National University of Singapore
School of Computing

CS3210 Assignment 3

Parallel Matrix Multiplication

using NVIDIA's CUDA

Syed Abdullah

November 20, 2017

Contents

1 Optimising through Shared Memory **2**
1.1 Overview . . . 2
1.2 Implementation Walkthrough . . . 2
1.2.1 `mm-cuda.cu` – Naive Implementation . . . 2
1.2.2 `mm-sm.cu` – Shared Memory Optimisation . . . 2
1.3 Results & Analysis . . . 4
1.3.1 Tabulation & Calculations . . . 4
1.3.2 Analysis . . . 4

2 Memory Banks Optimisation **5**
2.1 Problem Analysis & Implementation . . . 5
2.1.1 Bank Conflict in `mm-sm` . . . 5
2.1.2 Implementation . . . 6
2.2 Results & Analysis . . . 6

Chapter 1

Optimising through Shared Memory

1.1 Overview

In this section, we will be investigating the potential use of shared memory in `mm-cuda.cu`, so as to achieve a speedup in execution time.

1.2 Implementation Walkthrough

We have to first understand the rationale behind the implementation before we delve into the performance results and analysis.

1.2.1 `mm-cuda.cu` – Naive Implementation

In the naive CUDA implementation, for a $n \times n$ matrix, we will use a $n/32 \times n/32$-sized grid containing 32×32-sized blocks. Each block is actually a data decomposition of the $n \times n$ output matrix, C, like so:

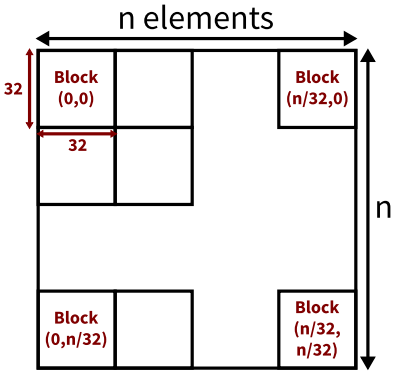

Figure 1.1: "Mapping" of block to output matrix C

In the `mm_kernel` method, the matrix multiplication that is performed is still similar to the sequential version. However, the thing to note is that one thread executes the multiplication for one output cell (i.e. does one row from A and one column from matrix B).

1.2.2 `mm-sm.cu` – Shared Memory Optimisation

The issue with the naive implementation is that the two input matrices are stored in the device's global memory. As such, when threads in a warp try to access the elements in the input matrices, there is a delay as the memory is located outside of the streaming multiprocessor itself. While this 'cost' is expected, the fact that the same input matrix elements are accessed more than once by multiple threads means that the 'cost' is not justified for subsequent accesses.

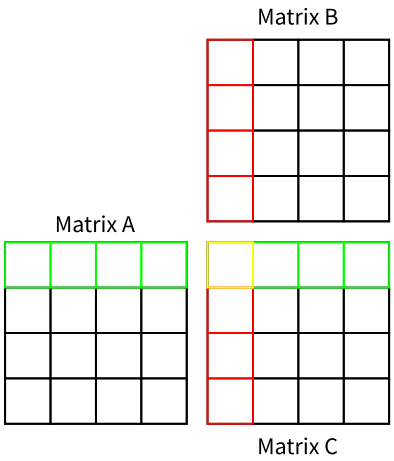

Figure 1.2: Memory reads on A and B for cells in C

In order to improve this communication time, we can load the appropriate matrices (in this case, A and B) onto the shared memory of the streaming multiprocessor. By doing so, we are putting the data "closer" to the core itself and thus, the data latency would be reduced for subsequent memory accesses.

Memory Size Constraints

While the shared memory is fast, it is also of a limited size, which is smaller than the global memory (that is in device DRAM). Furthermore, to simplify the code (i.e. avoid the use of dynamic memory allocation), we would want to use a fixed dimension for the matrix in the shared memory. Thus, we can define the size of each shared memory matrix to be $b \times b$, where b is the size of one dimension of the block (currently 32).

Implementation

```
1  __shared__ As[b][b]
2  __shared__ Bs[b][b]
3
4  C_result = 0.0 // cell result in C for thread
5
6  for m ← 0 to n (increment by b):
7      load A[y][m + x] & B[m + y][x] → As[y][x] & Bs[y][x]
8      wait for all threads
9      for k ← 0 to b:
10         C_result += As[y][k] + Bs[k][x]
11     end
12     wait for all threads
13     C[y][x] ← C_result
14 end
```

Figure 1.3: Sketch of implementation

As the input matrix will most likely be larger than $b \times b$ ($b = 32$), we would utilise this shared memory matrix as in the form of a rolling buffer. Thus, for each input matrix, we can divide it into chunks of size $n/b \times n/b$ (where n is the size of the input matrix). Each chunk gets loaded into the shared memory and then, a partial result of C is calculated.

By doing this, we reduce the number of times a matrix element gets loaded from global memory from b times (because each element in A and B is used to calculate a row/column (of size b) of C) to only 1 per warp.

1.3 Results & Analysis

The results were obtained by running `mm-seq` (without CUDA), `mm-cuda` and `mm-sm` with matrix sizes 512, 1024 and 2048 on the Jetson TK1 board. The experiment was run five times and the minimum execution time was selected, so as to minimize non-program related execution in our measurements (e.g. process swapped out).

