Process Time Comparison between GPU and CPU

Abhranil Das

July 20, 2011

Hamburger Sternwarte, Universität Hamburg

Abstract

This report discusses a CUDA program to compare process times on a GPU and a CPU for a range of task size. The hardware compared are the NVIDIA® Tesla™ C870 GPU and the quad-core Intel® Xeon® E5410 CPU. The operation is to compute the square root of an array. Due to GPU restrictions, the array size is fixed and the operation repeated to increase the task size. In the limiting case of long process times, the GPU emerges ~400 times faster.

1 Introduction

1.1 GPU and CPU architecture

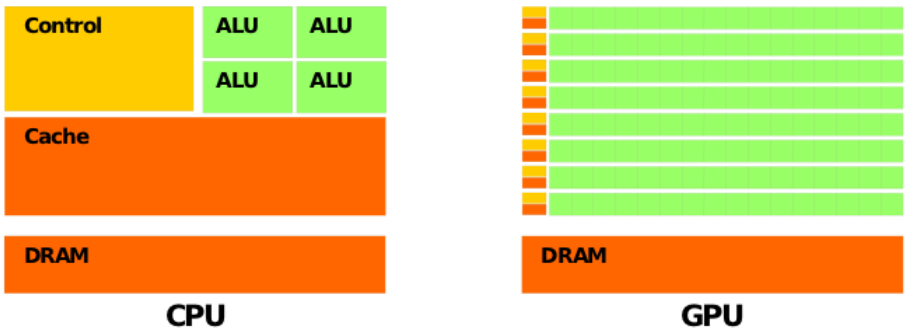


Figure 1: Differences in GPU and CPU hardware architecture

The primary difference between a GPU and CPU is that the hardware architecture of a GPU contains many more ALU's (Arithmetic Logic Units) than a typical CPU, and fewer components for the cache and flow control. This implies high arithmetic intensity of operation (ratio of arithmetical operations to memory operations), and a far better capability to process parallel arithmetic operations, in which the same operations in performed on many different data elements. This parallel arithmetic computation is just what is required by graphical applications, in which clusters of pixels are allocated simultaneous threads and rendered in parallel. However, any other application whose function can be parallelized is suitable to be modified and run on a GPU for faster results. An elementary example would be matrix multiplication.

1.2 NVIDIA CUDA

NVIDIA's Compute Unified Device Architecture (CUDA)[1] allows programmers to interact directly with the GPU and run programs on them, thus effectively utilizing the advantages of parallelization. CUDA is not a new programming language, rather an extension to C with GPU-specific commands, options and operations.[2]

Programs written in CUDA are compiled by NVIDIA's nvcc compiler and can be run only on NVIDIA's GPU's. The task of modifying the problem to be run as parallel operations, however, lies with the programmer. CUDA ensures only that after this modification the parallel data operations can be run on simultaneous GPU threads. A CUDA program may be run on any number of processor cores, and only the number of processors needs to be known to the runtime system.

A typical CUDA program consists of two parts: the main part of the program which executes serially on the CPU (the *host*), and the **kernel**, called by the main program, which is executed in parallel on the GPU (the *device*). Host functions like *printf* cannot be called from the kernel.

2 Hardware Specifications

The hardware used were the following:

2.1 GPU

NVIDIA® Tesla™ C870, based on the Tesla™ G80[3]

- Total amount of global memory: 1610416128 bytes
- Number of multiprocessors: 16
- Number of streaming processor cores: 128
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 16384 bytes
- Total number of registers available per block: 8192
- Warp size: 32
- Maximum number of threads per block: 512
- Maximum sizes of each dimension of a block: 512 x 512 x 64
- Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
- Processor core clock rate: 1.35 GHz
- Memory size: 1536 MB (Twenty-four pieces 16M × 32 GDDR3 136-pin BGA SDRAM)
- Memory clock: 800 MHz
- Voltage: 1.3 V ± 130 mV
- Power: 170.9 W

2.2 CPU

Intel® Xeon® Processor E5410[4]

