Evolution and trends in GPU computing

Marko Misic
University of Belgrade
28 PUBLICATIONS **111** CITATIONS
SEE PROFILE

Đorđe Đurđević
University of Belgrade
20 PUBLICATIONS **87** CITATIONS
SEE PROFILE

Milo Tomasevic
University of Belgrade
70 PUBLICATIONS **1,007** CITATIONS
SEE PROFILE

Evolution and Trends in GPU Computing

Marko J. Mišić, Đorđe M. Đurđević, and Milo V. Tomašević
University of Belgrade/School of Electrical Engineering, Belgrade, Serbia
{marko.misic, djordje.djurdjevic, mvt}@etf.bg.ac.rs

Abstract – Central Processing Units (CPUs) are task-parallel, latency-oriented processors, while Graphics Processing Units (GPUs) are data-parallel, throughput oriented processors. Besides their traditional use as graphics coprocessors, the GPUs have been used in recent years for general purpose computations, too. Rapid development of graphics hardware led to an extensive use in both scientific and commercial applications. Numerous papers report high speedups in various domains. This paper presents an effort to bring GPU computing closer to programmers and wider community of users. GPU computing is explored through NVIDIA Compute Unified Device Architecture (CUDA) that is currently the most mature application programming interface (API) for general purpose computation on GPUs.

I. Introduction

From their beginning, graphics processing units (GPUs) have been used for specialized, intensive computations in the domain of computer graphics. Rendering of 3D graphics is a good example of highly intensive parallel computation. It includes computations for both geometry (vertices) and rasterization (pixels). Strong pressure from the fast growing gaming industry and the demand for fast, high definition graphics led to constant innovation and evolution of GPUs towards highly parallel, programmable processors with lots of GFLOPS and high throughput.

Central processing units (CPUs) are task-parallel, latency-oriented processors with transistors devoted to caching and sophisticated flow control. On the contrary, GPUs are data-parallel and throughput-oriented processors that hide relatively expensive global memory accesses with extensive use of parallel threads. Contemporary CPUs could be considered as *multicore* processors, since they need only several threads to fulfill their full capacity, while GPUs are *manycore* processors that need thousands of threads for their full use. The GPUs could be seen as coprocessing units to the CPUs, which are suitable for problems that exhibit high regularity and arithmetic intensity. Typically, the sequential parts of the program are executed on CPU, while the compute-intensive parts are offloaded to GPU to speedup the whole process.

The GPUs are found in many applications. At the very beginning, they were used mostly in academia, but since they brought significant speedups in several domains they were adopted from wider research community. GPUs are nowadays used in computational physics, computational chemistry, life sciences, signal processing, as well as finances, oil and gas exploration, etc.

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (III44009 and TR32039). The authors gratefully acknowledge the financial support.

The rest of the paper is organized as follows. The second section presents a short history of modern GPUs, and gives a further discussion on General-Purpose computation on Graphic Processing Units (GPGPU). The third section presents an overview of the Compute Unified Device Architecture (CUDA) that exploits GPU computing power for general purpose computation. The fourth section concentrates on the issues related to CUDA program execution and performance optimization. The fifth section of the paper gives a brief discussion on application domains suitable for GPU computing and presents some programming primitives and common algorithms. Sixth section reveals some of the experiences and lessons learned by the authors of this paper. The final section briefly concludes the paper and discusses the future trends of GPU computing.

II. Towards general purpose computation on GPUs

In the early stages of their existence, the GPUs have been fixed-function processors used primarily for 3D graphics rendering. The entire execution was built around graphics pipeline that was made configurable to some extent, but not programmable [1]. GPUs were specialized to render a screen picture through several steps, as shown in Figure 1. About the same time, 3D graphics application programming interfaces (APIs) became popular, notably OpenGL and Direct3D component of Microsoft DirectX. They exposed several programmable features to the programmers. The direction towards more programmable processors was driven by the fact that programmers wanted to create more complex visual effects on the screen than those provided by the fixed-function graphics hardware. In these early days of the GPU programming, the programmer had to send the short programs to the GPU, by calling API functions. These programs processed graphic primitives.

A. *Graphics hardware evolution*

In graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the positions of triangle vertices or generating pixel colors. This data independence, as the dominant application characteristics, is the key difference between the design assumption for GPUs and CPUs [1]. It allows a usage of a substantial hardware parallelism to process the bulks of data.

