Runtime Comparison of CPU and GPU Using Portable Programming Models

Franz Wiesinger, Florian Nairz, and Michael Bogner

***Abstract*—Since increasing clock speeds are not enough to speed up computation, there exist several alternative options. One of them is parallelism. For some problems it is possible to use the graphics processor as a massive parallel system and gain high speedups. Since NVIDIA introduced the unified device architecture and AMD switched to the OpenCL programming model it is possible for everyone to achieve high speedups through massive parallel systems easily. In this paper CUDA and OpenCL are introduced and differences are shown. To point out that graphics processor can be used for common tasks too, there is a comparison on runtime of two different test cases. The results show why the problem size is a very important factor for decisions to make use of the massive parallelism.**

***Index Terms*—Runtime comparison, CUDA, OpenCL, graphics processor, parallelism, microprocessor.**

I. Introduction

Since the invention of the transistor there is the need to speed up the computation to solve problems of increasing complexity. In the past the hardware was the only limiting factor to achieve the goal of speeding up computation. Nowadays, software developers have to deal with multiple parallel platforms and technologies.

Until the not-so-distant past speedup was mostly gained by increasing the clock speed of a processor. Due to physical limitations this method is not feasible any longer.

Parallelism turned out to be a good alternative method to gain further speedups because it makes it possible to split a problem into several parts and solve more than one of them at a time.

Since the beginning of this century processor architectures turned from single-core to multi-core. This caused decreasing execution time because of parallelization. On the other hand, there are not only multi-core CPUs to obtain parallelism, but also GPUs (Graphics Processing Units).

Earlier GPUs used a fixed-function pipeline for calculating results. The different stages of calculation could be configured but not programmed. The next big thing was a unified graphics pipeline which can be fully programmed. This technique was optimized that it is now possible to provide a pool of unified cores that can be used for computing. Further, this turned the GPU into a massive parallel system that is able to solve more than graphical problems [1].

Manuscript received June 24, 2014; revised September 16, 2014.

The authors are with the Department of Embedded Systems Engineering, University of Applied Sciences Upper Austria, Hagenberg, Austria (e-mail: {Franz.Wiesinger, Florian.Nairz, Michael.Bogner}@fh-hagenberg.at).

A. Research Objectives

Up to now most software was written single threaded. This implies that instructions are executed one after another. The execution speed mainly depends on the clock speed of the processor. The parallelization of software speeds up execution time and therefore saves energy as well [2].

But not all problems can be parallelized and not every parallelized problem could provide remarkable speedups. The aim of this work is to find problem structures that are worth parallelization, find the break-even point and the speedup that could be achieved. Therefore we compared two often used algorithms using different programming models and their CPU equivalent by their runtime.

This paper is structured as follows: Section II discusses related work. In Section III we provide a short introduction to the differences between CPU and GPU. Section IV introduces to the different programming models used to write applications utilizing the GPU. Section V provides information about the test procedure and in the Section VI and Section VII a description of the implemented tests are given. In Section VIII the results are presented and discussed. Finally Section IX gives a future outlook.

II. Related Work

There already exist performance and runtime comparisons of programming models for multi-core and many-core systems. In [3] they provide a great comparison on performance of CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) on different devices but lacks the comparison of this calculations to runtimes of widespread used microprocessors. Ref. [4] and [5] provide excellent work in the field of designing efficient sort algorithms for massive parallel systems. Furthermore in [6] a bitonic sort algorithm is introduced that is able to sort input fields that's length is not a power of two efficiently, in-place and memory saving.

III. Differences between CPU and GPU

In general, CPUs are categorized into SISD and MIMD systems and GPUs are of type SIMD, according to the classification of Flynn [7]. The major differences between the CPU and the GPU are the number of cores, the focus of execution, clock speeds and the memory organization.

While the CPU was designed to optimize the instruction level focusing on sequential execution, the GPU could be seen as massive parallel co-processor optimized for data throughput. This allows GPU manufacturers to remove core control logic and cache memory and use the resources to

DOI: 10.7763/IJIEE.2015.V5.531

provide more simplified cores. Fig. 1 shows that there exist more ALUs (Arithmetic Logic Units) in a GPU than in a CPU and that control logic and caches were shrunk to a minimum. This enables the GPU to be highly performant when homogenous data should be processed [1].

