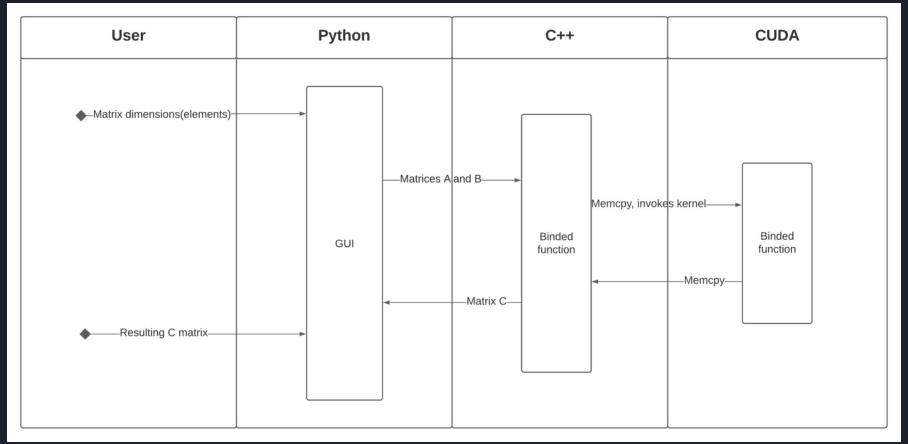
ECE277 Final Project Presentation

Henrik Larsson Hestnes

GPU-accelerated matrix-matrix multiply in Python

Proposed project

- Simple GUI in Python
  - Specify dimensions of matrices
  - Possibility to set the elements of the vector
- Use Pybind11 to bind a C++ function which invokes the kernel which performs the multiplication via shared memory on the device
- Possibility to print the result to the GUI



Sample of code for Python GUI

```python
import numpy as np
import sys
sys.path.append('build')
import gpu_library

print("\nYou have two matrices A(MxN) and B(NxK) that you want to multiply.")
M, N, K = input("Enter their dimensions seperated by comma(M, N, K): ").split(",")
M, N, K = int(M), int(N), int(K)

A = np.random.rand(M, N)
B = np.random.rand(N, K)

C_TRUE = A @ B

C_GPU_SHARED = np.zeros(M*K)
gpu_library.cuda_shared_matrix_multiply(A.reshape(M*N), B.reshape(N*K), C_GPU_SHARED, M, N, K)

if np.allclose(C_GPU_SHARED.reshape(M, K),C_TRUE):
    SEE_RESULT = input("Do you want to see the result?(y/n) ").lower()
    if SEE_RESULT == 'y':
        print(f"\nResult: \n{C_GPU_SHARED.reshape(M, K)}")
else:
    print(f"CUDA encountered a problem, so the resulting matrix is wrong")
```

Pybind11 bindings

```
PYBIND11_MODULE(gpu_library, m){
    m.doc() = "Library for doing GPU accelerated matrix multiply in Python";
    m.def("cuda_global_matrix_multiply", &global_matmul);
    m.def("cuda_shared_matrix_multiply", &shared_matmul);
    m.def("cpu_matrix_multiply", &cpu_matmul);
}
```

Binded C++ function - part 1

```
void gpu_matmul(const py::array_t<const double> a, const py::array_t<const double> b, py::array_t<double> c,
                int M, int N, int K, MEM_TYPE memory){

    unsigned int size_of_A = sizeof(double)*M*N;
    unsigned int size_of_B = sizeof(double)*N*K;
    unsigned int size_of_C = sizeof(double)*M*K;

    cudaError_t error;

    const pybind11::buffer_info h_buff_a = a.request();
    const pybind11::buffer_info h_buff_b = b.request();
    pybind11::buffer_info h_buff_c = c.request();

    const double *h_a, *h_b;
    double *h_c;
    h_a = reinterpret_cast<double*>(h_buff_a.ptr);
    h_b = reinterpret_cast<double*>(h_buff_b.ptr);
    h_c = reinterpret_cast<double*>(h_buff_c.ptr);

    double *d_a, *d_b, *d_c;
    error = cudaMalloc((void **)&d_a, size_of_A);
    error = cudaMalloc((void **)&d_b, size_of_B);
    error = cudaMalloc((void **)&d_c, size_of_C);

    if (error != cudaSuccess) {
        std::cout << "Error in cudaMalloc" << std::endl;
        throw std::runtime_error(cudaGetErrorString(error));
    }

    error = cudaMemcpy(d_a, h_a, size_of_A, cudaMemcpyHostToDevice);
    error = cudaMemcpy(d_b, h_b, size_of_B, cudaMemcpyHostToDevice);

    if (error != cudaSuccess) {
        std::cout << "Error in first cudaMemcpy" << std::endl;
        throw std::runtime_error(cudaGetErrorString(error));
    }
```

