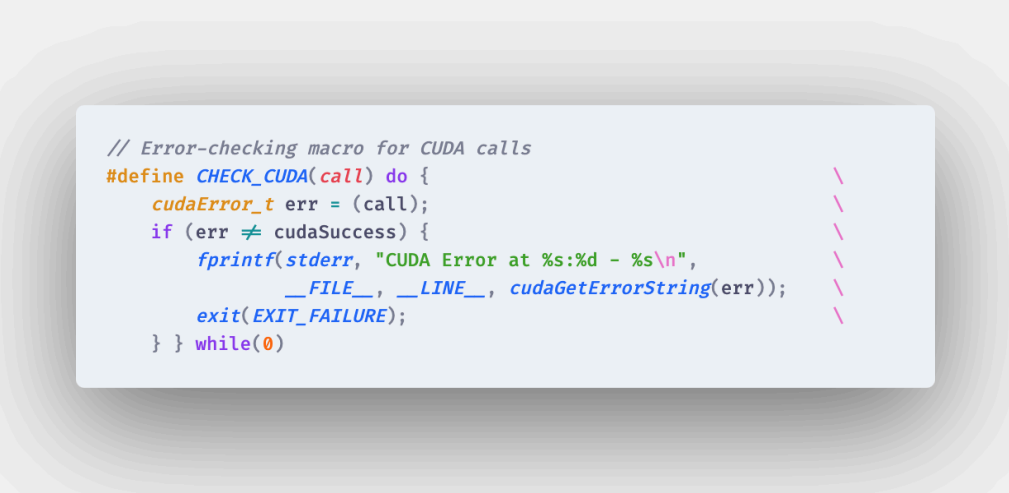
Parallel Computing - Assignment 3

Martin Štrekelj
Hallak mohamadAnas
Rishabh Gupta

Task 1 - CUDA Matrix multiplication

In CUDA programming, error handling and debugging is by default very inaccessible to the developers with vague error messages (if some) in the terminal console. To address this we implemented a `CHECK_CUDA` helper function that we later on used to wrap all CUDA related callbacks and ensure that we got a descriptive error message that would help us with debugging. The snippet below (see Figure 1) shows a macro that captures error codes from CUDA operations and terminates execution with detailed information when failure occurs. The error is printed to standard output.



```
// Error-checking macro for CUDA calls
#define CHECK_CUDA(call) do {                                   \
    cudaError_t err = (call);                                   \
    if (err ≠ cudaSuccess) {                                    \
        fprintf(stderr, "CUDA Error at %s:%d - %s\n",           \
                __FILE__, __LINE__, cudaGetErrorString(err));   \
        exit(EXIT_FAILURE);                                     \
    } } while(0)
```

Figure 1: Cuda error checking snippet

In the following section we are going to explain how we implemented the kernel function (operation that runs on the GPU) in order to calculate matrix multiplication.

The kernel function (See Figure 4) implements matrix multiplication using shared memory utilization and memory coalescing to ensure high performance and optimization for GPU execution. The kernel input argument is a reference to both input matrices and target matrix and the size of a matrix dimension. We added a __restrict__ prefix to tell the compiler that A, B and C do not overlap in memory, which results in more aggressive optimizations.
In lines on Figure 2 we partition the matrices into `TILE_DIM` size tiles, which in the final execution phase was set to 32 (during testing, we also tried 16, but we got better performance with 32 size tiles). Since we mark this as fast `__shared__` memory, we can programmatically cache this on kernel level and ensure optimal memory bandwidth utilisation.