The first attempts to make the commercially available graphics processors more programmable exposed the vertex shader programmability, in NVIDIA GeForce 3 [2]. Further development provided programmability of the pixel shaders and full 32-bit floating point arithmetic

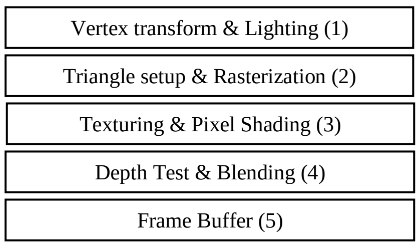
Vertex transform & Lighting (1)
Triangle setup & Rasterization (2)
Texturing & Pixel Shading (3)
Depth Test & Blending (4)
Frame Buffer (5)

Figure 1. Fixed-function graphics pipeline

support. The vertex and pixel shader programs essentially define the behavior of the stages 1 and 3 in the fixed graphics pipeline (Figure 1.), respectively. Also, as an important improvement, texture lookups were made possible from the vertex shader stage. This development could be seen as a trend to unify the functionalities of different graphics pipeline stages, since they essentially required the same processing capabilities. ATI Xenos GPU, made for the gaming console Xbox 360, was the first to allow the vertex and pixel shaders to execute on the same (non-specialized) processor. This unified approach proved to be successful and was consequently adopted in modern GPUs.

Also, graphics rendering made an impact to memory architecture of the GPUs. Typically, input data are processed only once during one stage and accessed in a strictly coherent way. Rendering algorithms usually access contiguous memory locations simultaneously which allows efficient memory bandwidth utilization. The GPU memory is organized in banks that allow simultaneous access to neighboring memory locations. The GPU memory architecture is built to emphasize bandwidth over latency, bringing more than 100 GB/s in recent GPUs. Whereas the CPU die area is dominated by cache memories, GPUs are dominated by floating-point datapath and fixed-function logic [1], thus efficiently hiding latency with massive parallelism.

On the other hand, the GPU memory organization had a very restricted memory access policy. Since the output data of the shader programs are pixels, they have predefined positions on the screen (and thus in the frame buffer). The output address of a pixel is always determined before the pixel is processed and the processor cannot change the output location of a pixel, so the pixel processors were incapable of memory "scatter" operations [3]. For several generations, GPUs lacked an efficient way to do the scatter operation, e.g. to write to a calculated (arbitrary) memory address.

B. Software support for general purpose computation

In the early stages of the GPU development, some researchers noticed the computational power of those processors and tried to use that power to solve general, compute-intensive problems. Because typical graphics scenes consist of primitives defined with fewer vertices than the number of pixels on modern displays, the programmable stage with the highest arithmetic rates in modern GPU pipeline is the pixel shader stage. Therefore, a typical GPU program used the pixel processor as the computation engine for the general-purpose computation on the GPU [3].

However, that was a challenging process, since it involved the usage of the graphics-oriented APIs, such as OpenGL or Direct3D, to launch a computation. Besides learning the basics of the API, one also needed to express a general-computation problem in the sense of graphics primitives and write the appropriate shader program. Again, output data needed to be converted from a pixel form, stored in a frame buffer, to data meaningful to the programmer. A very good example of such a programming model for the common problem of fluid simulation is found in [4]. Since OpenGL and Direct3D are primarily intended for graphics usage, no common features, like user data types or bitwise operators, were available. Also, attempts to do GPU programming were hindered by lack of dedicated tool chain – proper compiler support, debugging and profiling tools, and libraries.

Despite this quite awkward way of programming in the beginning, researchers demonstrated usefulness of GPUs for solving general-purpose problems in various domains. Papers [3] and [4] reported significant speedups in the domains of computational physics (fluid dynamics, game physics, etc.), computational biology (protein folding), and image processing (magnetic resonance imaging). Many more examples can be found in [3], including a survey of suitable programming techniques and data structures for GPU programming.

C. GPU computing era

The first fully unified GPU, NVIDIA GeForce 8800, brought an array of unified processors that execute logical graphics pipeline with the support of some fixed-function logic. Vertex shading, geometry shading, and pixel shading were executed on the same array of processors and used the same pool of resources. This design pursued much better workload balance, since different stages need different amount of resources during their execution that could be dynamically allocated.

The GeForce 8800 was also the first GPU to use scalar thread processors rather than vector processors [5]. It brought a wider instruction set to support C and other general-purpose languages, including integer and IEEE 754 floating-point arithmetic. Also, the GeForce 8800 eliminated memory access restrictions and provided load and store memory access instructions with byte addressing. It provided new hardware and instructions to support parallel computation, communication, and synchronization. Contemporary NVIDIA GPUs are programmed through simple C language extension and corresponding application programming interface or, more recently, through OpenCL API.

Similarly to NVIDIA, ATI (now AMD) also introduced unified shader architecture with Radeon 2000/3000 series of processors. Unlike NVIDIA that exposed GPU programmability through C language extension, AMD firstly exposed programmability through low-level programming interface, called Close-To-Metal that did not gain much success. Its successor, Stream SDK, included an open-source Brook+ language for GPGPU. In a more recent development, AMD switched to fully supported OpenCL, as a main platform to program their GPUs [6].

III. CUDA OVERVIEW

CUDA (Compute Unified Device Architecture) is a parallel computer architecture developed by NVIDIA, which came as a result of a need to simplify the use of GPUs for non-graphics applications. CUDA exposed a generic parallel programming model in a multithreaded environment, with support for synchronization, atomic operations, and eased memory access. Programmers do not need to use graphics API anymore to execute non-graphics computation.

A. Programming model

Essentially, the processing units of the GPU follow a single program multiple-data (SPMD) programming model, since many elements are processed in parallel using the same program. Processing elements (threads) can read data from a shared global memory (a “gather” operation) and write back to arbitrary locations in shared global memory (“scatter” operation). Generally, threads execute the same code that is inherent to SIMD execution model. However, since threads could follow different execution paths through branching in the same program, that leads to more general SPMD programming model.

Programs on the CUDA platform are executed in co-processing mode. Typically, the application consists of two parts. Sequential parts of the application are executed on the CPU (host) that is responsible for data management, memory transfers, and the GPU execution configuration. Compute-intensive, parallel parts of the code are executed on the GPU (device) as special functions (kernels) that are called by the CPU.

A kernel is executed by a large number of lightweight threads that run on the processing units called streaming multiprocessors (SMs). Every streaming multiprocessor is a SIMD processor, currently consisting of up to 32 scalar processors. Threads are organized in blocks executed on a single multiprocessor, and kernel execution is organized as a grid of thread blocks, as shown in Figure 2. Thread blocks executed on the same SM share the available resources, such as registers and shared memory. Depending on the commercial target niche, modern GPUs have from 2 up to 32 streaming multiprocessors. Due to this organization of thread execution, a kernel scales very well across any number of parallel processors. Efficient threading management in GPUs allows applications to expose a much larger amount of parallelism than available through hardware execution resources, with little or no penalty [1]. A compiled CUDA program executes on any size GPU automatically using more parallelism on GPUs with more processor cores and threads [5].

The number of threads in a block, and the number of blocks in a grid are specified through execution configuration of the kernel. Both block and grid could be multidimensional (1D, 2D and 3D) to support different memory access patterns. Every thread has a unique thread index within a block, and every block has a unique identifier within a grid. Threads can access those identifiers and dimensionality through special, built-in variables. Typically, every thread uses its own indices to make branching decisions, and to simplify memory addressing, when processing multidimensional data.

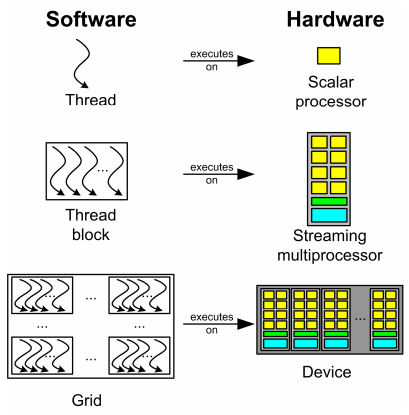

Figure 2. CUDA execution model

CUDA provides a way to synchronize threads in the same block using the barrier synchronization primitive. However, threads in different thread blocks cannot cooperate, since they could be executed on different streaming multiprocessors. Global synchronization is possible only through repeated kernel calls, which is one of the inherent limitations of manycore architectures.

B. Memory hierarchy

CUDA offers specific memory hierarchy to support fast, parallel execution. Threads have access to registers, local memory, shared memory, constant memory, texture memory, and global memory. All threads have access to slow, global device memory (DRAM). However, access to global memory is slow (approx. 200 cycles), in comparison to other memories, so other smaller memories could be used for temporary storage and reuse of data.. Every thread has an exclusive access to given (allocated) registers and local memory. Although private to the thread, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory [7]. Threads in a block could share data through per-block shared memory. Shared memory could be seen as a user-managed cache (scratchpad) memory as it could be accessed very fast (3-4 cycles). The programmer is responsible to bring the data to and move it from the shared memory. Also, threads can access constant memory, that is read-only and cached, and texture memory using special hardware units.

