hw3

ECE 3803

Jeff Epstein

Contents

1 Introduction

This assignment shows how to improve performance with shared memory, in the context of matrix algebra.

2 Assignment Part 1: Audio blur with shared memory (15 points)

Your audio blur program from the previous homework is a good candidate for using shared memory. The convolution kernel data, passed to the GPU code in parameter `gpu_blur_v`, is read frequently. If we copy that data into shared memory, we can reduce the number of global memory reads.

Your task is to modify the `audioblur.cu` file to store the convolution kernel in shared memory, and to read it from shared memory when performing the convolution. You will need to change

`cuda_blur_kernel` to copy the data and synchronize threads. You will also need to make sure that a pointer to shared memory is passed to `cuda_blur_kernel_convolution`. You do not need to modify `cuda_blur_kernel_convolution` itself. If you choose to dynamically allocate shared memory, you'll need to modify the kernel launch in `cuda_call_blur_kernel`.

Make sure that your program works correctly with convolution kernels of different sizes. You will need to be careful that the entire convolution kernel is copied into shared memory. You can test this by increasing the value of `GAUSSIAN_SIDE_WIDTH` in `audioblur.cuh`.

3 Assignment Part 2: Matrix multiply (20 points)

As you know, the product of matrices $\mathbf{C} = \mathbf{AB}$ can be calculated as

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

In other words, the value of each element of the result matrix $\mathbf{C}$ is determined by one row of $\mathbf{A}$ and one column of $\mathbf{B}$. We can visualize this algorithm as follows:

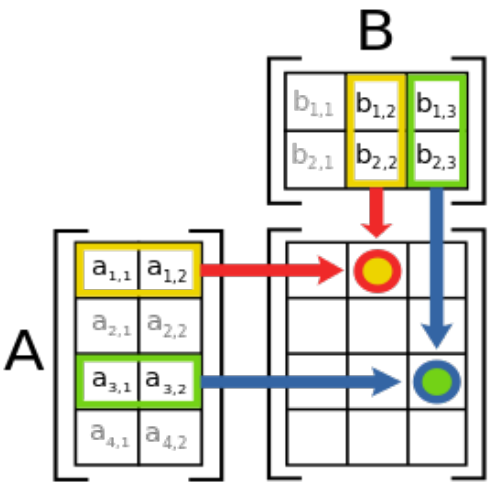

Figure 1: matrix multiply

This can be directly expressed in an imperative language such as C. Assuming we have two two-dimensional input arrays `A` and `B`, and a two-dimensional output array `C`, the following code will perform a matrix multiply.

```
#define C_WIDTH B_WIDTH
#define C_HEIGHT A_HEIGHT

void matrixMulCPU(float A[A_HEIGHT][A_WIDTH],
```

```
        float B[B_HEIGHT][B_WIDTH],
        float C[C_HEIGHT][C_WIDTH]) {
    for (i = 0; i < C_HEIGHT; i++){
        for (j = 0; j < C_WIDTH; j++){
            C[i][j] = 0;
            for (k = 0; k < A_WIDTH; k++){
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

}
```

We would like to implement this algorithm in CUDA in order to achieve better performance. The natural approach is to allocate one GPU thread for each value in the output matrix `C`, using a two-dimensional grid with two-dimensional blocks. Because each block has an upper limit of 1024 threads, we will need several blocks to cover a matrix of nontrivial size. In the diagram below, the current thread index (tx, ty) writes to position (x, y) in the output matrix. That thread needs to read all elements of row y in matrix `A` and all elements of column x in matrix `B`.

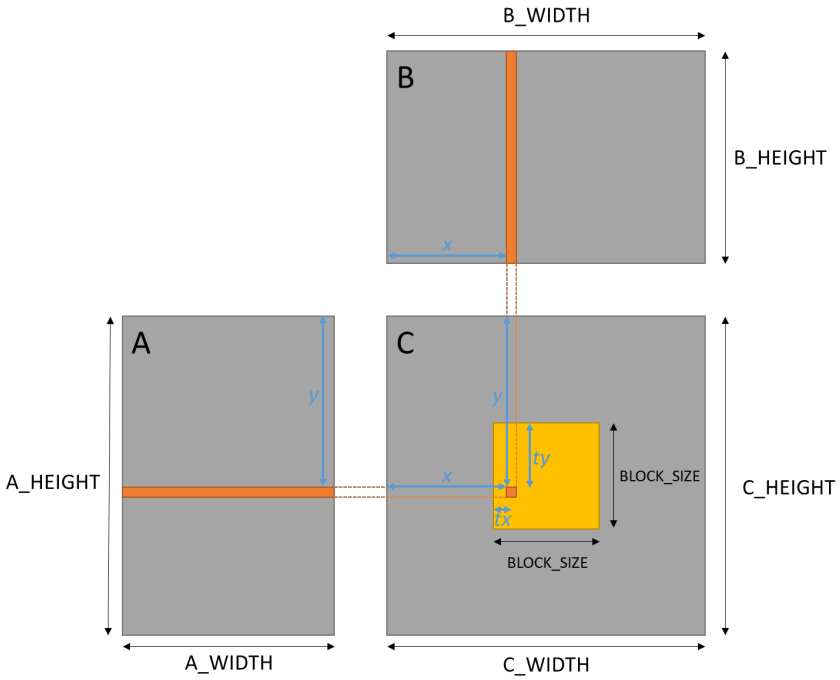

Figure 2: naive GPU matrix multiply

Your task is to implement this simple GPU matrix multiply kernel in the function `matrixMulCUDA` in the file `matrix.cu`. This function will read from arrays stored in global variables `d_A` and `d_B`,

and write its result to the global variable `d_C`. Do *not* use shared memory. Do not modify any other part of the file.

You are given starter code in `matrix.cu` that includes the CPU-based solution given above. Please read and understand the complete source code before you start coding. Use the `make` command to compile. You can run the CPU-based version with the command `make test-cpu` or `./matrix 0`. You can run your version in `matrixMulCUDA` with the command `make test-gpu-naive` or `./matrix 1`. Automated tests will be run which will notify you if your results differ from the CPU-based reference version.

4 Assignment Part 3: Matrix multiply with shared memory (40 points)

One glaring problem with the algorithm described in the previous section is that it is very inefficient with memory access: each thread performs `A_WIDTH` reads from global memory for each of the input matrices. Furthermore, these global memory reads are often repeated: every element of matrix `C` with a common `x` position will load the same entire row from matrix `A`.

We can use shared memory to minimize global memory reads. The basic idea is that threads will collaboratively load data from global memory in shared memory, where that data can be accessed by other threads.

A wrong approach Ideally, each global memory address would be read only once. So, for now, let's assume that input arrays `A` and `B` can be copied whole into shared memory. In that case, we need only one block: each thread copies into shared memory at most one value from the input arrays. Then, when both arrays have been fully copied, we can do the standard matrix multiply as in the previous section.

The following diagram shows this approach, in the special case that `BLOCK_SIZE` = `C_WIDTH` = `C_HEIGHT`.

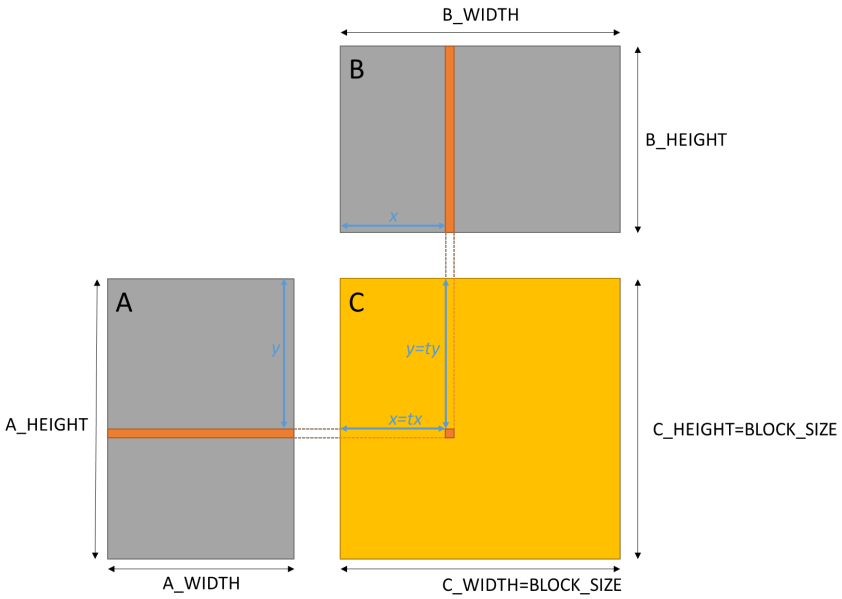

Figure 3: single-block shared-memory matrix multiply

Unfortunately, there are a few problems with this approach:

- Because both input arrays are loaded in their entirety into a single block's shared memory, we cannot use more than a single block. This limits utilized concurrency in the GPU, as only one multiprocessor will be active.
- The assumption that both input matrices will fit in shared memory holds only for small matrices. On recent GPU models, the maximum shared memory per block is 96 Kb, which would provide enough room for at most an `int` matrix of size 156×156.

So, it seems that we are forced to an alternative approach.

A better approach Instead of loading the input matrices as wholes into shared memory, we will use multiple thread blocks to process them in "chunks." We will partition the output matrix `C` into several submatrices, each of dimension `BLOCK_SIZE` $\times$ `BLOCK_SIZE`.

If we consider the data which is required by a single thread block to calculate all of its assigned outputs in matrix `C`, it will correspond to a number of square tiles from matrix `A` and matrix `B`. For example, if we consider a block size which is half of `A_WIDTH` and `B_HEIGHT`, then we can describe the problem as shown below.

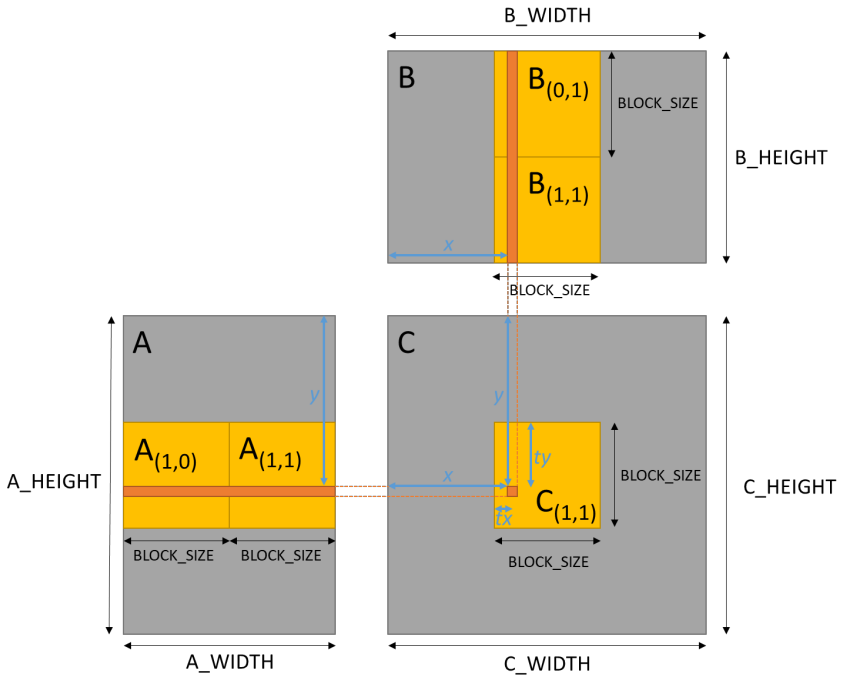

Figure 4: tiled shared-memory matrix multiply

In the diagram above, the current thread index (tx, ty) writes to position (x, y) in "chunk" $C_{(1,1)}$ in the output matrix `C`. That thread needs to read all elements of row y in matrix `A` and all elements of column x in matrix `B`, which can be found inside "chunk" $A_{(1,0)}$, $A_{(1,1)}$, $B_{(0,1)}$, and $B_{(1,1)}$. Other threads in the same block will also use some of those elements. Therefore, if the threads in the block handling chunk $C_{(1,1)}$ process the chunk in `A` from left to right, and the chunks in `B` from top to bottom, they will access all the values they need.

