1 Hands-on; GPU parallelization

```
$ lspci | grep -i nvidia
01:00.0 VGA compatible controller: NVIDIA Corporation GP108
[GeForce GT 1030] (rev a1)
01:00.1 Audio device: NVIDIA Corporation GP108 High Definition
Audio Controller (rev a1)

Installation Instructions:
$ wget https://developer.download.nvidia.com/compute/cuda/repos/
ubuntu2204/x86_64/cuda-keyring_1.0-1_all.deb
$ sudo dpkg -i cuda-keyring_1.0-1_all.deb
$ sudo apt-get update
$ sudo apt-get -y install cuda
$ sudo reboot
$ export PATH=/usr/local/cuda-12.0/bin${PATH:+:${PATH}}
$ export LD_LIBRARY_PATH=/usr/local/cuda-12.0/lib64${LD_LIBRARY_PATH:
+:${LD_LIBRARY_PATH}}
```

1. **Hello world I:** code24.cu Follow the steps below for executing Cuda code;

   ```
   nvcc -o code24 code24.cu
   ./code24
   ```

```
#include <stdio.h>
#include <unistd.h>
__device__ const char *STR = "HELLO WORLD!";
const int STR_LENGTH = 12;
__global__ void cuda_hello(){
  // blockIdx.x: Block index within the grid  in the x direction
  // threadIdx.x: Thread index within the block
  // blockDim.x,y,z # of threads in a block
  printf("Hello World from GPU! (%d ,%d) : %c ThreadID %d \n",
    blockIdx.x, threadIdx.x, STR[threadIdx.x % STR_LENGTH], (threadIdx
    .x +blockIdx.x*blockDim.x));
}
/*
  ./deviceQuery Starting...

  CUDA Device Query (Runtime API) version (CUDART static linking)

  Detected 3 CUDA Capable device(s)

  Device 2: "NVIDIA GeForce GT 1030"
  CUDA Driver Version / Runtime Version          12.0 / 11.8
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 1998 MBytes
    (2095185920 bytes)
  (003) Multiprocessors, (128) CUDA Cores/MP:    384 CUDA Cores
```

```
23   GPU Max Clock rate:                                1468 MHz (1.47 GHz)
24   Memory Clock rate:                                 3004 Mhz
25   Memory Bus Width:                                  64-bit
26   L2 Cache Size:                                     524288 bytes
27   Maximum Texture Dimension Size (x,y,z)             1D=(131072), 2D
       =(131072, 65536), 3D=(16384, 16384, 16384)
28   Maximum Layered 1D Texture Size, (num) layers      1D=(32768), 2048
       layers
29   Maximum Layered 2D Texture Size, (num) layers      2D=(32768, 32768),
       2048 layers
30   Total amount of constant memory:                   65536 bytes
31   Total amount of shared memory per block:           49152 bytes
32   Total shared memory per multiprocessor:            98304 bytes
33   Total number of registers available per block:     65536
34   Warp size:                                         32
35   Maximum number of threads per multiprocessor:      2048
36   Maximum number of threads per block:               1024
37   Max dimension size of a thread block (x,y,z):      (1024, 1024, 64)
38   Max dimension size of a grid size    (x,y,z):      (2147483647, 65535,
       65535)
39   Maximum memory pitch:                              2147483647 bytes
40   Texture alignment:                                 512 bytes
41   Concurrent copy and kernel execution:              Yes with 2 copy
       engine(s)
42   Run time limit on kernels:                         Yes
43   Integrated GPU sharing Host Memory:                No
44   Support host page-locked memory mapping:           Yes
45   Alignment requirement for Surfaces:                Yes
46   Device has ECC support:                            Disabled
47   Device supports Unified Addressing (UVA):          Yes
48   Device supports Managed Memory:                    Yes
49   Device supports Compute Preemption:                Yes
50   Supports Cooperative Kernel Launch:                Yes
51   Supports MultiDevice Co-op Kernel Launch:          Yes
52   Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0
53   Compute Mode:
54   < Default (multiple host threads can use ::cudaSetDevice() with
       device simultaneously) >
55   deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.0, CUDA
       Runtime Version = 11.8, NumDevs = 3
56   Result = PASS
57 */
58 int main() {
59   printf("Hello World from CPU!\n");
60   sleep(2);
61   int threads_per_block=12;
62   int blocks_per_grid=2;
63   cuda_hello <<< blocks_per_grid, threads_per_block >>> ();
64   cudaDeviceSynchronize(); /* Halt host thread execution on CPU until
       the device has finished processing all previously requested tasks
       */
65   return 0;
66 }
```

2. **Hello world II:** code25.cu Follow the steps below for executing Cuda code;

```
nvcc -o code25 code25.cu
./code25
```

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N 720 // number of computations
#define GRID_D1 20 // constants for grid and block sizes
#define GRID_D2 3  // constants for grid and block sizes
#define BLOCK_D1 12 // constants for grid and block sizes
#define BLOCK_D2 1 // constants for grid and block sizes
#define BLOCK_D3 1 // constants for grid and block sizes