- Number of Cores: 4
- Number of Threads: 4
- Clock Speed: 2.33 GHz (2327.501 MHz)
- FSB Speed: 1333 Mhz
- L2 Cache: 12 MB

- Instruction Set: 64-bit
- VID Voltage Range: 0.850V-1.3500V
- Maximum Thermal Design Power: 80 W

3 The Program

3.1 The Task

The objective of the program is simple. It generates an array up to a certain length of integers. It then computes the square roots of these, saving them to another array of the same length. This operation it performs on both the CPU and the GPU, and reports the process times taken on the two and their ratio. These three numbers are measured for a range of process size (total number of operations required for the process). An obvious way to regulate process size is to alter the array length. However, this freedom emerges to be restricted for the GPU, as discussed later. Therefore, with the array size fixed, the task is instead repeated, i.e. square roots of the same first array are computed and repeatedly saved to the second array, replacing the identical values before. In terms of utility this operation is an unproductive repetition, but in terms of analysing optimization, which is our goal, this serves equally well as any other, more utilitarian method of lengthening the process. In fact, as discussed later, it is a better technique to measure performance than lengthening the array. The repetition loop is lengthened in steps of 100. For each length of the loop, process times on the GPU, CPU and their ratio are measured 10 times, and the average of each is written to a file. The standard error (RMSD) for the two process times across the 10 samples are also logged.

3.2 The Program Code

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include <math.h>
4
5  // Kernel that executes on the CUDA device
6  __global__ void calculate(float *a_d, float *b_d, int N)
7  {
8    int idx = blockIdx.x * blockDim.x + threadIdx.x;
9    if (idx<N) b_d[idx] = sqrt(a_d[idx]);
10 }
11
12 // main routine that executes on the host
13 int main(void)
14 {
15 printf("Sample\tGPUt\tCPUt\tRatio\tGPU_error\tCPU_error\n");
16 remove("myspeedComparedata");
17 FILE *file;
18 for (int repeat=1; repeat<=50000; repeat+=100)
19 {
20 printf("Repeat_%d\n",repeat);
21 int samplesize=10;
22 float GPUsum=0, CPUsum=0, GPUsumsq=0, CPUsumsq=0;
23 for (int sample=0;sample<samplesize;sample++)
24   {
25   clock_t start, mid, stop;
26   double GPUt = 0.0; double CPUt = 0.0;
27   start=clock();
28
29   //GPU part
30
31   float *x_h,*y_h, *a_d, *b_d;                                    // Pointer to host
         & device arrays
32   const int N = 262144;                                          // Number of
         elements in arrays
33   size_t size = N * sizeof(float);
34   x_h = (float *)malloc(size); y_h = (float *)malloc(size);      // Allocate array
         on host
35   cudaMalloc((void **) &a_d, size); cudaMalloc((void **) &b_d, size);   // Allocate array
         on device
```