Binded C++ function - part 2

- M, N, K = 160
- BLOCK_SIZE = 32
- dim_grid = {5, 5}
- dim_block = {32, 32}

```
    switch(memory){
        case MEM_TYPE::GLOBAL: {
            unsigned int grid_cols = (K + BLOCK_SIZE - 1) / BLOCK_SIZE;
            unsigned int grid_rows = (M + BLOCK_SIZE - 1) / BLOCK_SIZE;
            dim3 dim_grid(grid_cols, grid_rows);
            dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE);
            gpu_global_matmul<<<dim_grid, dim_block>>>(d_a, d_b, d_c, M, N, K);
            break;
        }
        case MEM_TYPE::SHARED: {
            dim3 dim_grid(ceil((double)K / (double)BLOCK_SIZE), ceil((double)M / (double)BLOCK_SIZE));
            dim3 dim_block(BLOCK_SIZE, BLOCK_SIZE);
            gpu_shared_matmul<<<dim_grid, dim_block>>>(d_a, d_b, d_c, M, N, K);
            break;
        }
    }


    error = cudaMemcpy(h_c, d_c, size_of_C, cudaMemcpyDeviceToHost);

    if (error != cudaSuccess) {
        std::cout << "Error in last cudaMemcpy" << std::endl;
        throw std::runtime_error(cudaGetErrorString(error));
    }

    error = cudaFree(d_a);
    error = cudaFree(d_b);
    error = cudaFree(d_c);

    if (error != cudaSuccess) {
        std::cout << "Error in cudaFree" << std::endl;
        throw std::runtime_error(cudaGetErrorString(error));
    }
}
```

Kernel implementation

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

Kernel implementation

- Consecutive thread indices access consecutive global memory addresses
  - -> Coalesced memory access
    - -> Faster code
- Reduces number of global memory access
  - -> Faster code

