International Masaryk Conference for PhD students and young researchers (MMK2014)
Hradec Králové, 15-19 Dec. 2014, pp. 3354-3360, ISBN 978-80-87952-07-8

OPTIMIZING GENERAL PURPOSE COMPUTATIONS USING KEPLER BASED GRAPHICS ACCELERATORS

Sándor Szénási
John von Neumann Faculty of Informatics, Óbuda University, Budapest
szenasi.sandor@nik.uni-obuda.hu

Abstract

The programming of GPUs (Graphics Processing Units) is ready for practical applications; the largest industry players (including research centres, financial and analyst corporations) have already announced that they use these new devices for high computing applications. There are several well-known areas, like image processing, simulations and obviously 3D graphics, where we can use these devices very efficiently. In this paper, we would like to show, that beyond these well-known topics, GPU programming is able to speed-up more general purpose applications. The key is the data parallel nature of the algorithm, and the minimization of data transfers between CPU and GPU.

Key words: *GPGPU, GPU development, CUDA, data-parallel algorithms, speed-up, matrix operations*

1 INTRODUCTION

Originally, display adapters were responsible for displaying content on the screen; however, this has changed significantly in the last ten years. Nowadays, most of the available display adapters are capable of doing general computations. The first step in this direction was the appearance of the 3D accelerator cards in the 90s, and the integration of these into the main display adapters. Due to the integration of several new functions, these cards become more and more comprehensive devices, and in the last few years, we can execute general programs [1][2][3][4] using these. In case of NVIDIA hardware, we should use the CUDA environment to handle this process, this helps us to compile and run kernels in the GPUs.

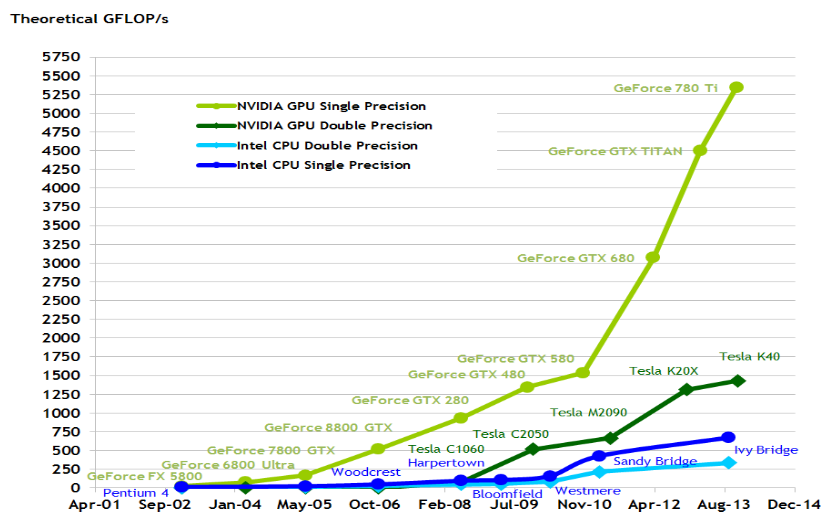

Fig. 1. Theoretical peak performance of GPUs and CPUs [5]

The main reason why we want to run applications in the GPU is the enormous computing capacity of these devices. As Fig. 1 shows, the peak performance of the graphics accelerator cards is about one or two magnitudes higher than the traditional CPUs. However, these

devices have several limitations, and this high performance is not available with all types of tasks. This paper introduces some aspects about when it is worth porting algorithms to the graphics cards.

2 GPGPU ADVANTAGES

There are several reasons for this high peak performance, the main ones of these are as follows [6]:

- Large number of processing units
- Complex and efficient memory architecture

1.1 Large number of processing units

The main direction of the traditional CPU development is to increase the number of cores. Similar to this, in the case of video display adapters, we can see more and more processing units too. In this case, we can talk about quite different dimensions; the number of processing units is more than one thousand in the top-level graphics cards.

Fig. 2. shows the architecture of a Kepler based streaming multiprocessor (manufactured by NVIDIA). As you can see, most of the die area is occupied by CUDA cores (green rectangles). One streaming multiprocessor (SMX) contains 192 pieces of processing units, and one graphics card contains several multiprocessors. These units are much simpler than the cores of a traditional CPU, but the high number of them leads to this increased performance.

