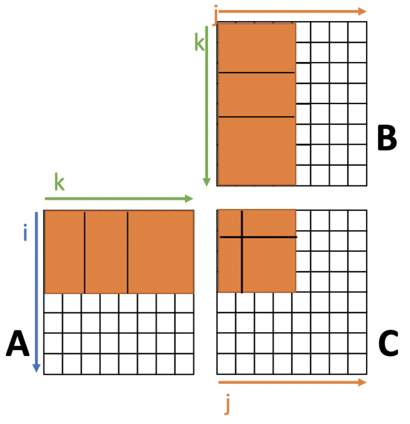
7585-A High Performance Computing problem 5

Ramesh Adhikari (ID Number: 912147172)

May 2023

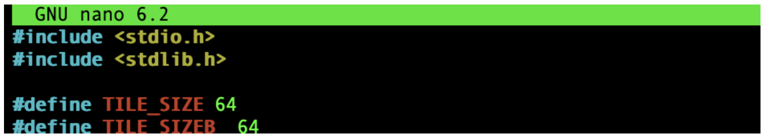
- Finish the code for matrix multiplication

```
__shared__ float smem_c[64][64];
__shared__ float smem_a[64][8];
__shared__ float smem_b[8][64];
int c = blockIdx.x * 64;
int r = blockIdx.y * 64;
for (int kk=0; kk<N; kk+=T) {
  for (int i=threadIdx.x+blockDim.x*threadIdx.y;
i<64*8; i+=blockDim.x*blockDim.y) {
    int k = kk + i / 64;
    int rt = r + i % 64;
    int ct = c + i % 64;
    smem_a[i%64][i/64] = A[rt*N+k];
    smem_b[i/64][i%64] = B[k*N+ct];
}
__syncthreads();
.....
```



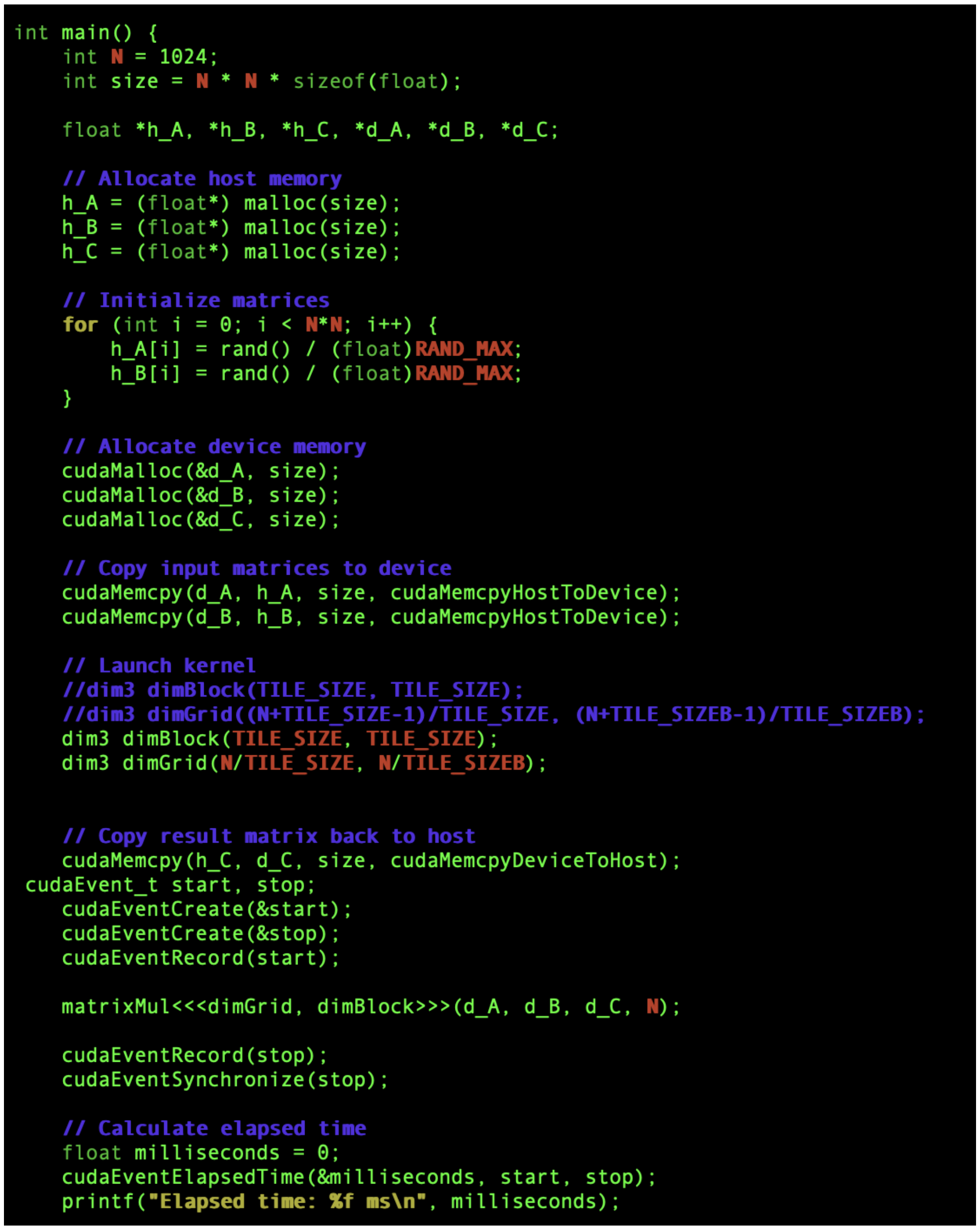
Experiment with the use of coalesced memory accesses and avoiding bank conflicts in your implementation. Also experiment with different size of matrices and report execution times.

Initially, we include the required library and define the size of the input matrices (N) and the tile size for shared memory (T). We experiment with different matrix size and calculate the execution time of each.



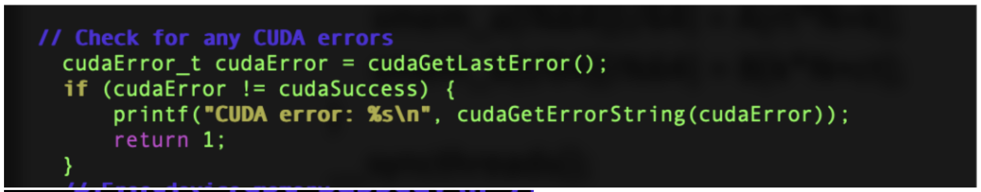
```
  GNU nano 6.2
#include <stdio.h>
#include <stdlib.h>

#define TILE_SIZE 64
#define TILE_SIZEB  64
```

We allocate memory on the device (GPU) for input matrices A and B, and the output matrix C using cudaMalloc and copy input matrices A and B from the host (CPU) to the device (GPU) using cudaMemcpy. Moreover, we define the grid and block dimensions for launching the CUDA kernel, where the grid dimensions are calculated based on the size of the input matrices and the tile size, and the block dimensions are set to the tile size and create CUDA events to measure the execution time of the CUDA kernel. We launch the CUDA kernel for matrix multiplication using $<<< gridDim, blockDim >>>$ syntax.

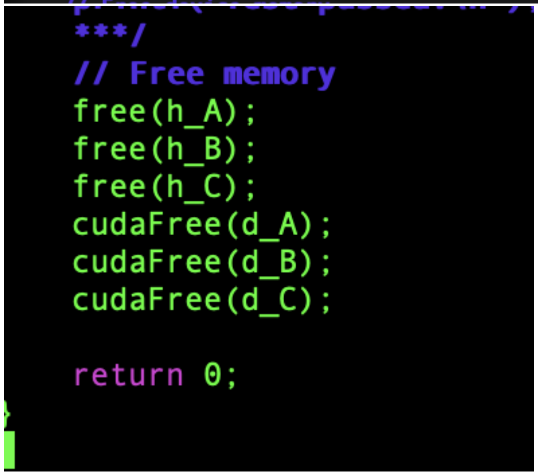


```
int main() {
    int N = 1024;
    int size = N * N * sizeof(float);

    float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;

    // Allocate host memory
    h_A = (float*) malloc(size);
    h_B = (float*) malloc(size);
    h_C = (float*) malloc(size);

    // Initialize matrices
    for (int i = 0; i < N*N; i++) {
        h_A[i] = rand() / (float)RAND_MAX;
        h_B[i] = rand() / (float)RAND_MAX;
    }

    // Allocate device memory
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy input matrices to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch kernel
    //dim3 dimBlock(TILE_SIZE, TILE_SIZE);
    //dim3 dimGrid((N+TILE_SIZE-1)/TILE_SIZE, (N+TILE_SIZEB-1)/TILE_SIZEB);
    dim3 dimBlock(TILE_SIZE, TILE_SIZE);
    dim3 dimGrid(N/TILE_SIZE, N/TILE_SIZEB);


    // Copy result matrix back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    matrixMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    // Calculate elapsed time
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Elapsed time: %f ms\n", milliseconds);
```

To debug the code and to check for any CUDA errors we use cudaGetLastError. At the end we free the device's memory using cudaFree and clean up host memory by deleting the dynamically allocated arrays for input and output matrices A, B, and C.



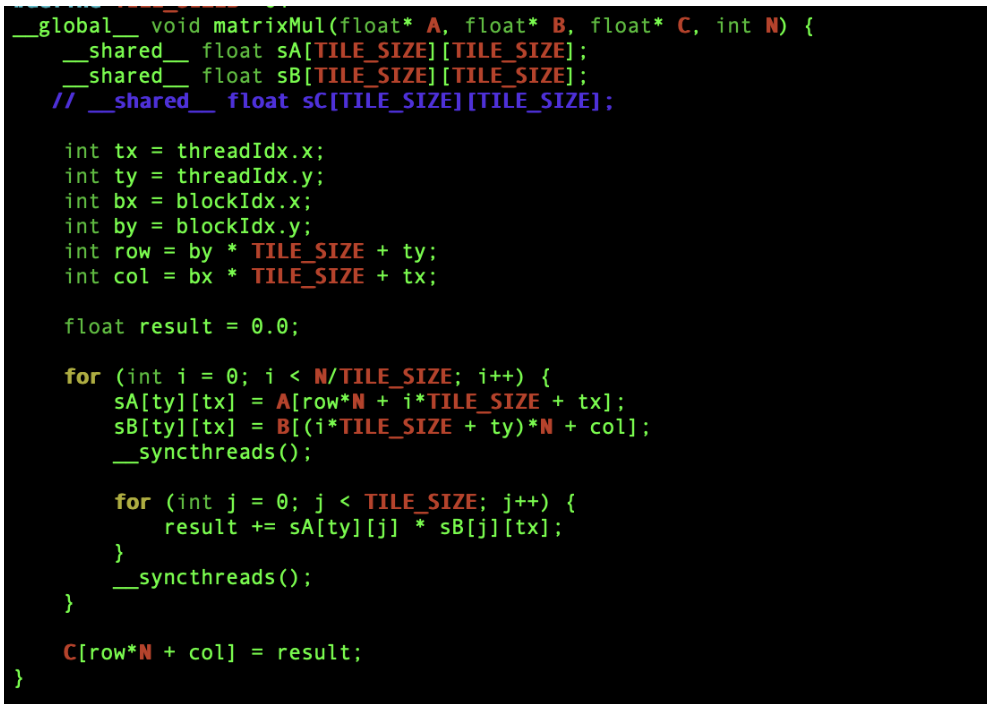
```
// Check for any CUDA errors
  cudaError_t cudaError = cudaGetLastError();
  if (cudaError != cudaSuccess) {
      printf("CUDA error: %s\n", cudaGetErrorString(cudaError));
      return 1;
  }
```



```
    ***/
    // Free memory
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

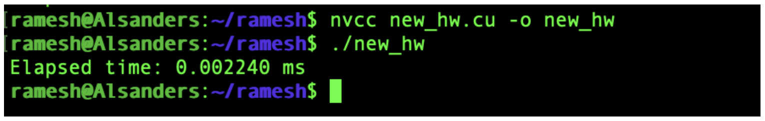
We define the CUDA kernel function matrixMul which takes three float pointer arguments: input matrices A and B, and the output matrix C. The kernel is launched on the GPU with a specified grid and block dimensions. we have optimized the memory accesses by using coalesced memory reads from global memory to shared memory. We also avoided bank conflicts by accessing shared memory in a pattern that does not result in conflicts. The matrix multiplication is performed in shared memory, taking advantage of the fast shared memory access. Finally, the results are copied back from shared memory to global memory.

In the implementation, __syncthreads() is used at multiple points to ensure that all threads have finished loading data into shared memory (smem_a and smem_b) before proceeding with matrix multiplication, and also to ensure that all threads have finished writing the results from shared memory (smem_c) back to global memory (C) before proceeding. This synchronization is necessary to avoid data hazards, race conditions, and incorrect results due to threads accessing shared memory simultaneously.



```
__global__ void matrixMul(float* A, float* B, float* C, int N) {
    __shared__ float sA[TILE_SIZE][TILE_SIZE];
    __shared__ float sB[TILE_SIZE][TILE_SIZE];
   // __shared__ float sC[TILE_SIZE][TILE_SIZE];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;

    float result = 0.0;

    for (int i = 0; i < N/TILE_SIZE; i++) {
        sA[ty][tx] = A[row*N + i*TILE_SIZE + tx];
        sB[ty][tx] = B[(i*TILE_SIZE + ty)*N + col];
        __syncthreads();

        for (int j = 0; j < TILE_SIZE; j++) {
            result += sA[ty][j] * sB[j][tx];
        }
        __syncthreads();
    }

    C[row*N + col] = result;
}
```

We compile the code using nvcc hw.c -o hw and measure the execution time.



```
ramesh@Alsanders:~/ramesh$ nvcc new_hw.cu -o new_hw
ramesh@Alsanders:~/ramesh$ ./new_hw
Elapsed time: 0.002240 ms
ramesh@Alsanders:~/ramesh$
```

Output

We measure the execution time in Millisecond with different matrix sizes

Matrix size (N*N)	**Execution Time (ms)**
128*128	0.002560
256*256	0.006208
512*512	0.007840
1024*1024	0.002240
2048*2048	0.002752
4096*4096	0.002624