BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)

CS G532 – High Perf. Het. Computing

Lab#5

Note: Please use programs supplied with this sheet. Do not copy from this sheet.

The lab has the following objectives:
Giving practice programs for CUDA.

Hello World Program

Consider the following program hello.cu. The kernel hello() is just like any other C function. The only difference is the decoration with the __global__ directive, which specifies that the hello function is supposed to be called from the host and run on the device. In devices of compute capability 3.5 and above, a __global__ function can be also called from the device. Kernels that are called from the host (i.e., __global__) are not supposed to return any value. They must be declared void. Any results that a kernel computes are stored in the device memory and must be explicitly transferred back to host memory. CUDA supports two more function specifiers: __device__ and __host__. A __device__ function can only be called from within a kernel, i.e., not from a host. A __host__ function can only run on the host. The __host__ is typically omitted unless used in combination with __device__ to indicate that the function can run on both the host and the device.

```
1.  #include <stdio.h>
2.  #include <cuda.h>
3.
4.  __global__ void hello()
5.  {
6.      printf("Hello world\n");
7.  }
8.
9.  int main()
10. {
11.    hello<<<1,10>>>();
12.    cudaThreadSynchronize();
13.    return 0;
14. }
```

Q?

1. Compile the above program using the following command. How many times should it print “Hello World”?

```
nvcc hello.cu -o hello
./hello
```

2. Change the code so that it will print 10000 times.
3. Can you make changes so that the string “Hello World by <thread id>” is printed?

Query device features

This is a program “deviceQuery.cu” to query the device features in your GPU.

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

int main ()
{
  int deviceCount = 0;
  cudaGetDeviceCount (&deviceCount);
  if (deviceCount == 0)
    printf ("No CUDA compatible GPU.\n");
  else
    {
      cudaDeviceProp pr;
      for (int i = 0; i < deviceCount; i++)
        {
          cudaGetDeviceProperties (&pr, i);
          printf ("Dev #%i is %s\n", i, pr.name);
        }
    }
  return 1;
}
```

Q?

1. Compile the above program and run it. Check how many GPU devices are present in this server.
2. Modify the above program with the following lines added. They will list more features of a device. Specifically find out how many SMs are there and warp size and maximum no of threads per block. Now modify “helloWorld.cu” program to generate maximum no of threads that will generate enough work for all cores with this information.

```
        printf(" --- General Information for device %d ---\n", i);
        printf("Name: %s\n", prop.name);
        printf("Compute capability: %d.%d\n", prop.major, prop.minor);
        printf("Clock rate: %d\n", prop.clockRate);
        printf("Device copy overlap: ");
```

```
6·          if (prop·deviceOverlap)
7·              printf("Enabled\n");
8·          else
9·              printf("Disabled\n");
10·         printf("Kernel execition timeout : ");
11·         if (prop·kernelExecTimeoutEnabled)
12·             printf("Enabled\n");
13·         else
14·             printf("Disabled\n");
15·         printf(" --- Memory Information for device %d ---\n", i);
16·         printf("Total global mem: %ld\n", prop·totalGlobalMem);
17·         printf("Total constant Mem: %ld\n", prop·totalConstMem);
18·         printf("Max mem pitch: %ld\n", prop·memPitch);
19·         printf("Texture Alignment: %ld\n", prop·textureAlignment);
20·         printf(" --- MP Information for device %d ---\n", i);
21·         printf("Multiprocessor count: %d\n",
22·                 prop·multiProcessorCount);
23·         printf("Shared mem per mp: %ld\n", prop·sharedMemPerBlock);
24·         printf("Registers per mp: %d\n", prop·regsPerBlock);
25·         printf("Threads in warp: %d\n", prop·warpSize);
26·         printf("Max threads per block: %d\n",
27·                 prop·maxThreadsPerBlock);
28·         printf("Max thread dimensions: (%d, %d, %d)\n",
29·                 prop·maxThreadsDim[0], prop·maxThreadsDim[1],
30·                 prop·maxThreadsDim[2]);
31·         printf("Max grid dimensions: (%d, %d, %d)\n",
32·                 prop·maxGridSize[0], prop·maxGridSize[1],
33·                 prop·maxGridSize[2]);
34·         printf("\n");
```

CUDA's Execution Model and Memory

GPU cores are essentially vector processing units, capable of applying the same instruction on a large collection of operands. So, when a kernel is run on a GPU core, the same instruction sequence is synchronously executed by a large collection of processing units called streaming processors, or SPs. A group of SPs that execute under the control of a single control unit is called a streaming multiprocessor, or SM. A GPU can contain multiple SMs, each running each own kernel. Nvidia calls this execution model Single-Instruction, Multiple Threads (SIMT). SIMT is analogous to SIMD. The only major difference is that in SIMT the size of the "vector" on which the processing elements operate is determined by the software, i.e., the block size. Threads are scheduled to run on an SM as a block. The threads in a block do not run concurrently, though. Instead they are executed in groups called warps. The size of a warp is hardware-specific. The current CUDA GPUs use a warp size of 32. At any time instance and based on the number of CUDA cores in an SM, we can have 32 threads active (one full active warp), 16 threads active (a half-warp active), or 8 threads active (a quarter-warp active). The benefit of interleaving the execution of warps (or their parts) is to hide

the latency associated with memory access, which can be significantly high. An SM can switch seamlessly between warps (or half- or quarter-warps) as each thread gets its own set of registers. Each thread actually gets its own private execution context that is maintained on-chip. This contradicts the arrangement used by multithreading on CPUs.

Consider the following program vectorAdd.cu. The host is responsible for generating two random integer arrays that are passed to the device. Upon completion of the vector addition on the device, the result data are transferred back to the host.

The program manages six pointers, three pointing to host memory addresses and three pointing to device memory addresses. By convention and in order to easily identify them, host pointers are prefixed by h and device pointers by d. Data are copied from the host to the device prior to calling the kernel (lines 49, 50), and from the device to the host after the kernel completes (line 60). The device memory that holds our data is called global memory. The reservation of the required space in global memory is done in lines 37-39. Each CUDA thread calculates just a single element of the result matrix (line 23). This is not an advisable design in general, but it clearly illustrates that CUDA threads can be extremely lightweight. Each thread calculates its global position/ID in the grid (line 21) and uses it to index the parts of the input that it will operate upon. Each core proceeds with the calculation, only if its ID permits (line 22). This check is required when the workload is not evenly distributed between the thread blocks.

