MONASH University

FIT3143 Tutorial Week 12

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (MU Malaysia)

INTRODUCTION TO GENERAL PURPOSE COMPUTING USING GPUS (GPGPU)

OBJECTIVES

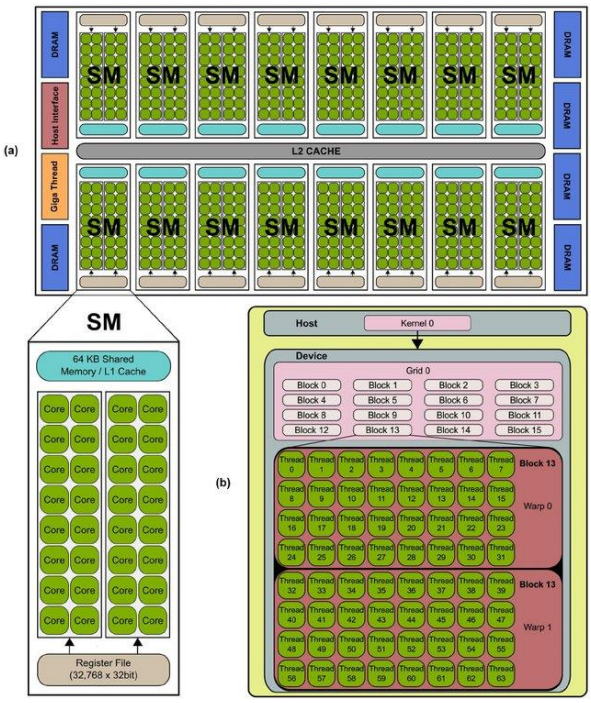
- Understand the concepts of the GPU architecture and GPGPU programming
- Understand the concepts of the CUDA programming framework

Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

QUESTIONS

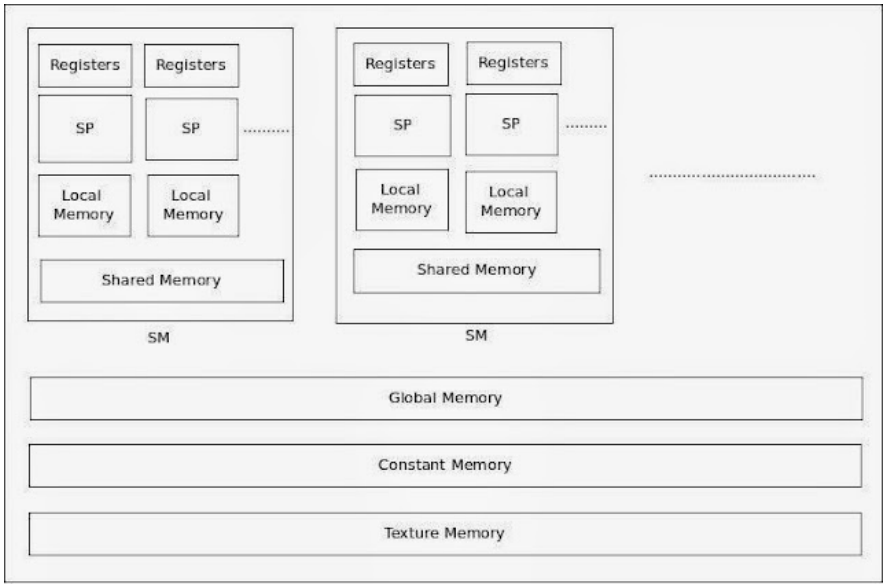
1) What are the components of a GPU in relation to the CUDA programming model? Within a GPU, what is a CUDA core?

GPU consists of grids which contain smaller components called as Stream Multiprocessors (SM) or blocks. Each SM consists of many Stream Processors (SP) on which actual computation is done using threads. Each SP is also called a CUDA core.



From the figure above, The GPU is comprised of a set of SMs. Each SM is comprised of several SP cores, as shown for the NVIDIA's Fermi architecture (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b). Source: doi:10.1371/journal.pone.0061892.g001