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

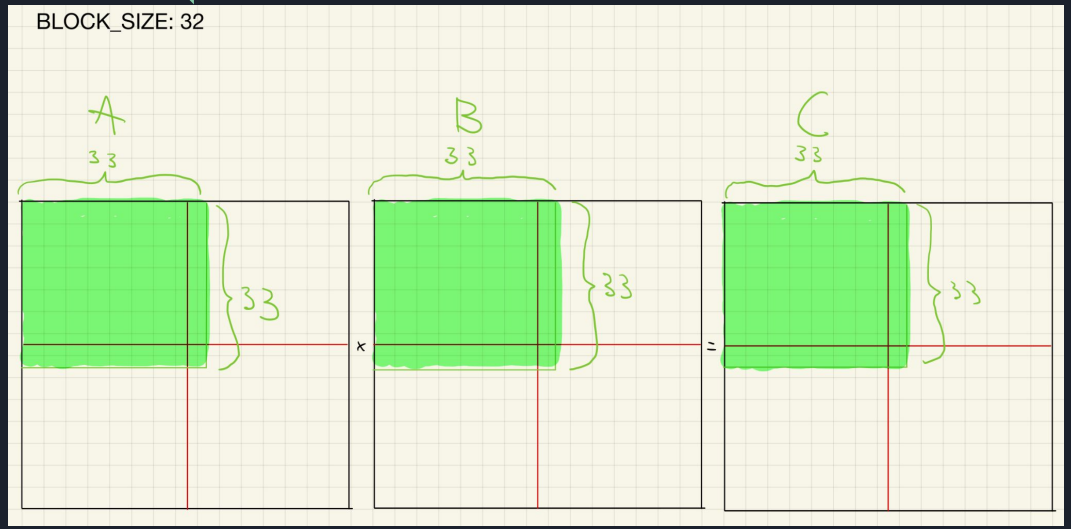
```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

Will this work on matrix dimensions which is not a factor of the block size?

YES!

