

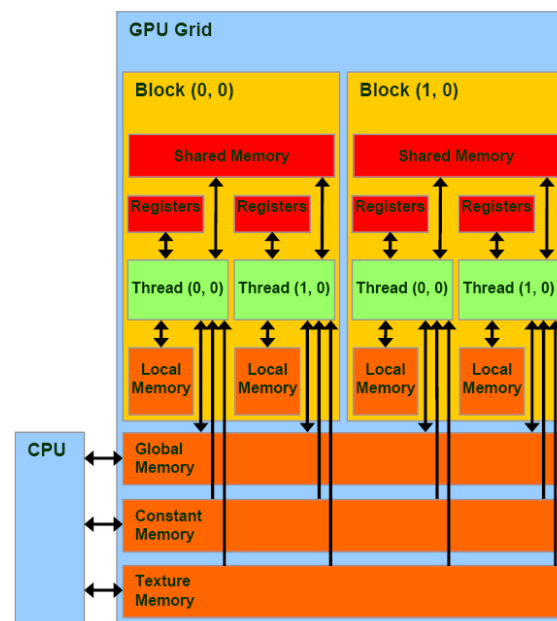
Introduction

Understanding the basic memory architecture of GPU devices is essential for developing hi-performance applications. One of the most important performance challenges is the best use of local multiprocessor memory resources such as shared memory, constant memory, and registers.

So far, we have only used the global memory and in this tutorial we will learn a bit more about other memories that are available on modern GPU cards. We will write our first CUDA kernel that makes use of the super-fast on-chip memory, which is the main goal of this tutorial.

Before we start coding, let's have a brief introduction into different kinds of memories that we can make use of. Remember that every type of memory has its pros and cons and knowing when to use which type is the only way to develop hi-performance applications. Do not worry too much if do not understand everything that is written below. For the purpose of this tutorial it is enough when you get the general idea about the differences between different memories.

The image below shows different types of memories such as registers, shared, texture, constant and global memory. Red colour denotes small but fast on-chip memories while orange colour denotes much larger but much slower off-chip memories.



Global Memory - Understanding how to efficiently use global memory is an essential requirement to becoming an adept CUDA programmer. Following is a brief discussion about global memory that should be sufficient to understand the performance differences between different implementations of matrix-matrix multiplication in the first exercise.

Global memory delivers the highest memory bandwidth only when the global memory accesses are coalesced so that the hardware can then fetch (or store) the data in the fewest number of transactions. CUDA Compute Capability devices (1.0 and 1.1) can fetch data in a single 64-byte or 128-byte transaction. If the memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp, which is undesirable. The performance penalty for non-coalesced memory operations varies according to the size of the data type. The CUDA documentation provides some rough guidelines for the expected performance degradation to expect for various size data types:

- 32-bit data types will be roughly 10x slower
- 64-bit data types will be roughly 4x slower
- 128-bit data types will be roughly 2x slower

Global memory access by all threads in the half-warp of a block can be coalesced into efficient memory transactions on G80 architecture when:

- The threads access 32, 64 or 128-bit data types
- All 16 words of the transaction must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words). This implies that the starting address and alignment are important.
- Threads must access the words in sequence: the k^{th} thread in the half-warp must access the k^{th} word. Note: not all threads in a warp need to access memory for the thread accesses to coalesce. This is called a "divergent warp".

Newer architectures such as the GT200 family of devices have more relaxed coalescing requirements than those just discussed. For purposes here, suffice to say that if you tune your code to coalesce well on a G80 CUDA-enabled device, it will coalesce well on a GT200 device.

Local Memory - is a memory abstraction that implies "local in the scope of each thread". It is not an actual hardware component of the multi-processor. In actuality, local memory resides in global memory allocated by the compiler and delivers the same performance as any other global memory region. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. Unfortunately, the relationship between automatic variables and

local memory continues to be a source of confusion for CUDA programmers. The compiler might choose to place automatic variables in local memory when:

- There are too many register variables.
- A structure would consume too much register space.
- The compiler cannot determine if an array is indexed with constant quantities. Please note that registers are not addressable so an array has to go into local memory -- even if it is a two-element array -- when the addressing of the array is not known at compile time.