In order to perform the pairwise multiplications, we can consider a single chunk from each of matrix `A` and `B` in isolation and sum this with the results of other chunk calculations. That is, consider the calculation of the value of `C[y][x]`. This requires a loop (of length `BLOCK_SIZE`) through each pair of chunks ($A_{(1,0)}$ with $B_{(0,1)}$; and $A_{(1,1)}$ with $B_{(1,1)}$). After loading $A_{(1,0)}$ and $B_{(0,1)}$ into shared memory and calculating their pairwise sum of products, we will load the next pair of chunks, $A_{(1,1)}$ and $B_{(1,1)}$, and add their pairwise sum of products to the earlier sum. And so we will continue through all necessary chunks. The final sum is the new value of (tx, ty) in `C`.

In other words, the procedure is:

1. Load a chunk of matrix `A` and matrix `B` into shared memory
2. Sum the products of the pairwise elements over range of 0 to `BLOCK_SIZE`
3. Sum the resulting value with the results from other chunk pairs

Your task is to implement this tile-based GPU matrix multiply kernel in the function `matrixMulCUDATiled` in the file `matrix.cu`. This function will read from arrays stored in global

variables `d_A` and `d_B`, and write its result to the global variable `d_C`. You must use shared memory. Do not modify any other part of the file.

You are given starter code in `matrix.cu`. Use the `make` command to compile. You can run your version in `matrixMulCUDATiled` with the command `make test-gpu-tiled` or `./matrix 2`. Automated tests will be run which will notify you if your results differ from the CPU-based reference version.

Hints:

- The kernel needs enough shared memory to store one "chunk" of submatrix from each input matrix. The number of elements in a chunk is equal to the number of threads in a block, so you can statically allocate two shared memory `BLOCK_SIZE` × `BLOCK_SIZE` arrays.
- Make sure to use `__syncthreads` appropriately. You will need it after each thread copies data from global memory to shared memory, and again before moving to the next chunk.

5 Assignment Part 4: Questions (15 points)

Answer the following questions in a plain text file named `hw3.txt`.

1. Let's find out if our modification to the audio blur program actually reduced the number of global reads, even though you probably won't see a significant performance improvement. We can measure the difference in global memory reads precisely. The Nsight Compute tool lets us profile our program's performance using various metrics. Use the following command:

   ```
   ncu --metric l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum yourcommand
   ```

   where *yourcommand* is the command used to run `blur-audio`, taken from `Makefile` for that assignment.

   The command will give output like this:

   ```
   Metric Name                                          Metric Unit Metric Value
   ---------------------------------------------------- ----------- ------------
   l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum     request    NNNNNNNNN
   ```

   where `NNNNNNNNN` is the number of global memory loads performed by the kernel. Note that this number will likely be lower than the actual number of global memory accesses, because concurrent reads within a warp will be combined due to memory coalescing, which we will discuss later.

   Give your program's reported memory loads both with and without shared memory.

2. Did your shared-memory kernel get better performance than the version without shared memory? Give the timing results achieved by each version when running on ICE. How do those results compare to the CPU version?

   Please note that timing will vary between runs, so take the average of several runs as the authoritative measure for each version.

3. How many global memory reads does does each version (with and without shared memory) of the matrix multiply program perform? Explain your answer by referring to relevant portions

of the source code.

How many global memory writes does does each version (with and without shared memory) of the matrix multiply program perform? Explain your answer by referring to relevant portions of the source code.

4. In the provided code, the vertical and horizontal dimension of each block is `BLOCK_SIZE`, which initially is set to 8, meaning that the number of threads per block is $\texttt{BLOCK\_SIZE}^2 = 64$. However, this value is not necessarily optimal.

   Occupancy is defined as the ratio of actually active warps to the maximum potentially active number of warps, and can be measured with the `ncu` command, introduced above. You can query multiple metrics by passing them as a comma-separated list to the command. The following metrics can be used to calculate the occupancy of a kernel:

Metric	Description
`sm__warps_active.avg.per_cycle_active`	average number of actually active warps
`sm__warps_active.max.per_cycle_active`	maximum number of actually active warps
`sm__maximum_warps_avg_per_active_cycle`	the kernel's maximum theoretically active warps

   The number of maximum theoretical warps is determined by hardware limitations, as well as the kernel's use of registers and shared memory. The actual number of warps is measured at runtime; to the extent it differs from the theoretic ceiling, the usual limiting factor is an unbalanced workload or too few blocks. Higher occupancy doesn't necessarily lead to higher performance, but it's a good factor to consider if you want to maximize use of available resources.