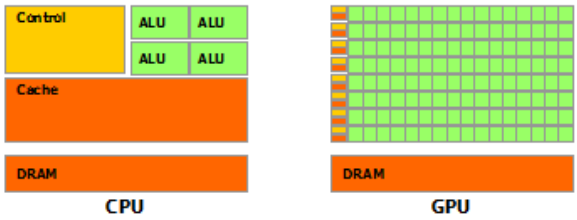

Fig. 1. Differences in the CPU and GPU architecture [8].

GPUs consist of several SMs (Streaming multiprocessors). Those are able to process hundreds of thread concurrently. Each of these SMs is divided into a high number (NVIDIA Fermi architecture: 32 compute cores for each of the 16 SMs) of compute-cores. Each of this cores consist of a FPU (Floating Point Unit) and a fully pipelined integer ALU [9].

Another difference is the memory model. All SMs share a global memory. Furthermore each SM provides a memory region where the registers of the compute-cores are placed and a shared memory region that is accessible from each compute-core within a SM very fast. Within a SM the cache is shared amongst all compute-cores [9].

Registers are the fastest way to access data but the memory region reserved for them is highly limited. If no more on-chip memory for saving registers is available local variables are stored in the global memory. Shared memory is an on-chip memory, which makes accessing it very fast, too. Accessing the global memory is slowest. It takes the GPU 200 – 800 cycles, depending on the hardware, accessing data that is located there [8], [9].

IV. Programming Models

TABLE I: Similarities of CUDA and OpenCL [1], [3]

OpenCL	CUDA
Common Identifiers	
Kernel	Kernel
Host Program	Host Program
NDRange (index space)	Grid
Work Group	Block
Work Item	Thread
Function Identifiers	
__kernel	__global__
No Identifier necessary	__device__
Variable Identifiers	
__constant	__constant__
__global	__device__
__local	__shared__

As already mentioned, recent GPU architectures allow the usage of the massive parallel system for solving problems that are not of graphical matter but for general purpose computation. To utilize these systems there exist several programming models like CUDA, OpenCL, DirectCompute, C++ AMP or OpenGL.

We focused on CUDA and OpenCL because those are the portable ones. The other models are either based on Microsoft's DirectX or are optimized for graphical matters.

While CUDA is designed to work with NVIDIA hardware only, OpenCL is an open standard that supports multiple platforms including FPGAs (Field-Programmable Gate Arrays), DSPs (Digital Signal Processors), multicore CPUs and accelerator-cards. But in general they share the basic ideas like the platform model, the memory abstraction or the execution model. Even the syntax is quite similar. This makes porting from CUDA to OpenCL relatively straightforward. You can find most syntactically differences located in Table I [1], [11].

A. Differences of CPU and GPU Programming

When programming multicore CPUs it is possible to execute different instructions on each core at different data at the same time. This is not possible when programming a GPU. On GPUs it is only possible to execute the same instruction on all cores, indeed on different data [1], [10].

To take advantage of the parallelism the code is usually split into two parts, the kernel, which is executed on the GPU and the host-code, that is executed on the CPU. Invoking the kernel is model-dependent, but similar. While it is good practice in CUDA C to compile the kernel at compile-time, it is mandatory to make use of a just-in-time compiler when using OpenCL [1], [10].

When writing programs for a GPU the first thing you have to do is to select an execution platform. Then your host code will have to allocate memory on the device where the data is copied to. After invoking the kernel you can copy back your results to the host's memory or process some further calculations on them. To obtain more speedup and utilize the different memory regions of the GPU there exist some keywords to place your data there. Table II provides the mapping between CUDA and OpenCL memory regions.

TABLE II: CUDA and OpenCL Memory Mapping [1]

OpenCL	CUDA
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Local Memory

V. Test Systems and Test Procedure

In this section the different test systems are presented as well as the test procedure and underlying test details.

A. Hardware Setup

The following test systems were available and used to determine the results. The first one has an Intel i5 core that is equipped with an internal on-chip GPU and a middle class graphics card. The second system consists of high-end components but the used CPU does not provide an on-chip GPU.

