Assignment 5 : Lab 4

Date	@September 26, 2022
Course	Parallel Computing
Finished?	☑
Tags	Lab

Programming Project : Transpose Experiments

Objective

- We are to write two tiled versions of the matrix transpose operation, one using global memory and one using shared memory.
  - These operations are to be launched with one thread per element, in $K \times K$ thread-blocks.
  - Our code should enable each thread in a tile to determine and write the element (i, j) of global output matrix.

Transpose of a Matrix

- In Linear Algebra, the transpose of a matrix is one of the most commonly used methods in matrix transformations.
- For a given matrix, the transpose of a matrix is obtained by interchanging rows into columns or columns to rows.

Matrix in C

- Matrices are stored as one dimensional arrays in CUDA/C in row-major order and are indexed as $i + j * N$ for the element $A[i, j]$, where N is the number of rows.

Starter Code

- The starter code provided for this assignment consists of three ways to transpose a given Matrix.
  - Transposing a Matrix using only CPU

        - We will utilize the output matrix obtained from this method to check and verify the correctness of the consequent methods that involve CUDA.
    - Transposing a Matrix using CUDA launched on a single thread
        - This method utilizes the GPU instead of the CPU. However, the entire operation is carried out by only a single thread.
        - The Matrix is stored in global memory and the read and writes from the `input` to the `output` matrix are sequential.
    - Transposing a Matrix using CUDA launched on one thread per row
        - This method instead of launching a kernel on a single thread, launches a kernel on a Block of size N. Where each thread will execute on a single row of the given matrix.
        - The matrix is stored in global memory and the read and writes from the `input` to the `output` matrix are sequential over each row. However, since the N threads are being executed in parallel, this method does speed-up the overall computation.
- However, there is still a lot of gain that can be achieved through tiling the original matrix and is discussed further as below.

TILED version

- By tiling, we simply mean that the entire original matrix of size $N \times N$ is divided into smaller $K \times K$ mini-chunks called `tiles`.
- Here, the original matrix is a square matrix of size 1024×1024.
- The tile size for this matrix is chosen to be 32×32.
- `gridDim` is kept 32×32 → Each block representing a single Tile.
- `blockDim` is kept 32×8 →Mapping each thread to a single entry of the matrix at any arbitrary location (x, y).

Non-shared Memory version (Global Memory)

- This kernel is launched in a non-shared memory fashion meaning we are still accessing the memory of the original matrix from the global (slower) memory.
- Through the tiling of the original matrix, we are able to achieve coalesced reads but the writes are still not coalesced.
- Since, we kept the `blockDim.y` equal to 8 instead of 32. We need to loop over the rest of the matrix in each tile (block). The `for` loop is for this purpose. It is also kept in the shared memory version for a similar reason.
- A code-block for this kernel is shown as under. Where, `TILE_DIM` is 32 and `BLOCK_ROWS` is 8.

```
// launched in tile fashion non-shared memory
    // doesn't use shared memory
    // Global memory reads are coalesced but writes are not
    // 1 Tile = 1 Block in the Grid
__global__ void transpose_tiled(float in[], float out[])
{
    int x = threadIdx.x + blockIdx.x * TILE_DIM;
    int y = threadIdx.y + blockIdx.y * TILE_DIM;

    for (int j=0; j < TILE_DIM; j+= BLOCK_ROWS)
        out[x*N + (y+j)] = in[(y+j)*N + x];
}
```

kernel launched in tile fashion using non-shared memory

Shared Memory version

- Here, we define a two-dimensional array called `tile` of dimension `TILE_DIM=32`.
- This `tile` variable is a shared variable → i.e. Each block among the 32×32 blocks will have it's own `tile` variable which is shared.
- Then from the global memory we need to read the entire original matrix into this shared variable.
- This read is a coalesced read.
- To avoid race conditions, we need to synchronize all the threads that are reading the global memory into the shared variable.
    - `__syncthreads();` command does exactly that. It acts as a barrier for threads that are finished before others and makes them wait for all thread completion before further instructions are executed.
- After all the threads are synced, we have to offset the block with new `x` and `y` indices.
- Now, we can simply flip the indices of row and column from the shared variable `tile` and write them in the output variable `out`.
- This write is now also a coalesced write.
- A code-block for this kernel is shown as under. Where, `TILE_DIM` is 32 and `BLOCK_ROWS` is 8.

```
// launched in tile fashion shared memory
    // Uses shared memory to achieve coalesign in both reads and writes
    // Uses the fact that tile.T can be mapped to Matrix.T
__global__ void transpose_tiled_shared(float in[], float out[])
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int x = threadIdx.x + blockIdx.x * TILE_DIM;
    int y = threadIdx.y + blockIdx.y * TILE_DIM;

    // read global memory -> tiles
    for (int j=0; j < TILE_DIM; j+= BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = in[(y+j)*N + x];

    // like the name suggests - sync threads
        // to avoid race conditions
    __syncthreads();

    x = threadIdx.x + blockIdx.y * TILE_DIM;
    y = threadIdx.y + blockIdx.x * TILE_DIM;

    for (int j=0; j < TILE_DIM; j+= BLOCK_ROWS)
        out[(y+j)*N + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

kernel launched in tile fashion using shared memory

Results

- The execution time for each method is given in the Table 1.

Index	Single Thread	One Thread Per Row	Tile : Non-shared Memory	Tile : Shared Memory	Numba + Cuda JIT
1	82.709 ms	2.38746 ms	0.148736 ms	0.083296 ms	62.98279 ms

- As it is visible from the Table 1, using the `shared memory` is the fastest way to transpose the given matrix.
- Also, using python with `numba` and `cuda.jit` renders the execution time of 62.98 ms which is faster than the Single Thread execution but still slower than all other CUDA/C execution.
    - `Note` : For all above executions, the original matrix is of size 1024×1024.