LAB 1: MatMul

Zhaorui Wang cwid: 20007447

Questions:

(1) How many floating operations are being performed in your dense matrix multiply kernel if the matrix size is N times N? Explain.

For each output element there are N multiplications and N - 1 additions so When N times N is small the total tflops is $N^2 \times (2N - 1)$. When N is large it is approximately $2N^3$.

(2) How many global memory reads are being performed by your kernel? Explain.

Assuming we have A and B arrays every element in A[i,k] is needed for computing in C[i,j]. Each thread can read the A element independently. This is the same for Matrix B. There are N reads from A and N reads from B for every C[i,j] calculation. This means that there are 2N reads. Since there are N^2 elements in C then the total global memory reads will be $2N^3$.

(3) How many global memory writes are being performed by your kernel? Explain.

There are N^2 writes because you write each value of C into global memory and C is a NxN matrix.

(4) Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

Several optimizations can be applied to achieve a performance speedup. Different tile sizes can be experimented with. For every program, there's always some autotuning or manual tuning that can be achieved. I've experimented with a few already using manual tuning.

I've also implemented 1D warptiling and the homework calls for shared memory and tiling as well so these are the easiest, but I've run out of time to do 2D warptiling. (can't get it to work). This way for each warp, there are 32 threads.

Another solution is to vectorize memory access so that the floats are stored in float4. This can be done to shared and global memory. This will reduce memory access by 4. I've also tried implementing this but I don't think it's possible with non square and non multiple of 2 matrices. I think a micro kernel that can conditionally handle parts of the matrix or some other solution is in order. This might be what cublas is doing.

Another solution that might be easier is to implement double buffering. This will allow for overlap of computation with data transfers to hide global memory access latency. The cuda compiler already implements some memory coalescing and potentially already does fused multiply add or loop unrolling. I am unsure so I explicitly declared these.

(5) Suppose you have matrices with dimensions bigger than the max thread dimensions allowed in CUDA. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.

1. Choose tiles that fit inside the thread block dimensions. Divide NxN into tiles.
2. Each thread block will compute one tile of the output C and repeat this step for each tile
3. Write C back when the partial sum has been accumulated.

```
for (int i = 0; i < round(N/T); i++) {
    for (int j = 0; j < round(N/T); j++) {
        //each block computes a c tile
        Initialize C_tile to 0;
        for (int k = 0; k < round(N/T); k++) {
            //a tile and b tile are loaded
            __syncthreads();
            //compute partial matmul
            for (int t = 0; t < T; t++) {
                C_tile += matrixMultiply(A_tile, B_tile)
            }
            __syncthreads();
        }
        //write the c tile back to global memory
    }
}
```

(6) Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

If the matrices are too large I think u can use a block algorithm that computes the partial matmul, then offloads it to the host. First, divide the matrices into blocks that can fit into the global memory or of the gpu. Let's say the block size is B. For each block in the C matrix, the corresponding A and B blocks will be loaded, then matmul can be performed whatever way is desired. The sub matrix block of C is then stored back to the disk or host memory if device memory is also overflowed. I think here would be easy to implement double buffering or asynchronous memory access so that computing and memory access are running at the same time.

```
for (int i = 0; i < cell(N/B); i++) {
    for (int j = 0; j < cell(N/B); j++) {
        //each block computes a C block
        C_block = zeros(B, B);
        for (int k = 0; k < cell(N/B); k++) {
            //load A and B into global memory
            __syncthreads();
            //compute partial matmul
            A_block = loadBlock(A, i, k);
            B_block = loadBlock(B, k, j);
            C_block += matrixMultiply(A_block, B_block); //perform kernel operations on A and B blocks and store to C
            }
            __syncthreads();
        }
        writeBlock(C, i, j, C_block); //write to device or host memory
    }
```

MY IMPLEMENTATION:

Naive Implementation:

The first order of business for this lab was to implement a naive matmul solution. This was easy enough.

```
// naive
__global__ void matrixMulGlobalNaive(float *A, float *B, float *C, int M, int N, int K) {

    // calculate index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // check if they are multiples of block
    if (row < M && col < N) {
        float sum = 0.0f;
        #pragma unroll
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

I essentially copied the matmul algorithm from the slides. Then I wrapped this with a kernel launcher that allocates and deallocates the memory, creates the matrices, and passes the dimensions into it. It also measures the performance of the kernel using cuda events.