Test system 1:

1) Intel® Core™ i5-3570K
2) ASRock Z77 Extreme4 Mainboard
3) Corsair 8 GiB DDR3-1333 CL9
4) Crucial M4 SSD 128GB, SATA 6Gb/s
5) NVIDIA GeForce GTX 460
6) Windows 7 Professional 64-bit, Service Pack 1
7) NVIDIA Driver version 314.22

Test system 2:

1) Intel® Core™ i7-3820
2) Asus Rampage IV Formula
3) Corsair 16 GiB DDR3-1866
4) Samsung 840 Pro SSD 256GB
5) NVIDIA GeForce GTX 680
6) Windows 8 Professional 64-bit
7) NVIDIA Driver Version 320.49

The NVIDIA driver versions differ because of erroneous behavior on test system 1.

B. Program Flow

The different algorithms are compared by their runtime. The tests are performed as black-box tests. Time measurement starts when input data is ready to be processed and stops when the result calculation has finished. Due to different compiling strategies of CUDA and OpenCL the compilation and device selection process are not taken into account. This should provide a fair runtime comparison because this procedure has only to be done once. Additionally all tests make use of the same program flow. First, a hardware platform is selected. After that the input data is prepared before starting time measurement and executing the operation. After that time measurement is stopped and the results are verified against their validity. All tests use the same set of input data. Kernel-code is only syntactically adopted.

For each test iteration there exist two cancelation points for reliable time measurement. The first cancelation point is if a test run consumes more than 20 seconds of CPU time and the second one is a maximum of 2^{20} iterations to prevent an overflow of the iteration counter. This means that each test run stops when at least one cancelation point has been reached.

All tests have been compiled using Microsoft Visual Studio 2010 with installed Service Pack 1. Only the 32-bit release version, compiled with the */O2* flag (*Maximize Speed*), was tested. Intrinsics have been deactivated for CPU version.

CUDA code was compiled using NVIDIA CUDA 5.0 SDK and NVIDIA Nsight Visual Studio Edition Version 3.0 (Build 3.0.0.13123). The default compiler options were set, including the */O2 – Maximize Speed* flag.

OpenCL code was compiled using the NVIDIA driver with the version number mentioned. If no further notes are present, the default options without any additional compiler flags were used to compile the code.

VI. Matrix Multiplication

The matrix multiplication is a distinguished example to demonstrate the power of SIMD systems. Every single result can be calculated without any dependencies. Every test multiplies square-matrices filled with random values of type *float*. The matrix height and width grow by a multiple of 16 which equals the used block-size.

A. CPU

The CPU implementation uses a straightforward sequential matrix multiplication algorithm. Therefore three nested loops are used to calculate the result.

B. GPU

The straight forward way to multiply two matrices on a GPU is that every threads calculates exactly one result as shown in Fig. 2. This is not very scalable because of the limited amount of threads per block.

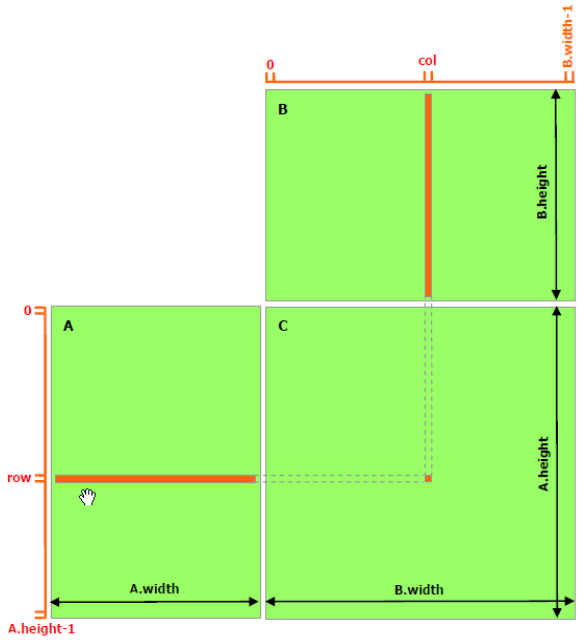

Fig. 2. Matrix multiplication without using shared memory. Every thread calculates one result [8].

But due to the fact that speedup is gained by making use of the different memory locations another algorithm was implemented [8].