1.3.1 Tabulation & Calculations

512			**1024**			**2048**		
`mm-seq`	`mm-cuda`	`mm-sm`	`mm-seq`	`mm-cuda`	`mm-sm`	`mm-seq`	`mm-cuda`	`mm-sm`
0.63	12.83	0.19	32.06	18.52	0.27	434.29	99.69	1.86

Table 1.1: Execution times (in seconds) for `mm-seq`, `mm-cuda` and `mm-sm`

Relative to	**512**			**1024**			**2048**		
	`mm-seq`	`mm-cuda`	`mm-sm`	`mm-seq`	`mm-cuda`	`mm-sm`	`mm-seq`	`mm-cuda`	`mm-sm`
`mm-seq`	1.00	0.05	3.32	1.00	1.73	118.74	1.00	4.36	233.49
`mm-cuda`	N.S.[1]	1.00	67.52	N/A	1.00	68.59	N.S.	1.00	53.59

Table 1.2: Execution time speedups for `mm-seq`, `mm-cuda` and `mm-sm`

1.3.2 Analysis

As we can see from the above calculations, the shared memory implementation allows us to achieve an execution time speedup of around 60, relative to the execution time of `mm-cuda`. This is consistent to our hypothesis that the global memory accesses are slower than shared memory accesses.

However, this speedup (between `mm-cuda` and `mm-sm`) is somewhat constant even with increasing matrix sizes, unlike the increasing speedup between `mm-seq` and `mm-cuda` with increasing matrix size. This is expected as the memory accesses are strongly correlated to the number of blocks, which in turn, is defined in our program as a part of the resultant matrix. Thus, with an increased matrix size, there would be more blocks that needs to be executed on one SM (Jetson only has 1 SM), which would also mean that more values need to be loaded from memory. With that correlation, the percentage of total execution time spent on memory accesses largely remains the same.

[1] Not significant

Chapter 2

Memory Banks Optimisation

In this chapter, we will be investigating the effects of bank conflicts in the `mm-sm` version of the matrix multiplication implementation.

2.1 Problem Analysis & Implementation

Bank conflicts occur when more than one thread in a half warp tries to access the same shared memory bank – thus, serialising the access to the particular memory location. As this project utilises Compute Capability 3.X, there are 32 32-bit shared memory banks.[1] Since our matrix is stored as a `float` (32-bit value), each element in the matrix is thus mapped to one memory bank.

2.1.1 Bank Conflict in `mm-sm`

In CUDA, a 2D shared memory array is assigned to banks in a row-major order. In order to investigate the possible bank conflict in the program, we can look at the memory accesses for a single iteration of the `mm-sm` program. The memory accesses are done by half warps. Partitioning of blocks into warps is done in a deterministic manner: consecutive, increasing thread IDs, starting from thread 0 in the first warp.[2] Thread IDs are, in turn, assigned in the following manner: $x + y \times Dx$, for the thread with index (x, y)[3]

As such, in the case of `mm-sm`, each row in the resultant matrix C is processed as a single warp. Thus, the memory accesses for the first iteration of the calculation loop executed in the warp is illustrated in the following diagram:

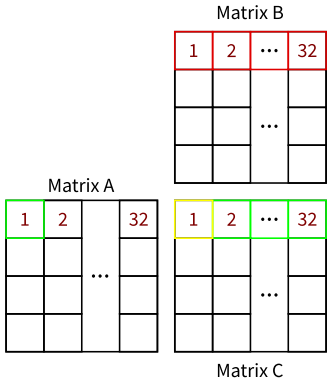

Figure 2.1: Memory access on a single iteration

While this may not suggest a memory bank conflict, as different banks in matrix B is being accessed, while in matrix A, it can be done through broadcast, the author postulates that the firmware on the Jetson board

[1] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-3-0`
[2] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture`
[3] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`

may not support the broadcast operation. Thus, in this case, bank conflict would occur on matrix A.

2.1.2 Implementation

To resolve this issue, we have to find a way to stagger the accesses such that there's no bank conflict. This is exactly the implementation that was used in `mm-banks`. The sketch of the implementation is presented below:

```
1  __shared__ As[b][2*b]
2  __shared__ Bs[2*b][b]
3
4  C_result = 0.0 // cell result in C for thread
5
6  for m ← 0 to n (increment by b):
7      load A[y][m + x] & B[m + y][x] → As[y][x] & Bs[y][x]
8      load A[y][m + x] & B[m + y][x] → As[y][x + b] & Bs[y + b][x]
9      wait for all threads
10     for k ← x to b+x:
11         C_result += As[y][k] + Bs[k][x]
12     end
13     wait for all threads
14     C[y][x] ← C_result
15 end
```

Figure 2.2: Sketch of implementation

As we can see above, we offset the accesses on matrix A, such that each thread in the half-warp accesses a different column in matrix A. However, in order to ensure that the matrix multiplication is correct, we shall also stagger the accesses on matrix B (due to the nature of matrix multiplication). Furthermore, in order to improve execution times, the buffer matrices of A and B are doubled, so that there's no need to perform modulo calculation for every iteration (which is expensive).

2.2 Results & Analysis

In order to analyse the shared memory bank conflicts, `nvprof` was executed and the value of the `shared_load_replay` event was collected. In particular: `nvprof --events shared_load_replay <program name>`.

mm-sm	mm-banks
8,388,608	0

Table 2.1: `shared_load_replay` event results

We can see that our implementation, `mm-banks`, eliminates all bank conflicts that is present in the `mm-sm` version.

512			**1024**			**2048**		
mm-sm	mm-banks	Speedup	mm-sm	mm-banks	Speedup	mm-sm	mm-banks	Speedup
0.19	0.23	0.826	0.27	0.33	0.818	1.86	2.25	0.826

Table 2.2: Execution times (in seconds) and Speedup for `mm-sm` and `mm-banks`

We can observe that the no bank conflict code does not lead to a speed up. In fact, it is actually the opposite. One possible reason as to why is due to the double loading of the arrays every time the buffer gets refreshed. As there are two times the number of elements, the initial loading time will increase by a factor of two. Thus, this increase may have outweighed the speedup from resolving the bank conflicts.