Constant Memory - is read only from kernels and is hardware optimised for the case when all threads read the same location. Amazingly, constant memory provides one cycle of latency when there is a cache hit even though constant memory resides in device memory (DRAM). If threads read from multiple locations, the accesses are serialised. The constant cache is written to only by the host (not the device because it is constant!) with `cudaMemcpyToSymbol` and is persistent across kernel calls within the same application. Up to 64KB of data can be placed in constant cache and there is 8 KB of cache for each multiprocessor. Access to data in constant memory can range from one cycle for in cache data to hundreds of cycles depending on cache locality. The first access to constant memory often does not generate a cache "miss" due to pre-fetching

Texture Memory - The easiest way to think of texture memory is as an alternative memory access path that the CUDA programmer can bind to regions of the GPU device memory (e.g., global memory). Texture references can be bound to the same, overlapping or different textures in memory. Each on-chip texture unit has some internal memory that buffers data from global memory. For this reason, texture memory can be used as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements discussed in previous articles do not apply to texture memory accesses. This is particularly useful when targeting previous generation CUDA-capable GPUs but may not matter as much with the relaxed coalescing requirements in newer hardware.

Since optimised data access is very important to GPU performance, the use of texture memory can (in the right circumstances) provide a large performance increase. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. CUDA provides 1D, 2D and 3D fetch capabilities using texture memory. Since a texture performs a read operation from global memory only when there is a cache miss it is possible to exceed the maximum theoretical memory bandwidth of the underlying global memory through the judicious use of the texture memory cache.

Especially consider using textures when:

- There is locality of reference so caching in texture memory can help
- Use of the texture cache can reduce the penalty for nearly coalesced accesses (such as a misaligned starting address) that cannot be made fully coalesced
- Each of the previous is problem dependent so some testing is required. Some authors have reported certain texture memory access patterns are many times faster than others

Other performance benefits of texture memory (over global and constant memory) include:

- Packed data may be broadcast to separate variables in a single operation.
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] by the texture unit hardware.
- Linear, bilinear, and tri-linear interpolation using dedicated hardware separate from the thread processors.

Shared Memory – is a very fast on-chip memory that is accessible by different threads as long as they are in the same block. We have learned that a kernel launch requires a specification of execution configuration to define the number of threads that compose a block and the number of blocks that are combined together to form a grid. It is important to note that threads within each block can communicate with each other through local multi-processor resources because the CUDA execution model specifies that a block can only be processed on a single multi-processor. In other words, data written to shared memory within a block is accessible to all other threads within that block, but it is not accessible to a thread from any other block (see the image on the previous page). Shared memory with these characteristics can be implemented very efficiently in hardware which translates to fast memory accesses for CUDA developers.

A multiprocessor takes four clock cycles to issue one memory instruction. Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency. As an example, the assignment operator in the code snippet below takes four clock cycles to issue a read from global memory, four clock cycles to issue a write to shared memory, and 400 to 600 clock cycles to read a float from global memory. Note: In the last tutorial we used the `__device__` keyword to denote device functions. In addition to this, the `__device__` variable type qualifier can be also used to denote a variable that resides in global memory.

```
__shared__ float shared[32];  
__device__ float device[32];  
shared[threadIdx.x] = device[threadIdx.x];
```

With a factor of 100x-150x difference in access time, it is no surprise that developers need to minimize accesses to global memory and reuse data within the local multiprocessor memories.

Since shared memory is on chip, accesses are significantly faster than accesses to global memory and the main optimisation is avoiding bank conflicts. Shared memory is divided into equally-sized memory modules that are called memory banks. Each memory bank holds a successive 32-bit value (just like int or float) so consecutive array accesses by consecutive threads are very fast. Bank conflicts occur when multiple requests are made for data from the same bank (either the same address or multiple addresses that map to the same bank). When this happens, the hardware effectively serializes the memory operations, which forces all the threads to wait until all the memory requests are satisfied. If all threads read from the same shared memory address then a broadcast mechanism is automatically invoked and serialization is avoided. Shared memory broadcasts are an excellent and high-performance way to get data to many threads simultaneously. It is worthwhile trying to exploit this feature whenever you use shared memory.