```
36    for (int i=0; i<N; i++) x_h[i] = (float)i;                                    // Initialize host
          array
37    cudaMemcpy(a_d, x_h, size, cudaMemcpyHostToDevice);                           // and copy it to
          CUDA device
38    // Do calculation on device:
39    int block_size = 4;
40    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
41    for (int i=0; i<=repeat; i++) calculate <<< n_blocks, block_size >>> (a_d, b_d, N);
42    cudaMemcpy(y_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);                // Retrieve result
          from device and store it in host array
43
44    mid=clock();
45    GPUt=(double)(mid-start)/CLOCKS_PER_SEC;
46    printf("%d\t%.2f\t",sample+1,GPUt);
47    GPUsum+=GPUt;GPUsumsq+=GPUt*GPUt;
48
49    //CPU part
50
51    float *a_h, *b_h;
52    a_h = (float *)malloc(size); b_h = (float *)malloc(size);
53    for (int i=0; i<N; i++) a_h[i] = (float)i;
54    for (int j=0; j<=repeat; j++)
55    {
56    for (int i=0; i<N; i++) b_h[i] = sqrt(a_h[i]);
57    }
58
59    stop = clock();
60    CPUt = (double) (stop-mid)/CLOCKS_PER_SEC;
61    printf("%.2f\t%.0f\n",CPUt,CPUt/GPUt);
62    CPUsum+=CPUt;CPUsumsq+=CPUt*CPUt;
63
64    //Cleanup
65
66    //free(x_h); free(y_h);free(a_h); free(b_h);
67    //cudaFree(a_d); cudaFree(b_d);
68    }
69 float GPUavg=GPUsum/samplesize, CPUavg=CPUsum/samplesize;
70 printf("Avg.\t%.3f\t%.3f\t%.1f\t%f\t%f\n",GPUavg,CPUavg,CPUavg/GPUavg,sqrt(GPUsumsq/
       samplesize-GPUavg*GPUavg),sqrt(CPUsumsq/samplesize-CPUavg*CPUavg));
71 file = fopen("myspeedComparedata","a+");
72 fprintf(file,"%d\t%.3f\t%.3f\t%.1f\t%f\t%f\n",repeat,GPUavg,CPUavg,CPUavg/GPUavg,sqrt(
       GPUsumsq/samplesize-GPUavg*GPUavg),sqrt(CPUsumsq/samplesize-CPUavg*CPUavg));
73 fclose(file);
74 }
75 }
```

3.3 Code Walkthrough

This is a thorough explanation of the functioning of the important parts of the code by line number.

6-10 This is the kernel that is invoked by the main program. It first allocates a thread ID to the simultaneous processes. Each thread then proceeds to compute the square root of an element of the array **a_d** and place it at the same position in **b_d**.

15 Prints the column headers on screen.

16 When the program is run multiple times, perhaps with different ranges, without this line it would append data to the same file. This line causes the previous data file to be removed at the start of each new run. At the end of each run and before the next, the data file generated should be backed up or renamed.

18 The loop over the range of repetitions of the task. Note carefully that here we have *three* loops. The innermost loop computes the square-root of the array n times. A loop over that causes this to be repeated for a number of samples. A loop over that, this one, repeats samples with n iterated through 1 to say, 50,000. This gives us a comparison of the GPU v/s CPU performance for a range of the size of the task.

21 The sample size is set to 10.

22 GPUsum and GPUsumsq are the variables that will contain the sum of the GPU times and the sum of their squares for the 10 samples. These will give us the average GPU time and the standard error in measurement. Similarly for the CPU.

23 The loop over the 10 samples.

25-27 The process times are recorded using the simple, but not very accurate **clock()** function. The sampling takes some care of its limited accuracy. These lines initiate the variables to be measured and start the clock.

31 The GPU calculation in this case was performed using four arrays. The first, **x_h**, resides on the host and contains data to be worked on. This data is transferred to an array **a_d** on the device. The necessary calculations on this are performed by the GPU and the results stored in array **b_d**, which is then copied back to the host memory in **y_h**.

