COMPARING PARALLEL OPERATION PERFORMANCE FOR PIXEL PROCESSING IN MOBILE NVIDIA GPU AND INTEL CPU IN WINDOWS 10

Valeriu Manuel IONESCU
University of Pitesti
Department of Electronics, Communications and Computers
Pitesti, Romania
valeriu.ionescu@upit.ro

Keywords: parallel processing, CUDA, bitmap, image processing, MangedCUDA

Abstract: *Parallel processing is necessary in many computer science domains. Graphics cards, even non-professional ones, are designed to perform very fast parallel processing tasks. This paper will provide an overview of recent mobile NVIDIA Graphics Processing Unit architecture, the benefits for using a GPU in parallel programming and will investigate the effect of operation grouping on the parallel processing performance of image pixels. The GPU performance will be compared to that of a multi core CPU designed for use in mobile platforms, in a pixel targeting image-processing algorithm.*

1. INTRODUCTION

More devices than ever are connected to a network and send data that needs to be processed and stored. Processing data faster is even more necessary as the data sources multiply and the server must keep pace with their number and the data volume. Image and video processing are some of the most intensive processing tasks.

Performing these parallel operations with a CPU which is characterized as a low latency processor is not optimal as current mobile CPUs and execute up to 8 threads in parallel (sometimes desktop CPUs are included in laptops but this massively decreases the devices mobility) and are usually assigned to GPUs that are known as throughput processors.

Low latency processors are optimized for low latency data access to information that resides in the cache and perform speculative execution in order to get to the result; this gives good results even in the serial sections of the algorithm.

High throughput processors are optimized to process parallel data while being optimized to handle memory latency. [1, 2]

For example a shading operation can imply many multiply and multiply-add and clamp operations.

The problem with throughput processors are stalls that appear when data to be processed is not ready because it depends on the result of a previous operation. One solution is to process on a single core as many parts of the operation needed to be executed as possible without having to wait for more data. Another is to always be able to move to a non stalled group and execute the operations there until one of the stalled groups gets the data and can resume the computation. Also the processing cores in a GPU need not be as complex as CPU cores. [3]

When moving data between the CPU and the GPU the limiting factor becomes the memory bandwidth; the solution in this case is to request data less often and perform more mathematical operations on it while being in the GPU.

This paper will test different logical and arithmetic operations on the CPU and on the

GPU and performance considerations will be drawn in the conclusions.

In the following paper the CPU is the host that executes functions and the GPU is the device that executes kernels. The kernel is executed by the entire GPU on multiple threads, as allowed by the GPU architecture, specified by the user or needed by the processing algorithm.

2. PROCESSING FLOW IN CUDA

First step of processing flow is copying the data from main memory to GPU memory. After this, CPU loads the program to the GPU which will execute it in parallel in each core, locally caching the data in the GPU. At the end of the operation the result from GPU memory is copied back to main memory. Starting with CUDA 6 the migration operations between the host (CPU) and the device (GPU) memory are performed by the system automatically. These operations are presented in Fig. 1.

CUDA characteristics:

- Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
- The Fermi architecture provides fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. With a configuration setup this type of behavior can be prevented however and the GPU can be forced to separately compute the addition and the multiplication.
- Fermi's integer ALU supports full 32-bit precision for all instructions. The integer ALU is optimized to efficiently support 64-bit and extended precision operations.
- Various instructions are supported, including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.

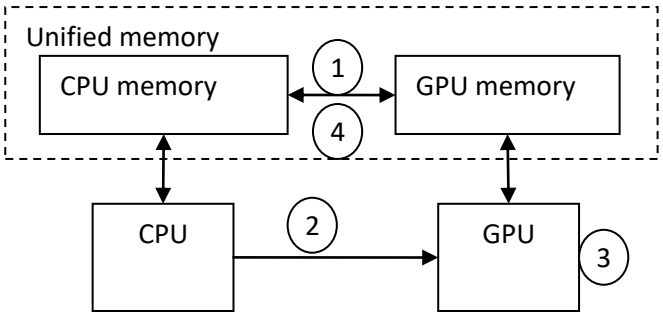

Fig. 1 CUDA processing flow: 1-load data to GPU; 2-load program to GPU and execute; 3-execute operations in GPU on many threads; 4-load data in CPU system memory

The characteristics for the tested GPU were obtained by using in CUDA C:

- First using the `cudaGetDeviceCount` function that returns the number of compute 2.0 capable devices on the system;
- For each of the devices found, the function `cudaGetDeviceProperties` with two parameters, first is the `cudaDeviceProp` and the second is the device number in case there are multiple GPUs connected to the system. Properties like `maxThreadsPerBlock` or `clockRate` are read from the device;

The resulting information is presented below:

```
CUDA Device #0
Major revision number:         5
Minor revision number:         2
Name:                  GeForce GTX 970M
Total           global           memory:
3221225472
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum          memory          pitch:
2147483647
Maximum threads per block:     1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  1024
Maximum dimension 3 of block:  64
Maximum    dimension    1    of    grid:
2147483647
Maximum dimension 2 of grid:   65535
```

```
Maximum dimension 3 of grid:   65535
Clock rate:                    1038000
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     10
Kernel execution timeout:      Yes
```

The Geforce 970M (using a Maxwell GM204 core) has 10 streaming multiprocessor (SM) units, giving 1280 Shader Processing Units and 3GB of RAM accessed through 192-bit memory interface.

3. CREATING THE CUDA PROJECT

The project was created using ManagedCUDA 7.0 that allows writing the project in C# while the kernel is written in CUDA C.

First, the kernel that runs on the GPU must be created and exposed in order to be used in C#. In order to do this, the function implemented in the kernel should be marked as *__global__*. This is a GPU kernel function launched by the CPU and must return void. If there are other functions that need to be called from the GPU (and executed also on the GPU) they should be implemented as __device__.

A sample kernel is presented in [4]. The implementation used in this paper uses a similar kernel, with various function implemented between the two parameters:

```
__global__ void VecAdd(const float* A,
const float* B, float* C, int N)
    {
    int i = blockDim.x * blockIdx.x +
threadIdx.x;
        if (i < N) C[i] = A[i] * B[i];
    }
```

The section highlighted is where the different array operations were implemented (such as division, modulo or masking).

Each kernel as an Id. Threads are grouped into blocks and blocks are grouped into a grid that is executed by the kernel.

In the code: `blockDim.x`, `blockIdx.x`, `threadIdx.x` allow the program to know where it is executed. They act similarly to a memory paging in an operating system and allows the identification of the thread running on a specific array element: the *blockIdx.x* is the block that the kernel is running and *blockDim.x* is the number of threads in one block; their product gives the only the block of threads location; finally *threadIdx.x* acts as an offset in that block. The *blockDim.x* is set/constant after the kernel is called, the id of the first block is 0 (*blockIdx.x*) and the id of the first thread in a block is also 0 (*threadIdx.x*). The process is illustrated in Fig. 2.

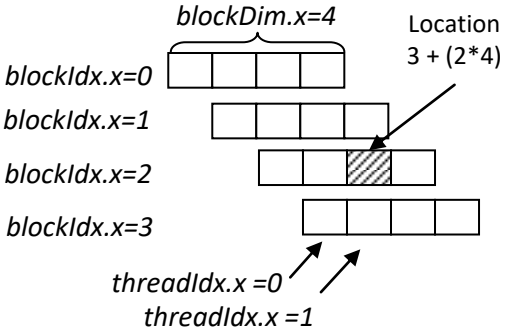

Fig. 2. Finding the thread that runs on a specific array element

For the given GPU, as there are 10 multiprocessing units (10 blocks are executed in parallel), and each can run 1024 threads per block (a 1024x1x1 configuration), the maximum number of parallel threads that the hardware allows is 10240.

All other processing is placed in blocks in a queue that wait for the previous blocks to finish.

For example for a 5.000.000 pixels image with 1024 threads per block there will be 4882 blocks waiting to have one of the 10 processors allocated.

The name of the resulted kernel is in the .ptx file.

After the kernel is compiled,

The C# program structure is the following:

-create the CUDA context, an instance of ManagedCuda: `CudaContext()`;

-load the precompiled kernel (.ptx file) using `CUmodule`;

-the function from the kernel is loaded, defined as a `static CudaKernel`;

-the array to be processed (bitmap array) is loaded;

-the data is copied to the device;

-the kernel is run;

-finally the data is copied from the device to the CPU.

Loading the images pixels in an array fast was made using an unsafe function with the Marshal.Copy(*pointer*, *rgbValueArray, startIndex, NumberOfBytes*) function instead of the GetPixel() method that is very slow (unsuitable for large images).

The steps were: locking the bitmap's bits; getting a pointer to the start of the bitmap; iterating through the image's bits using the step/stride (the width of a single row of pixels, rounded up to a four-byte boundary); finally reading the RGBA components. Bitmaps have the pixels are laid out in memory row by row therefore the iteration was made by going though rows first then columns.

4. CPU CODE IMPLEMENTATION

The laptop CPU used was Intel 4720HQ with 4 cores and 8 threads, running at 3.6GHz on all cores with 16GB of RAM, running Windows 10. The CPU over clock was made using the Intel XTU utility in order to obtain as much performance as possible from the CPU.

The CPU code implementation using C# created new threads with the function:

```
Thread thread1 = new Thread(new ThreadStart(A));
```

Each thread was processing a part of the array determined by a rounded number of array elements. The final thread was processing any extra array elements if the array division result between the array elements and the number of threads was not an integer number. Finally the threads were waited to be terminated using Join:

```
thread1.Join();
```

In both the GPU and the CPU code the timing was monitoried on the CPU using the stopwatch function:

```
var watch = Stopwatch.StartNew();
```

The time was recorded using the Elapsed property: `watch2.Elapsed.TotalMilliseconds`

5. RESULTS

The GPU code was run for an array with the size of 5000000 (results in Table 1 and Fig. 3).

Table 1. Running the processing for an array of size 5000000 multiple times in order to determine the average processing time, for different number of operations. Here mul- multiply and add -addition operation

Number and type of operations	1mul 1add	3mul 2 add	6mul 5 add	12mul 10 add	24 mul 20 add	36mul 30 add	48mul 40 add	72mul 70 add
Processing time (s)	0.022161	0.023309	0.023006	0.029186	0.033356	0.037844	0.041825	0.051044
	0.020638	0.021359	0.022631	0.026138	0.029561	0.034634	0.038348	0.047853
	0.021148	0.021661	0.022517	0.025828	0.029753	0.034334	0.03849	0.048188
	0.02162	0.021505	0.022366	0.025951	0.029838	0.034602	0.039005	0.047701
	0.021381	0.021429	0.021891	0.026065	0.029876	0.034608	0.038729	0.04774
	0.021163	0.021687	0.021911	0.026202	0.029871	0.034282	0.038972	0.047421
	0.021256	0.021777	0.022694	0.026355	0.029947	0.034204	0.03852	0.047944
	0.02141	0.021945	0.02238	0.02693	0.030274	0.034403	0.039071	0.047559
	0.0213	0.021757	0.02247	0.02668	0.029941	0.034374	0.038707	0.047862
	0.021331	0.021789	0.022391	0.025813	0.029677	0.034486	0.039623	0.047814
	0.021486	0.022028	0.02304	0.025368	0.030021	0.034195	0.038986	0.047875
	0.021114	0.022224	0.022864	0.025866	0.030063	0.034452	0.039304	0.047664
	0.021104	0.021806	0.022761	0.02595	0.029937	0.034229	0.039105	0.047614
	0.021096	0.021625	0.022394	0.026223	0.030605	0.034476	0.039306	0.047725
	0.021794	0.021589	0.022788	0.026162	0.030057	0.034701	0.038524	0.047789
	0.020844	0.021668	0.022256	0.026135	0.030433	0.034907	0.039377	0.047849
	0.022194	0.022149	0.023329	0.026259	0.030238	0.03469	0.038613	0.048023
	0.022765	0.023049	0.023155	0.02699	0.030513	0.034935	0.0386	0.047406
	0.020901	0.021739	0.022957	0.026349	0.030238	0.034556	0.03863	0.048025

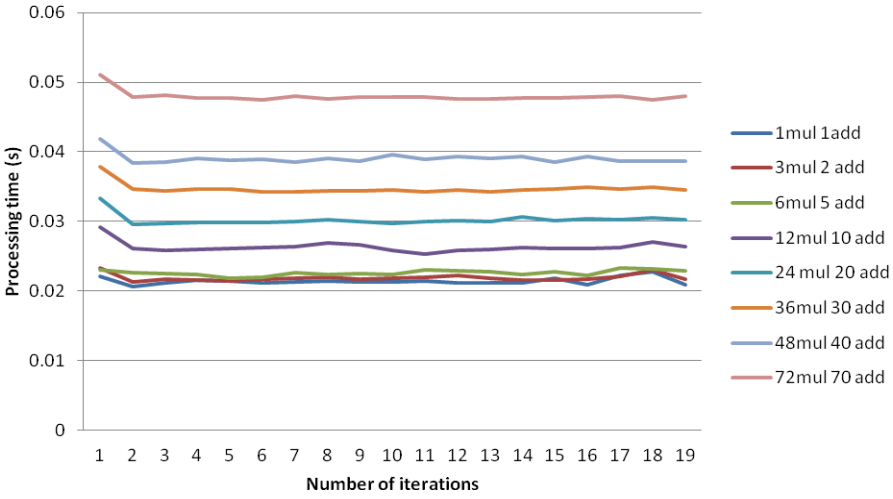

Fig. 3. Graphical representation of the processing times from Table 1. The results show that the time increase is liniar

Other operations were also tested (division, `__fdividef`, modulo, and masking) and the results are presented in Fig. 3. The conclusion is that depending on the operation type the processing time varies.

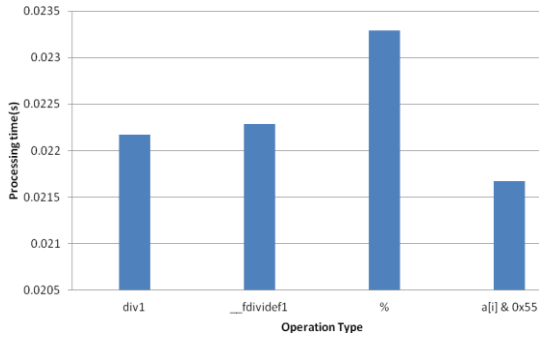

Fig. 3. Graphical representation of the average processing times from Table 2. There are differences in the processing times depending on the operation type

On the CPU, it was necessary to determine what the best threaded configuration is and the results for the tests on a different number of threads are presented in Table 2 and graphically in Fig. 4.

This shows that the best results are obtained when 4 threads are executed, with 8 thread result being very close. Running more threads than the system can handle (16 threads) results in a massive speed penalty as they have to wait to obtain access to the CPUs.

Table 2. Running the 2 addition and 3 multiply operation on the CPU for a different number of threads

1Thread Time (s)	2Threads Time (s)	4Threads Time (s)	8Threads Time (s)	16Threads Time (s)
0.029682	0.01552	0.014228	0.015899	0.026916
0.028231	0.015882	0.014348	0.015935	0.025735
0.028361	0.016104	0.014212	0.016583	0.026721
0.028368	0.015936	0.014073	0.016401	0.025821
0.028696	0.016053	0.014251	0.015971	0.026334
0.029032	0.015242	0.014462	0.016404	0.025576
0.028938	0.015816	0.01404	0.016098	0.026556
0.028449	0.016241	0.013648	0.017377	0.025969
0.02861	0.015664	0.01433	0.016969	0.026805
0.029507	0.015739	0.014444	0.016762	0.027769
0.02779	0.015675	0.014411	0.015763	0.025794
0.029192	0.015433	0.01527	0.016404	0.027824
0.028929	0.01627	0.014227	0.015722	0.030673

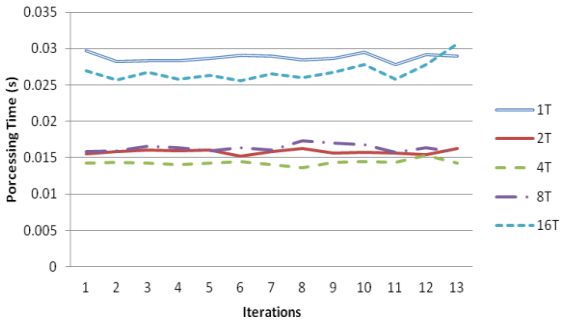

Fig. 4. Graphical representation of the processing times from Table 2. Running on 4 threads shows the best speed for this mobile CPU

Table 3. Best CPU results when running on 4 threads for a different number of operations

Operation type and count	1 add 1 mul	2 add 3 mul	5add 6mul	8add 10mul	10add 12 mul	72mul 70 add
Best time (s)	0.0615566	0.0738509	0.0794391	0.0815901	0.08677	0.148914

The best times were collected and placed in Table 3.

When comparing the CPU average result for the same number of operations with the result obtained on the GPU (for 72 multiply and 70 addition operations), we can see it is three times slower.

The average time however for the same number of operations for the CPU is 0.233931571 s, that is five times slower. The consistency of the results is better for the GPU.

6. CONCLUSIONS

The paper tested the processing of bitmap pixels on both mobile CPU and GPU. The pixel information was read from the bitmap as an array and the processing was performed with the results being read by the CPU. The execution time was recorded using the CPU as it is the one that will finally use the results.

The GPU showed more result consistency when compared to the CPU due to the fact that the CPU has other tasks/services that need to be processed in the background.

For the operations considered, the mobile GPU showed an up to 3 times performance improvement for the best times and an increase of 5 times in performance for the average values when compared to the CPU.

One further research related to this paper will be to determine how the performance is modified with the size of the structure used for the GPU processing.

7. ACKNOWLEDGMENT

The author thanks Andrei Elena for the support in testing the CPU and GPU implementation presented in this paper.

8. REFERENCES

[1]. NVIDIA, NVIDIA Tesla P100, Whitepaper, WP-08019-001_v01, 2016

[2]. Whitepaper NVIDIA GeForce GTX 980, v1.1, 2014, Accessed December 2016, Web: https://goo.gl/wo2jfj

[3] Sanders. Jason, Kandrot. Edward, “CUDA by example: an introduction to general-purpose GPU programming”, Addison Wesley, ISBN 978-0-13-138768-3, July 2010

[4]. Michael Kunz, "managedCuda”, Accessed December 2016, Web: https://kunzmi.github.io/managedCuda/