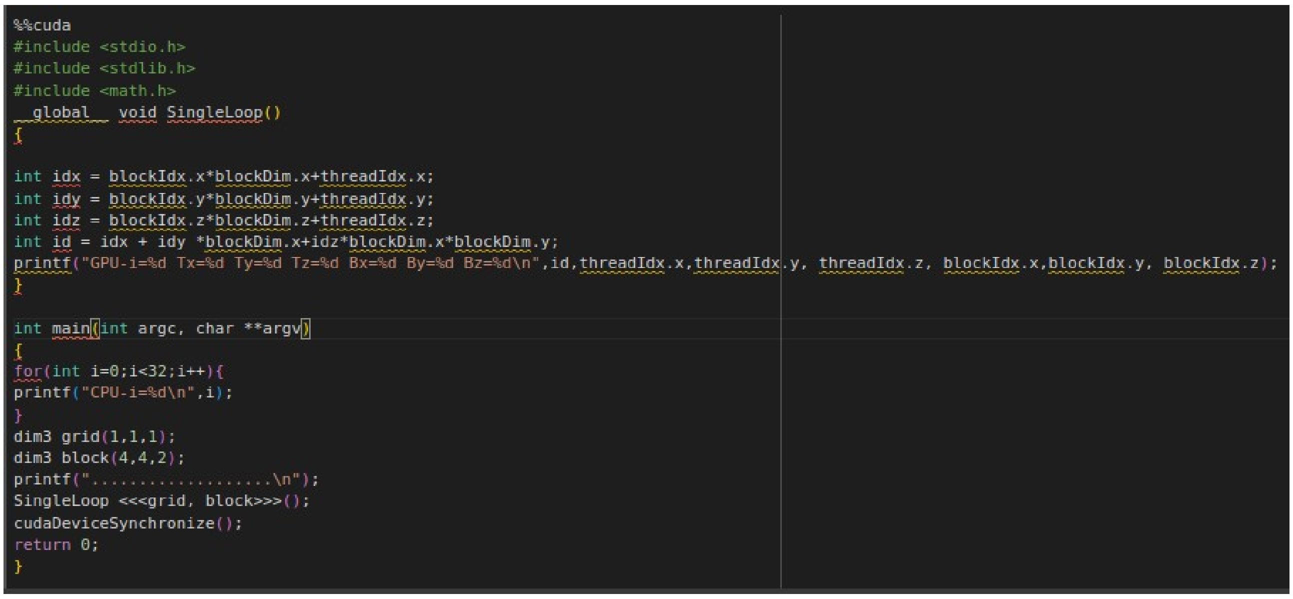
Nithin S
221IT085

IT301 Lab Assignment 09

Q1.To understand device variables execute the following program and analyze the result for following.