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}



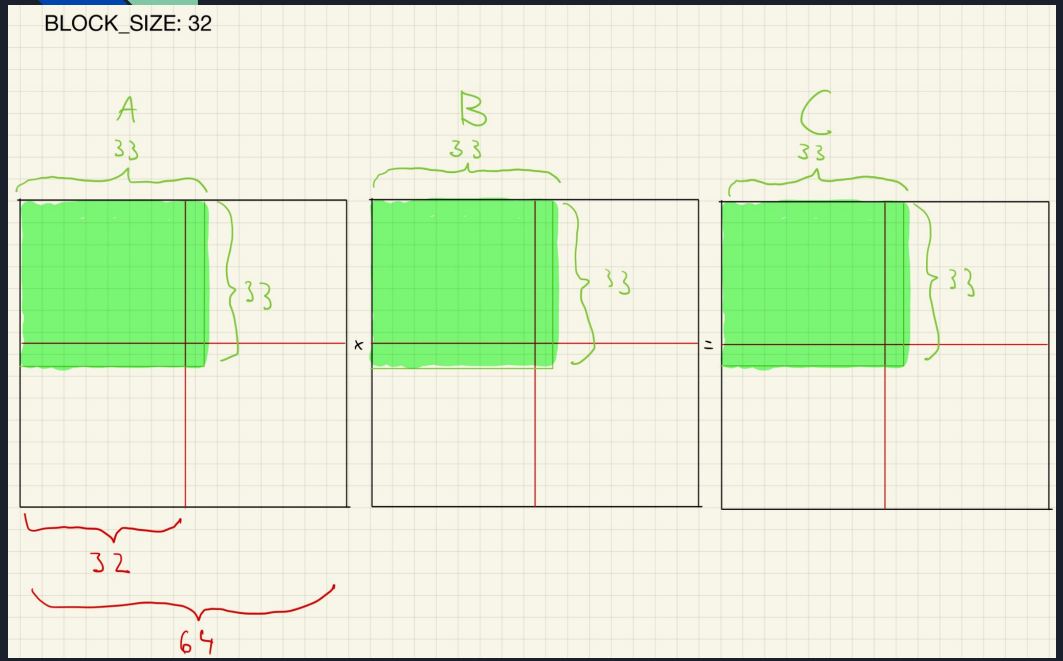

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}



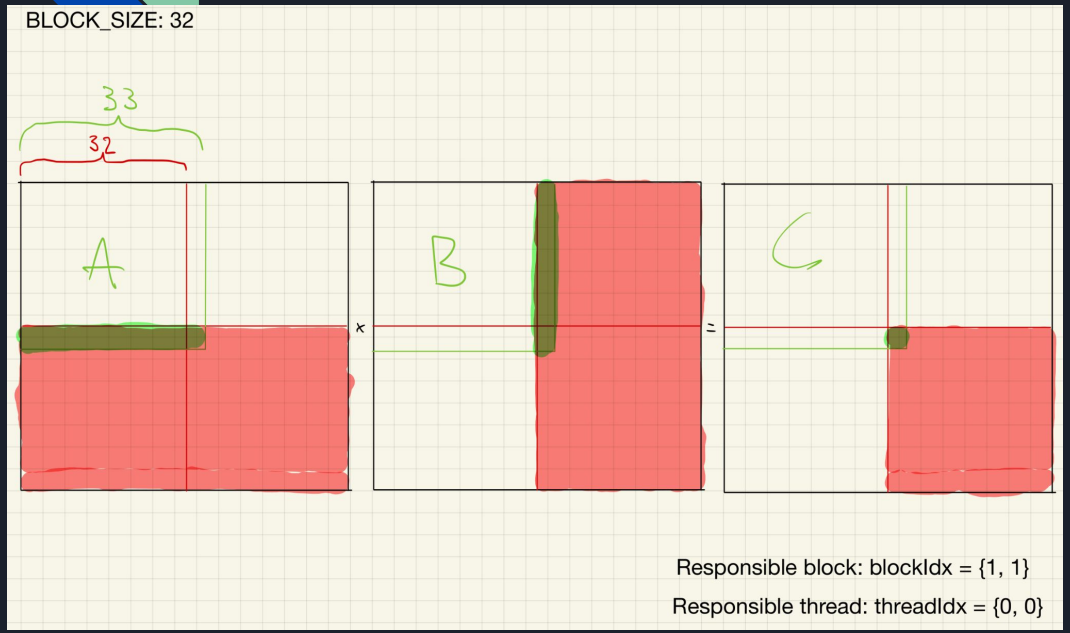

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}



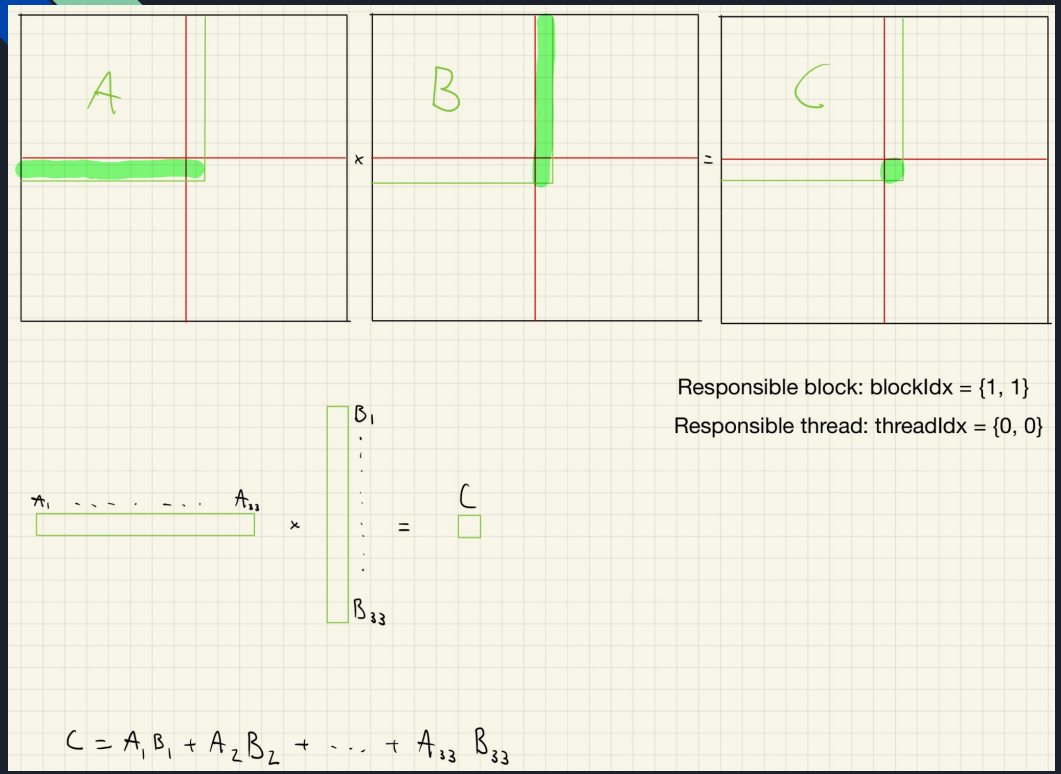

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}



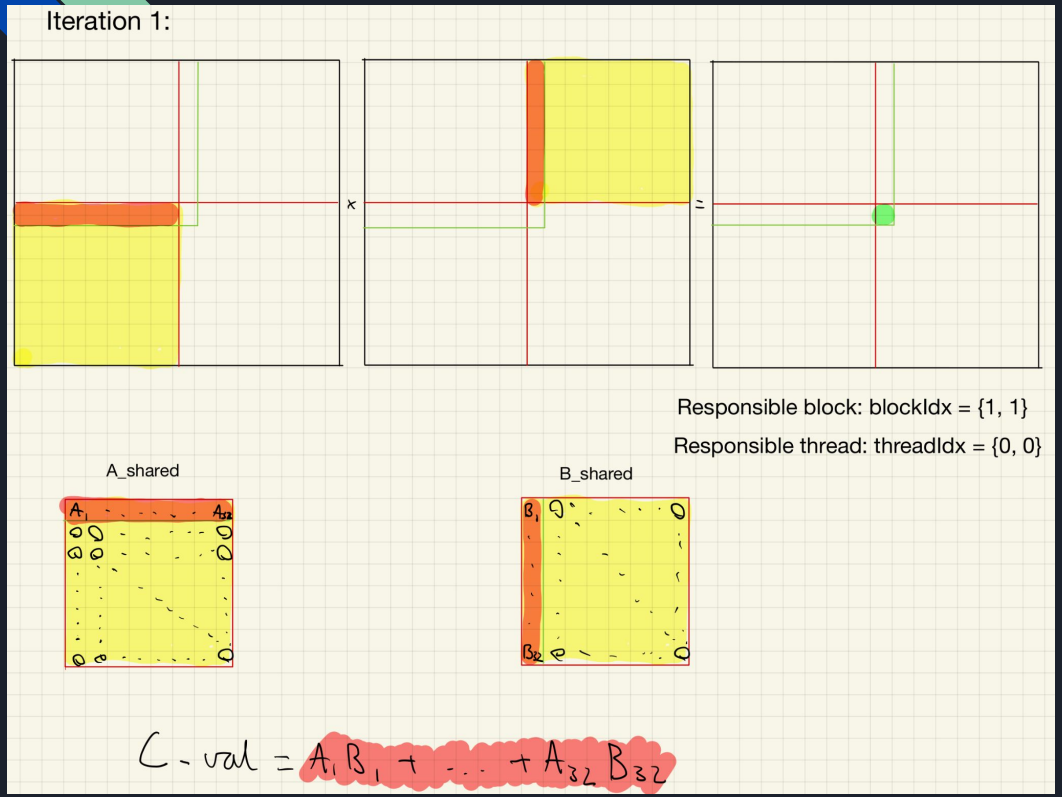

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}



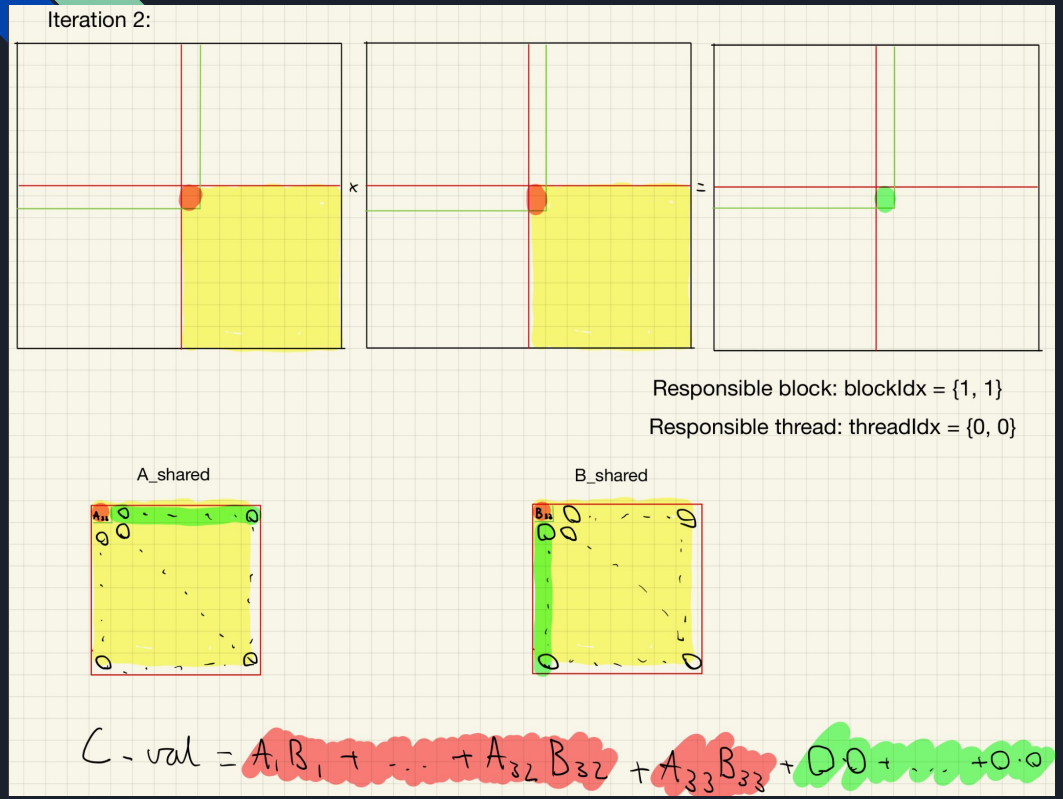

