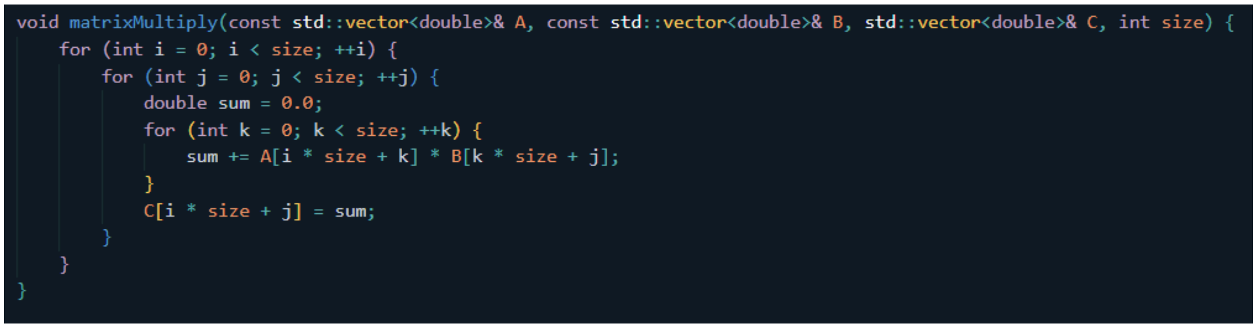
HPC Tutorial 12 Report
CS22B2012 K Aditya Sai

This report analyses the performance of CUDA parallel code on the multiplication of two matrices of size 5000. The values range from 88.0 to 888.0.

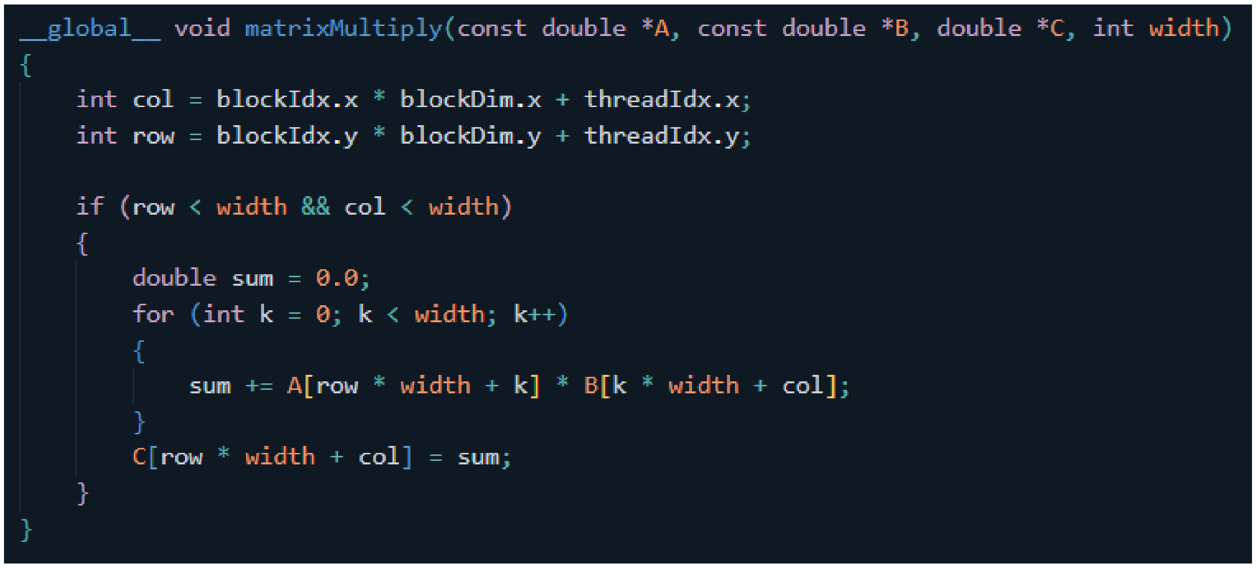
The Host device is throwing an instance of **bad_alloc()** when a matrix of size 10000 x 10000 was allocated.