32 The array size is 262144. Why this number is discussed later.

34 Memory for N floating point numbers each is allocated for the host arrays

35 and for the device arrays. **cudaMalloc** is a CUDA function.

36 The host array is initialized with integers.

37 The host array **x_h** is copied to the device array **a_d**. This uses the CUDA function **cudaMemcpy**.

39-40 Sets the block size and number of blocks.

41 This is the call from the host to the kernel, which shall execute on the device. The kernel is called **repeat** number of times, causing the operation on the device to be repeated as many times. The computed values are stored in **b_d** on device. Note that the host program only continues after the device operation has completed.

42 **cudaMemcpy** copies back the results from **b_d** on device to **y_h** on host memory.

44-46 **clock()** measures the time again, records the difference as the GPU process time and displays it.

47 The process time and its square are added to the respective variables. This ends the GPU part. The next part performs the same operation on a CPU.

51-52 Memory is allocated for the arrays.

53 **a_h** is initiated with integers.

54-57 The square root of the entire array is computed and stored in **b_h** in repeated replacements. This is done **repeat** times.

59-61 **clock()** measures the time again, calculates the difference, records it as the CPU process time, and displays it and the ratio.

62 The CPU process time and its square are added to the respective variables.

66-67 These commented lines, supposed to free the memory for the arrays, raised issues that are discussed later. The issue was resolved when the lines were commented out, and the program ran fine without needing to free this memory.

69 Computes the process time averages from the sums.

70 Prints the average process times and their ratio (ratio of the averages, not average of the ratios) and the standard errors. **S.E.** $= \sqrt{\bar{x^2} - \bar{x}^2}$.

71-73 Writes the results to the data file. Note that this operation works inside the loop, so that the same file is opened, written to and closed after the sampling batch for every loop length. This is more convenient over writing to the file at one go per execution of the program for the following reasons:

- It is convenient from the program structure to write the statistics from the sampling to the file at the end of every iteration, and not have to store it to be written later.
- The more important reason is that one might need to interrupt the program at any stage in the middle of its range and yet not lose the data generated up to that point. Also, if the program meets a problem like a segfault and crashes, the data is still saved and may be backed up.

4 Results

4.1 Output

The program displays information from each sample on screen, but writes only the statistics from each sampling to file, without information from all the samples. The first allows easy debugging, while the second allows easy analysis and plotting of the final data.

The following are the initial output on the screen and file for a range of loop length starting from 1001 and incremented by 100.

4.1.1 Sample Screen Output

Sample	GPUt	CPUt	Ratio	GPU error	CPU error
Repeat 1001					
1	0.17	1.48	9		
2	0.01	1.48	148		
3	0.00	1.48	inf		
4	0.00	1.47	inf		
5	0.02	1.47	74		
6	0.01	1.47	147		
7	0.01	1.47	147		
8	0.01	1.47	147		
9	0.02	1.47	74		
10	0.01	1.47	147		
Avg.	0.026	1.473	56.7	0.048415	0.004554
Repeat 1101					
1	0.01	1.62	162		
2	0.00	1.62	inf		
3	0.02	1.62	81		
4	0.01	1.62	162		
5	0.01	1.62	162		
6	0.00	1.62	inf		