Code

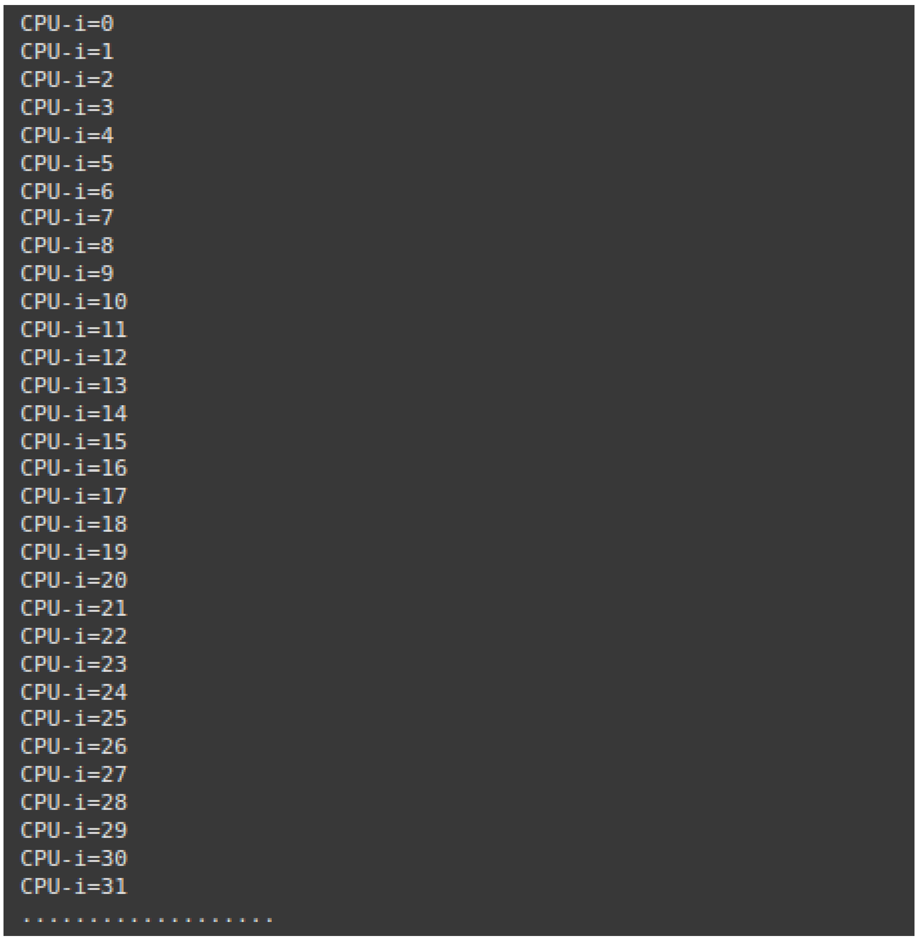


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(4,4,2);
printf("..................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
...................
```

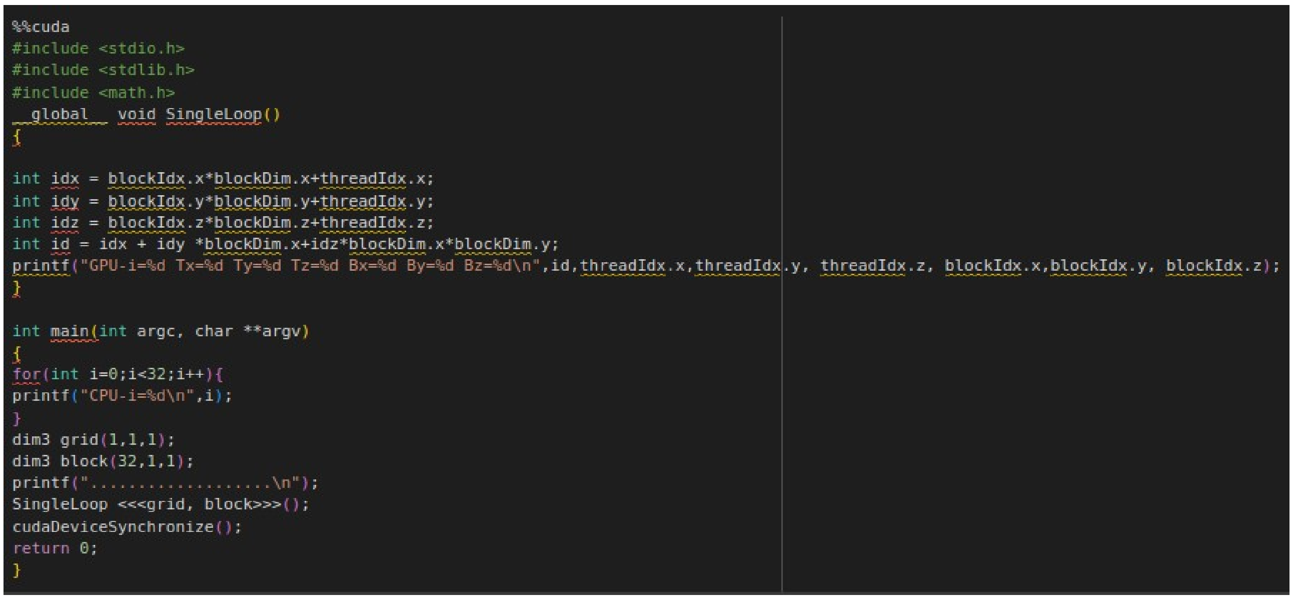
Analysis

This CUDA code is structured to demonstrate how threads in a GPU are indexed and mapped in a three-dimensional grid and block structure. The SingleLoop kernel calculates a unique identifier id for each thread based on its position in a 3D grid (gridDim) and block (blockDim) using thread and block indices (threadIdx and blockIdx). It then prints each thread's id along with its individual thread and block indices, which help illustrate how threads are organized in GPU programming.

In the main function, a CPU-based loop first prints "CPU-i" from 0 to 31, likely to show some sequential CPU-side operations. Then, the SingleLoop kernel is launched with a grid of one block (dim3 grid(1,1,1)) and a block size of 4x4x2 (totaling 32 threads). This kernel configuration implies that a single 3D block containing 32 threads is launched, and each thread outputs its own indices and calculated id. The call to cudaDeviceSynchronize() ensures that the CPU waits for all GPU threads to complete their execution before the program terminates. The output provides a clear insight into how each thread's identifier is derived based on its position, making this code useful for understanding thread indexing in CUDA.

a) Consider following dimensions and observe result. Threads in x direction is 32.
dim3 grid(1,1,1);
dim3 block(32,1,1);

Code

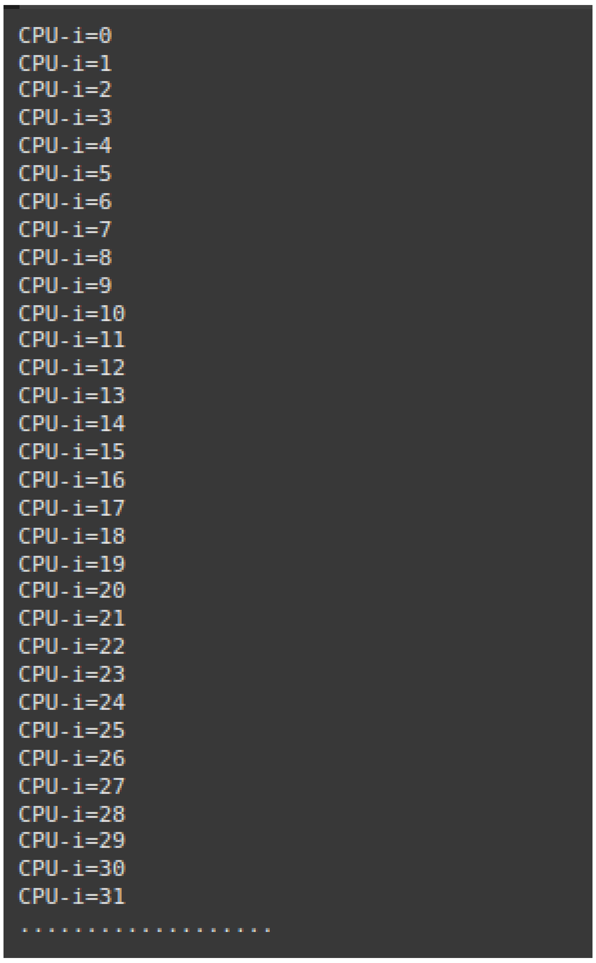


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(32,1,1);
printf("...................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
...................
```

Analysis

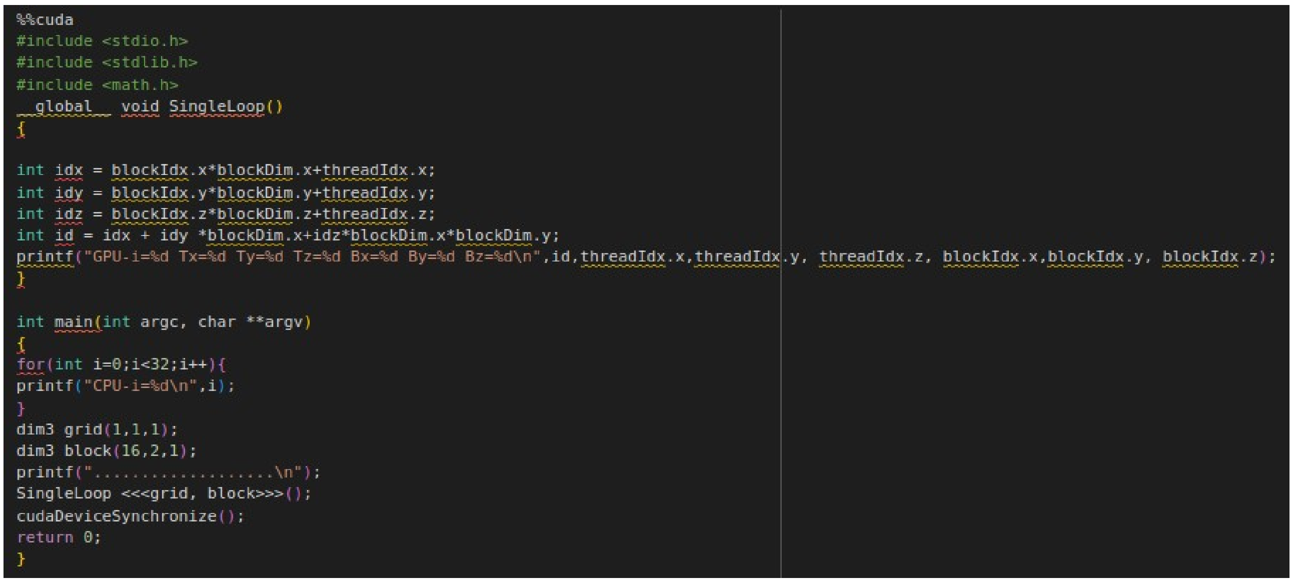
This CUDA program demonstrates thread indexing and how threads are uniquely identified within a block. The `SingleLoop` kernel calculates a unique thread ID (`id`) based on the thread and block indices in each dimension. Specifically, `idx`, `idy`, and `idz` represent the thread's position in the x, y, and z dimensions, respectively, across the grid and block structures. The final `id` calculation combines these indices to create a unique identifier for each thread based on its 3D coordinates.

In this specific setup, `dim3 grid(1,1,1);` and `dim3 block(32,1,1);` mean that there is only one block in the grid, and this block has 32 threads in the x-direction and 1 thread in both the y- and z-directions, giving a total of 32 threads. Each thread in the block will output its unique ID along with the values of `threadIdx` and `blockIdx` for all three dimensions, which are used for the ID calculation.

The `main` function also includes a loop that prints "CPU-i" from 0 to 31, demonstrating sequential execution on the CPU before the GPU kernel is launched. This is followed by the GPU kernel call, where the kernel's print statements reveal the thread organization, with `threadIdx.x` ranging from 0 to 31 and `threadIdx.y` and `threadIdx.z` staying at 0. The `cudaDeviceSynchronize()` function ensures that the GPU execution completes before the program terminates.

b) Consider following dimensions and observe result. Threads in x direction is 16. y is 2.
dim3 grid(1,1,1);
dim3 block(16,2,1);

Code

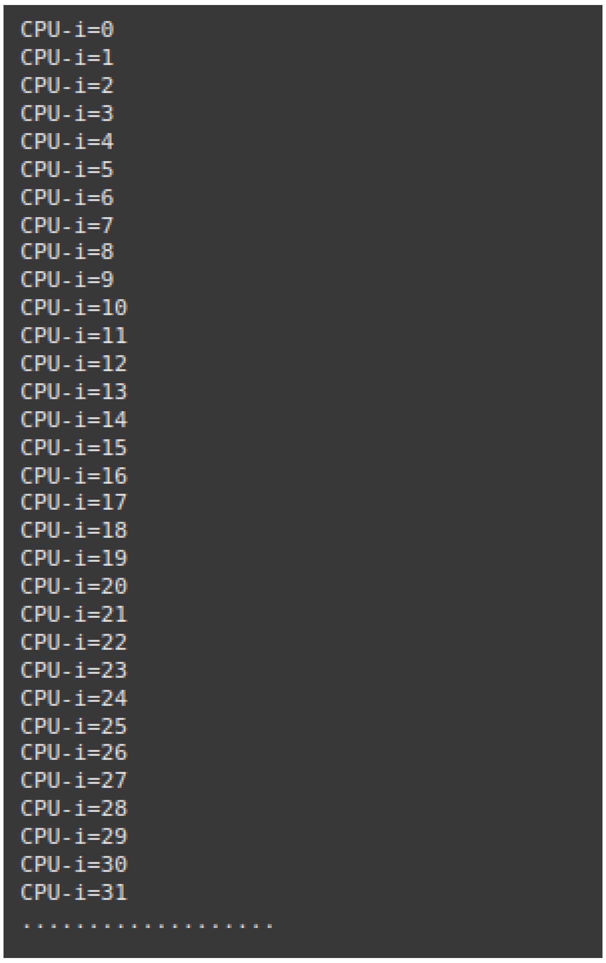


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(16,2,1);
printf("...................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
...................
```

Analysis

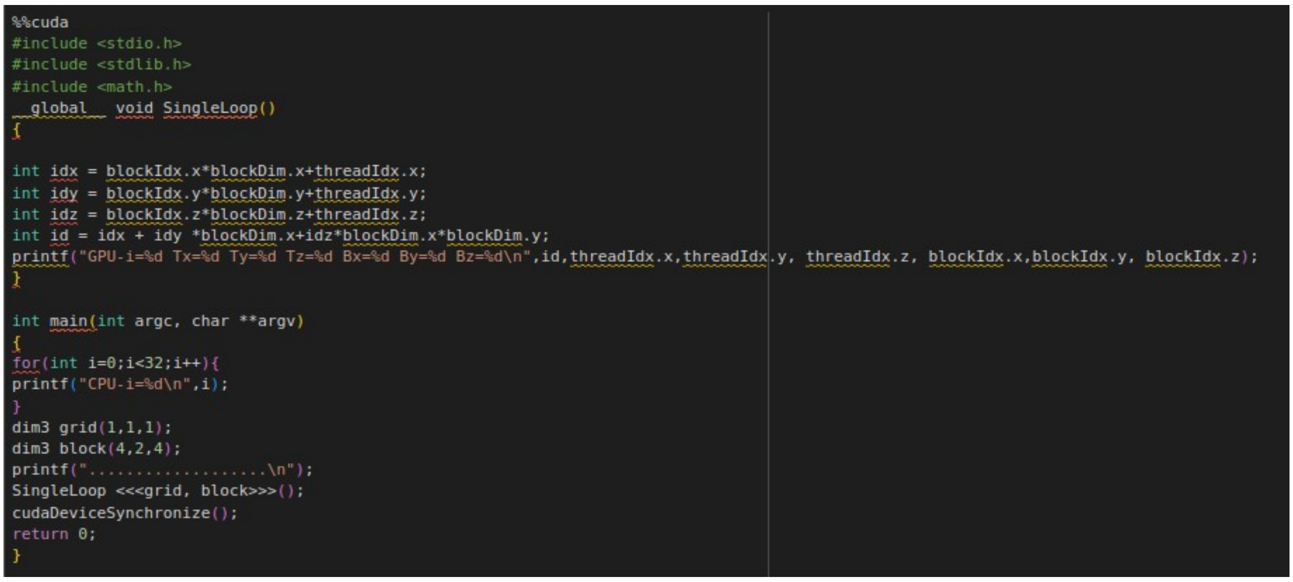
This CUDA code provides a demonstration of how threads are organized and uniquely identified within a 3D block structure. The `SingleLoop` kernel function calculates a unique identifier (`id`) for each thread in the block by combining its x, y, and z coordinates, represented by `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. These indices are used in the `id` calculation along with the block's dimensions (`blockDim.x`, `blockDim.y`, and `blockDim.z`) to create a unique linear index.

In this setup, `dim3 grid(1,1,1);` and `dim3 block(16,2,1);` specify that there is a single block in the grid with 16 threads along the x-axis, 2 threads along the y-axis, and 1 thread along the z-axis, resulting in a total of $16\times2\times1=32$ threads. Each thread outputs its calculated `id`, along with its individual thread and block indices for all three dimensions (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`, `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`).

In the `main` function, a loop on the CPU prints "CPU-i" from 0 to 31 to indicate sequential CPU-side processing. Then, the `SingleLoop` kernel is launched on the GPU, with `cudaDeviceSynchronize()` ensuring the CPU waits until all GPU threads complete. In the kernel's output, you would observe `threadIdx.x` ranging from 0 to 15 and `threadIdx.y` ranging from 0 to 1, while `threadIdx.z` remains constant at 0. This setup allows the code to illustrate how threads are organized in a two-dimensional block structure within CUDA.

c) Consider following dimensions and observe result. Threads in x direction is 4. y is 4 and z is 2.
dim3 grid(1,1,1);
dim3 block(4,2,4);

Code

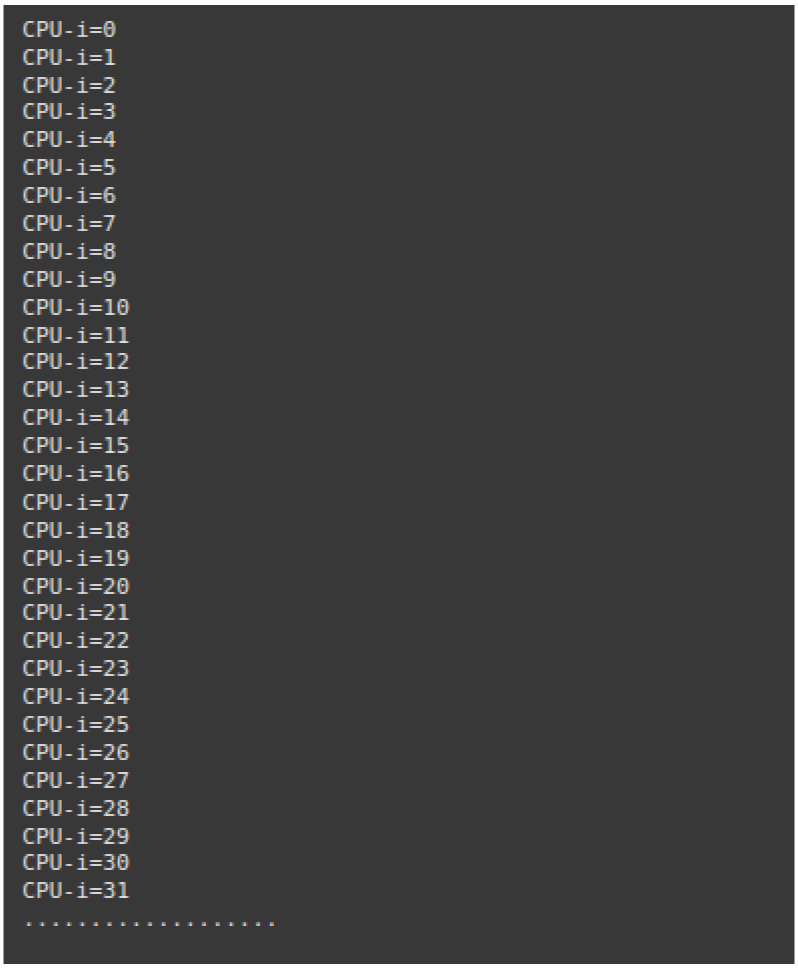


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(4,2,4);
printf("..................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
..................
```

Analysis

This CUDA program uses a kernel function, `SingleLoop`, to illustrate how threads within a block are indexed in a three-dimensional structure. The `SingleLoop` kernel calculates a unique identifier (`id`) for each thread by combining its position in the x, y, and z dimensions (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`) with the block's dimensions (`blockDim.x`, `blockDim.y`, and `blockDim.z`). This unique `id` calculation helps visualize thread indexing within the GPU.

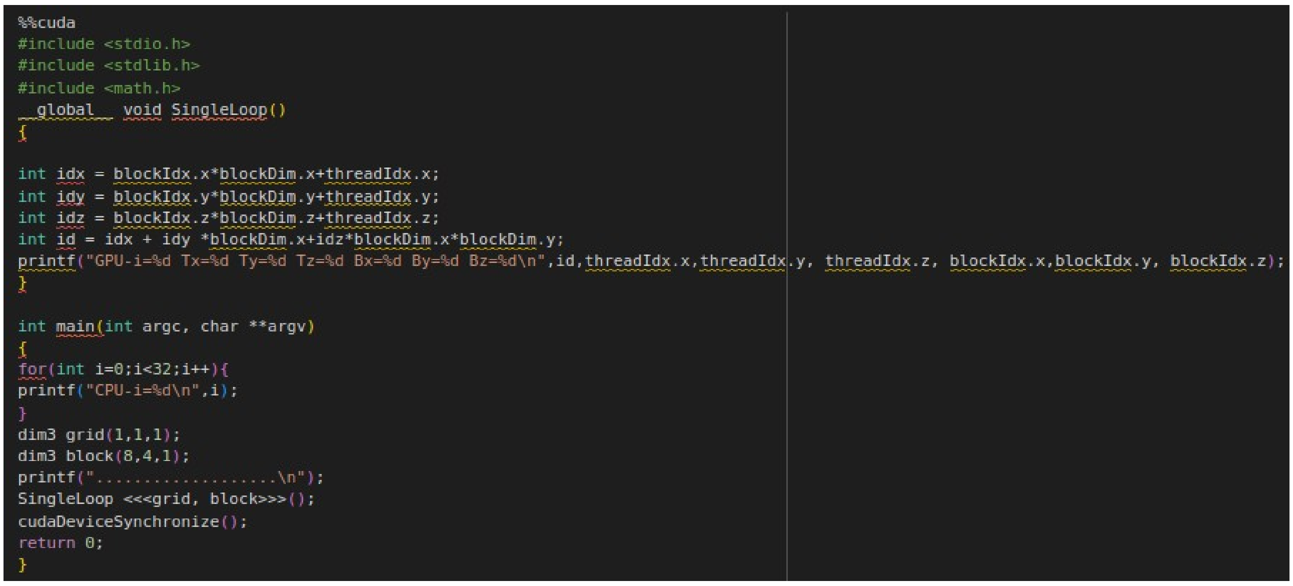
The grid and block dimensions are defined as `dim3 grid(1,1,1);` and `dim3 block(4,2,4);`, which means that there is a single block containing 4 threads along the x-axis, 2 threads along the y-axis, and 4 threads along the z-axis. This configuration results in 4×2×4=32 threads in total within the block.

In the `main` function, the CPU first runs a loop that prints "CPU-i" for values from 0 to 31, demonstrating sequential CPU-side execution. Following this, the `SingleLoop` kernel is launched on the GPU. Each thread in the kernel outputs its calculated `id` along with its thread and block indices for the x, y, and z dimensions (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`, `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`). The call to `cudaDeviceSynchronize()` ensures that the CPU waits for all GPU threads to complete before continuing.

With this configuration, `threadIdx.x` ranges from 0 to 3, `threadIdx.y` from 0 to 1, and `threadIdx.z` from 0 to 3, allowing the output to illustrate how each thread is indexed in a 3D block layout. This setup provides a useful demonstration of how thread coordinates map to a unique identifier, showcasing the spatial organization of threads in CUDA's 3D block model.

d) Consider following dimensions and observe result. Mention the Threads in each direction.
dim3 grid(1,1,1);
dim3 block(8,4,1);

Code

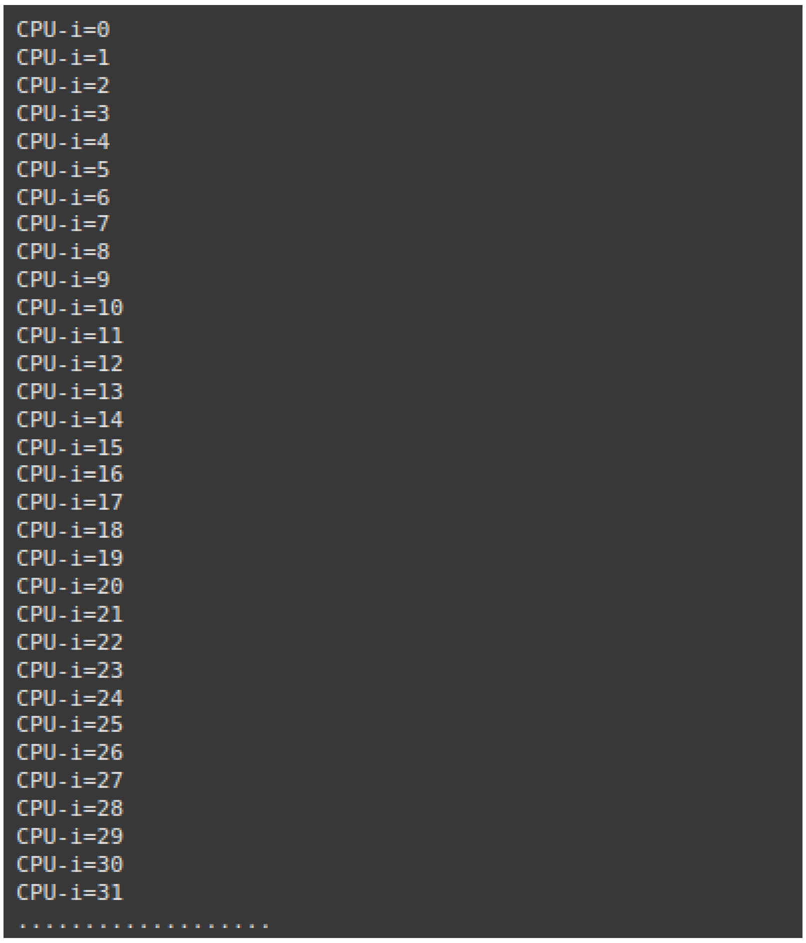


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(8,4,1);
printf("...................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
...................
```

Analysis

This CUDA code uses a kernel function `SingleLoop` to demonstrate thread indexing within a 3D block structure. The `SingleLoop` kernel calculates a unique identifier (`id`) for each thread by combining its coordinates (`threadIdx.x`, `threadIdx.y`, and `threadIdx.z`) within the block. These coordinates are combined with the block dimensions (`blockDim.x`, `blockDim.y`, and `blockDim.z`) to generate a unique `id` for each thread.

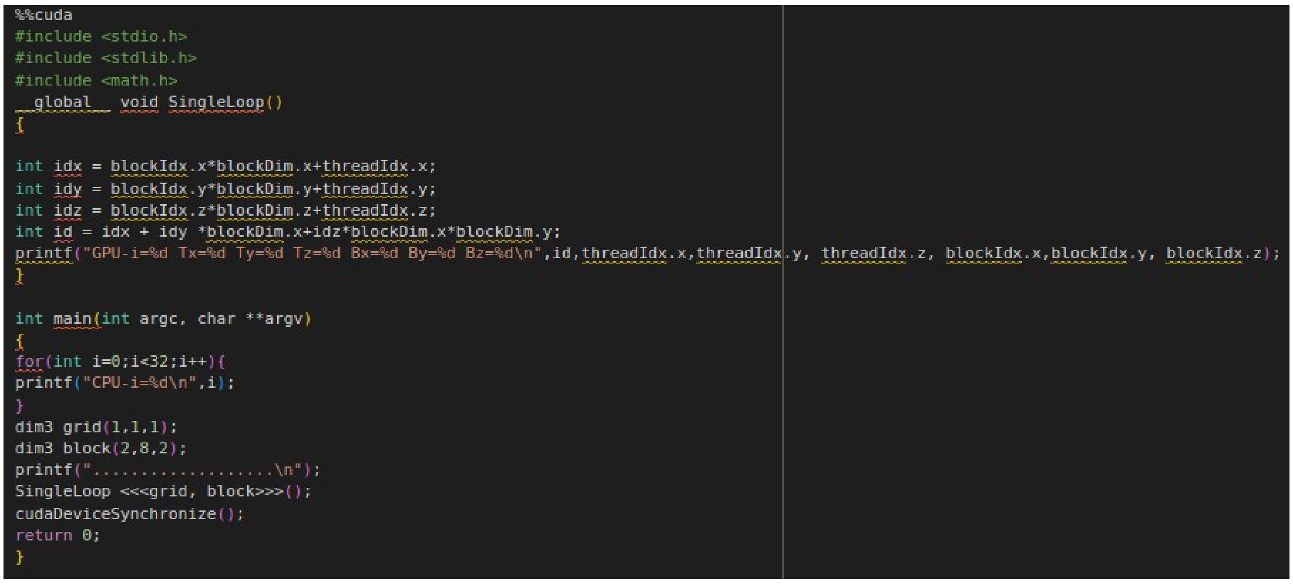
In this setup, the grid and block dimensions are defined as `dim3 grid(1,1,1);` and `dim3 block(8,4,1);`, meaning there is a single block with 8 threads along the x-axis, 4 threads along the y-axis, and 1 thread along the z-axis, totaling 8×4×1=32 threads.

The `main` function begins with a CPU loop that prints "CPU-i" from 0 to 31, simulating CPU-side sequential execution. Then, the GPU kernel `SingleLoop` is launched, with each thread printing its unique `id` and its thread indices (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`) along with block indices (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`). Since there is only one block in this grid, all block indices are zero. The `cudaDeviceSynchronize()` function ensures the CPU waits until all GPU threads finish executing before the program terminates.

In this configuration, `threadIdx.x` will range from 0 to 7, `threadIdx.y` from 0 to 3, and `threadIdx.z` will remain constant at 0. This code effectively demonstrates how threads are organized in a two-dimensional block (with no variation along the z-axis) within CUDA, showcasing thread indexing and how unique IDs are derived from 3D thread layouts.

e) Consider following dimensions and observe result. Mention the Threads in each direction.
dim3 grid(1,1,1);
dim3 block(2,8,2);

Code

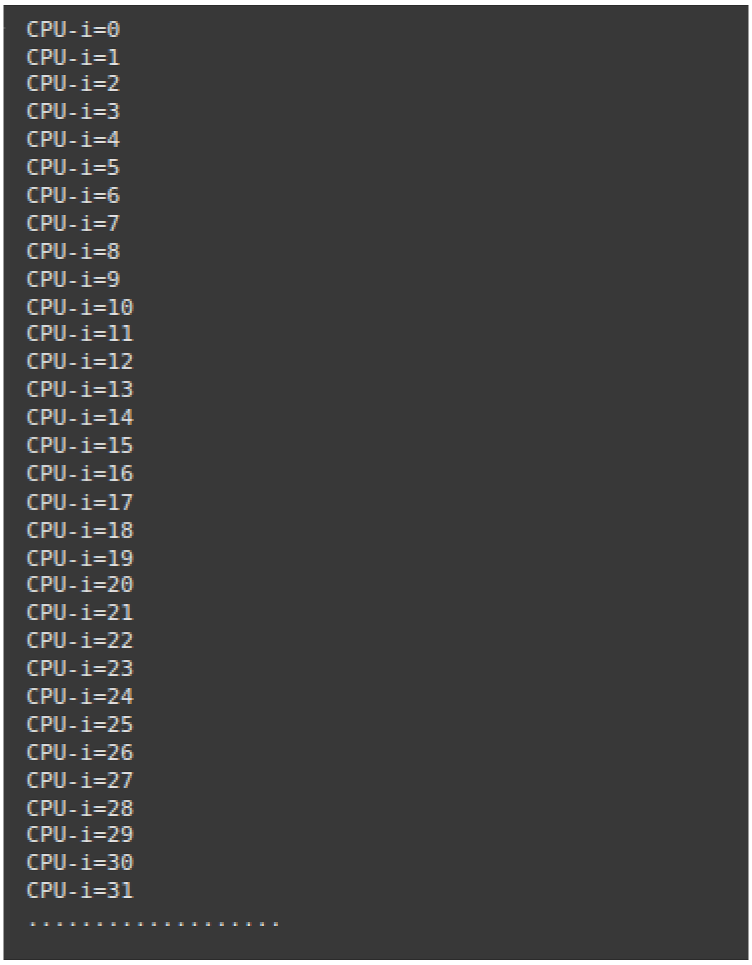