Serial Code:



```cpp
void matrixMultiply(const std::vector<double>& A, const std::vector<double>& B, std::vector<double>& C, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            double sum = 0.0;
            for (int k = 0; k < size; ++k) {
                sum += A[i * size + k] * B[k * size + j];
            }
            C[i * size + j] = sum;
        }
    }
}
```

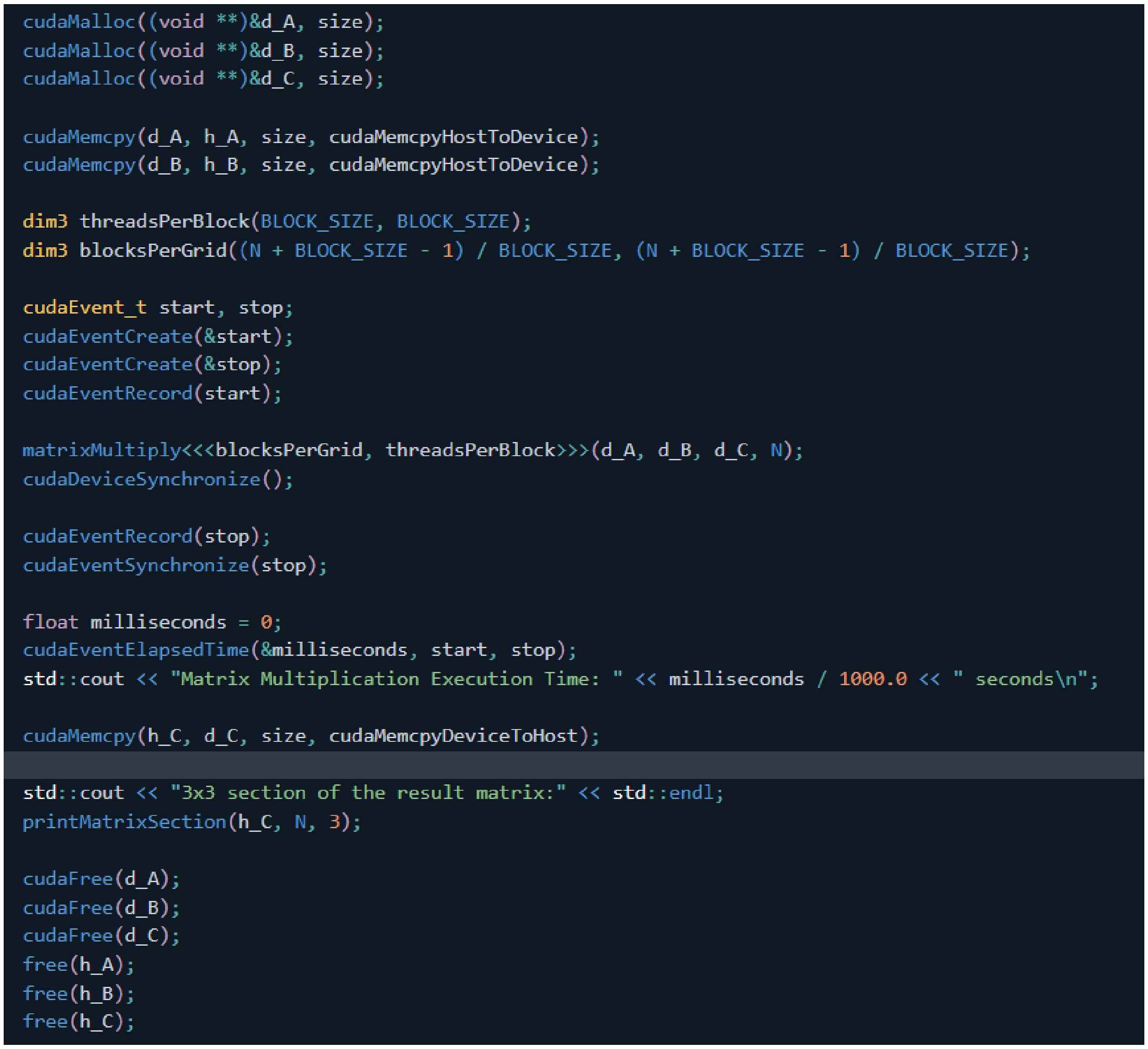
Parallel Code :



```cpp
__global__ void matrixMultiply(const double *A, const double *B, double *C, int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < width && col < width)
    {
        double sum = 0.0;
        for (int k = 0; k < width; k++)
        {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}
```

This parallel Code uses the inbuilt dim construct.
It uses the dim construct to make a 2d block of threads (32, 32).

