COL380 Assignment 4
Parallel & Dist. Programming | CUDA

Chinmay Mittal (2020CS10336)

Algorithm & Design Choices

Reading The Input

The input matrices are read at the CPU and sent to the GPU. I use a 2D grid of pointers to store the inputs at the GPU and CPU sides. Each pointer points to a block of elements. If the block is zero, the pointer is NULL; this *saves space by not storing and sending zero blocks*.

I have modified the size of the blocks to 32 x 32 for efficient processing on the GPU side. While reading the inputs, I reconstruct the input into 32 x 32 blocks and do not store empty blocks.

These matrices are first read at the CPU side; I used the two-bit unsigned int to store these matrices, which will save space and transfer time. After the input is read on the CPU side, I allocate space for these matrices on the GPU and start an *asynchronous transfer to the GPU*. I use asynchronous transfer and *streams* to send these matrices parallelly to the GPU.

Matrix Multiplication Kernel

I organise the blocks in the grid and the threads in the block in a 2D structure, I have a 2D grid of blocks, and each block is a 2D grid of 32 x 32 threads. If the matrix is not divisible by 32, it will be effectively padded by zeroes to the right and the bottom to make its dimension divisible by 32.

Each block of threads is responsible for computing a block in the output matrix, and each thread is responsible for computing an output element.

Hence the number of threads will be N^2, and the number of blocks will be $(N/32)^2$.

I have implemented the *tiled caching algorithm* that uses the faster-shared memory available to a block of threads.

A block of threads is supposed to compute an output block which is the multiplication of a row of blocks with a column of blocks. In tiled caching, we load two tiles (blocks), one from A's row and the other from B's column, into the *shared memory*, do all the processing required on these

blocks for computing the output and store the partial results. This is effective since it uses the faster-shared memory and optimises for cache hits amongst the threads. While multiplying two tiles, each thread multiplies a row of elements in the first tile with a column in the second and computes the partial output.

To account for the fact that the matrix can be sparse if any of the two tiles to be loaded into the shared memory are empty, we skip their multiplication.

We also return to the CPU a 2-D array indicating whether the output block computed is zero. This is effective because otherwise, the CPU must do this processing sequentially.

Writing the Output

The output is stored as an N^2 matrix; each element in this matrix is computed by one thread. The GPU also computes a 2-D grid indicating which output block is zero. These are copied from the GPU to the CPU; the CPU can effectively count and write the non-zero blocks to the output file without processing all blocks.

Optimisations

1. Preventing the use of *cudaMallocManaged*, unified memory accesses are slower. Hence, I have explicitly created, managed and synchronised CPU RAM and GPU global memory. Hence each process can access the data from memory closer to it.
2. Storing the inputs effectively: I do not store/send zero blocks, saving space and transfer time.
3. Data types: I have used the smallest datatype wherever possible, for e.g. using 16-bit ints and 8-bit characters instead of 32-bit ints. This saves space and transfer time.
4. Shared Memory and Tile Caching: I have made efficient use of caching into faster-shared memory available to block of threads.
5. Using 32 x 32 blocks. Instead of using smaller block sizes as provided in the inputs, I have used larger thread blocks, which save the overhead of thread creation and scheduling.
6. Asynchronous Data Transfer using streams: I have parallelised the data transfer of the two matrices to the GPU using CUDA streams and Asynchronous Memory transfer.
7. Handling sparsity: I do not multiply zero blocks to save time exploiting the sparsity of the input.

Results

Here are the results on a few sample matrices created. Times are in (ms)

S. No	N	M	K1 (A)	K2 (B)	K3 (C)	Reading & send Input	Matrix Mult	Copy Output	Write Output
1	24	4	12	12	19	2249	~0	~0	16
2	24	8	5	5	6	2254	~0	~0	~0
3	8	4	4	4	4	2258	~0	~0	~0
4	4096	8	52450	52450	262144	2850	580	22	841
5	32768	8	16800	16800	68517	2675	18079	2962	446
6	32768	4	3.3e7	3.3e7	6.71e7	106031	308401	3353	44301
7	32768	4	65300	65300	518873	1834	18939	3625	1201
8	32768	8	8.8e6	8.8e6	16777216	91244	308893	3542	32620