ME759
High Performance Computing for Engineering Applications
University of Wisconsin-Madison
Fall 2013
Assignment 3

Due: September 30, 2013 @ 11:59 PM

1. Write a CUDA application that does the following:

- It allocates on the device memory to hold an array of 16 integers.
- It launches a kernel on two blocks, each block having eight threads. The kernel is set up such that each thread computes the sum of its thread index and block index, and then saves the result (an integer) into the array for which memory was allocated on the device. Subsequently, the kernel uses a printf statement to print out what it saved into the device array.
- The host copies back the data stored in the device array into a host array of size 16.
- The host uses a loop to print to screen the 16 values stored in the host array.

NOTE: pass as an argument to the kernel the pointer to device memory where you are to save the information generated by each thread.

Solution:
The program code 'p1.cu' is run on euler and it is running successfully. Please find the file in the .zip folder as p1.cu. The code is as follows:

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

__global__
void simpleKernel(int* data)
{

    data[(blockIdx.x* blockDim.x)+threadIdx.x] = blockIdx.x + threadIdx.x;

//to print the result in the device array

     printf("\n %d   + \t %d \t %d",threadIdx.x, blockIdx.x, data[(blockIdx.x
* blockDim.x)+ threadIdx.x]);

}

int main()
{
      int hostarray[16], *devarray,i;
//allocate memory on the device (GPU)
      cudaMalloc((void**) &devarray,sizeof(int)*16);

//invoke GPU kernel, with two blocks each having eight threads
      simpleKernel<<<2,8>>>(devarray);
//bring the result back from the GPU into the host array
```

```
cudaMemcpy(&hostarray, devarray, sizeof(int)*16, cudaMemcpyDeviceToHost);


//printing hte result
printf("\n Values stored in hostarray: ");

for(i=0;i<16;i++)
{
printf("\t %d",hostarray[i]);
}

//release the memory allocated on the GPU
cudaFree(devarray);

return 0;
}
```

2. This exercise will provide an opportunity to understand how a problem can be solved using different execution configurations; i.e., number of blocks and threads in an execution grid. To this end, assume that you start with two arrays of random numbers (double precision) between -10 and 10. You are supposed to add these two arrays of length N on the GPU and eventually get the result into a host array of length N . You will have to run a loop to investigate what happens when the value of N increases by factors of 2 from N = 2^10 to N = 2^20 . Specifically, report in a plot the amount of time it took for the program to add the two arrays as a function of the array size. Plot the inclusive time using a blue line. Plot the exclusive time using a red line. Here's how this would work:

• Allocate space on the host for arrays hA, hB, and hC, refC, and then populate hA and hB with random numbers between -10 and 10. Each of these arrays is of size N .
• Store the result of hA+hB into refC
• Allocate space on the device for dA, dB, and dC
• [For inclusive timing, start the timing now]
• Copy content of hA and hB into dA and dB, respectively.
• [For exclusive timing, only start the timing now]
• Invoke kernel that sums up the two arrays
• [For exclusive timing, stop the timing now]
• Copy the content of dC back into hC
• [For inclusive timing, stop the timing now]
• Report the amount of time required to complete the job
• Confirm that the numbers in refC and hC are identical within 1E-12

Here's an important point. You'll have to do the sequence of steps above twice (plot all the result on the same plot though). The first time, the number of threads in a block is going to be 32. As the value of N increases, you'll have to adjust the number of blocks you launch to get the job done. The second time around, you are going to use 1024 threads in a block (this is currently the largest number of threads that you can have in a block). Again, as the value of N increases, you'll have to adjust the number of blocks you launch to get the job done.

NOTE: Use a significantly thicker line to plot the 32-thread case.

Solution:

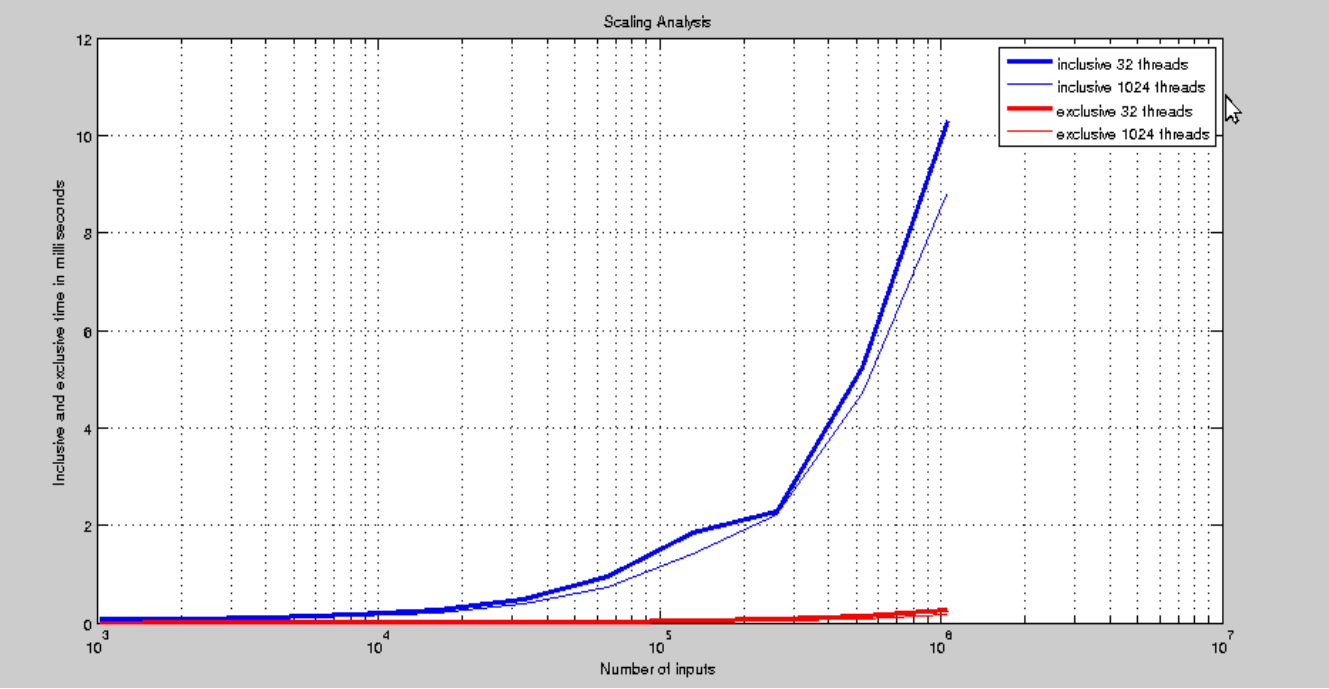
The program code ‘p2t.cu’ is attached in the .zip folder. It is compiled and run successfully on euler. The result for both 32 threads in a block and 1024 threads in a block has been printed in the same program. The code is as follows:

```
#include<stdio.h>
#include<cuda.h>
#include<math.h>
#include<sys/time.h>

__global__
void Matadd(double* A, double* B, double* C, int N)
 {

    int i = blockIdx.x* blockDim.x +threadIdx.x;
    if(i<N)
      C[i] = A[i] + B[i];
      __syncthreads();
 }

int main()
{

for(int j=10;j<=20;j++)
{

      cudaEvent_t start1,start2,start3,stop1,stop2,stop3,start4,stop4;
      float time1,time2,time3, time4;
      int i;
      int N = pow(2,j);
      size_t size = N * sizeof(double);
printf ("\n The value of N is %d",N);

cudaEventCreate(&start1);
cudaEventCreate(&stop1);

cudaEventCreate(&start2);
cudaEventCreate(&stop2);

cudaEventCreate(&start3);
cudaEventCreate(&stop3);

cudaEventCreate(&start4);
cudaEventCreate(&stop4);


//allocate input matrices hA, hB, hC,refC in host memory
double* hA = (double*)malloc(size);
double* hB = (double*)malloc(size);
double* hC = (double*)malloc(size);
double* refC = (double*)malloc(size);
```

```
for(i=0;i<N;i++)
{
hA[i] = rand()%20-10;
hB[i] = rand()%20-10;

refC[i] = hA[i] + hB[i];
 }
//allocate memory on the device (GPU)
double* dA;
cudaMalloc(&dA,size);
double* dB;
cudaMalloc(&dB,size);
double* dC;
cudaMalloc(&dC,size);

//timing start for inclusive timing
cudaEventRecord(start1, 0);

//copy vectors from host memory to devie memory
      cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);

      cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

//invoke GPU kernel, with two blocks each having eight threads


int threadsperblock = 32;
int blockspergrid = (N + threadsperblock - 1)/ threadsperblock;

//timing start for exclusive timing
cudaEventRecord(start2, 0);
Matadd<<<blockspergrid,threadsperblock>>>(dA,dB,dC,N);
//timing stop for exclusive timing
cudaEventRecord(stop2, 0);
cudaEventSynchronize(stop2);

cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);

//timing stop for inclusive timing
cudaEventRecord(stop1, 0);
cudaEventSynchronize(stop1);


//timing start for inclusive timing

cudaEventRecord(start3, 0);


//copy vectors from host memory to devie memory
      cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);

      cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);
```

```
//invoke GPU kernel, with two blocks each having eight threads
threadsperblock = 1024;
blockspergrid = (N + threadsperblock - 1)/ threadsperblock;

//timing start for exclusive timing
cudaEventRecord(start4, 0);
Matadd<<<blockspergrid,threadsperblock>>>(dA,dB,dC,N);

//timing stop for exclusive timing
cudaEventRecord(stop4, 0);
cudaEventSynchronize(stop4);

//bring the result back from the device memory into the host array
cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop3, 0);
cudaEventSynchronize(stop3);

for (i=0;i<N;i++)
{
if(fabs(refC[i] - hC[i]) > 1e-12)
{
printf("Erratic Value \n");
exit(1);
}
}


cudaEventElapsedTime(&time1,start1,stop1);

cudaEventElapsedTime(&time2,start2,stop2);

printf("\n The inclusive time and exclusive time for 32 threads in
microseconds for 2 to power %d is %f and %f respectively \n",j,time1,time2);


cudaEventElapsedTime(&time3,start3,stop3);

cudaEventElapsedTime(&time4,start4,stop4);

printf("\n The inclusive time and exclusive time for 1024 threads in
microseconds for 2 to power %d is %f and %f respectively \n",j,time3,time4);

free(hA);
free(hB);
free(hC);
free(refC);

cudaFree(dA);
cudaFree(dB);
cudaFree(dC);

}
return 0;
}
```

plot in log-scale is as follows:



The plot on linear-scale is as follows: