Lab No 5: **Date:**

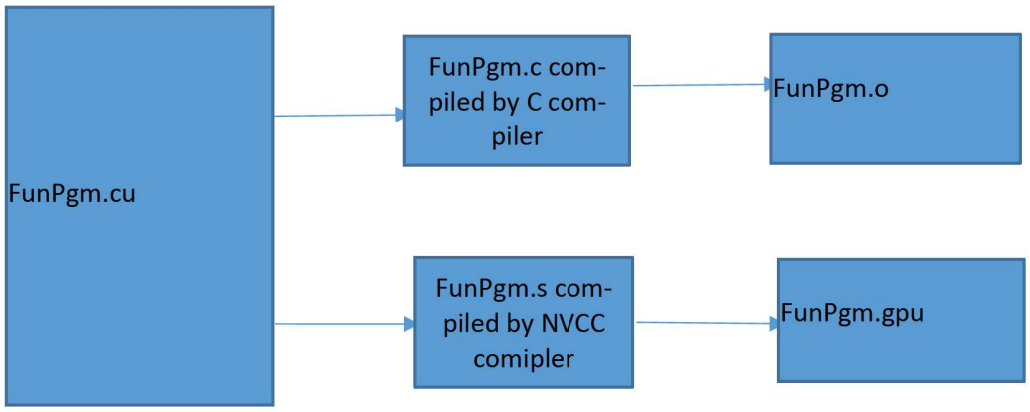
Programs on arrays in CUDA

Objectives:

In this lab, student will be able to

1. Know the basics of Computing Unified Device Architecture (CUDA).
2. Learn program structure of CUDA.
3. Learn about CUDA 1D blocks and threads
4. Write simple programs on one dimensional arrays
5. Learn mathematical functions in CUDA

About CUDA: CUDA is a platform for performing massively parallel computations on graphics accelerators. CUDA was developed by NVIDIA. It was first available with their G8X line of graphics cards. CUDA presents a unique opportunity to develop widely-deployed parallel applications. The CUDA programs are compiled as follows.



FunPgm.cu is compiled by both C compiler and Nvidia CUDA C compiler (NVCC compiler). If you have both main.c and Funpgm.cu then you can call cuda API's in main.c but keep in mind that you cannot call kernel from main.c. To call the kernel file extension must be .cu.

As in OpenCL CPU is the host and its memory the host memory and GPU is the device and its memory device memory. Serial code will be run on host and parallel code will be run on device.

1. Copy data from host memory to device memory.
2. Load device program and execute, caching data on chip for performance.
3. Copy result from device memory to host memory.

CUDA threads, blocks and grid

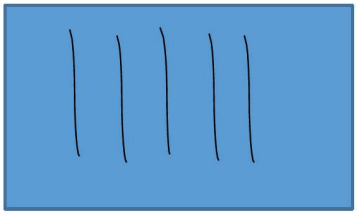
Thread – Distributed by the CUDA runtime. A single path of execution there can be multiple threads in a program.

(identified by threadIdx)

CUDA Thread

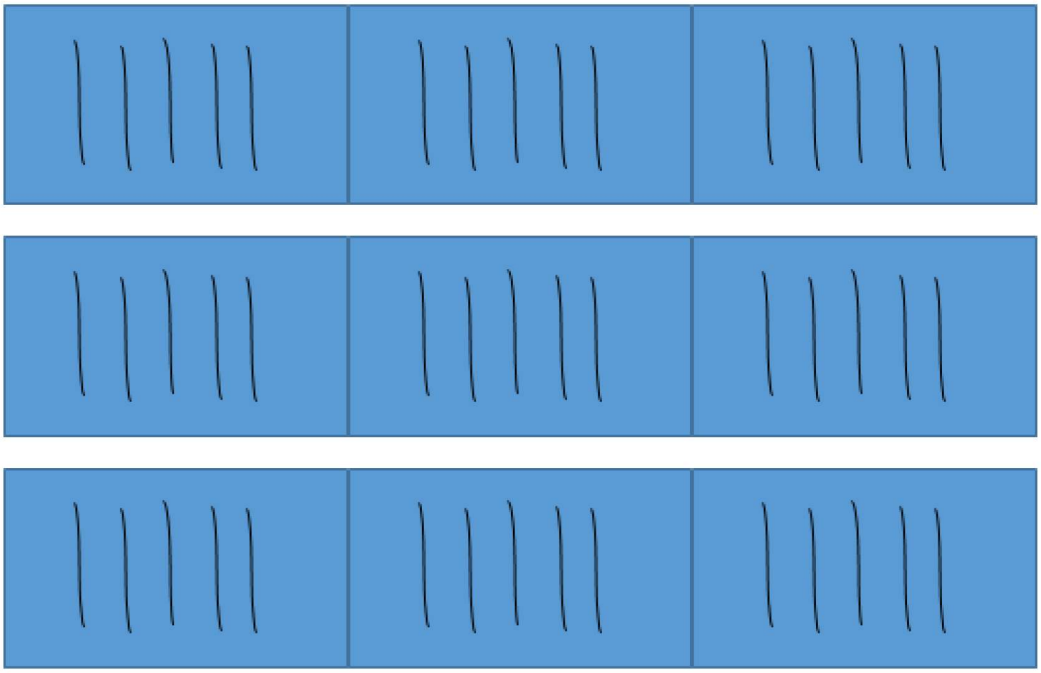
Block – A user defined group of 1 to 512 threads.