```
1· / / File : vectorAdd · cu
2· #include <stdio·h>
3· #include <stdlib·h>
4· #include <cuda·h>
5·
6· static const int BLOCK_SIZE = 256;
7· static const int N = 2000;
8·
9· #define CUDA_CHECK_RETURN(value) {           \
10·     cudaError_t _m_cudaStat = value;         \
11·     if (_m_cudaStat != cudaSuccess) {        \
12·         fprintf(stderr, "Error %s at line %d in file %s\n",          \
13·                 cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);    \
14·         exit(1);                                                      \
15·       } }
16·
17· __global__ void vadd (int *a, int *b, int *c, int N)
18· {
19·   int myID = blockIdx·x * blockDim·x + threadIdx·x;
20·   if (myID < N)
21·     c[myID] = a[myID] + b[myID];
22·}
23·
24·int main (void)
25·{
```

```
26.  int *ha, *hb, *hc, *da, *db, *dc;      // host (h*) and device (d*) pointers
27.  int i;
28.
29.  ha = new int[N];
30.  hb = new int[N];
31.  hc = new int[N];
32.
33.  CUDA_CHECK_RETURN (cudaMalloc ((void **) &da, sizeof (int) * N));
34.  CUDA_CHECK_RETURN (cudaMalloc ((void **) &db, sizeof (int) * N));
35.  CUDA_CHECK_RETURN (cudaMalloc ((void **) &dc, sizeof (int) * N));
36.
37.  for (i = 0; i < N; i++)
38.    {
39.      ha[i] = rand () % 10000;
40.      hb[i] = rand () % 10000;
41.    }
42.
43.  CUDA_CHECK_RETURN (cudaMemcpy (da, ha, sizeof (int) * N,
   cudaMemcpyHostToDevice));
44.  CUDA_CHECK_RETURN (cudaMemcpy (db, hb, sizeof (int) * N,
   cudaMemcpyHostToDevice));
45.
46.  int grid = ceil (N * 1.0 / BLOCK_SIZE);
47.  vadd <<< grid, BLOCK_SIZE >>> (da, db, dc, N);
48.
49.  CUDA_CHECK_RETURN (cudaThreadSynchronize ());
50.  // Wait for the GPU launched work to complete
51.  CUDA_CHECK_RETURN (cudaGetLastError ());
52.  CUDA_CHECK_RETURN (cudaMemcpy (hc, dc, sizeof (int) * N,
   cudaMemcpyDeviceToHost));
53.
54.  for (i = 0; i < N; i++)
55.    {
56.      if (hc[i] != ha[i] + hb[i])
57.        printf ("Error at index %i : %i VS %i\n", i, hc[i], ha[i] + hb[i]);
58.    }
59.
60.  CUDA_CHECK_RETURN (cudaFree ((void *) da));
61.  CUDA_CHECK_RETURN (cudaFree ((void *) db));
62.  CUDA_CHECK_RETURN (cudaFree ((void *) dc));
63.  delete[]ha;
64.  delete[]hb;
65.  delete[]hc;
66.  CUDA_CHECK_RETURN (cudaDeviceReset ());
67.
```

```
68.  return 0;
69.}
```

Q?

1. The nvcc compiler driver can be instructed to report the outcome of memory allocation process with the -Xptxas –v or --ptxas-option=-v switches.

```
1.  nvcc -Xptxas -v vectorAdd.cu
```

A device's compute capability determines the maximum number of registers that can be used per thread. If this number is exceeded, local variables are allocated in the run-time stack, which resides in off-chip memory and it is thus slow to work with. This off-chip memory is frequently called local memory, but it is actually the global memory that it is used for this purpose. The "local" specifier just conveys the fact that whatever resides there is only accessible to a particular thread. Local memory locations can be cached by the L1 cache, so performance may not suffer much, but the outcome is application-specific. The Nvidia compiler will automatically decide which variables will be allocated to registers and which will spill over to local memory.

Timetaken for GPU code can be measured using

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
vadd <<< grid, BLOCK_SIZE >>> (da, db, dc, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Timetaken %f\n", milliseconds);
```

2. Change the grid and block sizes as per the SMs and warpSize, registers, shared memory and check the time taken.
3. Increase N to big value and check.

End of lab5