As illustrated in Fig. 3 this algorithm divides the matrices into *BLOCK_SIZE* parts that are loaded into the blocks shared memory which is fast accessible for all threads within a block. This provides a single access to the slower global memory region where a consecutive memory region is loaded to a fast accessible memory region. This method is known as *strip mining*.

That is the algorithm we used for the tests.

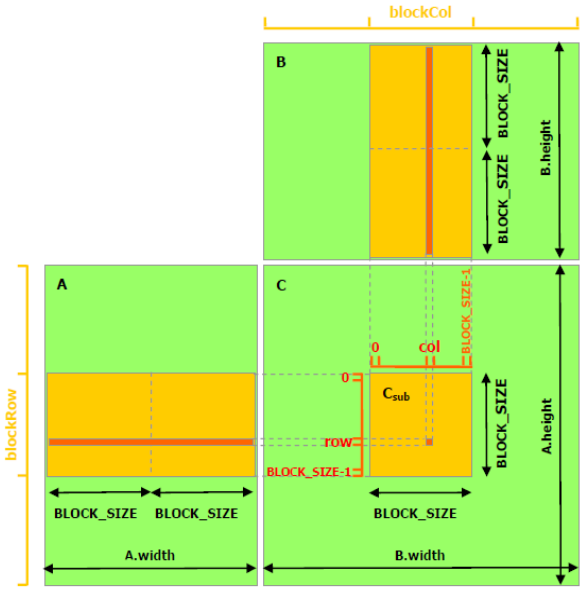

Fig. 3. Matrix multiplication using shared memory. Every block calculates a part of the result [8].

VII. Sorting

Sorting is an operation that is assumed to be available in almost every high level programming language. But when one must sort a huge amount of data it is sometimes a time consuming operation. This could be accelerated using the GPU. This test compares the runtime of sorting a given input of random unsigned integers. Every test processes the same input data to provide repeatable results. Before tests are started, files, containing the input data for every test, are produced as well as the input data is restored after every iteration. The tests are processing input vectors with a length that is a power of two.

A. CPU

The CPU implementation uses the std::sort algorithm that is shipped with the STL implementation of Microsofts Visual Studio 2010.

B. GPU

The GPU implementation uses a bitonic sort network to sort the input vector. As shown in [4] and [5], there exist more efficient sort algorithms but they are not in-place. So we decided to implement an in-place sorting algorithm that could easily be adopted to sort an input sequence whose length is not a power of two, see [6].

1) Bitonic sequence

A sequence is called sorted when the elements contained are monotonic non-increasing (or non-decreasing). A bitonic sequence consists of two sorted sequences, a monotonic increasing and a monotonic decreasing one. Therefore a sorted sequence is also bitonic when its decreasing (or increasing) part is null [6].

2) Sort algorithm

Fig. 4 shows a bitonic sort network with 16 inputs. The input passes the network from the left to the right. The arrows are comparators and the arrow points at the larger figure.

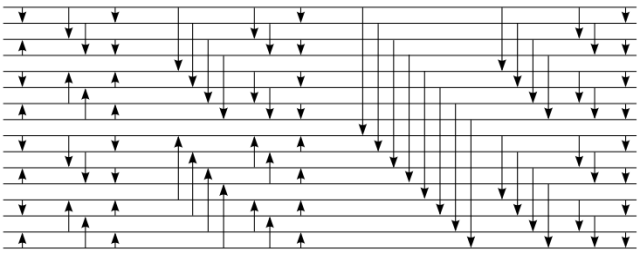

Fig. 4. Bitonic sort network.

When implementing this sort network directly as shown in Fig. 4, one thread for each input element is created and elements are getting compared twice. One time in the thread of the first element and again in the second element's thread where they are almost ordered correctly. So we used a slightly optimized algorithm. We only create one thread for two elements that should be compared. This thread loads the two values that should be processed, compares them and stores them back. So we get rid of unnecessary comparisons and only need to enqueue half of the kernels.

We fixed the number of threads (CUDA) respectively the workgroup size (OpenCL) to provide reproducible results.

VIII. Runtime Comparison and Analysis

In this section the test results for each test introduced in the former section are presented and discussed.

A. Matrix Multiplication

Fig. 5 illustrates the runtimes of test system 1. As expected, the plain CPU algorithm scales with its runtime complexity.