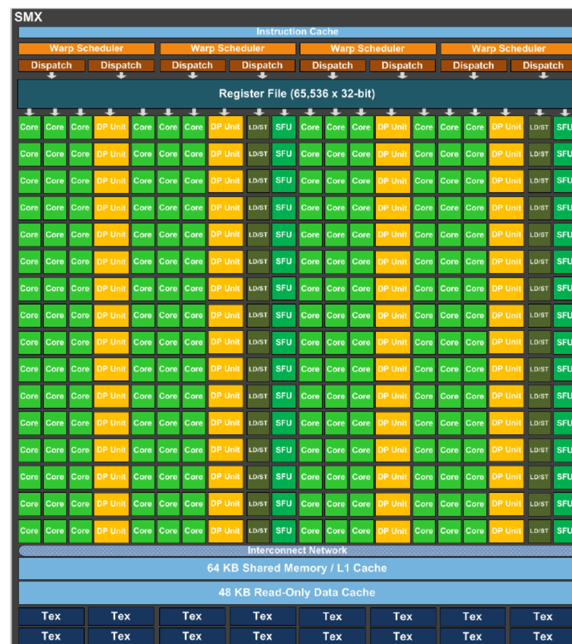

Fig. 2. NVIDIA Kepler architecture [7]

1.2 Memory architecture

Current GPUs have very complex and efficient memory hierarchy. As Fig. 3 shows, a Kepler based GPU contains the following memory regions:

- Fast on-chip memories (registers, shared memory)
- Slower off-chip memories (global memory)

- Cache regions (L1, L2)

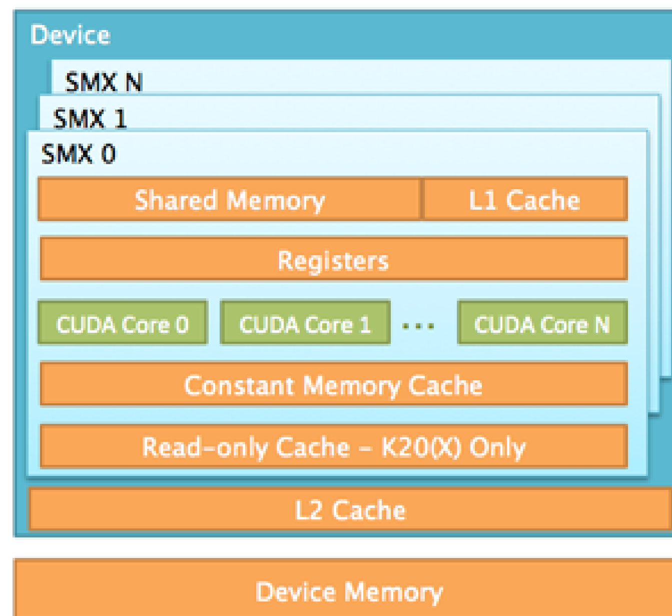

Fig. 3. Memory hierarchy of Kepler based multiprocessors [8]

Registers are similar to the common registers of traditional CPUs. There is a surprisingly large number of registers in each streaming multiprocessor (65536 32-bit registers in Kepler SMX), however we have to take into consideration that these devices are usually running large number of threads. If we scheduled the maximum number of threads in a multiprocessor (2048 in the case of Kepler), then each thread can work with only 32 registers.

The shared memory and the L1 cache are basically the same. Developers can configure the multiprocessor to use the given on-chip memory as an L1 cache, or as a shared memory available for the programmer. This memory is fast but the size is very limited (64K per SMX in the case of Kepler architecture).

The global memory is the largest and slowest memory region. To speed up the memory access procedure, a quite large (1536KB per SMX in case of Kepler architecture) L2 cache exists between the multiprocessor and the device memory.

3 DISADVANTAGES OF GPU PROGRAMMING

3.1 Data-parallel execution

The main advance of the GPUs is the large number of execution units. It is clear that we have to run at least as many threads as the number of CUDA cores to utilize the graphics card (for example, in the case of Tesla K40 accelerators, the number of cores is 2880). In practice, due to the fact that the GPUs latency hiding is based on the fast context switching, we have to run 3-4 times more threads parallel to reach the peak performance. As you can see, we can only implement highly parallelisable tasks in the GPU only efficiently.

GPUs are basically created for 3D graphics acceleration. Nowadays, these are available for general purpose computations, but the architecture is basically developed for the original purpose. In 3D graphics tasks (rendering, effects, etc.) we usually run thousands of parallel threads (for example, these can calculate the appropriate colour for each pixel in the screen). These threads are independent of each other from the data point of view (no one has to wait

for the results of others), but all of them execute the same algorithm. We can call this as "data-parallelism" and that is the main operating principle of GPUs.