(identified by blockIdx)



CUDA Block

Grid – A group of one or more blocks. A grid is created for each CUDA kernel function



CUDA GRID

Some of the calculations for indexing the thread is given below.

```
1D grid of 1D blocks
__device
int getGlobalIdx_1D_1D(){
return blockIdx.x *blockDim.x + threadIdx.x;
}
1D grid of 2D blocks
__device
int getGlobalIdx_1D_2D(){
return blockIdx.x * blockDim.x * blockDim.y
 + threadIdx.y * blockDim.x + threadIdx.x;
}
1D grid of 3D blocks
__device
int getGlobalIdx_1D_3D(){
return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
 + threadIdx.z * blockDim.y * blockDim.x
 + threadIdx.y * blockDim.x + threadIdx.x;
}
2D grid of 1D blocks
__device__ int getGlobalIdx_2D_1D(){
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int threadId = blockId * blockDim.x + threadIdx.x;
```

```
return threadId;
}
2D grid of 2D blocks
__device
int getGlobalIdx_2D_2D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y)
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
2D grid of 3D blocks
__device
int getGlobalIdx_2D_3D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
 + (threadIdx.z * (blockDim.x * blockDim.y))
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
3D grid of 1D blocks
__device
int getGlobalIdx_3D_1D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * blockDim.x + threadIdx.x;
```

```
return threadId;
}
3D grid of 2D blocks
__device
int getGlobalIdx_3D_2D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * (blockDim.x * blockDim.y)
 + (threadIdx.y * blockDim.x) + threadIdx.x;
rcturn thrcadId;
}
3D grid of 3D blocks
__device
int getGlobalIdx_3D_3D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
 + (threadIdx.z * (blockDim.x * blockDim.y))
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
```

Solved Exercise: Program to add two numbers.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void add(int *a, int *b, int *c)
        {
                *c = *a + *b;
        }

int main(void) {
                int a, b, c;                    // host copies of variables a, b & c
                int *d_a, *d_b, *d_c;       // device copies of variables a, b & c
                int size = sizeof(int);

                // Allocate space for device copies of a, b, c
                cudaMalloc((void **)&d_a, size);
                cudaMalloc((void **)&d_b, size);
                cudaMalloc((void **)&d_c, size);
                // Setup input values
                a = 3;
                b = 5;
// Copy inputs to device
                cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
                cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
                // Launch add() kernel on GPU
                add<<<1,1>>>(d_a, d_b, d_c);
                // Copy result back to host
                cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
                printf("Result : %d",c);
                // Cleanup
                cudaFree(d_a);
                cudaFree(d_b);
                cudaFrcc(d_c);
                return 0;
        }
```

Explanation:

add is the function which runs on device.

```
cudaMalloc ((void **)&d_a, size);
```

cudaMalloc will allocate memory of size bytes given as second argument to variable passed as first argument.

```
cudaMemcpy (Destination,Source,Size,Direction);

cudaMemcpy (d_a, &a, size, cudaMemcpyHostToDevice);

cudaMemcpy (&c, d_c, size, cudaMemcpyDeviceToHost);
```

cudaMemcpy copies the variables from host to device or device to host based on the direction which is either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Value of size bytes long is copied from source to destination.

cudaFree frees the memory allocated by cudaMalloc.

The add function is called like this add<<<1,1>>>(d_a,d_b,d_c). The add is followed by three angular brackets then the number of blocks, threads per block then corresponding closing angular brackets then how many arguments the function add takes is enclosed within parenthesis. If you want to add N elements you can achieve it in two ways either having N blocks or having N threads.

That is pass an array with following function calls

add<<< N,1>>> (d_a,d_b,d_c) or add<<< 1, N>>> (d_a,d_b,d_c)

Few Mathematical functions in CUDA:

Major Single-Precision floating point functions: Single precision functions work on float value(32 bit). A float value is stored in IEEE 754 format.

Function	**Description**
sqrtf(x)	Square root function
expf(x)	Exponentiation function. Base = e
exp2f(x)	Exponentiation function. Base = 2
exp10f(x)	Exponentiation function. Base = 10
logf(x)	Logarithmic function. Base=e
log2f(x)	Logarithmic function. Base=2
log10f(x)	Logarithmic function. Base=10
sinf(x)	sine function

cosf(x)	cos function
tanf(x)	tan function
powf(x,y)	power function
truncf(x)	truncation function
roundf(x)	round function
ceilf(x)	ceil function
floorf(x)	floor function

Major Double-Precision floating point functions: Double precision functions work on double value(64 bit). A double value is stored in IEEE 754 format.

Function	**Description**
sqrt(x)	Square root function
exp(x)	Exponentiation function. Base = e
exp2(x)	Exponentiation function. Base = 2
exp10(x)	Exponentiation function. Base = 10
log(x)	Logarithmic function. Base=e
log2(x)	Logarithmic function. Base=2
log10(x)	Logarithmic function. Base=10
sin(x)	sine function
cos(x)	cos function
tan(x)	tan function
pow(x,y)	power function
trunc(x)	truncation function
round(x)	round function

ceil(x)	ceil function
floor(x)	floor function

Steps to execute a CUDA program is provided in the form of video which is made available in individual systems.

Lab Exercises:

1. Write a program in CUDA to add two vectors of length N using
   a) block size as N b) N threads
2. Implement a CUDA program to add two vectors of length N by keeping the number of threads per block as 256 (constant) and vary the number of blocks to handle N elements.
3. Write a program in CUDA to process a 1D array containing angles in radians to generate sine of the angles in the output array. Use appropriate function.

Additional Exercises:

1. Write a program in CUDA to perform linear algebra function of the form $y = \alpha x + y$, where x and y are vectors and α is a scalar value.
2. Write a program in CUDA to sort every row of a matrix using selection sort.
3. Write a program in CUDA to perform odd even transposition sort in parallel.