Lab 03

February 2022

1 Introduction : Shared Memory

Shared memory is on-chip memory and is much faster than local and global memory. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. Since the shared memory is available to all the threads in a block there are chances of race conditions that might take place, the race condition amongst them prevented by using `__syncthreads()` which forces to stop further execution till all the threads have completed their execution within the block.

2 Code Implementation : Vector addition extended

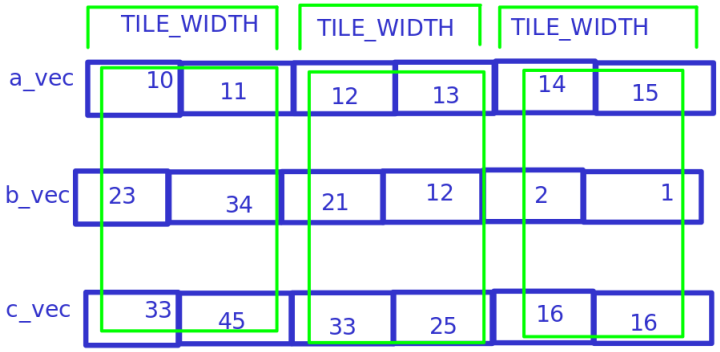
In the previous lab we performed vector addition using CUDA. In the previous implementation we performed a single addition on every thread. In this lab we will see how we can extend simple vector addition by incorporating shared memory.

Since shared memory is allocated and shared within a block of threads. There are some key things to keep in mind while using shared memory. In order to maximise performance the heavy operations are performed in shared memory. The kernel essential does 3 things, allocate shared memory, move data to shared memory , perform computation using shared memory, moving data back to GPU memory. In the case of vector addition there may not be a direct visible speedup due to addition being a simple operation, however the speedup becomes more visible in matrix multiplication.

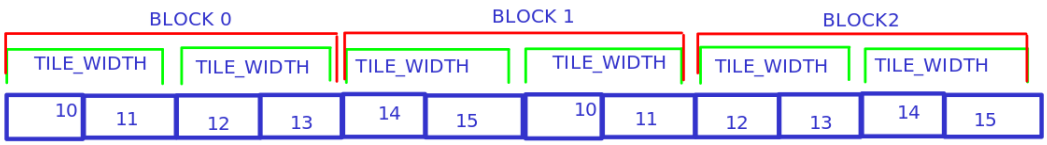
Lets take a closer look at vector addition. Say we have 3 vectors `a_vec`, `b_vec`, `c_vec`, where `a_vec` and `b_vec` are the vectors to be added and `c_vec` stores the result.

In order to perform the vector addition we split the vector into chunks of size `TILE_WITDTH` into the shared memory.

Say a_vec is of size 6 and `TILE_WIDTH` of size 2. The we create a block of containing 3 threads where each thread process `TILE_WIDTH` size of elements.




```
1       __global__ void vector_add(float *a, float *b, float *c){
2           __shared__ int a_[TILE_WIDTH];
3           __shared__ int b_[TILE_WIDTH];
4
5           int num_tiles =  LENGTH/(TILE_WIDTH*TILE_WIDTH)+1;
6           if (blockDim.x*blockIdx.x+threadIdx.x == 0)
7                   printf("%d\n", num_tiles);
8
9           for(int j=0; j < num_tiles; j++)
10                  for(int k = 0 ; k <TILE_WIDTH; k++){
11                          int index = blockDim.x * blockIdx.x + j*TILE_WIDTH + k;
12
13                          if(index < LENGTH){
14
15                                  a_[k] = a[index];
16                                  b_[k] = b[index];
17                                  __syncthreads();
18
19                                  auto  sum = a_[k] + b_[k];
20                                  c[index] = sum;
21                          }
22                          __syncthreads();
23          }
24
25  }
```

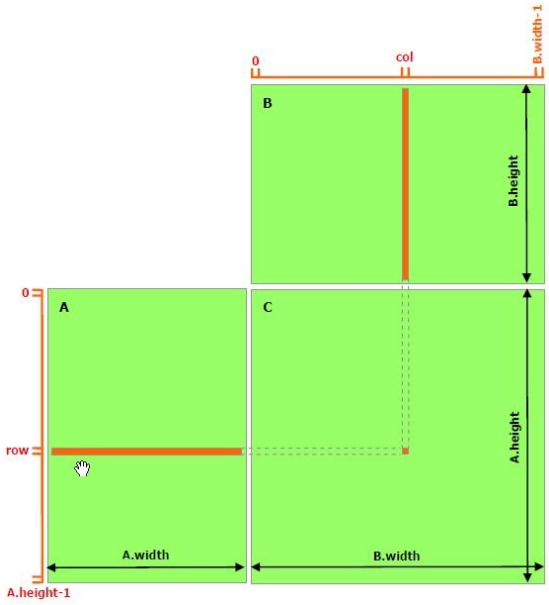


The outer loop essentially governs how many tiles each block will be working on. Line 10 maps the element within each tile to the global array index. Lines 11-17 perform addition and store it back.