```
7       0.01    1.63    163
8       0.01    1.61    161
9       0.01    1.62    162
10      0.02    1.62    81
Avg.    0.010   1.620   162.0   0.006325        0.004395
Repeat  1201
1       0.00    1.77    inf
2       0.01    1.77    177
3       0.01    1.77    177
4       0.01    1.77    177
5       0.00    1.77    inf
6       0.02    1.77    88
7       0.00    1.77    inf
8       0.01    1.77    177
9       0.01    1.76    176
10      0.02    1.77    88
Avg.    0.009   1.769   196.6   0.007000        0.002847
...
```

Note the (relatively) much longer GPU time at the first sample of the first iteration. This has been observed to be consistent in every execution, and never recurs in any subsequent sample. This is discussed further later in the article.

As you can see, the null time returned from the GPU may sometimes cause the ratio to become infinite. This owes itself to the inaccurate **clock()** function. Part of the purpose of sampling is to compensate for this. The average GPU time was never zero for any sampling in all the runs, even though individual times often were. Please also note that, as already mentioned, the 'average' ratio is the ratio of the average GPU and CPU times, not the average of the ratios, so that the infinite values pose no problem.

4.1.2 Sample File Output

As mentioned before, only the loop length followed by the statistical values from the sampling are written to file:

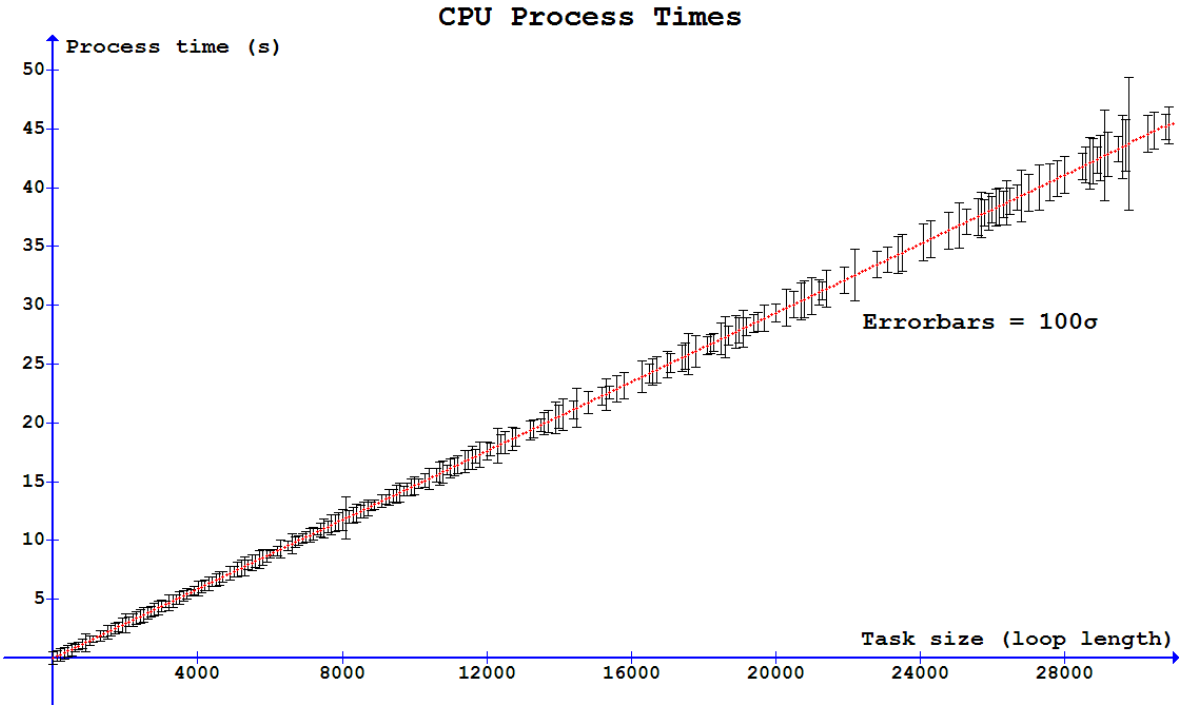
```
1001    0.026   1.473   56.7    0.048415        0.004554
1101    0.010   1.620   162.0   0.006325        0.004395
1201    0.009   1.769   196.6   0.007000        0.002847
...
```

4.2 Plots

After several trials, the optimum parameters for obtaining a descriptive output were determined. Then the program was run with the task size (loop length) incremented from 1 to 30701 in steps of 100. This was done in several different runs with a different range each time and with periodic backups of the data after each run.

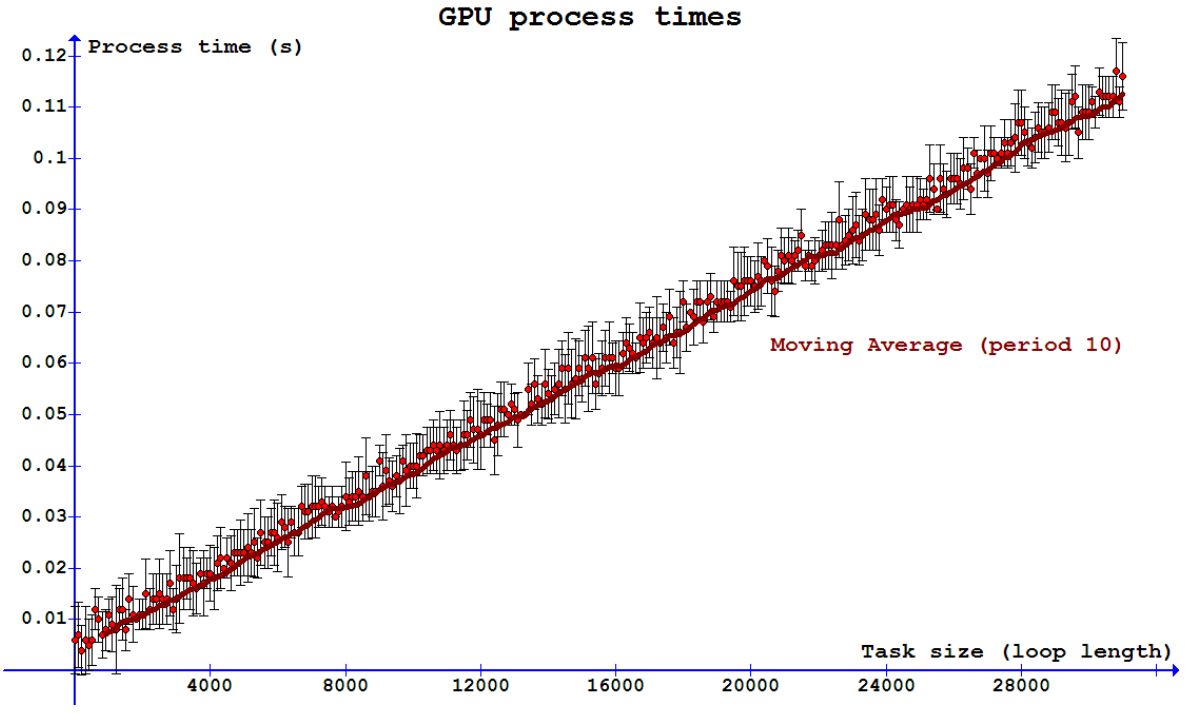