__global__ void hello(void) // this is the kernel function called for each thread
{
  // we use the CUDA variables {threadIdx, blockIdx, blockDim, gridDim} to determine a unique ID for each thread
  int myblock = blockIdx.x + blockIdx.y * gridDim.x; // id of the block
  int blocksize = blockDim.x * blockDim.y * blockDim.z;      // size of each block (within grid of blocks)
  int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x + threadIdx.x; // id of thread in a given block
  int idx = myblock * blocksize + subthread; // assign overall id/index of the thread
  int nthreads=blocksize*gridDim.x*gridDim.y; // Total # of threads
  int chunk=20; // Vary this value to see the changes at the output
  if(idx < chunk || idx > nthreads-chunk) { // print buffer from within the kernel is limited so only print for first and last chunks of threads
    if (idx < N){
      printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-thread index within block=(%d,%d,%d) => thread index=%d \n", blockIdx.x, blockIdx.y, gridDim.x, gridDim.y, threadIdx.x, threadIdx.y, threadIdx.z, idx);
    }
    else
      {
  printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-thread index within block=(%d,%d,%d) => thread index=%d [### this thread would not be used for N=%d ###]\n", blockIdx.x, blockIdx.y, gridDim.x, gridDim.y, threadIdx.x, threadIdx.y, threadIdx.z, idx, N);
      }
  }
}

int main(int argc,char **argv)
{
  // objects containing the block and grid info
  const dim3 blockSize(BLOCK_D1, BLOCK_D2, BLOCK_D3);
  const dim3 gridSize(GRID_D1, GRID_D2, 1);
  int nthreads = BLOCK_D1*BLOCK_D2*BLOCK_D3*GRID_D1*GRID_D2; // Total # of threads
```

```
38   if (nthreads < N){
39     printf("\n========== NOT ENOUGH THREADS TO COVER N=%d
       ==========\n\n",N);
40   }
41   else
42     {
43       printf("Launching %d threads (N=%d)\n",nthreads,N);
44     }
45   hello<<<gridSize, blockSize>>>(); // launch the kernel on the
       specified grid of thread blocks
46   // Need to flush prints, otherwise none of the prints from within
       the kernel will show up
47   // as program exit does not flush the print buffer.
48   cudaError_t cudaerr = cudaDeviceSynchronize();
49   if (cudaerr){
50     printf("kernel launch failed with error \"%s\".\n",
51      cudaGetErrorString(cudaerr));
52   }
53   else
54     {
55       printf("kernel launch success!\n");
56     }
57   printf("That's all!\n");
58   return 0;
59 }
```

3. **Vector Addition:** code26.cu Follow the steps below for executing Cuda code;

```
nvcc -o code26 code26.cu
./code26
```

```
// https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/
// https://github.com/olcf-tutorials/vector_addition_cuda
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define n 100000 // Size of array

/* CUDA KERNEL Compute the sum of two vectors
 * Each thread takes care of one element of C
 * C[i] = A[i] + B[i]
 */
__global__ void add_vectors(double *a, double *b, double *c)
{
  int id = blockIdx.x*blockDim.x+threadIdx.x; // Get our global thread
     ID
  if (id < n) // Make sure we do not go out of bounds
    c[id] = a[id] + b[id]; /* Compute the element of C */
}

int main( int argc, char* argv[] )
{
  double *h_a; // Host input vectors
  double *h_b;
  double *h_c;

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

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

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

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

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

  // Copy data from host arrays h_a and h_b to device arrays d_a and
    d_b
  cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
  cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

  int thr_per_blk = 1024; // blockSize. Number of threads in each
    thread block
```

```
51   int blk_in_grid = ceil( (float) n/thr_per_blk ); // gridSize. Number
       of thread blocks in grid
52
53   add_vectors<<<blk_in_grid, thr_per_blk>>>(d_a, d_b, d_c); // Execute
       the kernel
54
55   cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost ); // Copy data
       from device array d_c to host array h_c
56
57   // Sum up vector c and print result divided by n, this should equal
       1 within error
58   double sum = 0;
59   for(i=0; i<n; i++)
60     sum += h_c[i];
61   printf("final result: %f\n", sum/n);
62
63   cudaFree(d_a); // Free device memory
64   cudaFree(d_b);
65   cudaFree(d_c);
66
67   free(h_a); // Free host memory
68   free(h_b);
69   free(h_c);
70
71   printf("--------------------------------\n");
72   printf("__ENDED__\n");
73   printf("--------------------------------\n");
74   printf("N                  = %d\n", n);
75   printf("Threads Per Block  = %d\n", thr_per_blk);
76   printf("Blocks In Grid     = %d\n", blk_in_grid);
77   printf("--------------------------------\n\n");
78
79   return 0;
80 }
```