Because of the CPU-intensity of this operation the runtimes of CPU and GPU are almost equal when multiplying square matrices with 48 columns and rows. When multiplying square matrices with 64 columns and rows the runtime of the CPU has already doubled, compared to the GPU runtime. This is shown in a detailed view as in Fig. 6. This is not really surprising because the plain CPU algorithm has cubic runtime complexity. The GPU is still able to compute the results in parallel. Increasing data also relativize the time amount of copying data from the host to the device and from the device to the host.

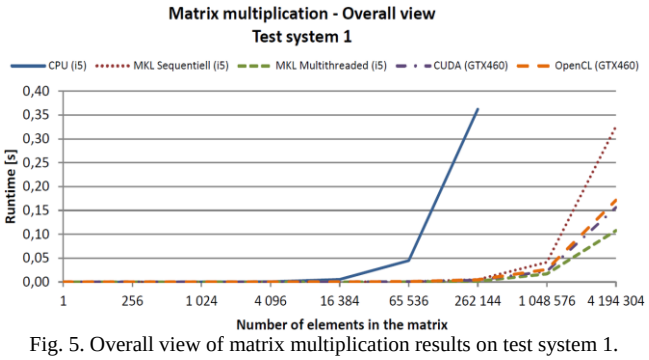

Fig. 5. Overall view of matrix multiplication results on test system 1.

This time amount of copying data also causes the higher runtimes of the GPU algorithms at smaller input sequences.

To show that there already exist some efficient implementations and algorithms on a CPU, we implemented two additional tests that make use of Intel's MKL (Math Kernel Library) in Version 11 Update 5, which is a highly optimized math library. One test uses the single-threaded and the other one the multi-threaded MKL variant. As could be seen in Fig. 5 the multi-threaded variant of Intel's MKL is able to beat the runtime of the GPU algorithm on test system 1. This is possible because this library is designed to make use of the CPU's SIMD hardware. But one should keep in mind that this library is hardware dependent and not free of charge.

Both GPU tests achieve similar runtimes, but with increasing matrix sizes the CUDA version turns out to be slightly faster than the OpenCL one.

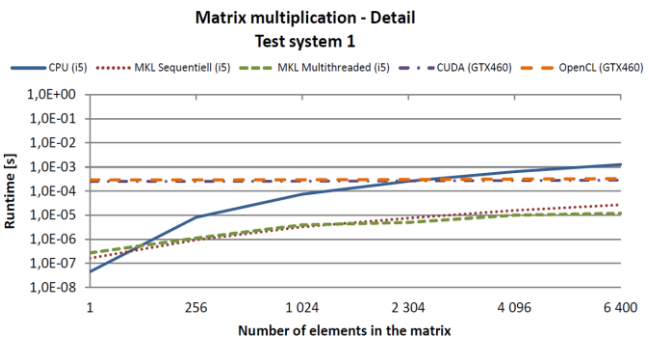

Fig. 6. Detail view of matrix multiplication results on test system 1.

Fig. 7 illustrates the runtimes of test system 2. The break-even point is at the same number of elements in a matrix as already pointed out for test system 1, as shown in the detailed view in Fig. 8.

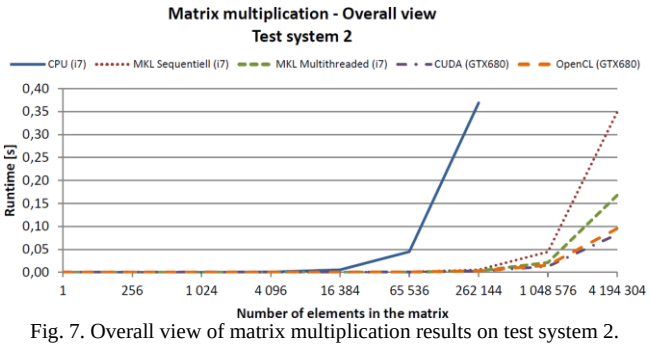

Fig. 7. Overall view of matrix multiplication results on test system 2.

Again, the CUDA version is slightly faster than the OpenCL one. Note that the runtimes are shorter than on test system 1 in general, except for MKL runtimes. The faster runtimes are explained by a more modern GPU in test system 2.

The differences between the MKL runtimes may be caused by the fact that the CPU in test system 1 is equipped with an internal GPU that is utilized.