The following are the graphs of the 10-sample average of the CPU and GPU process times with absolute error bars across the range of task size.

4.2.1 CPU Process Times



As one can see, the absolute errors increase with task size. The relative errors, however, do not vary much. In any case, the errors are ignorably small compared to the values themselves (the bars plotted are 100σ).

4.2.2 GPU Process Times

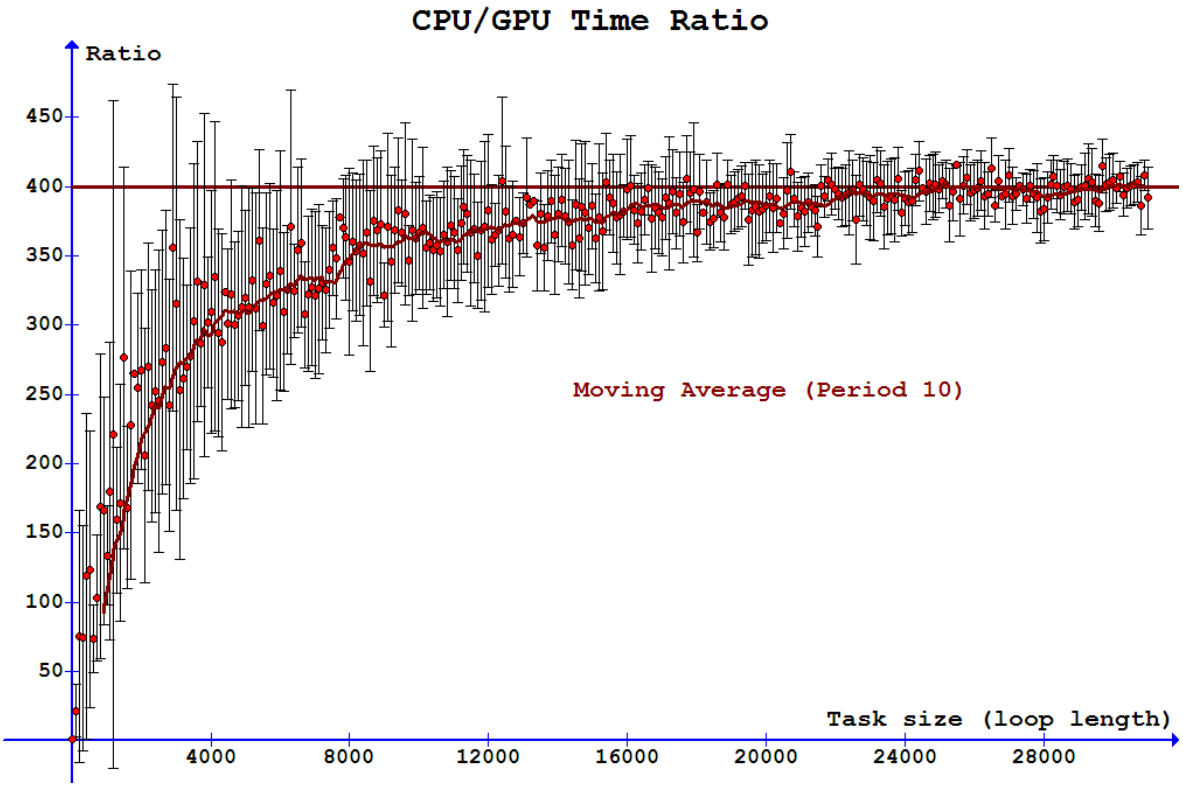


The errorbars here are not scaled. The errors appear larger than for the CPU times as the vertical scale of this plot is smaller (GPU times are much faster). The errors for the GPU and CPU process times are in fact of the same order of 10 ms.

The trendline fitted is a moving average with period 10.

4.2.3 CPU/GPU Ratio

The following is the plot of the ratio of CPU and GPU process times against task size:



The error bars here represent absolute error. However, they are not the statistical standard error of the ratio over the samples. As mentioned before, there was no statistics performed on the ratios. The ratio recorded to file was that of the average CPU and GPU process times. The errors represented here are computed from the errors of these two values, following:

$$\frac{d(\frac{a}{b})}{\frac{a}{b}} = \frac{d(a)}{a} - \frac{d(b)}{b}$$

The absolute errors are computed back by multiplying the modulus of these relative errors with the ratios.

5 Inferences

- For smaller tasks, the GPU is not much faster than the CPU as the data transfer overhead between host and device costs more than time saved by parallelization. However, that is a fixed cost, and well made up for by the parallelization in the limit of large tasks.
- The ratio between GPU and CPU speeds, however, does not keep rising with increasing task size. It reaches an asymptote of about a 400-fold efficiency. This occurs when the transfer overheads take negligible time compared to that taken by the actual arithmetical computations on the device.
