IT301 - Parallel Computing

Assignment 9

Name: Niraj Nandish

Roll No: 191IT234

Seting up Google Collab for CUDA

```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```

```
!nvcc --version
```

```
    nvcc: NVIDIA (R) Cuda compiler driver
    Copyright (c) 2005-2018 NVIDIA Corporation
    Built on Wed_Apr_11_23:16:29_CDT_2018
    Cuda compilation tools, release 9.2, V9.2.88
```

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

```
%load_ext nvcc_plugin
```

Program 1: Details of device

```
%%cu
#include<stdio.h>
int main()
{
int devcount;
cudaGetDeviceCount(&devcount);
printf("Device count:%d\n",devcount);
for (int i = 0; i < devcount; ++i)
{
// Get device properties
printf("\nCUDA Device #%d\n", i);
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, i);
printf("Name:%s\n", devProp.name);
printf("Compute capability: %d.%d\n",devProp.major ,devProp.minor);
printf("Warp Size %d\n",devProp.warpSize);
printf("Total global memory:%u bytes\n",devProp.totalGlobalMem);
printf("Total shared memory per block: %u bytes\n", devProp.sharedMemPerBlock);
printf("Total registers per block : %d\n",devProp.regsPerBlock);
printf("Clock rate: %d khz\n",devProp.clockRate);
printf("Maximum threads per block:%d\n", devProp.maxThreadsPerBlock);
for (int i = 0; i < 3; ++i)
printf("Maximum dimension %d of block: %d\n", i, devProp.maxThreadsDim[i]);
for (int i = 0; i <= 2; ++i)
printf("Maximum dimension %d of grid: %d\n", i, devProp.maxGridSize[i]);
printf("Number of multiprocessors:%d\n", devProp.multiProcessorCount);
}
return 0;
}
```

```
Device count:1

CUDA Device #0
Name:Tesla K80
Compute capability: 3.7
Warp Size 32
Total global memory:3407020032 bytes
Total shared memory per block: 49152 bytes
Total registers per block : 65536
Clock rate: 823500 khz
Maximum threads per block:1024
Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64
Maximum dimension 0 of grid: 2147483647
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535
Number of multiprocessors:13
```

Program 2: Hello World

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

__global__ void helloworld(void)
{
printf("Hello World from GPU\n");
}
```

```
int main() {
helloworld<<<1,10>>>();
printf("Hello World\n");
return 0;
}
```

```
    Hello World
```

Observation

The `helloworld` function has `__global__` keyword so the function will be run from the device after it's called from the host. We can see that the output of the program is just `Hello World` while we don't see the output of the `helloworld` function as the `cudaDirectiveSynchronize()` directive is missing. As a result, the CPU doesn't wait for the GPU to finish its execution and so the program is terminated before the GPU finishes executing their threads.

Program 3: Program to perform c[i] = a[i] + b[i]

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
// Get global thread
int id = blockIdx.x*blockDim.x+threadIdx.x;
// 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 = 1000;
//time varibales
struct timeval t1, t2;
// Host input vectors
double *h_a, *h_b;
//Host output vector
double *h_c;
// Device input vectors
double *d_a, *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] = i+1;
h_b[i] = i+1;
}
// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
gettimeofday(&t1, 0);
// Execute the kernel
vecAdd<<<1,1000>>>(d_a, d_b, d_c, n);
cudaDeviceSynchronize();
gettimeofday(&t2, 0);
// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
for(i=0; i<n; i=i+100)
printf("c[%d]=%f\n",i,h_c[i]);
double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000.0;
printf("Time to generate: %3.10f ms \n", time);
// 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;
}
```

```
c[0]=2.000000
c[100]=202.000000
c[200]=402.000000
c[300]=602.000000
c[400]=802.000000
c[500]=1002.000000
c[600]=1202.000000
c[700]=1402.000000
c[800]=1602.000000
c[900]=1802.000000
Time to generate: 0.1690000000 ms
```

Execution time for given inputs

No. of Blocks	No. of Threads	Execution time(ms)

No. of Blocks	No. of Threads	Execution time(ms)
1	100	0.124
1	50	0.177
2	50	0.121

Observation

In the first execution, we used 1 block with 100 threads, so we see all the values of `c[i]`. Then we executed the program with 1 block and 50 threads, so we see only the output of first 50 values in `c[i]`. Finally we executed with 2 blocks with 50 threads per block, so here the first block calculates the first 50 values while the second block calculates the remaining 50 values.

Output

a) vecAdd<<<1,100>>>(d_a, d_b, d_c, n)

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=102.000000
c[60]=122.000000
c[70]=142.000000
c[80]=162.000000
c[90]=182.000000
Time to generate: 0.1240000000 ms
```

b) vecAdd<<<1,50>>>(d_a, d_b, d_c, n)

```
c[0]=2.000000
c[10]=22.000000
c[20]=42.000000
c[30]=62.000000
c[40]=82.000000
c[50]=0.000000
c[60]=0.000000
c[70]=0.000000
```

```
c[80]=0.000000

c[90]=0.000000

Time to generate: 0.1770000000 ms
```

c) vecAdd<<<2,50>>>(d_a, d_b, d_c, n)

```
c[0]=2.000000

c[10]=22.000000

c[20]=42.000000

c[30]=62.000000

c[40]=82.000000

c[50]=102.000000

c[60]=122.000000

c[70]=142.000000

c[80]=162.000000

c[90]=182.000000

Time to generate: 0.1210000000 ms
```