Due to these restrictions, the number of problems effectively solvable using GPUs is very limited. Fortunately, we do not always have to implement the entire algorithm in the graphics card; it is possible to create programs using the CPU and the GPUs together.

1.3 Separate memory regions

Another bottleneck of CPU/GPU programming is the separated memory hierarchy. The GPU has its own memory areas. As it was introduced in the previous section, it has several advantages. In contrast, this can cause several disadvantages too. The GPU and the CPU have separate memory areas, and both of the devices can use only their own memory. This means that in case of hybrid approaches (when we want to use both the CPU and GPU) we have to copy data from host memory to device memory and vice versa. This copy is usually done via the PCIe bus, and the low bandwidth of this can cause significant performance degradation.

This latter drawback further limits the area of effectively implementable problems. According to this, we should be very careful in the software design phase [9], and make good decisions about what parts of the code we want to implement as a GPU kernel.

4 RUNTIME MEASUREMENTS

As you can see, GPU development has several advantages and disadvantages. We can use this technique only for a limited number of problems, but in these cases we can reach significantly better performance. There are several well-known areas, like image processing [10], simulations, optimizations[11] and obviously 3D graphics, where we can use these devices very efficiently.

In this paper, we would like to show, that beyond these well-known topics, GPU programming is able to speed-up more general applications. The key is the data parallel nature of the algorithm, and the minimization of data transfers between CPU and GPU.

Working with matrices is not quite as spectacular an area as the previously mentioned ones. In the case when we do not have to often transfer the sub results between the CPU and the GPU, we can achieve very good runtimes. Most of the basic matrix operations are very implementable using the data-parallel fashion. Therefore, if we can compensate for the time wastage caused by the input and output data copy procedure, it may be worth using graphics accelerators for general purpose computing.

In this paper, we introduce the performance of two basic matrix operations, addition and multiplication (of random [12] values). The main difference between them is the number of operations according to the number of the elements.

In the case of squared matrices, where the number of elements is M = N*N:

- Matrix addition: number of operation is M
- Matrix multiplication: number of operations is M*M

Test environment:

- CPU configuration: Intel® Core™ i7-2400 (Sandy Bridge)
- GPU configuration: NVIDIA Tesla K40c (GK110B)

1.4 Matrix addition

Fig. 4 shows the measured runtimes in the case of matrix addition. We have run the same test using different matrix sizes (we investigate N*N matrices). As you can see, if we strictly focus only on the addition part of the algorithm, the GPU code is faster than the CPU (except some very small N values). However, the overall runtime of the GPU code (including the memory transmissions) is always greater. That is what we expect, because in this case the ratio of calculation/memory transfers is too low to efficiently utilize the GPU resources. It is clear that if we need only one matrix addition, it is not worth doing it with the GPU.