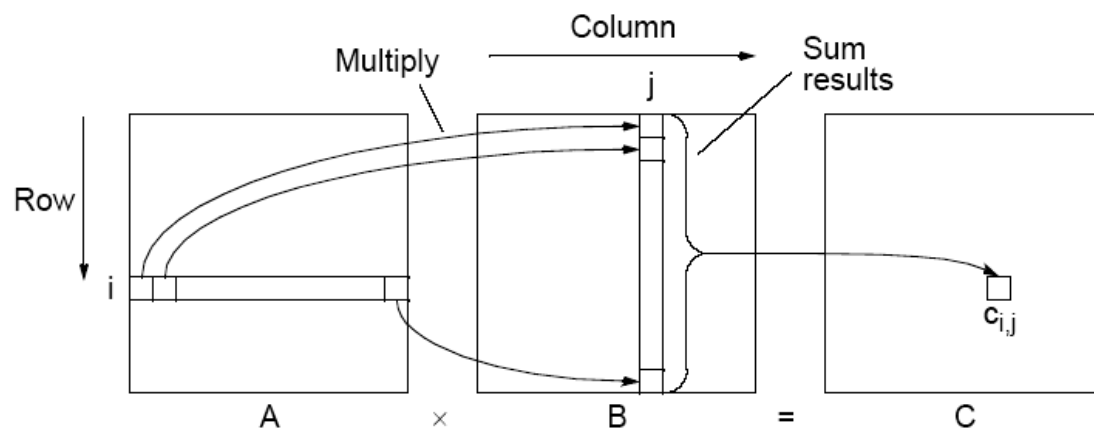
Memory Coalescing - One word that you won't hear much in CPU programming is memory coalescing. In CUDA, you typically have 32 threads (a warp) working together in unison. So many times, you have 32 threads, each wanting to write a 32-bit result to the main GPU memory. In order to accomplish this task with an absolute minimal number of memory transactions, it is best if the memory accesses are coalesced such that there are no bank conflicts. [CUDA Programming Guide](#) is a great place to look for more details.

Exercise – matrix multiplication

Matrix multiplication is an important application in HPC and appears in many applications

$$C = A * B$$

where A, B, and C are matrices (two-dimensional arrays)



First, we will write a CPU function that does matrix-matrix multiplication. Let's start by adding the headers. The `#include <omp.h>` header is added so that we can use the OpenMP CPU timing functions.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <omp.h>
```

To multiply two matrices on CPU you could write a function similar to this:

```
void matrixMultiplyCPU(float *a, float *b, float *c, int width)
{
    float result;

    for(int row=0; row<width; row++)
    {
        for(int col=0; col<width; col++)
        {
            result = 0;
            for(int k = 0; k<width; k++)
            {
                result += a[row * width + k] * b[k * width + col];
            }
            c[row * width + col] = result;
        }
    }
}
```

This function would need n^3 number of multiplications, n^3 number of additions and sequential time complexity of $O(n^3)$. This function is easy to parallelise on GPU and we will implement two different versions, one that only uses global memory and one that uses fast shared memory.

Without looking at the source code below, try to think how you could implement a simple GPU version (using global memory only) based on the CPU version above. Once you have an idea about how you would do this, you can compare it with the simple GPU implementation below.

```

__global__ void matrixMultiplySimple(float *a, float *b, float *c, int width)
{
    //calculate the row and column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

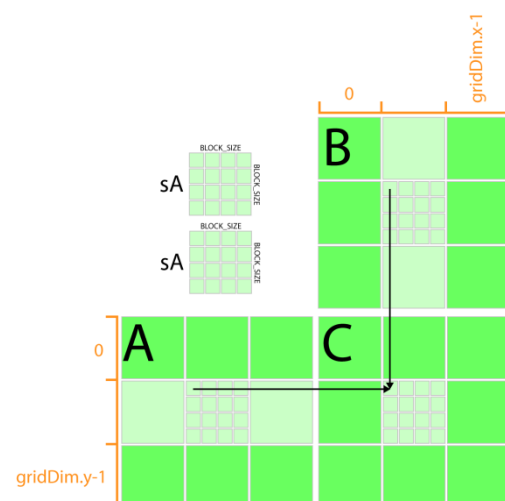
    float result = 0;

    //do dot product between row of a and column of b
    for(int i=0; i<width; i++)
    {
        result += a[row*width+i] * b[i*width+col];
    }

    //write out this thread's result
    c[row*width+col] = result;
}

```

In this example, one thread computes one c element and needs to access the entire row of a and the entire column of b array from global memory. This is probably the simplest example of matrix-matrix multiplication on GPU. The next implementation shows how you can make use of the shared memory to avoid unnecessary accesses to global memory through data reuse.



In this example we divide the matrices into tiles of 16x16 threads. Every tile (block) loads the content from the global memory to the fast shared memory and will compute a sub-part of the final matrix. The whole computation in each block is divided into phases where at each phase we load the data from global to shared memory and add the individual multiplications into the result variable.

```

__global__ void matrixMultiplyOptimised(float *a, float *b, float *c, int width)
{
    //create shorthand names for threadIdx & blockIdx
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //allocate 2D tiles in __shared__ memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    //calculate the row & column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float result = 0;
}

```

```

//loop over the tiles of the input in phases
for(int p=0; p<width/TILE_WIDTH; p++)
{
    //collaboratively load tiles into __shared__
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty) * width + col];

    //wait until all data is loaded before allowing any thread in
    this block to continue
    __syncthreads();

    //do dot product between row of s_a and column of s_b
    for(int i=0; i<TILE_WIDTH; i++)
    {
        result += s_a[ty][i] * s_b[i][tx];
    }

    //wait until all threads are finished with the data before
    allowing any thread in this block to continue
    __syncthreads();
}

//write out this thread's result
c[row*width+col] = result;
}

```

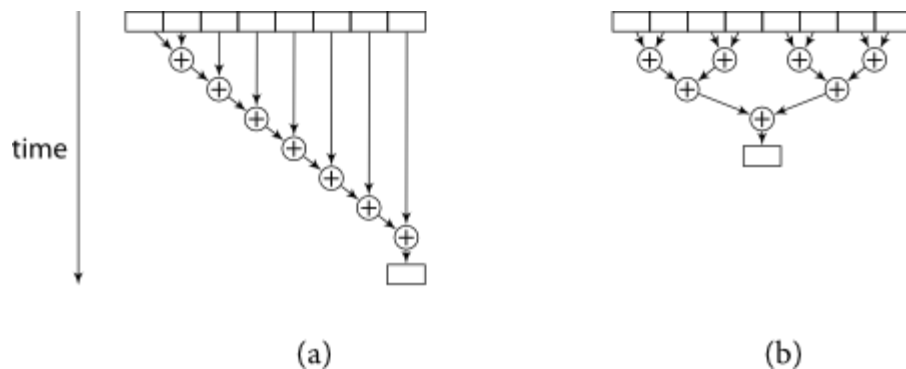
The main point here is to demonstrate the potential use of the shared memory and its benefits rather than the actual algorithm. Do not worry too much if it looks too complex at first. You can still look into Programming Massively Parallel Processors book, which explain in in great detail.

At the point you one CPU function and two different kernels to do the same thing. Have a look at the main() in the source code below and go step-by-step through it to launch different kernels and measure their execution times.

Exercise 2 – parallel reduction sum

Parallel reduction is an important data parallel primitive, which is relatively easy to implement in CUDA but harder to get it right. Check [this presentation](#) by NVIDIA, which explains the whole idea of parallel reduction and also different levels of optimisations, which you do not have to worry about just yet, For the purpose of this and the next exercise we will use a basic implementation that makes use of the shared memory.

The image below shows the difference between a straightforward sequential sum (a) and a parallel reduction sum (b).



Let's start by adding some headers:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <omp.h>
```

Create main function and specify the size of array that will be used:

```
int N = 50000000;
```

Get the device properties and find out its maximum number of threads:

```
cudaDeviceProp deviceProperties;
cudaGetDeviceProperties(&deviceProperties, 0);
int numThreads = deviceProperties.maxThreadsPerBlock;
```

Set the grid and block sizes and print out the details. Here we use the detected maximum number of threads per block to specify block dimension. Then the number of blocks that are necessary to compute the array of N size is $N/\text{numThreads}$. However, we use $(N+\text{numThreads}-1)/\text{numThreads}$ to make sure we have always enough blocks. For example in the case when $N < \text{numThreads}$.

```
dim3 block(numThreads, 1, 1);
dim3 grid((N+numThreads-1)/numThreads, 1, 1);

printf("\n===== \n");
printf("Single-GPU reduction sum \n");
printf("===== \n \n");
printf("Total number of elements to sum: %i \n", N);
printf("Kernel launch configuration: %i blocks of %i threads \n", grid.x, block.x);
```

Allocate host memory:

```
int *data_h;
data_h = (int*)malloc(N*sizeof(int));
```

Generate random data, calculate sum on host and measure the time:

```

srand(time(NULL));
for (int i=0; i<N; i++) data_h[i] = rand()%10;
int sumCPU = 0;
printf("Calculating sum on host...\n");
float startCPU = omp_get_wtime();
for (int i=0; i<N; i++) sumCPU += data_h[i];
float endCPU = omp_get_wtime();
float timeCPU = (endCPU-startCPU)*1000;

```

Now, let's allocate device memory and copy the host memory to device:

```

int *data_d;
int *blockSum_d;

cudaMalloc((void **)&data_d, N * sizeof(int));
cudaMalloc((void **)&blockSum_d, grid.x * sizeof(int));

cudaMemcpy(data_d, data_h, N * sizeof(int), cudaMemcpyHostToDevice);

```

Start the GPU timer;

```

float timeGPU;
cudaEvent_t start;
cudaEvent_t stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

```

Launch level 0 kernel. This might be all that is necessary if the number of elements in our array is less than the maximum number of threads per block. If this is not the case then we will launch consequent kernels that will sum the previous kernel sums. If it sounds too confusing have a look at the picture on slide 5 in [this presentation](#).

```

printf("Level 0 kernel summing %i elements with %i blocks of %i threads...\n",
N, grid.x, block.x);
reduceKernel<<<grid, block, block.x*sizeof(int)>>>(data_d, blockSum_d, N);

```

If necessary launch the same kernel recursively until we have the final sum stored in the first index of blockSum_d:

```

int remainingElements = grid.x;
int level = 1;
while(remainingElements > 1)
{
    int numThreads = (remainingElements<block.x) ?
nextPow2(remainingElements) : block.x;
    int numBlocks = (remainingElements + numThreads - 1) / numThreads;
    printf("Level %i kernel summing %i elements with %i blocks of %i
threads...\n", level, remainingElements, numBlocks, numThreads);
    reduceKernel<<<numBlocks, numThreads,
numThreads*sizeof(int)>>>(blockSum_d, blockSum_d, remainingElements);
    remainingElements = numBlocks;
    level++;
}

```

Stop the timer, copy the value from the first index of blockSum_d to host and print the results:

```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&timeGPU, start, stop);

int sumGPU;
cudaMemcpy(&sumGPU, blockSum_d, sizeof(int), cudaMemcpyDeviceToHost);
printf("CPU result: %i    processing time: %f ms\n", sumCPU, timeCPU);
printf("GPU result: %i    processing time: %f ms\n", sumGPU, timeGPU);
```

Indeed, you might need the actual kernel. To better understand what is this kernel doing have a look on slide 8 [in this presentation](#) or ask me.

```
__global__ void reduceKernel(int *input, int *output, int N)
{
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    extern __shared__ int sdata[];
    sdata[tid] = 0.0f;

    if(i < N)
    {
        sdata[tid] = input[i];
        __syncthreads();

        for(int s=1; s < blockDim.x; s*=2)
        {
            if (tid%(2*s) == 0) sdata[tid] += sdata[tid + s];
            __syncthreads();
        }

        if(tid == 0) output[blockIdx.x] = sdata[0];
    }
}
```

And this little function that gives you the next power of 2 number:

```
int nextPow2(int x)
{
    --x;

    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;

    return ++x;
}
```

Source code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <omp.h>

#define TILE_WIDTH 16

//a simple version of matrix_multiply which issues redundant loads from off-
chip global memory
__global__ void matrixMultiplySimple(float *a, float *b, float *c, int width)
{
    //calculate the row and column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float result = 0;

    //do dot product between row of a and column of b
    for(int i=0; i<width; i++)
    {
        result += a[row*width+i] * b[i*width+col];
    }

    //write out this thread's result
    c[row*width+col] = result;
}

//an optimized version of matrix_multiplication which eliminates redundant
loads
__global__ void matrixMultiplyOptimised(float *a, float *b, float *c, int
width)
{
    //create shorthand names for threadIdx & blockIdx
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //allocate 2D tiles in __shared__ memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    //calculate the row & column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float result = 0;

    //loop over the tiles of the input in phases
    for(int p=0; p<width/TILE_WIDTH; p++)
    {
        //collaboratively load tiles into __shared__
        s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
        s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];

        //wait until all data is loaded before allowing any thread in
this block to continue
        __syncthreads();

        //do dot product between row of s_a and column of s_b
        for(int i=0; i<TILE_WIDTH; i++)
        {
```

```

        result += s_a[ty][i] * s_b[i][tx];
    }

    //wait until all threads are finished with the data before
    //allowing any thread in this block to continue
    __syncthreads();
}

//write out this thread's result
c[row*width+col] = result;
}

void matrixMultiplyCPU(float *a, float *b, float *c, int width)
{
    float result;

    for(int row=0; row<width; row++)
    {
        for(int col=0; col<width; col++)
        {
            result = 0;
            for(int k = 0; k<width; k++)
            {
                result += a[row * width + k] * b[k * width + col];
            }
            c[row * width + col] = result;
        }
    }
}

int main(void)
{
    //the width of the matrix (not the number of total elements)
    int N = 1024;

    //grid and block size
    dim3 block(TILE_WIDTH, TILE_WIDTH);
    dim3 grid(N/block.x, N/block.y);

    //host memory pointers
    float *a_h = NULL;
    float *b_h = NULL;
    float *c_h = NULL;

    //allocate host memory
    size_t memSize = (N*N) * sizeof(float);
    a_h = (float *) malloc(memSize);
    b_h = (float *) malloc(memSize);
    c_h = (float *) malloc(memSize);

    //generate random input
    for(int i = 0; i < N*N; ++i)
    {
        a_h[i] = (float)(rand()/RAND_MAX);
        b_h[i] = (float)(rand()/RAND_MAX);
    }

    //device memory pointers
    float *a_d = NULL;
    float *b_d = NULL;
    float *c_d = NULL;

```

```

//allocate device memory
cudaMalloc((void**)&a_d, (N * N) * sizeof(float));
cudaMalloc((void**)&b_d, (N * N) * sizeof(float));
cudaMalloc((void**)&c_d, (N * N) * sizeof(float));

//copy input to the device
cudaMemcpy(a_d, a_h, (N * N) * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(b_d, b_h, (N * N) * sizeof(float), cudaMemcpyHostToDevice);

//get start time
float cpuStart = omp_get_wtime();

printf("Measuring CPU execution time...\n");
matrixMultiplyCPU(a_h, b_h, c_h, N);

//get end time
float cpuEnd = omp_get_wtime();
float cpuTime = (cpuEnd-cpuStart)*1000;

//CUDA events to measure time
cudaEvent_t start;
cudaEvent_t stop;
float simpleKernelTime;
float optimisedKernelTime;

//start timer
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

//launch simple kernel multiple times
printf("Measuring execution time of the simple kernel...\n");
matrixMultiplySimple<<<grid, block>>>(a_d, b_d, c_d, N);

//stop timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&simpleKernelTime, start, stop);

//start timer
cudaEventRecord(start, 0);

//launch optimised kernel multiple times
printf("Measuring execution time of the optimised kernel...\n");
matrixMultiplyOptimised<<<grid, block>>>(a_d, b_d, c_d, N);

//stop timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&optimisedKernelTime, start, stop);

//print out execution times
printf("Naive CPU implementation time: %f ms\n", cpuTime);
printf("Naive GPU implementation time: %f ms\n", simpleKernelTime);
printf("Optimised GPU implementaion time: %f ms\n",
optimisedKernelTime);

//free device memory
cudaFree(a_d);
cudaFree(b_d);
cudaFree(c_d);

```

```
    //free host memory  
    free(a_h);  
    free(b_h);  
  
    return 0;  
}
```