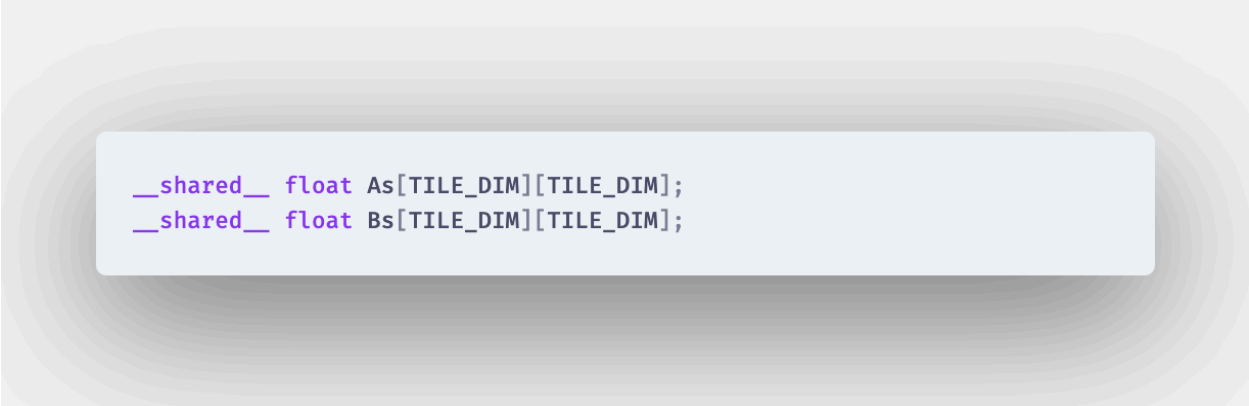


Figure 2: Shared memory inside kernel function snippet

In the next part (See Figure 3) we compute the global row and column index for writing into the result matrix and create a temporary variable that would store the calculated value before we write it into global memory.

Figure 3: Finding index for thread and temporary compute value snippet

Following the setup we then dive into the algorithm implementation (See Figure 4). The outer loop iterates through the tiles along the shared dimensions of matrices A and B. For each tile we do the following operations:

1. We load a tile from matrix `A` into shared memory `As` and matrix `B` into shared memory `Bs`. Here we are careful to check boundaries to handle edge cases.
2. We call `__syncthreads()` to ensure that all threads have finish step 1 before we proceed with computing
3. Then we compute the partial value for this tile using an unrolled loop. We used compiler optimisation technique (#pragma unroll) to get a boost in performance.
4. We call `__syncthreads()` again before moving to the next tile

After the values are calculated we write them into the target matrix. Here we are careful that we check boundaries.

```
__global__ void matMulTiled(
    const float *__restrict__ A,
    const float *__restrict__ B,
    float *__restrict__ C,
    int N)
{
    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];

    // Compute global row and col index for C
    int row = blockIdx.y * TILE_DIM + threadIdx.y;
    int col = blockIdx.x * TILE_DIM + threadIdx.x;
    float value = 0.0f;

    // Loop over tiles along the shared dimension (which is N here)
    for (int t = 0; t < (N + TILE_DIM - 1) / TILE_DIM; ++t)
    {
        int tiledCol = t * TILE_DIM + threadIdx.x; // column index in A
        int tiledRow = t * TILE_DIM + threadIdx.y; // row index in B

        // Boundary check ← inside of our tile
        if (row < N && tiledCol <  )
            As[threadIdx.y][threaNIdx.x] =  [row * N + tiledCol];
        else                                A
            As[threadIdx.y][threadIdx.x] = 0.0f;

        // Load B tile into shared memory
        if (col < N && tiledRow <  )
            Bs[threadIdx.y][threaNIdx.x] =  [tiledRow * N + col];
        else                                B
            Bs[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads();

        // Compute partial dot product for this tile
#pragma unroll
        for (int i = 0; i < TILE_DIM; ++i)
        {
            value += As[threadIdx.y][i] * Bs[i][threadIdx.x];
        }
        __syncthreads();
    }

    // Write the result to global memory if within bounds
    if (row < N && col <  )
    {                   N
        C[row * N + col] = value;
    }
}
```

Figure 4: Whole kernel function snippet

In the following section we are going to explain how we implemented the main function, that consists:

- Allocating memory on host (cpu) and device (gpu)
- Generating random matrices
- Copying matrices from cpu to gpu
- Configuring (Block & Grid) and running the kernel function
- Timing the computation procedure
- Validating on CPU
- Deallocating memory and cleanup

```
int main()
{
    int N = 1024;

    size_t bytes = N * N * sizeof(float);

    // Allocate RAM for our matrices, which is the same size as MATRIX_SIZE * sizeof(float)
    float *h_A, *h_B, *h_C;
    CHECK_CUDA(cudaMallocHost(&h_A, bytes));
    CHECK_CUDA(cudaMallocHost(&h_B, bytes));
    CHECK_CUDA(cudaMallocHost(&h_C, bytes));

    for (int i = 0; i < N *  ; ++i)
        h_A[i] = static_casN<float>(rand()) / RAND_MAX;
    for (int i = 0; i < N *  ; ++i)
        h_B[i] = static_casN<float>(rand()) / RAND_MAX;

    // Allocate GPU memory for our matrices, which is the same size as MATRIX_SIZE * sizeof(float)
    float *d_A, *d_B, *d_C;
    CHECK_CUDA(cudaMalloc(&d_A, bytes));
    CHECK_CUDA(cudaMalloc(&d_B, bytes));
    CHECK_CUDA(cudaMalloc(&d_C, bytes));

    // Enable asynchronous operations
    cudaStream_t stream;
    CHECK_CUDA(cudaStreamCreate(&stream));

    // Copy matrices A and B to the device asynchronously
    CHECK_CUDA(cudaMemcpyAsync(d_A, h_A, bytes, cudaMemcpyHostToDevic , stream));
    CHECK_CUDA(cudaMemcpyAsync(d_B, h_B, bytes,ecudaMemcpyHostToDevic , stream));
                                                e
    // REST OF CODE  .....
```

Figure 5: Generating matrices and allocating memory on host and GPU

On Figure 5 we demonstrate the lines of code that first allocate the pinned memory that will be used for a fast sync of matrices to the GPU device. We generate two matrices of size N (in the current implementation it is of size 1024) and copy them to the GPU. We continue (see Figure 6) with handling configuration to instruct CUDA on how to handle the matrix multiplication. First we set up the thread organization with `TILE_DIM x TILE_DIM` threads per block and calculate the grid size so it matches

the entire matrix. Then we launch the kernel function (described above) with this configuration, default shared memory set to 0 and using the specified CUDA stream that enables async operations. When the kernel execution completes we copy the result from GPU to CPU memory asynchronously. We run `cudaStreamSynchronize(stream)` to ensure all operations in the stream have completed before continuing with the execution. The whole operation is timed and result duration is written to the standard output. At the end we verify the calculated results with a simple CPU computation of matrix multiplication on the first `20x20` elements and verify the results. We finish the main execution with deallocating memory both on CPU and GPU and exiting the program.

```
// REST OF CODE  .....

// Configure grid and block dimensions
dim3 block(TILE_DIM, TILE_DIM); // threads x threads -> eg 16×16
dim3 grid((N + TILE_DIM - 1) / TILE_DIM, (N + TILE_DIM - 1) / TILE_DIM);

double sharedMemoryForKernel = 0;
double startTime = omp_get_wtime();
matMulTiled<<<grid, block, sharedMemoryForKernel, stream>>>(d_A, d_B, d_C,  );
CHECK_CUDA(cudaPeekAtLastError());                                        N
double endTime = omp_get_wtime();
double durationMS = (endTime - startTime) * 1000;

// Copy result matrix C back to host asynchronously
CHECK_CUDA(cudaMemcpyAsync(h_C, d_C, bytes, cudaMemcpyDeviceToHos , stream));
                                                t
// Wait for all operations in the stream to finish
CHECK_CUDA(cudaStreamSynchronize(stream));

std::cout << "Kernel execution time (excluding transfers): "
          << durationMS << " ms" << std::endl;

// (Optional) Validate a few entries of the result for correctness
bool correct = true;
for (int i = 0; i < 20 && correct; ++i)
{
    for (int j = 0; j < 20 && correct; ++j)
    {
        double ref = 0.0;
        for (int k = 0; k <  ; ++k)
            ref += h_A[i * N +  ] * h_B[k * N +  ];
        if (fabs(h_C[i * N +  & - ref) > 1e-3) j
        {                      j
            correct = false;
            std::cerr << "Mismatch at (" << i << ", " << j << "): "
                      << h_C[i * N +  ] << " vs " << ref << std::endl;
        }                           j
    }
}

if (correct)
    std::cout << "Result verification passed!" << std::endl;
else
    std::cerr << "Result verification failed!" << std::endl;

// Clean up all allocated resources
CHECK_CUDA(cudaStreamDestroy(stream));
CHECK_CUDA(cudaFree(d_A));
CHECK_CUDA(cudaFree(d_B));
CHECK_CUDA(cudaFree(d_C));
CHECK_CUDA(cudaFreeHost(h_A));
CHECK_CUDA(cudaFreeHost(h_B));
CHECK_CUDA(cudaFreeHost(h_C));

return 0;
```