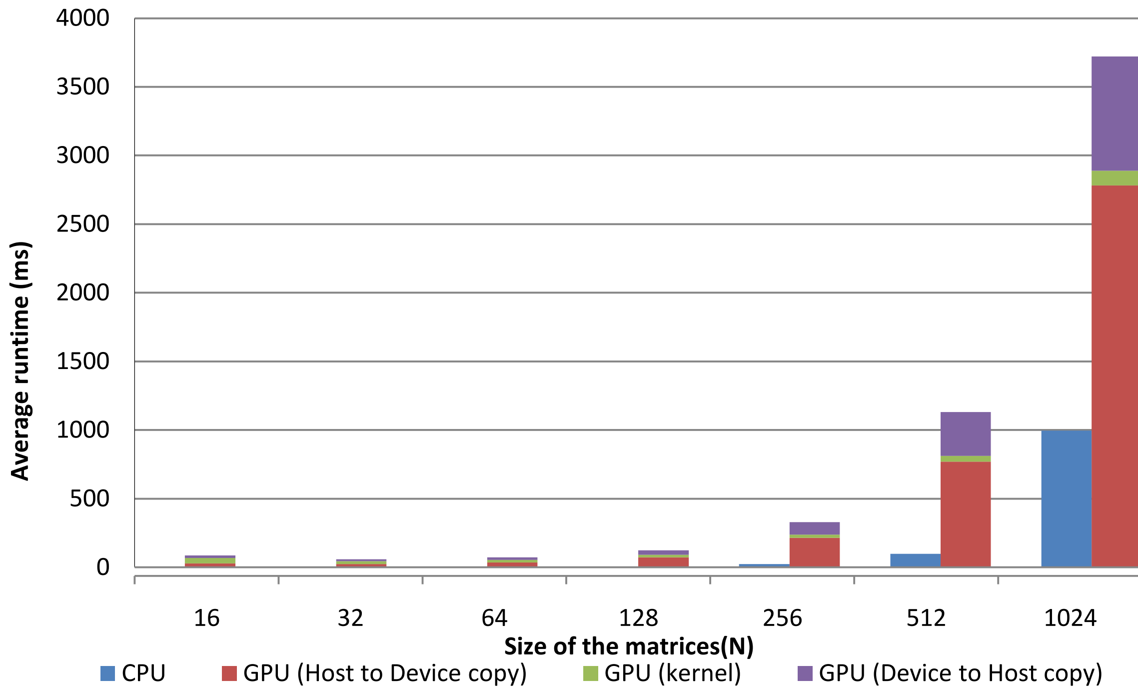

Fig. 4. Runtime of matrix addition algorithms

In practice, it is more general that we have to do additional calculations using the input data and derived values. In our second test, we: 1) transfer the input matrices to the GPU memory 2) execute the addition K times 3) transfer the final results back to the CPU. The last column of Table 1 shows the smallest K value, where the full runtime is equal for the CPU and the GPU in case of different matrix sizes. As you can see, in the case of larger matrices, this number is quite small, if we do two or more additions, the GPU becomes faster.

Table 1. Runtime of matrix addition algorithms

Matrix size (N)	CPU runtime (ms)	GPU runtime (ms)	GPU details: Host to Device memory copy	GPU details: Kernel	GPU details: Device to Host memory copy	Minimum number of operations
16	0.16	86.68	29.38	40.12	17.18	n/a
32	0.30	59.58	24.66	20.60	14.32	n/a
64	0.98	73.28	34.90	19.78	18.60	n/a
128	4.36	125.50	72.86	19.98	32.66	n/a
256	24.20	328.64	214.62	23.28	90.74	332
512	98.16	1132.46	770.96	40.54	320.96	19
1024	996.60	3722.18	2782.20	107.08	832.90	5
2048	4224.74	12435.24	9196.98	333.32	2904.94	4

1.5 Matrix multiplication

The results of the matrix multiplication runtime evaluation are shown in Fig. 5. The difference between the CPU and GPU runtimes are more significant in this case. Except for some small N values, the GPU is faster, and the results are better in case of larger matrix sizes. You can see that in the case of medium and large matrices, it is worth using the GPU for computations. The calculations are so fast that this can balance the memory transfer overhead.

