Com 4521 Parallel Computing with GPUs: Lab 06

Spring Semester 2022

Author: Dr Paul Richmond

Lab Assistants: Robert Chisholm, Jack Ashurst, John Charlton

Department of Computer Science, University of Sheffield

Learning Outcomes

- Understand how to use shared memory to improve performance of a memory bound algorithm (matrix multiply)
- Understand how to use tiling in shared memory for matrix multiply
- Understand how to manually calculate occupancy of a kernel at runtime
- Demonstrate how to automatically discover the thread block size which achieves highest occupancy

Prerequisites

Download CUDA occupancy calculator spreadsheet from the NVIDIA website.

https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls

Exercise 01

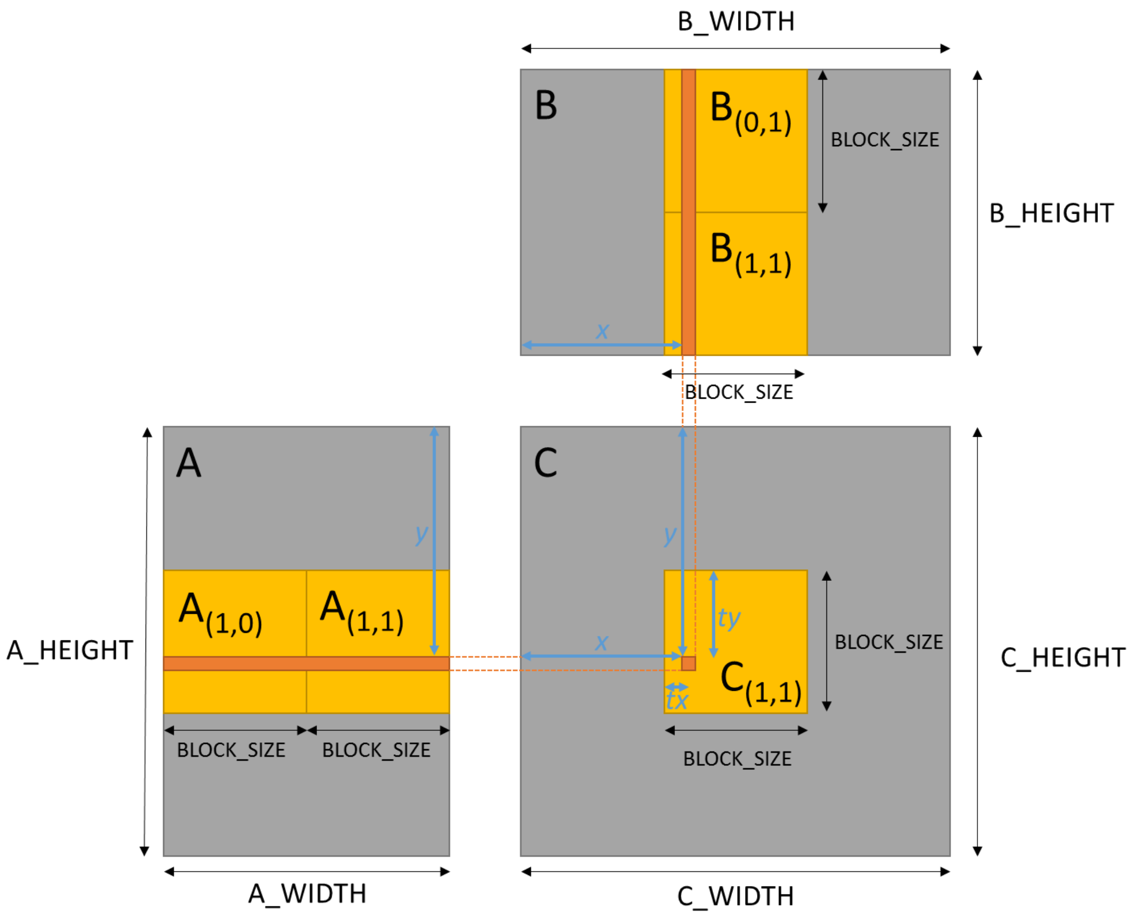

Figure 1 - Blocked matrix multiply (See text for description). Note: indices of sub block are given as (y, x)

For exercise one we are going to modify an implementation of matrix multiply (provided in exercise01_rows.cu). The implementation provided multiplies a Matrix A by a Matrix B to produce a Matrix C. The widths and heights of the matrices can be modified but for simplicity must be a factor of the `BLOCK_SIZE` (which has been predefined as a macro in the code). The implementation is currently very inefficient as it performs `A_WIDTH × B_HEIGHT` memory loads to compute each product (value in matrix C). To improve this we will implement a blocked matrix multiply which uses CUDAs shared memory to reduce the number of memory reads by a factor of `BLOCK_SIZE`. First note the performance of the original version and then modify the existing code to perform a blocked matrix multiply by implementing the following changes.

To implement a blocked matrix multiply will require that we load `NUM_SUBS` square sub matrices of the matrix A and B into shared memory so that we can compute the intermediate result of the sub matrix products. In the example Figure 1 (above) the sub matrix $C_{(1,1)}$ can be calculated by a square thread block of `BLOCK_SIZE` by `BLOCK_SIZE` where each thread (with location `tx`, `ty` in the square thread block) performs the following steps which requires two stages of loading matrix tiles into shared memory.