Memory Management :

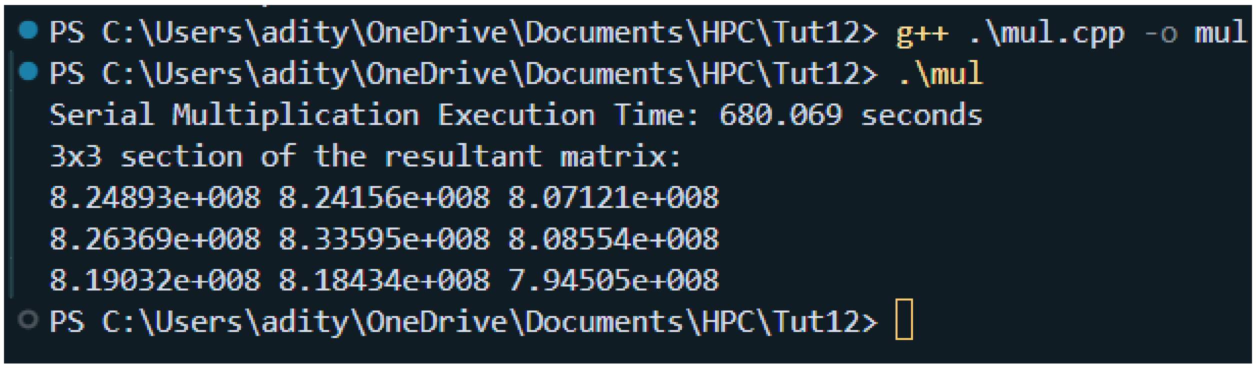


```
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 blocksPerGrid((N + BLOCK_SIZE - 1) / BLOCK_SIZE, (N + BLOCK_SIZE - 1) / BLOCK_SIZE);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

matrixMultiply<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
cudaDeviceSynchronize();

cudaEventRecord(stop);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
std::cout << "Matrix Multiplication Execution Time: " << milliseconds / 1000.0 << " seconds\n";

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);


std::cout << "3x3 section of the result matrix:" << std::endl;
printMatrixSection(h_C, N, 3);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C);
```

Number of Threads per block = (32, 32)
Number of Blocks = (N + threads_per_block_x - 1) / (threads_per_block_x), (N + threads_per_block_y - 1) / (threads_per_block_y)

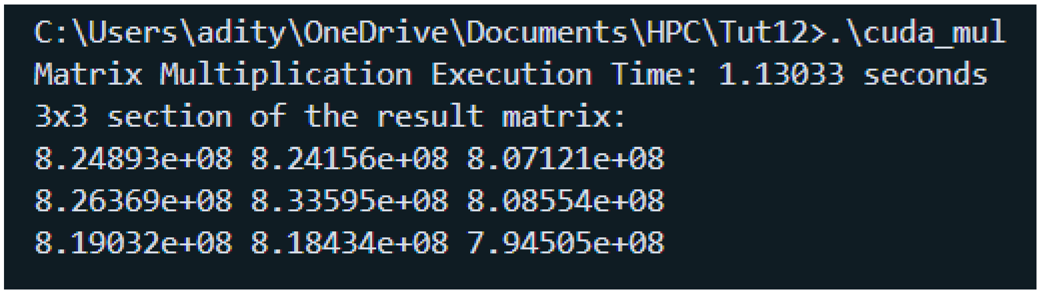
Outputs :

Serial :



```
PS C:\Users\adity\OneDrive\Documents\HPC\Tut12> g++ .\mul.cpp -o mul
PS C:\Users\adity\OneDrive\Documents\HPC\Tut12> .\mul
Serial Multiplication Execution Time: 680.069 seconds
3x3 section of the resultant matrix:
8.24893e+008 8.24156e+008 8.07121e+008
8.26369e+008 8.33595e+008 8.08554e+008
8.19032e+008 8.18434e+008 7.94505e+008
PS C:\Users\adity\OneDrive\Documents\HPC\Tut12>
```

Parallel :



```
C:\Users\adity\OneDrive\Documents\HPC\Tut12>.\cuda_mul
Matrix Multiplication Execution Time: 1.13033 seconds
3x3 section of the result matrix:
8.24893e+08 8.24156e+08 8.07121e+08
8.26369e+08 8.33595e+08 8.08554e+08
8.19032e+08 8.18434e+08 7.94505e+08
```

Speedup:

S= T(1) / T(p) = 680.069 / 1.13033 = 601.655268

Observations :

- Compared to matrix addition, matrix multiplication is a much more computationally expensive task
- Still we see that the usage of CUDA gives a much better speedup
- We can come to the conclusion that the max performance of the GPGPU is gotten in case of computationally expensive tasks