   We'd like to identify the best value of `BLOCK_SIZE` experimentally, in order to maximize performance. Run your shared-memory matrix multiply program with various different configured values of `BLOCK_SIZE` and note the performance and occupancy of each. Recall that timing and occupancy will vary between runs with the same configuration, so take the average of several runs as the authoritative measure for each configuration.

   If you hardcoded numeric block size values in your solution in the shared-memory matrix multiply program, then you need to change it to use the `BLOCK_SIZE` macro instead.

   Show the results of your experiments, giving the value of `BLOCK_SIZE`, execution time, and calculated occupancy for each configuration. Specify if you use the average or maximum active warps metric. Indicate the optimal thread block size and value of `BLOCK_SIZE`.

5. We want to improve our image blur program from the previous homework by using shared memory. Within a thread block, each thread loads one pixel from global memory into shared memory. Assume we have a square *blur area*, corresponding to the portion of the image of which the average will be taken for a central pixel. When possible, the blurring code will read from shared memory, but if pixels in the blurring area fall outside those copied to shared memory, it will have to access global memory. Assume the thread block is square.

   For example, if the blur area is 1x1 (i.e. no blur), then every access can be served from shared memory. If the blur area is 3x3, then in order to calculate the average pixel value for those pixels at the edge and corners of the block, shared memory must be used.

Develop a method to calculate the proportion of shared memory accesses relative to global memory accesses. Explain your method. Express your method using mathematical notation, code, or pseudocode. Test your method by giving the proportion of shared memory accesses under each of the following circumstances:

(a) blur area 21x21, block size 16x16 (the default parameters of the original assignment)

(b) blur area 21x21, block size 32x32

(c) blur area 33x33, block size 32x32

(d) blur area 33x33, block size 1x1

Hint: for an arbitrary pixel in the image, the total number of memory accesses must be equal to the number of pixels in the blur area. Some of those accesses may be serviced from shared memory, depending on (a) the blur area, (b) the block size, and (c) the position of the pixel within the current block's shared memory.

6 Rules

6.1 Development environment

Your code will be tested in our ICE environment, as described in the syllabus. Please test your code in that environment before submission. Your code must work with the operating system and CUDA development tool chain installed there.

You may use the CUDA libraries, as described in the ICE tutorial; as well as standard libraries that form part of the compiler. Do not use any other libraries. Do not use CUDA libraries or features that have not yet been introduced in this course.

6.2 Evaluation

Your submitted work will be evaluated on the following metrics, among others:

- your submission's fulfillment of the goals and requirements of the assignment. That is, does your program work?
- your submission's correct application of the techniques and principles studied in this course. That is, does your program demonstrate your understanding of the course material?
- your submission's performance characteristics. That is, does your program run fast enough?

Failure to satisfy the assignment's requirements, to apply relevant techniques, or to meet performance expectations may result in a grade penalty.

Only work submitted before the due date will be evaluated. Exceptions to this policy for unusual circumstances are described in the syllabus.

Ensure that your program compiles and runs without errors or warnings in the course's development environment. If your code cannot be compiled and run, it will not be graded.

In addition, you are obligated to adhere to the stylistic conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables, parameters, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- Where appropriate, code should be commented to describe its purpose and method of operation. If your undocumented code is so complex or obscure that the grader can't understand it, a grade penalty may be applied.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in assigning your grade.

6.3 Academic integrity

You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else, including code written by other students. Code may not be developed collaboratively. Do not use any artificial intelligence resource at any point in your completion of this assignment. Please read the course syllabus for detailed rules and examples about academic integrity.

7 Submission

Submit only your completed `audioblur.cu`, `matrix.cu`, and `hw3.txt` files on Gradescope. Do not submit binary executable files. Do not submit data files.

7.1 Acknowledgements

This assignment based on work by Paul Richmond.