3 Matrix Multiplication

3.1 Linear Addressing

An element in a 2D matrix can be accessed by either referring to the 2D coordinates or by linearizing the coordinates. This is equivalent of flattening a 2D matrix. An element at position x,y in 2D can be indexed as y*WIDTH + x, where WIDTH is number of columns. Say an element in location 0,2 would be at location 0 * WIDTH + 2 i.e. index 2 in flattened array. Similarly an element at location 1,2 would be at location 1 * WIDTH + 2, note that this is equivalent of skipping y times the total number of element in each row.



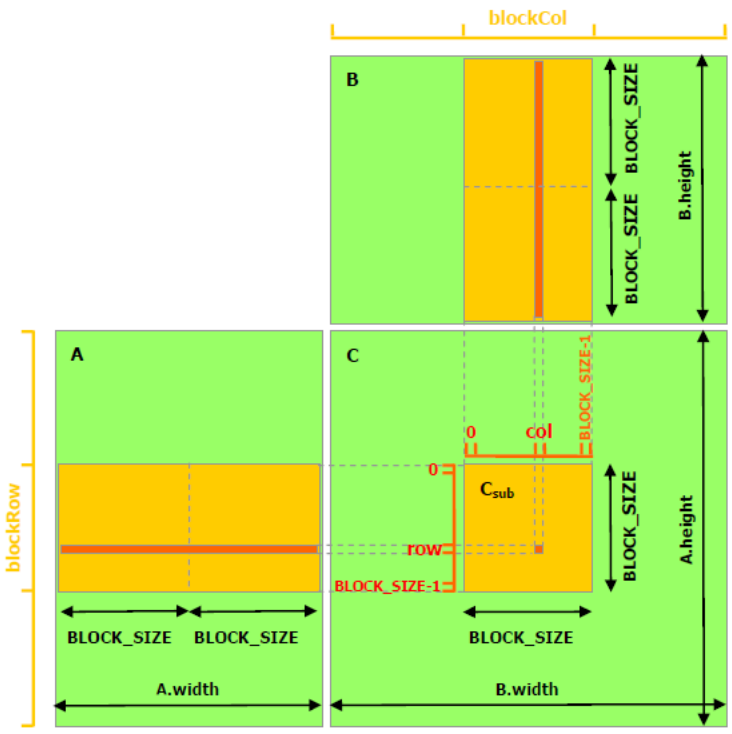
Matrix multiplication is a highly parallel operation. For each element in the resultant matrix can be computed as the sum of row-column element wise multiplication.

eg. $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} =$

$\begin{bmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix}$

This operation is highly parallel. Each element of the resultant matrix can be calculated and stored in parallel due to the independence from other elements of the resulting matrix.

Each thread of the GPU can be used to calculate each element of resultant matrix and can be further improved upon with the usage of shared memory, since shared memory allows for even faster operations. From here on BLOCK will refer to block elements loaded in shared memory and not thread block. In order to do so each multiplication can be performed in a block by block fashion. In the figure below we see how matrix multiplication can be performed in a tiled fashion using shared memory. For eg. lets take matrix of size $64 \times 64(DIM)$ and lets assume the `BLOCK_SIZE` to be 2 for now. We can create a grid of size `DIM/BLOCK_SIZE` $\times$ `DIM/BLOCK_SIZE` blocks where each block contains `DIM` $\times$ `DIM` threads. Now within each block we process matrix multiplication using shared memory. We slide a window over the row and column that we process using shared memory. We then copy the contents of original array in the shared memory, note that each element can be accessed in parallel using `threadIdx.x` & `threadIdx.y` for eg. the column can be calculated as `TILE_WIDTH * blockIdx.x + threadIdx.x`. We create the local shared memory matrices and map elements from original matrices to the shared memory. Eg. while loading an element from original matrix to shared memory can be done by
`Nds[threadIdx.y][threadIdx.x] = Nd[(m*TILE_WIDTH + threadIdx.y)*DIM + col];` where Nds is matrix thats accessing shared memory and Nd is one of the original matrix to be multiplied and m is the current `TILE/BLOCK` index.



Within the TILE the value can simply be calculated as matrix multiplication in a parallel fashion as

```
for(int k = 0; k < TILE_WIDTH; k++)
val += Mds[threadIdx.x][k]*Nds[k][threadIdx.y];
```

This value is then stored back to the result matrix.

4 Tasks

Let there be two square matrices m1 and m2 of a given size. Perform matrix multiplication in the following manner.

1. Matrix multiplication serially on CPU.
2. Simple Matrix multiplication in parallel on GPU.
3. Matrix multiplication using shared memory on GPU.

Observe the time taken for matrices of size 100, 1000, 10000.

5 Deliverables

Implement 2 different kernels for with/without shared memory in the same .cu file. Also add the code for timing the events and show the speedup graphs for the with/without shared memory implementations.

1. .cpp file for cpu implementation (matmul.cpp)
2. .cu file for GPU implementation (matmul.cu)
3. Makefile for the same

6 Upload Instructions

Upload both the files as a .zip file with the following naming convention - lab2_⟨roll_no⟩.zip

7 References

:

- https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/