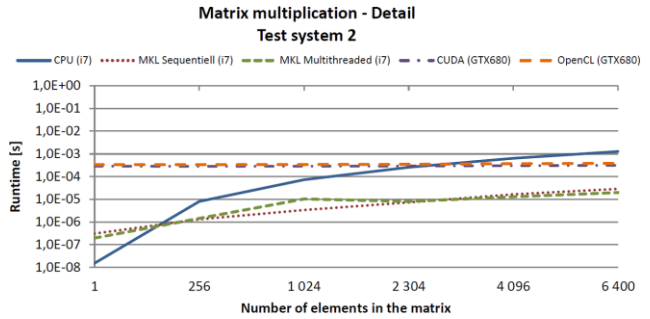

Fig. 8. Detail view of matrix multiplication results on test system 2.

Fig. 9 shows the runtime differences between the two CPUs. The runtimes are almost the same for the plain CPU implementation because the CPUs operate at almost identical clock speeds and the turbo-boost clock is even the same.

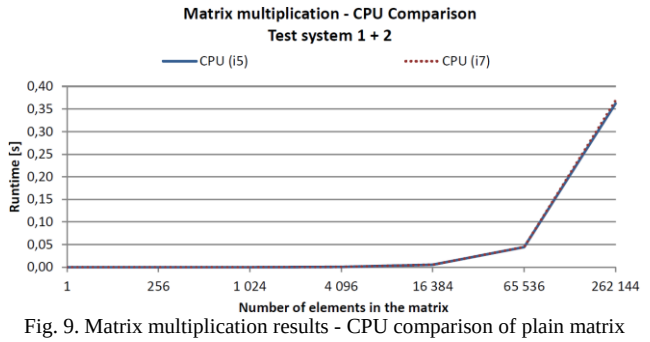

Fig. 9. Matrix multiplication results - CPU comparison of plain matrix multiplication algorithm on test system 1 and 2.

The GPU runtimes are compared in Fig. 10. The more modern GPU on test system 2 achieves an obvious improvement on runtime. As already mentioned, the CUDA version dominates on both systems.

B. Sorting

Fig. 11 shows the results of test system 1. It turns out that sorting big input sequences is obvious faster on a GPU than on a CPU.

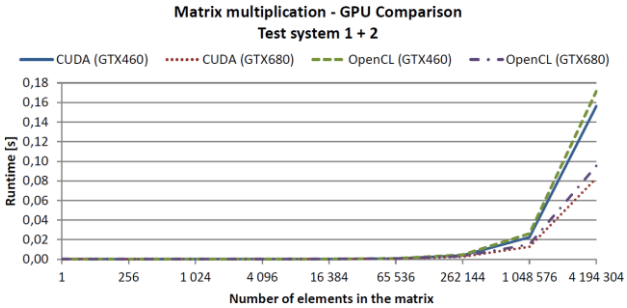

Fig. 10. Matrix multiplication results - GPU comparison of test system 1 and 2.

Even though the GPU algorithm is hardly optimized, Fig. 12 shows that already input sequences containing 2^{14} elements are sorted faster on a GPU than using the std::sort algorithm. The CPU runtime is already twice the GPU runtime when sorting input sequences containing 2^{19} elements.

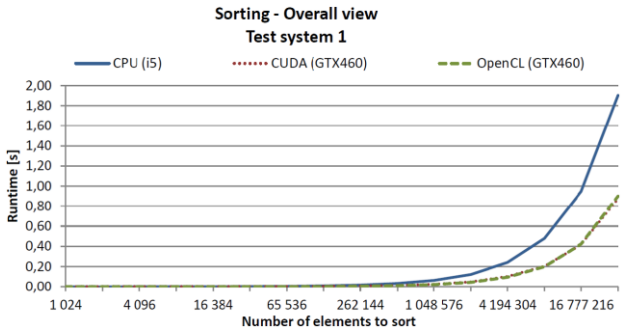

Fig. 11. Overall view of sorting results on test system 1.

The OpenCL algorithm is slower than the CUDA one until the input sequence contains more than 2^{23} elements. Only when the number of elements to sort is increasing further, CUDA wins again. This is influenced by the fixed threads per block (CUDA) respectively the workgroup size (OpenCL). However, the absolute runtime difference between CUDA and OpenCL is less than 3%.