```
inline std::pair<double, double> runMatrixMulNaive(int M, int N, int K, int blockWidth, int blockHeight) {

    float *d_A, *d_B, *d_C;
    float *h_A = new float[M * K];
    float *h_B = new float[K * N];
    float *h_C = new float[M * N];
    float *h_C_ref = new float[M * N];

    //initialize matrices with random values
    for (int i = 0; i < M * K; ++i) h_A[i] = static_cast<float>(rand()) / RAND_MAX;
    for (int i = 0; i < K * N; ++i) h_B[i] = static_cast<float>(rand()) / RAND_MAX;

    //allocate device memory
    allocateDeviceMemory(&d_A, &d_B, &d_C, M, N, K);
    cudaMemcpy(d_A, h_A, M * K * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, K * N * sizeof(float), cudaMemcpyHostToDevice);

    //define grid and block dimensions
    dim3 blockDim(blockWidth, blockHeight, 1);
    dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (M + blockDim.y - 1) / blockDim.y, 1);

    //launch kernel and measure performance
    auto result = measurePerformance([&]() { matrixMulGlobalNaive<<<gridDim, blockDim>>>(d_A, d_B, d_C, M, N, K); }, M, N, K);

    //copy results back to host
    cudaMemcpy(h_C, d_C, M * N * sizeof(float), cudaMemcpyDeviceToHost);
    freeDeviceMemory(d_A, d_B, d_C);

    //free memory
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;
    delete[] h_C_ref;

    //return tflops and ms
    return result;
}
```

In comparison.cu there is a driver that passes the blockwidth and height and matrices dimensions that are specified by the user.

<u>Tiled Implementation:</u>

Ok, so this algorithm I was actually able to implement pretty easily too. I was able to follow the algorithm from cuda reading and from the slides. It's pretty straight forward in theory. I added my own optimizations to it, but basically a B and A tile from matrix B and A are assigned to shared memory. Then they are loaded to the correct indexes. Afterwards a sync is called to make sure this is done. Then I used partial warp tiling (1D) to compute the sum into an accumulator. Afterwards sync again before the next tile. When done with this, write to C. Similar to the naive algorithm there is a kernel launcher that configures it.

```
//shared memory tiles
__shared__ float As[TILE_SIZE][TILE_SIZE];
__shared__ float Bs[TILE_SIZE][TILE_SIZE];

//number of tiles in the k dimension
int numTiles = (K + TILE_SIZE - 1) / TILE_SIZE;

//iterate over tiles
#pragma unroll
for (int t = 0; t < numTiles; t++) {
    //load A_tile into shared memory
    #pragma unroll
    for (int i = 0; i < MICRO_TILE_ROWS; i++) {
        int rowA = rowTile + ty + i * BLOCK_DIM_Y;
        int colA = t * TILE_SIZE + tx;
        if (rowA < M && colA < K)
            As[ty + i * BLOCK_DIM_Y][tx] = A[rowA * K + colA];
        else
            As[ty + i * BLOCK_DIM_Y][tx] = 0.0f;
    }

    //load B_tile into shared memory
    #pragma unroll
    for (int i = ty; i < TILE_SIZE; i += BLOCK_DIM_Y) {
        int rowB = t * TILE_SIZE + i;
        int colB = colTile + tx;
        if (rowB < K && colB < N)
            Bs[i][tx] = B[rowB * N + colB];
        else
            Bs[i][tx] = 0.0f;
    }

    __syncthreads();  //sync here to make sure both are loaded

    //compute using partial warp tiling 1 dimension warp tiling.
    #pragma unroll
    for (int k = 0; k < TILE_SIZE; k++) {
        float bVal = Bs[k][tx];