- The errors in the ratio reduce with increasing task size. In addition to the statistical error, the inaccurate **clock()** funtion, as mentioned before, was also lending to the error owing to its discreteness. However, this discreteness becomes small compared to the time being measured for long process times, reducing the error in the measured time.

6 Discussions

6.1 Array length is 262144

The earliest version of the program was run at first with arrays much smaller. But then the array size was increased as it was the most natural method of lengthening the process size. For large arrays, however, the GPU times were unnaturally fast, almost the same as for very small arrays, say 100-long. Inspection revealed that the square root operation was in fact not being performed correctly by the GPU. A print statement was inserted to display sample computed values. This showed that the CPU was computing correctly, but the GPU was returning nonsense integers.

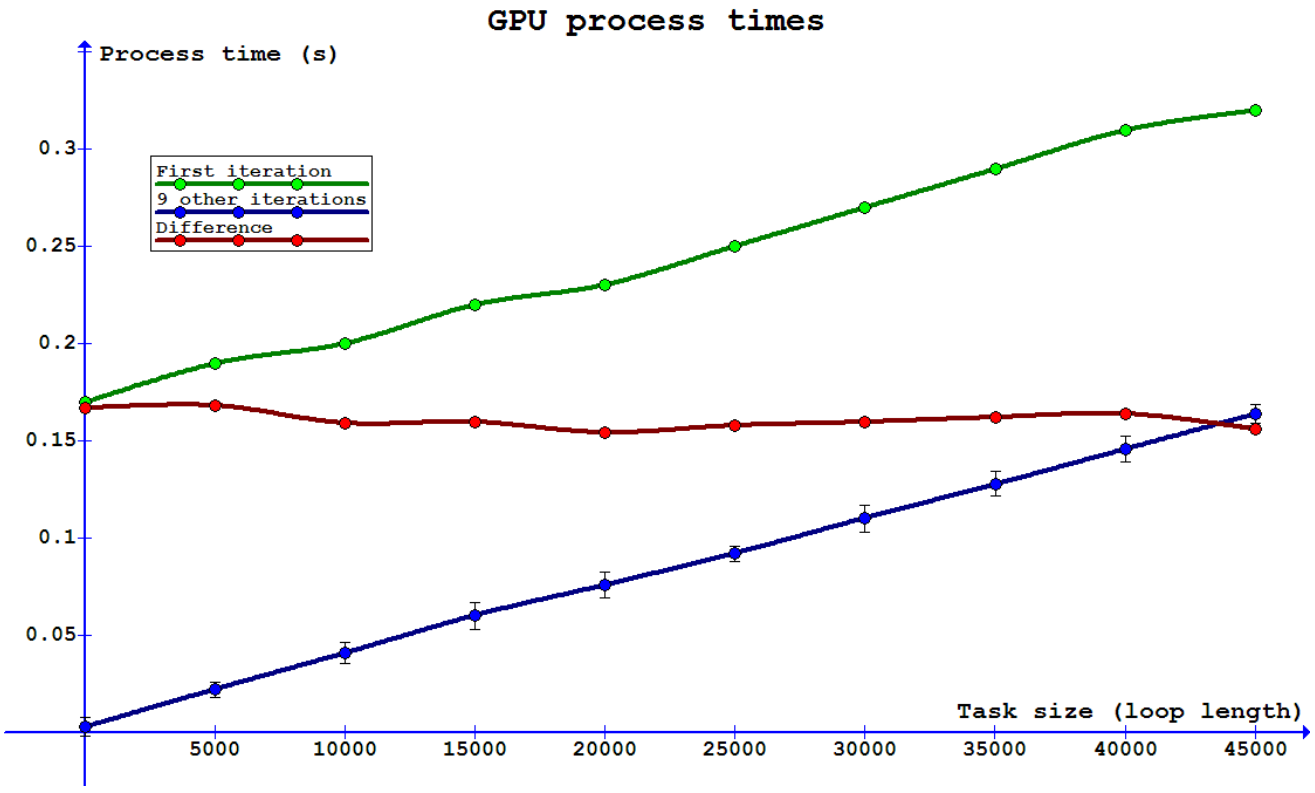
Since every data element of the array is worked on by a thread, and there is a fixed number of available GPU threads, the guess was that this issue may be related to the array size. So by trial and error (method of bisection), the array size was altered and the output observed, until the size of 262144 was reached. This is the longest array for which the GPU can successfully perform computations.

It turns out that $262144 = 2^{18}$. It also emerged from further inspection that it is equal to the no. of GPU cores (**128**) X block size (**4**, as set in line 39 of code) X maximum no. of threads/block (**512**). This is the maximum number of simultaneous threads that may be processed by the Tesla™ C870.

6.2 Longer GPU time on first iteration

As was mentioned in the section with the sample screen output, the GPU at the first iteration in any execution of the program returns a statistically significantly longer process time. This deviation is systematic, recurring irrespective of the loop length we start from.