1. Load the two sub-matrices $A_{(1,0)}$ and $B_{(0,1)}$ into shared memory. For these sub matrices each thread should
   a. Load an element of the sub matrix $A_{(1,0)}$ into shared memory from the matrix A at position `(ty+BLOCK_SIZE, tx)`

    b. Each thread should load an element of the sub matrix $B_{(0,1)}$ into shared memory from the matrix B at position `(ty, tx+BLOCK_SIZE)`
    c. Synchronise to ensure all threads have completed loading sub matrix values to shared memory
2. Compute the dot product of each row in sub-matrix $A_{(1,0)}$ with each column in the sub-matrix $B_{(0,1)}$, storing the result in a local variable. Achieved through the following steps.
    a. Iterate from `0` to `BLOCK_SIZE` to multiply row `ty` of $A_{(1,0)}$ (from shared memory) by column `tx` of $B_{(0,1)}$ (from shared memory) to calculate the sub matrix product value.
    b. Store the sub matrix product value in a thread local variable.
    c. Synchronise to ensure that all threads have finished reading from shared memory.
3. Repeat steps 1 & 2 for the next sub-matrix (or matrices in the general case), adding each new dot product result to the previous result. For the example in the figure, there is only one more iteration required to load the final 2 sub matrices. E.g.
    a. Load an element of the sub matrix $A_{(1,1)}$ into shared memory from Matrix A at position (`ty+BLOCK_SIZE`, `tx+BLOCK_SIZE`)
    b. Load an element of the sub matrix $B_{(1,1)}$ into shared memory from matrix B at position (`ty+BLOCK_SIZE tx+BLOCK_SIZE`,)
    c. Synchronise to ensure all threads have completed loading sub matrix values to shared memory
    d. Iterate from `0` to `BLOCK_SIZE` to multiply row `tx` of $A_{(1,1)}$ (from shared memory) by column ty of $B_{(1,1)}$ (from shared memory) to calculate the sub matrix product value.
    e. Add this sub matrix product value to the one calculated for the previous sub matrices.
4. Store the sum of the sub-matrix dot products into global memory at location `x`, `y` of Matrix C.

Following the approach described for the example in Figure 1 modify the code where it is marked TODO to support the general case (any sizes of A and B which are a multiple of `BLOCK_SIZE`). Test and benchmark your code compared to the original version. Do not worry about shared memory bank conflicts until you have implemented your code (it is unlikely that you have introduced any). Once you have a working solution see if you can understand what the stride of your accesses is to understand why your memory access pattern is conflict free. Any shared memory bank conflicts can be resolved by either inverting the storage of your shared memory tiles or by padding your shared memory storage.

Note: When building in release mode the CUDA compiler often fuses a multiply and add instruction which causes an improvement in performance but some small loss of accuracy. To ensure that you test passes you should either

1. Implement a small epsilon value and compare the difference of the host and device version to this to ensure that it is within an acceptable range.
2. Modify the `nvcc` flag to avoid the use of fused multiply and add with `–fmad=false`

Exercise 01 Part 2

We would like to know the occupancy of the matrix multiply example. We can calculate the theoretical occupancy through the following formula (which will not include the effect of any limitations of the availability of shared memory);

$$Occupancy = \frac{maxBlocksPerMultiProcessor * \ TPB}{maxThreadsPerMultiProcessors * multiProcessorCount}$$

Where TPB is threads per block and $maxBlocksPerMultiProcessor$ and $maxThreadsPerMultiProcessors$ can be queried from the device properties.

2.1 Calculate the occupancy of the kernel and print it to the console.
2.2 Modify the build arguments to output the register and shared memory usage. You can now calculate the occupancy by considering the impact of the limited amount of shared memory per multi processors within the CUDA_Occupancy_Calculator.xls spread sheet. Does this differ from what you have calculated?

Exercise 02

It would benefit performance to achieve optimal occupancy in our matrix multiplication code. We can do this using the CUDA Occupancy API however our code has a statically defined block size. Make a copy of your code from the previous exercise and make the following changes.

2.1 Implement a function (`requiredSM`) to dynamically calculate shared memory usage based on a given block size. For now use the exiting `BLOCK_SIZE` macro.
2.2 Modify the kernel launch to dynamically allocate shared memory for the kernel. Within your kernel you will need to create pointers to the A and B submatrix as the kernel launch will have allocated a single linear block. You will also need to update the indexing of the sub matrices as your shared memory sub matrices are now a 1D array. Test your code using the existing `BLOCK_SIZE` to ensure you have not introduced any errors.
2.3 Modify the macro `BLOCK_SIZE` so that it is instead a variable in CUDA constant memory. Calculate the `BLOCK_SIZE` which gives best theoretical occupancy by implementing the following;
    2.3.1 Use the `cudaOccupancyMaxPotentialBlockSizeVariableSMem` function to find the thread block size with best occupancy. See API docs for more details.
    2.3.2 Apply the following formula to ensure that the square root of the thread block size is an integer (i.e. the root has no decimal part) and also a power of two number.

$$ThreadsPerBlock = 4^{floor\left(\frac{\log_e(ThreadsPerBlock)}{\log_e(4)}\right)}$$

    2.3.3 Calculate the block size using `sqrt` function
    2.3.4 Copy the block size to constant memory.
2.4 Modify your code to print the block size with the highest occupancy to the console.
2.5 Test your code and then build and execute in release mode to compare with the original version with lower occupancy.