        #pragma unroll
        for (int i = 0; i < MICRO_TILE_ROWS; i++) {
            int rowIndex = ty + i * BLOCK_DIM_Y;
            accum[i] = __fmaf_rn(As[rowIndex][k], bVal, accum[i]);
        }
    }

    __syncthreads();  //sync to wait for next tile
}

//write back to global memory
#pragma unroll
for (int i = 0; i < MICRO_TILE_ROWS; i++) {
    int rowC = rowTile + ty + i * BLOCK_DIM_Y;
    if (rowC < M && col < N)
        C[rowC * N + col] = accum[i];
}
```

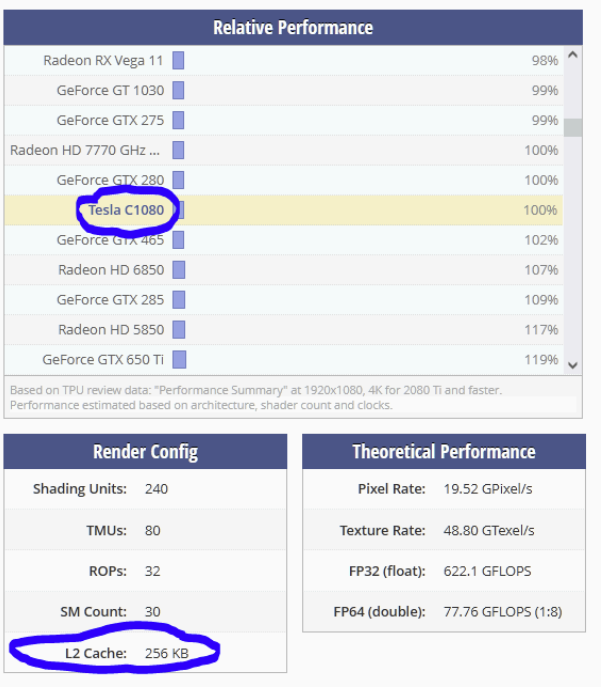
The problem with this algorithm is when giving square tiles and square blocks the performance speed up is terrible, only 1-1.5x the improvement over naive. This is because on Volta, the architecture that we use for this lab and what I have at home, the l2 and l1 cache are quite big and these represent the global memory. In order to overload this you would need 16 megabytes of data in l2 and l1. Theoretically it can store 4million

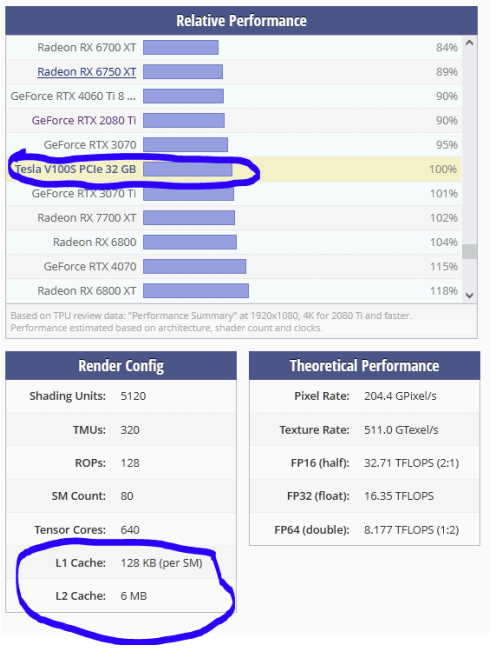
float points! In the slides and class they are using original teslas which don't even have l1 cache and very small l2 cache. This means that the amount of global memory is huge compared to previous architectures. This also means device memory read and writes are minimized already. There could also be some partial memory coalescing going as well.

I tried different methods like double buffering and vectorizing the memory loads to improve performance. Eventually I found that if I load the blocks as nonsquare like 32x8 or 64x4 I was able to achieve massive performance without compute problems, like 2-3x initially. Overall speed up compared to naive was 3-4x now. For the same matrices I was within 60% of cublas. I've kept block size to a maximum of 256 threads, maybe there's some improvement when using 512 threads but I didn't test, I've mainly tested different block shapes.

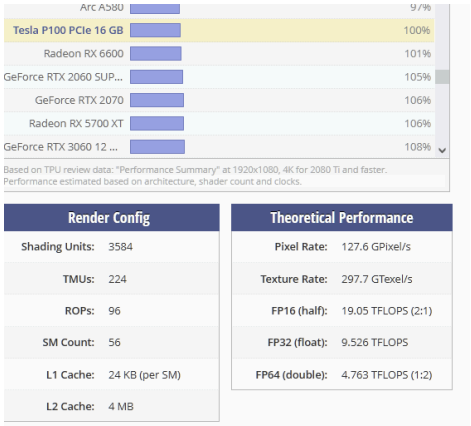
V100S Volta, our GPUs

Tesla which is similar to the slides gpus architecture (GTX 280) . It has 64x less global memory!