```
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

[M, N, K] = [33, 33, 33], BLOCK_SIZE = 32, Responsible blockIdx = {1, 1}, Responsible threadIdx = {0, 0}




```cpp
int block_row = blockIdx.y;
int block_col = blockIdx.x;

int row = threadIdx.y;
int col = threadIdx.x;

int global_row = block_row * BLOCK_SIZE + row;
int global_col = block_col * BLOCK_SIZE + col;
```

```cpp
for(int i = 0; i < ceil((double)N/(double)BLOCK_SIZE); i++){
    __shared__ double A_shared[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double B_shared[BLOCK_SIZE][BLOCK_SIZE];

    if(global_row < M && col + i * BLOCK_SIZE < N){
        const double* A_block = &A[N * block_row * BLOCK_SIZE + i * BLOCK_SIZE];
        A_shared[row][col] = A_block[N * row + col];
    }
    else{
        A_shared[row][col] = 0;
    }

    if(row + i * BLOCK_SIZE < N && global_col < K){
        const double* B_block = &B[K * i * BLOCK_SIZE + block_col * BLOCK_SIZE];
        B_shared[row][col] = B_block[K * row + col];
    }
    else{
        B_shared[row][col] = 0;
    }

    __syncthreads();

    for(int j = 0; j < BLOCK_SIZE; j++){
        C_val += A_shared[row][j] * B_shared[j][col];
    }
```

```cpp
__syncthreads();

for(int j = 0; j < BLOCK_SIZE; j++){
    C_val += A_shared[row][j] * B_shared[j][col];
}

__syncthreads();
```

```cpp
if(global_row < M && global_col < K){
    C[global_row * K + global_col] = C_val;
}
```

Extra: comparing execution times

- Added more bindings
  - C++ CPU
  - GPU global memory
- Option to show execution time in GUI

Execution times with M, N, K = 160

```
CPU in C++ time: 0.021621011997922324
GPU with only global memory time: 0.0653340360004222
GPU with shared memory time: 0.0009912850000546314
Python time: 0.04182699699958903
```

```
CPU in C++ time: 0.021748513001512038
GPU with only global memory time: 0.06215431500095292
GPU with shared memory time: 0.0011924200007342733
Python time: 0.019945928997913143
```

```
CPU in C++ time: 0.02218994500071858
GPU with only global memory time: 0.061548968998977216
GPU with shared memory time: 0.001000150998152094
Python time: 0.0008918780004023574
```

Thank you!

Questions?
Email: hahestne@stud.ntnu.no