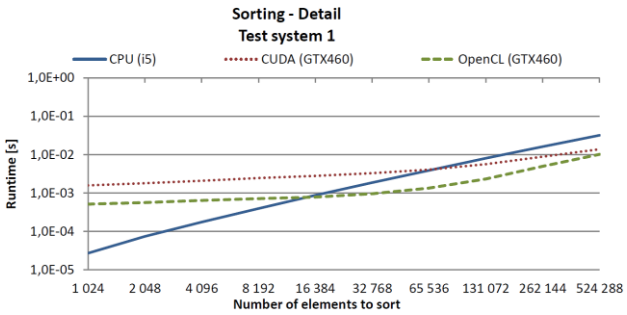

Fig. 12. Detailed view of sorting results on test system 1.

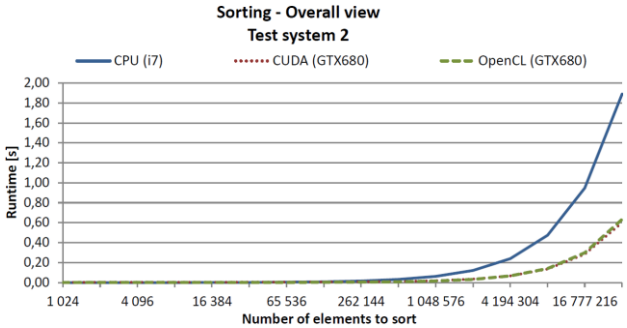

Fig. 13. Overall view of sorting results on test system 2.

On test system 2 the result is almost the same. As shown

in Fig. 13. The detail view, illustrated in Fig. 14, shows that the GPU algorithm is faster than its CPU equivalent for input sequences containing more than 2^{14} elements. The CPU version already needs twice the runtime of the GPU version for input sequences greater than 2^{18} elements.

It turns out that the OpenCL version is faster than the CUDA variant, when the input sequences contain 2^{14} to 2^{22} elements. Only when longer input sequences should be sorted, the CUDA variant dominates again.

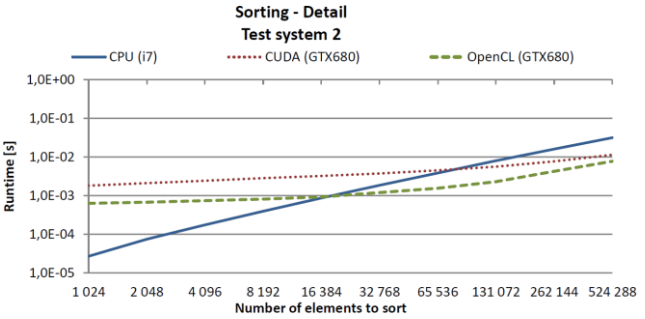

Fig. 14. Detailed view of sorting results on test system 2.

As with matrix multiplication, the CPU does not cause big changes in runtime, as shown in Fig. 15. Again it is influenced by almost similar clock speeds.

When comparing the GPU runtimes for test system 1 and 2, as illustrated in Fig. 16, one could see that the GPU of test system 2 sorts the given input sequences faster than the GPU of test system 1.

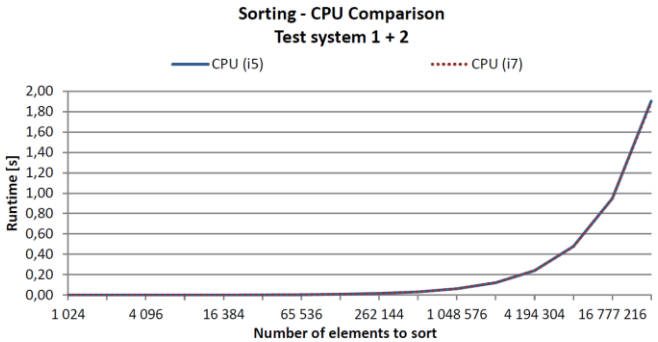

Fig. 15. Sorting results - CPU comparison of test system 1 and 2.

The more modern GPU of test system 2 speeds up the runtime of sorting, relatively to the runtime of the other GPU, by 31%. The absolute difference between CUDA and OpenCL is less than 4%.