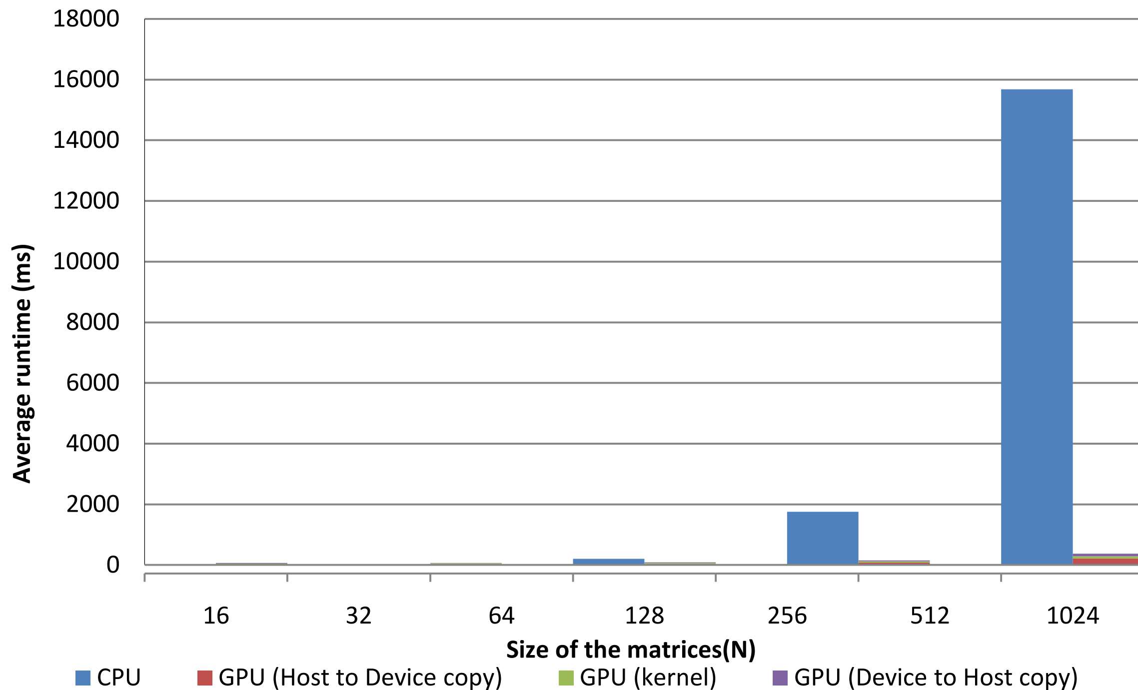

Fig. 5. Runtime of matrix multiplication algorithms

In the case of CPU implementation, we used a single threaded sequential naive algorithm for the multiplication. In the case of the GPU, the implementation of the multiplication is based on the freely available CUBLAS library.

5 RESULTS

It is difficult to clearly make a decision in the matrix addition case. As you can see, the calculation itself is significantly faster using the GPU, but the memory transfer overhead causes higher overall runtime. It is worth checking Table 1, where you can see that if we need multiple additions, using the same (or already available) data, we should take into consideration the GPU kernel.

The matrix multiplication tests give clearer results. As it was expected, in the case of small matrices, the CPU was faster. But beyond a critical size limit, the GPU code becomes significantly faster. In these cases, this speed advantage can compensate the time waste caused by the memory transfers.

It is worth noting that the previous tests correspond to the worst case scenario from the data transfer point of view. We had to transfer both input matrices to the GPU and transfer the resulting matrix back to the CPU. In practice, it is more common to do a chain of calculations, where several steps use the previous sub-results. In these cases, the drawback caused by the memory transfers is less significant.

Acknowledgements

The authors would like to thank NVIDIA Corporation for providing graphics hardware for the GPU benchmarks through the CUDA Teaching Center program. The authors also wish to thank the members of the CUDA Teaching Center at Óbuda University for their constructive comments and suggestions. The content of the paper however does not necessarily express the views of these companies and persons, the authors take full responsibility for any errors or omissions.

References

[1] S. Sergyán, Z. Vámossy, B. Szántó, and P. Pozsegovics, "Távolsági mértékek képi adatbázisok tartalom alapú keresésében," in *Informatika a felsőoktatásban 2011 konferencia*, Debrecen, 2011, pp. 435-442.

[2] K. Par and O. Tosun, "Real-time traffic sign recognition with map fusion on multicore/many-core architectures," *Acta Polytechnica Hungarica*, vol. 9, no. 2, pp. 231-250, 2012.

[3] A. Keszthelyi, "About passwords," *Acta Polytechnica Hungarica*, vol. 10, no. 6, pp. 99-118, 2013.

[4] S. Sergyán and L. Csink, "Consistency Check of Image Databases," in *2nd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence*, Timisoara, 2005, pp. 201-206.

[5] NVIDIA, "CUDA C Programming Guide," 2014.

[6] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[7] NVIDIA, "Whitepaper. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110," 2012.

[8] K. Goss. (2013, May) Kelly Goss's blog. [Online]. http://acceleware.com/blog/32

[9] J. Tick, "Potential Application of P-Graph-Based Workflow in Logistics," in *Aspects of Computational Intelligence: Theory and Applications: Revised and Selected Papers of the 15th IEEE International Conference on Intelligent Engineering Systems 2011, INES 2011.*, Poprad, 2013, pp. 293-303.

[10] G. Windisch and M. Kozlovszky, "Improvement of texture based image segmentation algorithm for HE stained tissue samples," in *2013 IEEE 14th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov. 2013, pp. 273-279.

[11] E. Toth-Laufer, M. Takacs, and I. J. Rudas, "Interactions Handling Between the Input Factors in Risk Level Calculation," in *IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Nov. 2013, pp. 71-76.

[12] G. Györök and M. Tóth, "On Random Numbers in Practice and their Generating Principles and Methods," in *International Symposium on Applied Informatics and Related Areas (AIS)*, 2010, pp. 1-6.

Contact

Dr. Sándor Szénási, Ph.D.
Óbuda University, John von Neumann Faculty of Informatics, Institute of Applied Informatics
Bécsi út 96/B. Budapest, Hungary
Tel: +36 (1) 666-5532
email: szenasi.sandor@nik.uni-obuda.hu