Even when compared to the previous generation there's a huge global memory

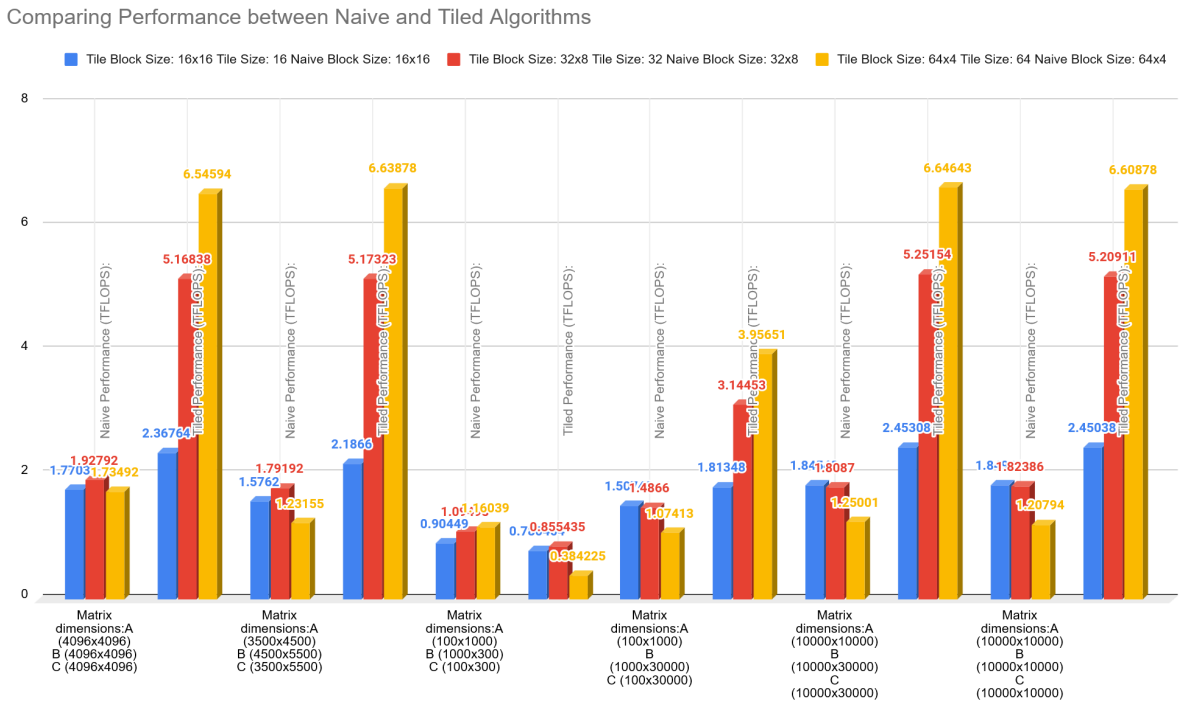


improvement. Volta's two biggest changes were the tensor cores and the increase in l1 cache.

BENCHMARKS:

Now for the fun stuff, benchmarking. I had several code edits throughout this process due to time constraint so I will add acknowledgements for where it is necessary.

First we will compare naive against tiled with different matrix sizes all other than one are not multiples of block or 16 for that matter.



The only problem I had here was I forgot to copy the memory to the gpu, so the memory is allocated and whatever garbage values are there are being multiplied. I didn't rerun everything but just to confirm there's no massive slow down here's results for 10k squared with memory copied in.

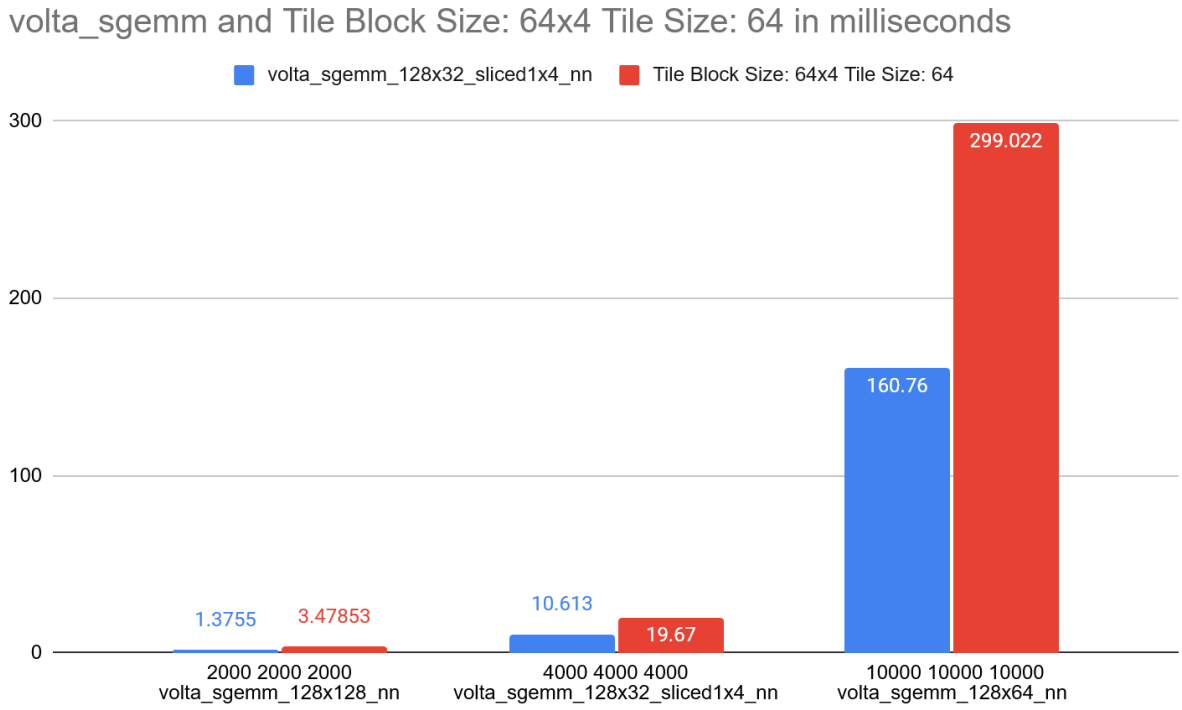
Matrix dimensions:
A (10000x10000)
B (10000x10000)
C (10000x10000)
Performance Results:
Tile Block Size: 16x16 Tile Size: 16
Naive Block Size: 16x16
Naive Execution Time (ms): 1083.07
Tiled Execution Time (ms): 844.566
Naive Performance (TFLOPS): 1.8466
Tiled Performance (TFLOPS): 2.36836
Performance Results:
Tile Block Size: 32x8 Tile Size: 32
Naive Block Size: 32x8
Naive Execution Time (ms): 1117.36
Tiled Execution Time (ms): 418.841
Naive Performance (TFLOPS): 1.78994
Tiled Performance (TFLOPS): 4.77511
Performance Results:
Tile Block Size: 64x4 Tile Size: 64
Naive Block Size: 64x4
Naive Execution Time (ms): 1588.94
Tiled Execution Time (ms): 341.067
Naive Performance (TFLOPS): 1.2587
Tiled Performance (TFLOPS): 5.86398

In the new data to the side there is a slow down for when tile block size is 64x4 but not enough to be much slower than its naive implementation or any naive implementation. It's still about 4x faster than any naive implementation.

