CPU-GPU Benchmarking Test

Egemen Ulucam
Department of Engineering
Izmir Institute of Technology
City: Izmir, Turkey
E-mail: egemenulucam@gmail.com

***Abstract*—This paper stated that the advantage of data parallelism of the graphics processing units (GPUs) in computing large number of data, has significant speed gain compared to their CPU counterparts. With help of the CUDA and OpenGL, we have an example to show the speed gain. The example has two parts, in part one creating a cube object and rotating it in CPU. In part two same rotating operations are done in GPU to comparison. The example's performance of GPU has increased with using task parallelism (multiple streams).**

***Keywords:* Central Processors Unit (CPU) Graphics Processor Units (GPUs), Multiple Streams, Simulations, Performance Comparison, CUDA, OpenGL.**

I. Introduction

This paper shows that Data level parallelism can increase the performance of a large data set if the application is suitable for the parallelism. A simulation of a subdivided cube object that is made up of 3,538,944 vertices will be rotated first in CPU and then will be rotated with the combination of using a CPU with a GPU. The rotation operation is basically a matrix multiplication as it can be seen in Figure 1. Each vertex can be multiplied with rotation matrix separately; therefore data-level parallelism can be applied. And to increase performance even more, we will use task-level parallelism and parse our large vertex matrix into smaller parts. Why and how these parallelism techniques are effective will be explained later. To simulate OpenGL [1] is used and for parallel programming CUDA C is used.

$$\underbrace{\overset{\text{Rotation Matrix}}{\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}}_{4 \times 4} \times \underbrace{\begin{array}{cccccccc} V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & & \\ -1 & -1 & 0 & 0 & -1 & 0 & & \\ 1 & 0 & 0 & 1 & -1 & -1 & \cdots\cdots & \\ -5 & -5 & -5 & -5 & -5 & -5 & & \\ 1 & 1 & 1 & 1 & 1 & 1 & & \end{array}}_{4 \times \text{VertexNumber}}$$

Theta(θ): Rotating Angle
VertexNumber = 3538944
4 x 3538944 = 14155776 Floating points operations

Fig. 1. Rotation Operation.

II. Background

A. CUDA

Compute Unified Device Architecture (CUDA) is a platform for parallel computing and programming model developed by Nvidia. Developers can use programming languages like C, C++, Python, Fortran and MATLAB to parallel programming, and do calculations on Nvidia's GPUs [2].

B. OpenGL

OpenGL provides a way to communicate between the programmer and the graphics hardware. It is possible to make 2D and 3D graphics.

III. Sequential Execution Part

A. CPU Part

Using C language, I created a cube object and drew it on the screen with the help of OpenGL (Figure 2.a) and started to subdivide the polygons (Figure 2.b is obtained). The purpose is to create more vertices to have a large data set, so that data-level parallelism can make sense. After creating 3,538,944 vertices I rotated the cube π/360 radian in each loop. That means doing a matrix multiplication between the rotation matrix and the vertex matrix as it shown in Figure 1. Four floating point multiplication is calculated for each vertex. Therefore, a total of 14,155,776 floating point calculations were done for one rotation operation. I rotate the cube for π/2 radian total and that means rotation operation is called for 180 times in CPU. To simplify things, I did not mention about translation operations. But before and after rotation, to rotate the object in its own center, we need to transfer the object to the world origin, then rotate it, and then transfer it back to where we want the object to be. Each rotation operation includes these steps even if it is done on CPU or on GPU. For each 3D vertex, translating to the world center means three adding operations and translating back to the object position means three more adding operations are done. But for simplicity, I will not mention about it in calculations.

B. GPU Part

The same cube object is created (Figure 2.c and Figure 2.d) for the operations will be done on GPU but in green tones to distinguish. The calculations are done on GPU, but the drawing of polygons is done on CPU.

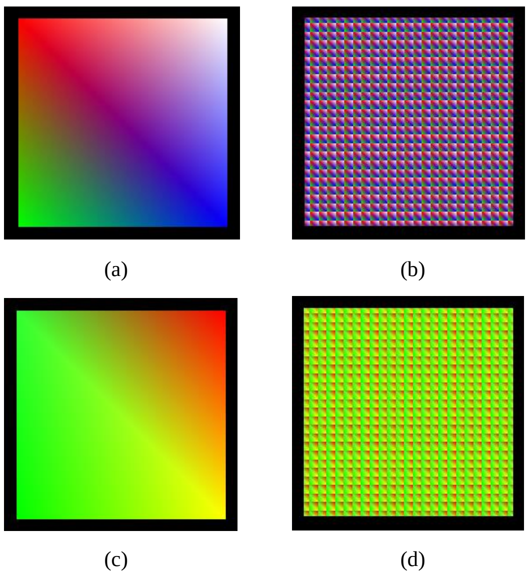


Fig. 2. Front Face of Cubes. (a) Cube created for CPU. (b) Eight times subdivided version of the cube created for CPU. (c) Cube created for GPU. (d) Eight times subdivided version of the cube created for GPU.

IV. Data Level Parallelism

As a definition, data-level parallelism is a data set that doing the same task independently can be parsed into smaller parts and can be done concurrently [3]. As an example, vector additions can be parsed into pieces and each element can be summed independently. In Figure 3 two vectors with 12 elements are separated into four pieces, and the addition of these four pieces concurrently is an example of data parallelism.

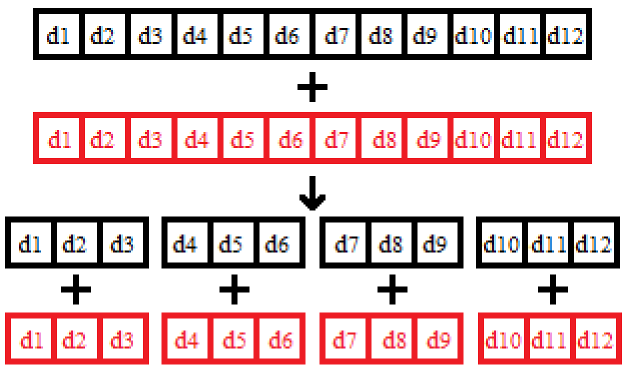
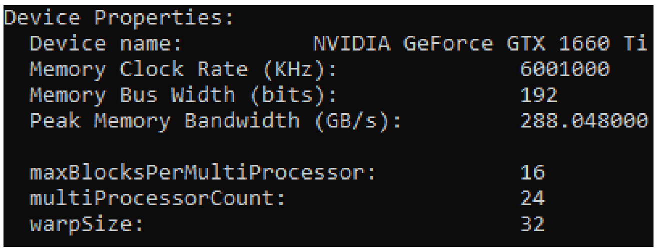

d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12
+
d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12
d1 d2 d3
d4 d5 d6
d7 d8 d9
d10 d11 d12
+
d1 d2 d3
d4 d5 d6
d7 d8 d9
d10 d11 d12

Fig. 3. Sequential vs Data Level Parallelism.

For our simulation, each vertex can be multiplied with a rotation matrix independently. Because our vertices occur of 4 elements, we can assume each vertex is a vector with length of four.

In CUDA programing to make calculations on GPUs (devices) first we need to send the data from host (CPU) to GPU. This task is bottlenecked with the bandwidth of the hardware. After copying the data to the device, we use kernels to launch our operations. When launching a kernel, CUDA needs to specify the size of the grid and blocks. The grid holds blocks and the blocks hold threads. Our calculations are run in threads. Not all the threads can work concurrently. Streaming multiprocessors (SMs) of the GPU devices controls scheduling. Multiple blocks can be assigned to each SM and SMs can execute each block as 32-thread warps. Warps are scheduling units of the SMs, and they usually occur in 32 threads on today's devices.

For our rotation operation, we first copy the vertex matrix from host to device. In my kernel launch, the grid has 27648 blocks, and each block is occurring of a total of 1024 (4 x 32) threads with two dimensions. After the kernel launch the calculated vertex matrix is copied back to the host. The simulation results may and will change according to the device properties. The test results are obtained according to the device in Figure 4.

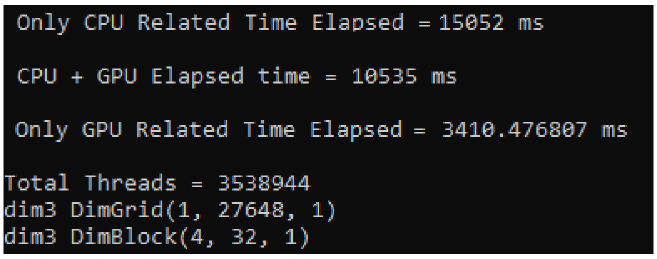


```
Device Properties:
  Device name:          NVIDIA GeForce GTX 1660 Ti
  Memory Clock Rate (KHz):              6001000
  Memory Bus Width (bits):              192
  Peak Memory Bandwidth (GB/s):         288.048000

  maxBlocksPerMultiProcessor:           16
  multiProcessorCount:                  24
  warpSize:                             32
```

Fig. 4. Device Properties.

The simulation results can be seen in figure 5.



```
 Only CPU Related Time Elapsed = 15052 ms

 CPU + GPU Elapsed time = 10535 ms

 Only GPU Related Time Elapsed = 3410.476807 ms

Total Threads = 3538944
dim3 DimGrid(1, 27648, 1)
dim3 DimBlock(4, 32, 1)
```

Fig. 5. Data Parallelism Simulation Result.

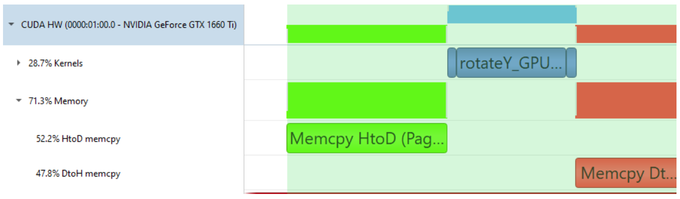
The performance improvement of data parallelism is nearly 43%. Could performance be further improved? Even if the elapsed time to do calculations is 3410 milliseconds (ms) total rotation takes 10535 ms. That is because the drawing operation is executed in sequential order. It could be executed in parallel if we separate each polygon of the cube and draw them separately, but in this application we didn't.

There is also one more way to increase performance and it is task-level parallelism.

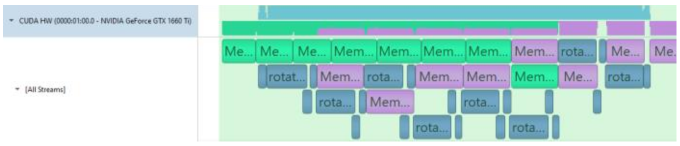
V. Task Level Parallelism

The task level parallelism is that instead of executing separated data concurrently, we separate data to execute tasks concurrently. In the previous topic we mentioned that data on the host are copied to the device. After that, the kernel launch is done. And if needed which in our application we need, the calculated data copied back to the host from the device. All these tasks are executed in sequential order.

If we parse our vertex matrix into smaller chunks and send these chunks to the device, we can start calculations as soon as the first chunk arrives. This pipelining method is done by using multiple streams in CUDA [4]. Number of streams is also an important topic but, in this paper, we will work with 8 streams just to get the basic idea about task level parallelism. The task parallelism in the timeline for our application can be seen in Figure 6. The copying operations in a single stream take nearly twice as long as the kernel execution. But with task parallelism the kernel execution starts as soon as the first chunk is copied, and the tasks are executed concurrently.



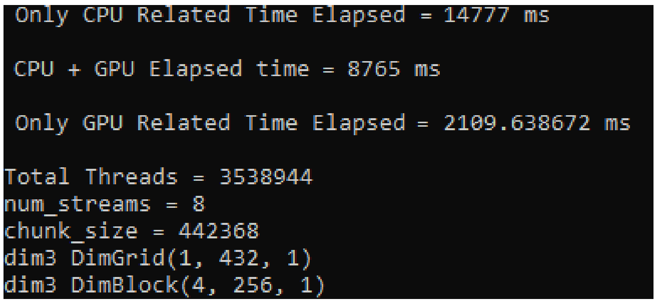
(a) With Single Stream



(b) With Eight (8) Streams

Fig. 6. Execution of tasks in timeline. For single stream green is copying data from host to device, red is copying data from device to host and purple is kernel launch. For eight steams green is copying data from host to device, pink is copying data from device to host and purple is kernel launch.

The results of the simulation with 8 streams can be seen in Figure 7.



```
 Only CPU Related Time Elapsed = 14777 ms

 CPU + GPU Elapsed time = 8765 ms

 Only GPU Related Time Elapsed = 2109.638672 ms

Total Threads = 3538944
num_streams = 8
chunk_size = 442368
dim3 DimGrid(1, 432, 1)
dim3 DimBlock(4, 256, 1)
```

Fig. 7. Simulation Result with Eight Streams.

The total time elapsed in the GPU operations has dropped from 3410 ms to 2109 ms with task level parallelism. Also, total time in simulation has dropped 20%.

ACKNOWLEDGMENT

We would like to thank Dr. Isıl Oz for giving CENG443 Heterogeneous Parallel Programming Course.

REFERENCES

[1] OpenGL Home Page, https://www.opengl.org.

[2] CUDA Home Page, http://developer.nvidia.com/object/cuda.html.

[3] Kirk, D. B., &amp; Hwu, W.-mei W. (2013). Heterogeneous Parallel Computing. In Programming Massively Parallel Processors A Hands-onApproach (second, pp. 10–22). essay, Elsevier Inc.

[4] Rennich, S. (n.d.). CUDA C/C++ Streams and Concurrency. Retrieved from https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.