.3 Development Process and Analysis of Program Behavior

3.3.1 Shared Memory Access Patterns

I initially began optimizing the matrix multiplication routine by implementing the use of shared on-chip memory. This resulted in more than double the performance of the naive version, achieving more than 200 Gflops/s (see figure 2). I then attempted to vary the memory access patterns to global memory and shared memory, with all attempts worsening performance. It was only later that I realized the memory access patterns already coalesced and avoided bank conflicts. See algorithm 1

3.3.2 Block Sizes

In tandem with varying memory access patterns (see section 3.3.1), I varied block sizes and conditionals to evaluate their effects on performance. For the most part, varying conditionals (i.e. nesting them, keeping them separate, using conditionals to avoid computations of

Algorithm 3 Calculating Four Elements of AB in Parallel

$A\_idx_0 = row_0 * N + threadIdx.x$
$A\_idx_1 = row_1 * N + threadIdx.x$
$B\_idx_0 = col_0 + threadIdx.y * N$
$B\_idx_1 = col_1 + threadIdx.y * N$
$\_c = \{0, 0, 0, 0\}$
for $stride = 0; stride < gridDim.x; ++stride, A\_idx_{0\&1} += incr_A, B\_idx_{0\&1} += incr_B$ **do**

$$\text{A}\_tile_{[:]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|} \hline A\_idx_{0_{ty_0,tx_0}} & \cdots & A\_idx_{0_{ty_0,tx_{n-1}}} \\ \hline \vdots & \ddots & \vdots \\ \hline A\_idx_{1_{ty_{n-1},tx_0}} & \cdots & A\_idx_{1_{ty_{n-1},tx_{n-1}}} \\ \hline \end{array}\right\}_{|2n=64|}}_{|n=32|}$$

$$\text{B}\_tile_{[:]} \leftarrow \underbrace{\left.\begin{array}{|c|c|c|} \hline B\_idx_{0_{ty_0,tx_0}} & \cdots & B\_idx_{1_{ty_0,tx_{n-1}}} \\ \hline \vdots & \ddots & \vdots \\ \hline B\_idx_{0_{ty_{n-1},tx_0}} & \cdots & B\_idx_{1_{ty_{n-1},tx_{n-1}}} \\ \hline \end{array}\right\}_{|n=32|}}_{|2n=64|}$$

`__syncthreads()`
$\_c[0] += A\_tile_0[threadIdx.y][:] * B\_tile_0[:][threadIdx.x]$
$\_c[1] += A\_tile_0[threadIdx.y][:] * B\_tile_1[:][threadIdx.x]$
$\_c[2] += A\_tile_1[threadIdx.y][:] * B\_tile_0[:][threadIdx.x]$
$\_c[3] += A\_tile_1[threadIdx.y][:] * B\_tile_1[:][threadIdx.x]$
`__syncthreads()`
$C[row_0][col_0] = \_c[0]$
$C[row_0][col_1] = \_c[1]$
$C[row_1][col_0] = \_c[2]$
$C[row_1][col_1] = \_c[3]$

specious elements of AB, etc.) resulted in imperceptible effects on performance. Initial experimentation with varying block sizes pointed to the fact that larger block sizes were better. Due to a lack of time and suspicion that there would be a marginal effect on performance, rectangular block sizes were not investigated.

3.3.3 Program Optimizations

I also took advantage of CUDA compiler pragmas to unroll inner for-loops and refactored the calculation of threads' corresponding A and B indexes so that the multiplications were done outside loops (i.e., indexes into the next chunks of A and B were calculate solely through an increment; see algorithm 1).

3.3.4 Unrolling the 1-Element Calculation Loop

Next I realized I could unroll the loop over C by loading consecutive tiles at the same time. This was equivalent to doubling the shared memory tiles in their respective dimensions. Though this halved the number of times threads had to sync for loads and computations, this only resulted in fairly modest performance improvement (about 20 Gflops/s; see figure 2).

3.3.5 Interleaving Loads and Computations

Following the approach taken in section 3.3.4, I realized I could interleave the loads to the next tiles with computations from the current tiles. This again halves the number of times threads had to sync for loads and computations as they now they only need to sync prior to running the next computation (see algorithm 2).

3.3.6 Calculating Multiple Elements of AB in Parallel

Next, I attempted to increase ILP by having each thread calculate multiple elements in parallel. Since threads in a block are already calculating adjacent elements, it seemed logical for threads attempt to calculate elements in adjacent blocks. In this way, the threads doubled the Flops per double-load (calculating four elements with four loads as opposed to one element with two loads; see algorithm 3). This resulted in another doubling of performance (about 480 Gflops/s) as compared to interleaving computations and loads. Due to time constraints and the fact that the program was already hitting the limits of shared memory, I was not able to test how much more performance could be gained from this approach.

4 Future Work

In the future, it would be interesting to delve deeper into how the different CUDA libraries increase ILP and achieve greater, more consistent levels of performance, especially at non-static block sizes.

References

CPU World. Intel xeon e5-2640 v3 specifications, 2017. URL `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2640%20v3.html`.

NVIDIA. *NVIDIA Tesla K20 GPU Accelerator BD-07317-001_v05 | January 2015 Board Specification*, 2015. URL `http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf`.

NVIDIA Corporation. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110/210. Technical report, 2014.