For the rest of the results we see that the naive is mostly block dimension independent. This makes sense as there's no warptiling or any special techniques to optimize threading. With the tiling algorithm, we see massive improvement from moving from 16x16 block size and tile size of 16 to 32x8 and tile size of 32 and 64x4 with tile size of 64. With tile size of 64 we achieve our highest performance, of 6.6 tflops in some cases. Cublas on the same can achieve about 13tflops. The theoretical peak of V100S is 16.3 tflops as seen previously in the picture of global memory size. As to why we can't achieve

Cublas level of performance, I was not able to implement 2D warptiling and Cublas probably has different kernels for different matrix sizes so it's impossible for me solely to compete with them there. Cublas can also take advantage of tensor cores for speed up using mixed precision. Either way I am at 50% of cublas with just a shared memory tiling algorithm and about 3-4x faster than the naive algorithm.

Cublas comparison:



Error check:

There is an error check program that allows you to check the error rate between tiled and naive. It computes the matrix with tiled and naive implementations then the cpu implementation and compares them.The error rate is very close between the two and both have an error rate of 0% with a 10^-3 threshold. Here are some results from that.

Matrix Dimensions: 1270 x 1254 x 721	
Running Tiled GPU Matrix Multiplication (Tile Size = 64)...	
Error Percentage: 0%	
Mean Squared Error:	9.07E-11
Max Absolute Error:	6.10E-05
Running Naïve GPU Matrix Multiplication...	
Error Percentage: 0%	
Mean Squared Error:	9.09E-11
Max Absolute Error:	6.10E-05
Matrix Dimensions: 1041 x 1247 x 139	
Running Tiled GPU Matrix Multiplication (Tile Size = 16)...	
Error Percentage: 0%	
Mean Squared Error: 3.78281e-12	
Max Absolute Error: 1.52588e-05	
Running Naïve GPU Matrix Multiplication...	
Error Percentage: 0%	
Mean Squared Error: 3.78185e-12	
Max Absolute Error: 1.52588e-05	
Matrix Dimensions: 535 x 792 x 414	
Running Tiled GPU Matrix Multiplication (Tile Size = 32)...	
Error Percentage: 0%	
Mean Squared Error: 2.70443e-11	
Max Absolute Error: 3.8147e-05	
Running Naïve GPU Matrix Multiplication...	
Error Percentage: 0%	