It appeared that the additional time required was nearly the same every time, about 160 ms. The program was modified to check this over a long range of widely spaced loop lengths. The values plotted are the GPU time for the first sample; the average and standard deviation for the 9 others; and the difference between the times:



As can be seen clearly, the process time for the first iteration is deviated from the average of the 9 others by a difference that is significantly more than their statistical error.

A first explanation of the issue would be that memory needs to be allocated and arrays need to be created in the first iteration. However, these operations are also performed in every other iteration. The earlier version of the program with lines 66-67 uncommented, in which at the end of each iteration the memory for the host and device arrays are freed, also exhibited this issue, and the additional time was the same. In that case the memory allocation, array creation and initiation were steps that were clearly performed in every iteration; yet the first was reported to take longer. It may also be ruled out that the memory operations require time to initiate when first performed on an idling GPU. This is because if the program is stopped at any time and executed again immediately, this additional time always appears again in the first sample of the first iteration.

The complete data presented in this report have been generated from multiple runs of the program at different times with different ranges. Since the first iteration of every execution was known to have this issue, the runs were started from one iteration before, so that the correct numbers for the first iteration may be 'patched' in.

References

[1] CUDA homepage: `http://www.nvidia.com/object/cuda_home_new.html`

[2] CUDA Programming Guide: `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`

[3] NVIDIA specs: `http://www.nvidia.com/docs/IO/43395/C870-BoardSpec_BD-03399-001_v04.pdf`

[4] Intel Datasheet: `http://www.intel.com/Assets/en_US/PDF/datasheet/318589.pdf`