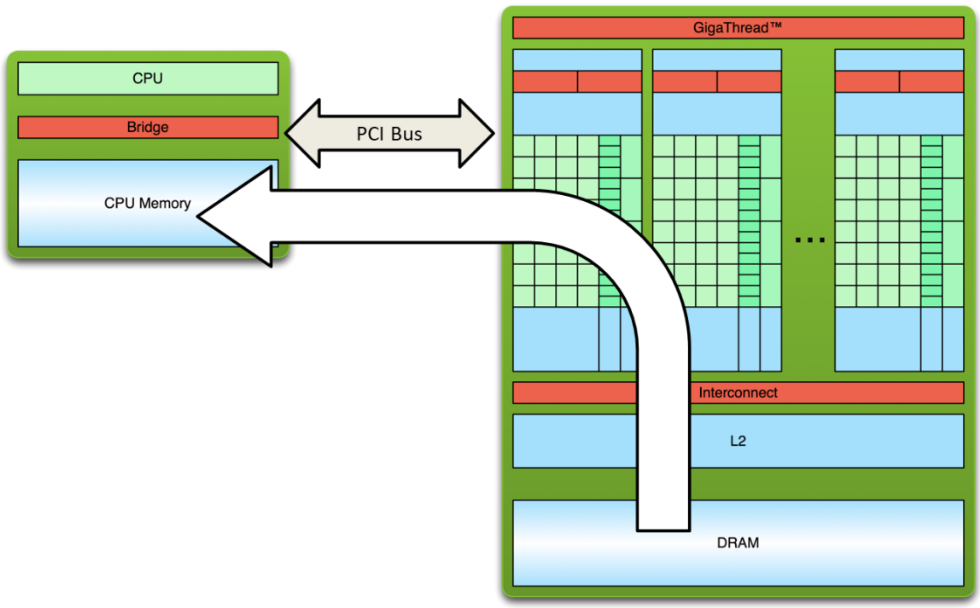
2) What are the different memories used in a GPU?



With reference to the figure above:

a) Local Memory - Each SP uses Local memory. All variables declared in a kernel (a function to be executed on GPU) are saved into Local memory.
b) Registers - Kernel may consist of several expressions. During execution of an expression, values are saved into the Registers of SP.
c) Global Memory - It is the main memory of GPU. Whenever a memory from GPU is allocated for variables by using cudaMalloc() function, by default, it uses global memory.
d) Shared Memory - On one SP, one or more threads can be run. A collection of threads is called a Block. On one SM, one or more blocks can be run. Shared memory is shared by all the threads in one block. Shared memory is used to reduce the latency(memory access delay). How? See, Global memory is very big in size as compared to shared memory. So, search time for a location of variable is lesser for shared memory compared to global memory. Keep in mind, shared memory in one SM is shared by all threads in one block. When we must use shared memory for a variable, it should be prefixed with keyword __shared__ during its declaration. For e.g., __shared__ int x.
e) Constant Memory - Constant memory is also used to reduce latency. But constant memory is used in only those situations, when multiple threads has to access same value. How constant memory reduces the latency, I will explain now. Suppose there are 32 threads in one block. Let all of them are using the same variable. Hence there will be 32 accesses from global memory. Now if we store the variable in constant memory. Then first thread will access the value of a variable and will broadcast this value to other threads in half warp. This value will be saved in a cache and will be provided to the threads of another half warp. Hence total accesses will be just one instead of 32. Constant memory can be used for a variable by prefixing keyword __constant__ in the variable declaration (e.g., __constant__ int x).
f) Texture Memory - Texture memory is again used to reduce the latency. Texture memory is used in a special case. Consider an image. When we access a particular pixel, there are more chances that we will access surrounding pixels. Such a group of values which are accessed together are saved in texture memory.

3) With the aid of an illustration, explain the simple processing flow of a CUDA programming model (i.e., from CPU to GPU and vice versa).



A brief answer:

With reference to the figure above:

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

For a more in-depth illustration steps, please refer to slides #9 to #11 in Week 11c lecture notes.

4) The following CUDA C code implements a parallel vector addition. By referring to this code, explain the following code syntaxes:

a) `__global__`
b) `blockIdx.x`
c) `blockDim.x`
d) `threadIdx.x`
e) `cudaMalloc`
f) `cudaMemcpy`
g) `vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);` - *focus on the angular brackets and the parameters in it.*
h) Continuing from (g), why is it important to have threads linked to a block? Why can't we just have blocks or what is the benefit of having multiple threads within a block?

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
```

```
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
```

```
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1 within
error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

a) The __global__ decorator specifies this is a CUDA kernel, which is executed on the device (GPU) and called from the host (CPU).
b) blockIdx contains the blocks position in the grid, ranging from 0 to gridDim-1.
c) blockDim.x contains the number of threads in the block.
d) threadIdx is the threads index inside of it's associated block, ranging from 0 to blockDim-1.
e) Given a pointer cudaMalloc will allocate memory in the devices global memory space and set the past pointer to point to this memory.
f) By calling cudaMemcpy we initiate a memory transfer between host and device or vice versa.
g) We launch our kernel with a modified C function syntax that lets us specify the grid and block sizes. In the triple angular brackets, the first parameter indicates number of blocks while second parameter indicates the number of threads per block.
h) Blocks are executed on SMs. Each SM has multiple SPs where the threads are executed on these SPs. By calling multiple blocks but a single thread per block, this represents an inefficient approach in maximizing the performance of GPU for parallelism. Besides multiple threads in a block can access a shared memory, which further improves the parallel performance of a parallel algorithm.

Note: Answers to the questions above were obtained by referring to the following online sites:

- https://www.comrevo.com/2017/05/interview-questions-on-cuda-programming.html
- https://www.tutorialspoint.com/cuda/cuda_threads.htm

- https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/