Mean Squared Error: 2.69517e-11	
Max Absolute Error: 3.05176e-05	

Conclusion:

This was a pretty straightforward homework assignment that I got carried away on and was not able to finish in time. I asked for a 1 to 2 day extension to finish this and carry out the benchmarks. I spent a lot of time implementing different solutions to get tiled to be much faster than naive. I found out that different block shapes and 1D warp tiling was achievable in time. In terms of things I tried I did, double buffering, 2D warptiling, vectorized memory access ,unrolling loops, and faster add multiply. In square matrices that are nice multiples of 16, these methods will work great, but when they are of arbitrary size it becomes much harder. I'm sure if an implementation is to branch to different kernels for different parts of the matrices or different shapes of matrices, more performance can be achieved. I ran out of time to implement branching logic and micro kernel selection.

Another reason why I spent so much time on this was I wasn't confident in my theory of the larger global memory on Volta. I was not convinced that this could make the naive program that much faster than if the global memory was smaller.

The basic concepts with kernel just looking at being just one thread of the program you are running and barrier synchronizing to wait for memory load and unload were important to the implementation. I also learned a lot about memory hierarchy and streaming multiprocessor hierarchy with warps, blocks and thread characteristics for different architectures. The easiest tiled algorithm was straightforward to implement but was not fast enough.

Overall this was a cool project and I had fun benchmarking and checking for error rate. I spend my free time benchmarking GPUs anyway so the work was something I like to do and would spend hours doing this if I could. Matrix Mul isn't very interesting but its kernel optimizations are the fundamentals of all cuda programming and achieving performance here is applicable to all kernels.