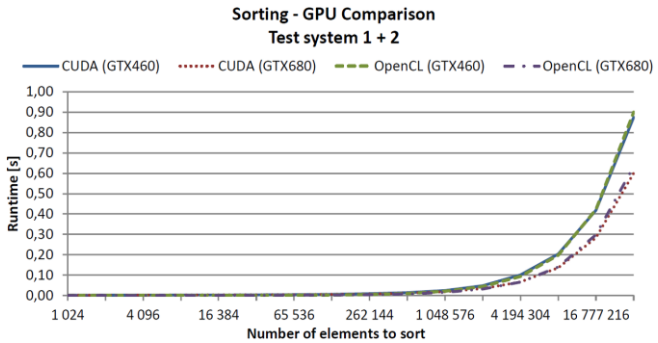

Fig. 16. Sorting results - GPU comparison of test system 1 and 2.

IX. Conclusions

Modern chip architectures promote the use of the GPU for solving problems of non-graphical matters. The unified compute cores and a freely programmable graphic pipeline enables everyone to take advantage of a cost-effective massive parallel system.

The tests show that the obtainable speedup is influenced by the problem size and data dependencies. The precision of the used data types should be taken into account, too. Because many GPUs are optimized for single floating point and integer operations.

Overall it could be said that tasks with data parallelism fit best to obtain as much speedup as possible. Often, nested loops can benefit from the parallelization as well.

Both programming models provide a similar interface. So it mostly depends on favor and the time reserved for a training period which model is best. We did not experience any remarkable performance differences that exclude a programming model from selection. CUDA performed slightly better when multiplying matrices and OpenCL achieved better runtimes in the sorting test.

CUDA offers a comprehensive develop environment including a wide range of useful libraries. The biggest disadvantage of the proprietary language is the hardware limitation, because CUDA code is only runnable on NVIDIA hardware.

OpenCL leaves some room for improving the develop environment but gaps are getting closed continuously. The biggest advantage is that OpenCL is a standard which is implemented by many manufacturers. This enables source portability with minor modifications.

References

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Burlington, MA: Elsevier Inc., 2010, ch. 1-3, pp. 11.

[2] H. Göbel and H. Siemund, *Einführung in Die Halbleitertechnik (Introduction to Semi-Conductor Technology)*, Springer-Verlag, 2008, ch. 1, pp. 11.

[3] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," presented at the International Conference on Parallel Processing (ICPP), Taipei, September 13-16, 2011.

[4] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters 21.02*, 2011, pp. 245-272.

[5] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. the 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE Computer Society*, Rome, 2009, pp. 1-10.

[6] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with CUDA based on bitonic sort," in *Proc. the 8th International Conference on Parallel Processing and Applied Mathematics: Part I (PPAM'09)*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski (Eds.), Springer-Verlag, 2010, pp. 403-410.

[7] M. J. Flynn, "Very high-speed computing systems," in *Proc. the IEEE 54.12*, 1966, pp. 1901-1909.

[8] NVIDIA Corporation. (February 2014). CUDA C Programming Guide, Design Guide PG-02829-001, Version 6.0. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[9] NVIDIA Corporation. (2009). Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Version 1.1. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[10] J. Sanders and E. Kandrot, *CUDA by Example An Introduction to General-Purpose GPU Programming*, Boston, MA: Pearson Education Inc., 2011, ch. 3, pp. 5.

[11] Khronos Group. (2013). *The OpenCL Specification*. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

Franz Wiesinger is a professor for software engineering and operating systems at the Department of Embedded Systems Engineering of the University of Applied Sciences Upper Austria.

His research interests include software architecture and design, performance optimization in embedded systems, compiler construction and code generation.

He is a member of the research group Embedded Systems at Hagenberg, Austria.

Florian Nairz is a postgraduate currently serving the master degree program embedded systems design at the University of Applied Sciences Upper Austria.

He is interested in parallel computing, system programming for various computer architectures, software architectures and operating systems.

Michael Bogner is a professor for software and systems engineering at the Department of Embedded Systems Engineering of the University of Applied Sciences Upper Austria.

His research interests include software modeling, parallel computing, system programming for various computer architectures, and cybernetics.

Prof. Bogner is a member of the research group Embedded Systems at Hagenberg, Austria.