UC San Diego

ECE 277 Project Presentation

GPU Accelerated Common Matrix Operations with CUDA and Pybind

GPU Accelerated Common Matrix Operations with CUDA and Pybind

- GPU is good at parallel computing such as matrix computation.
- This project creates a Python library and bind it with C++ using Pybind.
- Users can implement common matrix operations such as addition and multiplication on Python with GPU acceleration.
- The project only focuses on 2D matrices

GPU Accelerated Common Matrix Operations with CUDA and Pybind

User	Python	C++	CUDA
Input matrix and select operation type	Pybind	Pass data to GPU	Calculate with GPU
Get the result		Receive data from GPU	Return the result

UC San Diego

Operation Types

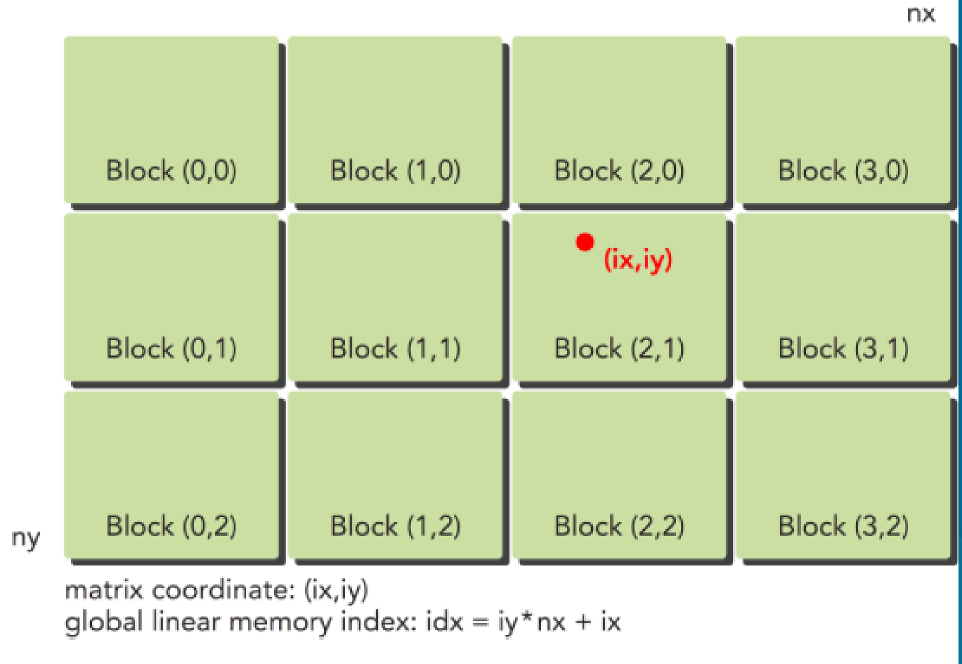
1. Addition
2. Subtraction
3. Multiplication
4. Transposition

*The project only focuses on 2D and integer matrices.

Addition & Subtraction

2D grid and 2D block partition

- Map one thread to one matrix element
- One threadblock processes a sub-block of a matrix (multiple columns and multiple rows)



Multiplication

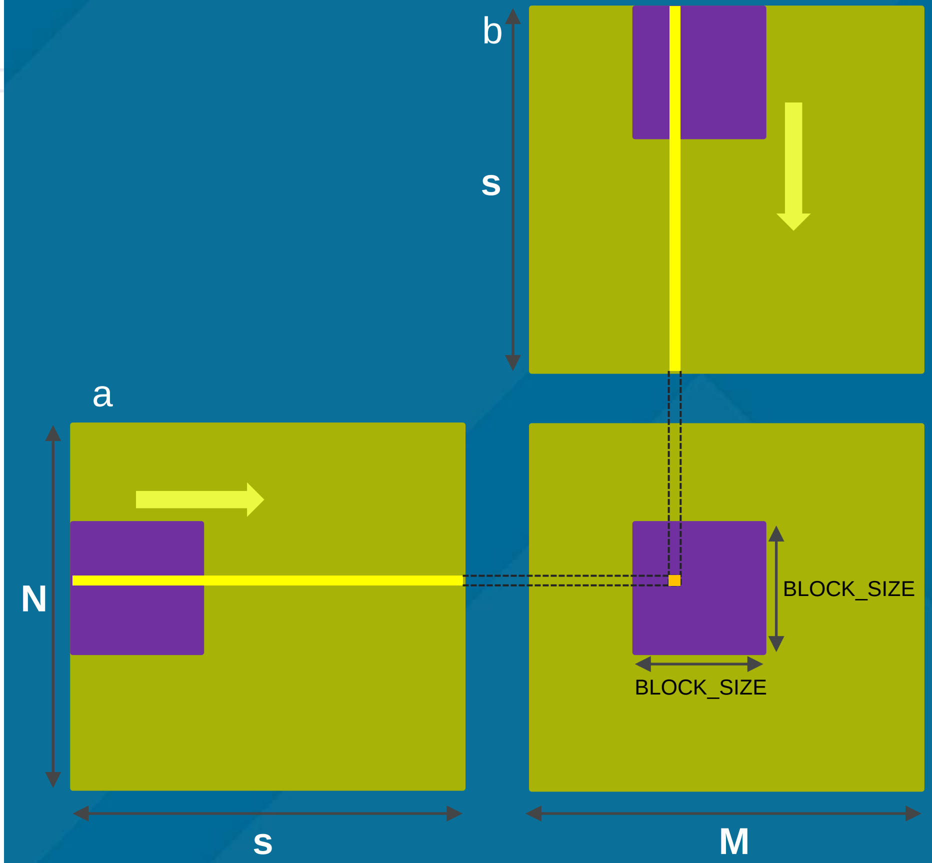
Use Shared Memory (SMEM) to accelerate the computation

- If we don’t use SMEM, the program will access Global Memory (GMEM) multiple times during execution. Since the latency of GMEM is relatively high, the efficiency of code execution will be badly reduced.
- We introduced SMEM for optimization. We mainly use the Write Once/ Read Many (WO/RM) of SMEM to reduce the latency during data transferring.

Multiplication

Matrix a multiplies Matrix b

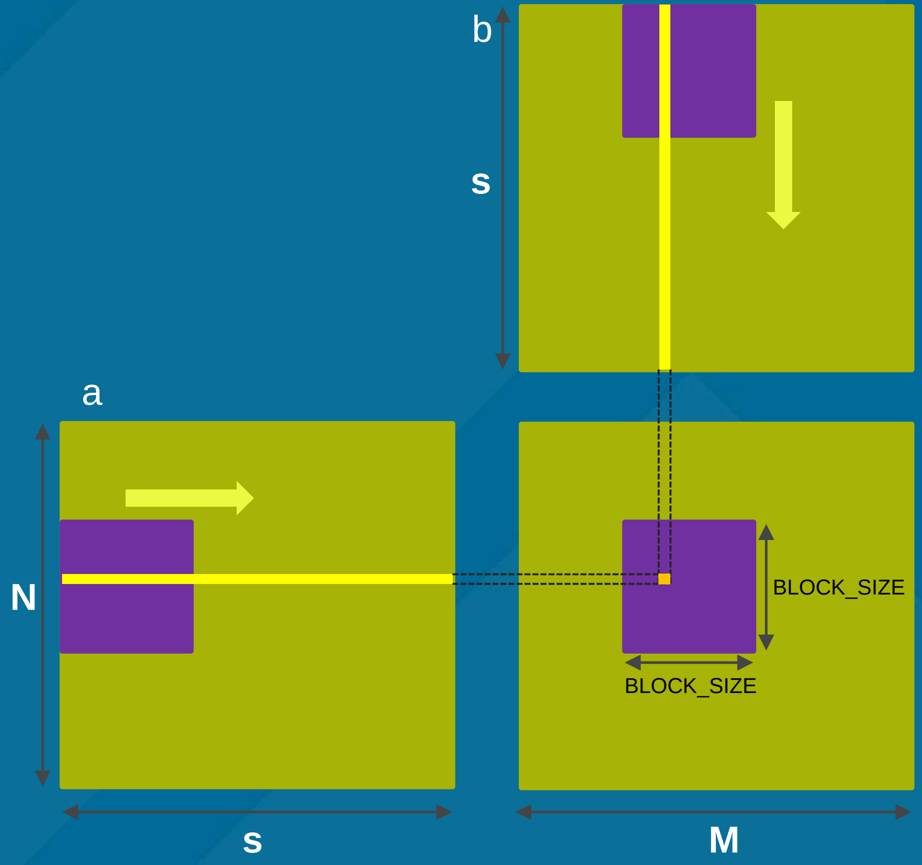
```
__global__ void kernel_mmul(int* A, int* B, int* C, int M, int N, int s, dim3 blk)
{
    __shared__ int smem_m[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ int smem_n[BLOCK_SIZE][BLOCK_SIZE];
    unsigned int row = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int col = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int tmp = 0;

    if (row + col == 0)
        printf("cuda matrix ((%d, %d)(%d, %d)) multiplication\n", N, s, s, M);

    for (int stride = 0; stride <= s / blk.y; stride++) {
        int idm = stride * blk.y + row * s + threadIdx.x;
        if (row < N && blk.y * stride + threadIdx.x < s) {
            smem_m[threadIdx.y][threadIdx.x] = A[idm];
        }
        else {
            smem_m[threadIdx.y][threadIdx.x] = 0;
        }
        int idn = stride * blk.y * M + col + threadIdx.y * M;
        if (col < M && blk.y * stride + threadIdx.y < s) {
            smem_n[threadIdx.y][threadIdx.x] = B[idn];
        }
        else {
            smem_n[threadIdx.y][threadIdx.x] = 0;
        }
        __syncthreads();
        for (int i = 0; i < blk.y; i++) {
            tmp += smem_m[threadIdx.y][i] * smem_n[i][threadIdx.x];
        }
        __syncthreads();
    }
    if (row < N && col < M)
    {
        C[row * M + col] = tmp;
    }
}
```



Multiplication

Matrix a multiplies Matrix b

1. First, we define two tiles that exist in SMEM. The size of the square tile is the size of the current Block.
2. In the kernel function, each thread plays two roles: 1. Assign the data in the GMEM to the SMEM; 2. Calculate the value in the result matrix.
3. My codes use the method of moving tiles. While copying and calculating, tiles are moved towards certain direction. As shown in the figure, one tile will move to the positive direction of the x-axis of Matrix a, and the other tile will move to the positive direction of the y-axis of Matrix b, and the step size is the side length of the tile.

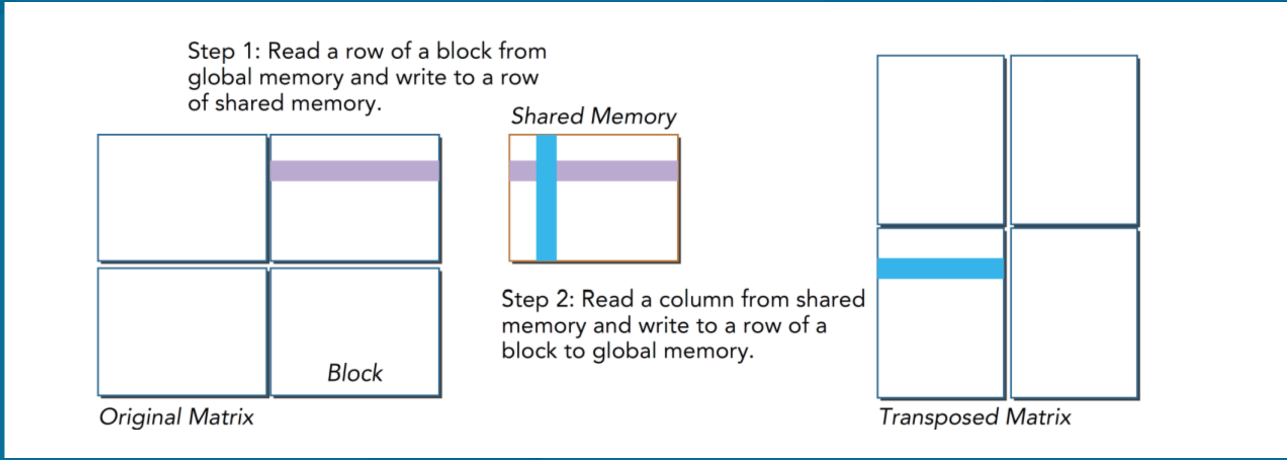


Transposition

Transpose with SMEM

1. Row-wise GMEM read, Row-wise SMEM write
2. Column-wise SMEM read, Row-wise GMEM write

Both GMEM read and write can be coalesced access so we can make use of it to accelerate the operation.



Test Result

Addition

```
M=10240, N=10240
cuda matrix (10240, 10240) addition
Pass
Python Time: 97789.59120000001 ms
GPU Time: 513.2112000000006 ms
```

Subtraction

```
M=10240, N=10240
cuda matrix (10240, 10240) subtraction
Pass
Python Time: 107122.3083 ms
GPU Time: 488.4299999999939 ms
```

Multiplication

```
M=256, N=256
cuda matrix ((256, 128)(128, 256)) multiplication
Pass
Python Time: 5290.227 ms
GPU Time: 185.84150000000045 ms
```

Transposition

```
M=10240, N=5120
cuda matrix (5120, 10240) transposition
Pass
Python Time: 29242.1385 ms
GPU Time: 323.91989999999999 ms_
```

UC San Diego