```
%%cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
__global__ void SingleLoop()
{

int idx = blockIdx.x*blockDim.x+threadIdx.x;
int idy = blockIdx.y*blockDim.y+threadIdx.y;
int idz = blockIdx.z*blockDim.z+threadIdx.z;
int id = idx + idy *blockDim.x+idz*blockDim.x*blockDim.y;
printf("GPU-i=%d Tx=%d Ty=%d Tz=%d Bx=%d By=%d Bz=%d\n",id,threadIdx.x,threadIdx.y, threadIdx.z, blockIdx.x,blockIdx.y, blockIdx.z);
}

int main(int argc, char **argv)
{
for(int i=0;i<32;i++){
printf("CPU-i=%d\n",i);
}
dim3 grid(1,1,1);
dim3 block(2,8,2);
printf("...................\n");
SingleLoop <<<grid, block>>>();
cudaDeviceSynchronize();
return 0;
}
```

Output



```
CPU-i=0
CPU-i=1
CPU-i=2
CPU-i=3
CPU-i=4
CPU-i=5
CPU-i=6
CPU-i=7
CPU-i=8
CPU-i=9
CPU-i=10
CPU-i=11
CPU-i=12
CPU-i=13
CPU-i=14
CPU-i=15
CPU-i=16
CPU-i=17
CPU-i=18
CPU-i=19
CPU-i=20
CPU-i=21
CPU-i=22
CPU-i=23
CPU-i=24
CPU-i=25
CPU-i=26
CPU-i=27
CPU-i=28
CPU-i=29
CPU-i=30
CPU-i=31
...................
```

Analysis

This CUDA code demonstrates how threads are indexed within a 3D block structure using a kernel function, `SingleLoop`. Inside the kernel, each thread calculates its unique identifier (`id`) by combining its x, y, and z coordinates within the block (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`) with the block's dimensions (`blockDim.x`, `blockDim.y`, and `blockDim.z`). The unique `id` helps in identifying each thread's specific location in the 3D block layout.

The grid and block dimensions in this code are defined as `dim3 grid(1,1,1);` and `dim3 block(2,8,2);`, which means there is a single block with 2 threads along the x-axis, 8 threads along the y-axis, and 2 threads along the z-axis, totaling 2×8×2=32 threads.

In the `main` function, a loop first runs on the CPU, printing "CPU-i" from 0 to 31 to represent sequential CPU processing. Then, the `SingleLoop` kernel is launched on the GPU, with each thread printing its computed `id` along with its `threadIdx` and `blockIdx` values. Since there is only one block, `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` will all be zero for all threads. The call to `cudaDeviceSynchronize()` ensures that the CPU waits for all GPU threads to finish before exiting.

For this block configuration, `threadIdx.x` will range from 0 to 1, `threadIdx.y` will range from 0 to 7, and `threadIdx.z` will range from 0 to 1. This setup illustrates how threads are arranged in a 3D block with more variation along the y-axis, showing how CUDA threads are organized and indexed in a multi-dimensional layout, and how unique IDs are generated for each thread in such a configuration.