Figure 6: Kernel run configuration, computation, verification and memory deallocate snippet

To run the benchmarks on the assignment we calculated an average of 10 runs per matrix size. Benchmarks were run on matrix sizes of 512x512, 1024x1024, 2048x2048, 3072x3072, 4096x4096, 5120x5120, 6144x6144, 7168x7168, 8192x8192, 9216x9216 and finally the size of 10240x10240. The calculated results are shown in the table below and are visualized on Figure 7:

#	Size	Avg. Kernel Execution Time (ms)	Verification
1	512	0.6559	✅
2	1024	0.8774	✅
3	2048	2.7054	✅
4	3072	6.2955	✅
5	4096	10.8898	✅
6	5120	15.6473	✅
7	6144	22.163	✅
8	7168	29.8723	✅
9	8192	38.5997	✅
10	9216	48.6334	✅
11	10240	59.5007	✅

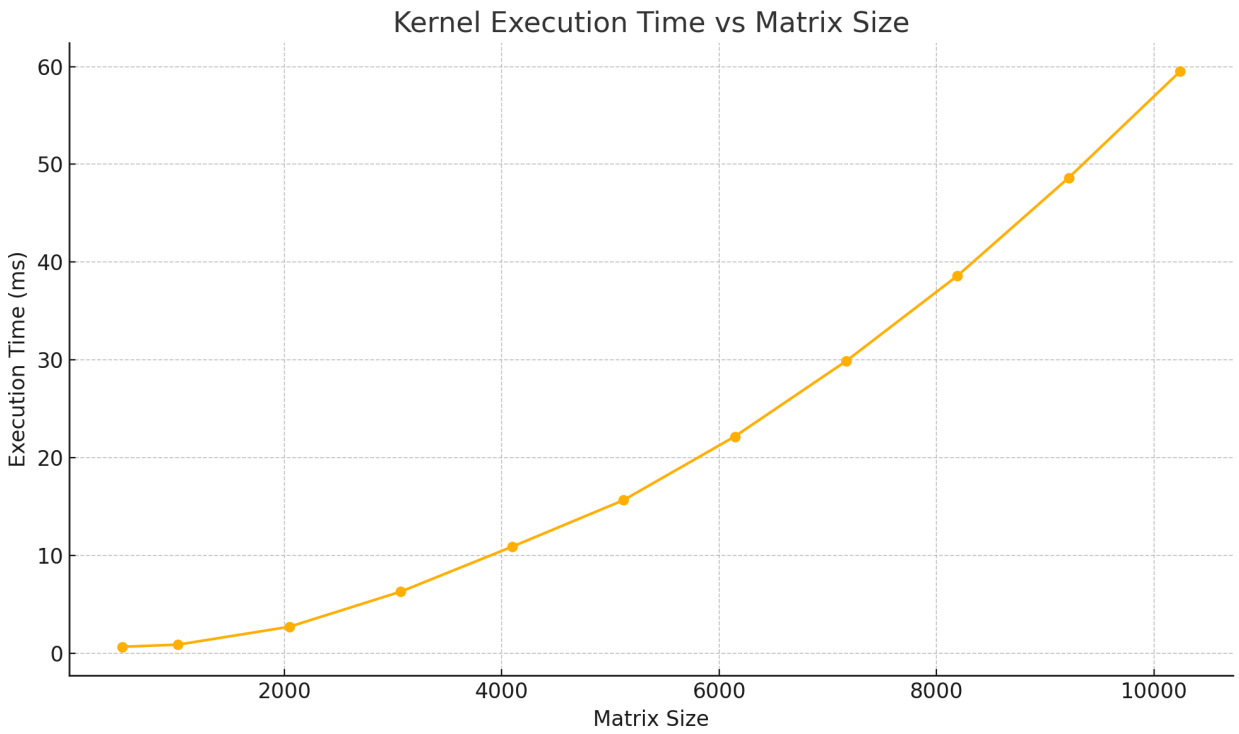


Figure 7: Kernel Execution Time vs Matrix size graph

To study the time complexity, we plotted the empirical data against the theoretical time complexities. We observed that the growth rate is faster than O(n), but not as steep as O(n²). This suggests that the algorithm exhibits sub-quadratic behavior, likely with a time complexity around O(n^1.5).

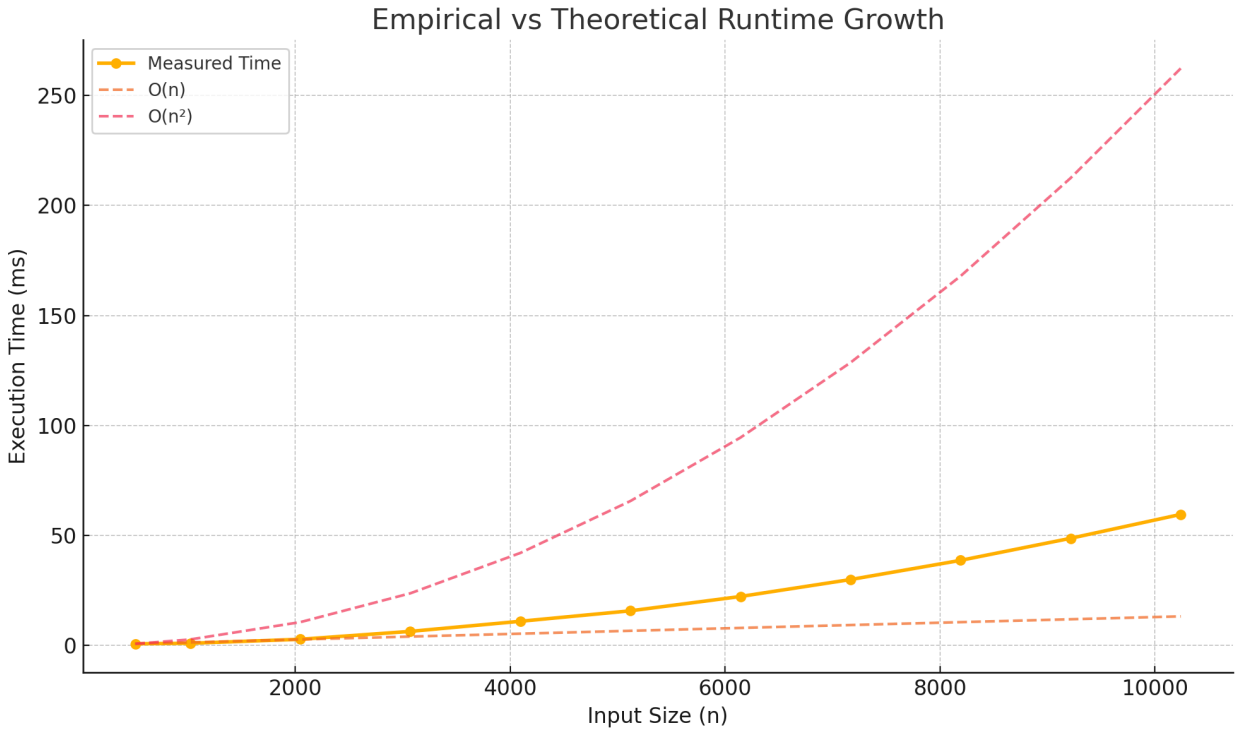


Figure 8: Kernel Execution Time vs Matrix size with quadratic fit