IV. PERFORMANCE CONSIDERATIONS

Although it is not difficult to write a correct CUDA program that can run on any CUDA device, performance can vary greatly depending on the resource constraints of the particular device architecture. So, performance concerned programmers still need to be aware of them to make a good use of a contemporary hardware.

A. Thread scheduling and execution

Logically, threads are executed in thread batches, called thread blocks. Physically, however, every thread block is executed on a SM in 32-thread units called warps. Threads in a block are partitioned into warps based on their indices. Consecutive threads could be considered to reside in the same warp. Threads in a block can execute in any order relatively to each other, so the synchronization is needed to avoid race conditions.

Warps are used as an execution optimization technique to hide latency of the long-latency operations, such as global memory accesses or floating-point operations. Every SM implements zero-overhead warp scheduling policy. The selection of warps ready for execution does not impose any additional time, so the waiting time of warp instructions is hidden by executing instructions from other warps.

All threads in a warp execute the same instruction when selected for execution. This SIMD style of execution works well when all threads in a warp execute the same path. But, since GPUs allow threads to follow different execution paths, a problem arises when threads within a warp diverge. In those situations, the hardware sequentially executes both control paths for all threads in a warp and thus creates a performance penalty. The programmer is responsible to structure the code in a way to avoid incoherent branching. A good example of how *if-then-else* construct is executed on SM can be found in [1], and a good survey on flow control mechanisms used in GPUs in [3].

B. Memory access

Global memory access is slow (approx. 200 cycles) in comparison with access to other memories in the hierarchy. But, since GPU applications process vast bulks of data, global memory accesses should be carefully considered to reach the peak bandwidth available on the GPUs. Also, to achieve high bandwidth utilization, the GPU memory is organized in banks. On NVIDIA Tesla GPU architecture, the memory is divided in 16 banks. Each bank can serve one memory request per cycle. To allow simultaneous accesses, consecutive memory locations are located in consecutive memory banks. Multiple simultaneous accesses to the same bank result in a bank conflict and conflicting accesses are then serialized.

When accessing global memory, peak bandwidth utilization is achieved when all threads in a half of the warp access consecutive memory locations. Memory accesses are then combined into a single memory transaction and data are delivered at high rates. This technique is called memory coalescing. Essentially, programmers should organize memory accesses in a way that follows those favorable patterns.

C. Shared memory

In order to achieve high performance on the GPUs, many computations should be done between two global memory accesses. To achieve better reuse of data and improve arithmetic intensity, threads can cooperate through per block shared memory. Usually, data set is partitioned into subsets that fit into the shared memory. Each thread block loads the subset from the global memory to shared memory and then performs computation on the given elements. Threads use shared memory for temporary storage and cooperation. Each thread block then writes the results from shared memory to global memory. Loading and storing of data should be done in a coherent way to exploit memory coalescing. The above described technique is often referred to as “tiling” technique for data processing on the GPUs.

D. Resource limitations

Every streaming multiprocessor in the GPU has a limited set of execution resources such as registers, thread block slots, thread slots, and shared memory. These resources are dynamically partitioned among threads during the execution. One can run a smaller number of threads that require many registers each or a large number of threads that require few registers. Dynamic partitioning of the resources gives more flexibility to compilers and programmers, but can also result in underutilization of resources, and thus performance degradation. More one this topic can be found in [1] and [7].

E. Streams

Beside the massive data parallelism through the kernel execution, CUDA offers one more way to overlap tasks. CUDA exposes the concept of streams which presents a kind of a task-parallelism between CPU and the GPU [8]. Stream represents a queue of GPU operations such as memory copies and kernel launches that get executed in a specific order. Streams can help to accelerate GPU applications by overlapping kernel execution and memory copies between the CPU and the GPU. More on concurrent execution between CPU and GPU can be found in [8] and [9].

V. APPLICATION DOMAINS

The GPU performance and computational power offer a lot for particular classes of applications. A very good survey of characteristics that application needs to satisfy to be successfully mapped for GPU execution could be found in [4].

First, parallelism should be substantial. The GPU utilization depends on a high degree of parallelism in the workload. The GPU also needs a substantial workload to hide memory latency, so computational requirements should be large. Thus, high throughput is more important than latency in GPU systems. Number of numeric computations performed per memory transaction should be an order of magnitude higher in order to preserve arithmetic intensity. On the other hand, execution divergence should be avoided as much as possible. Since threads are run in batches, if threads within a batch diverge, there is a penalty in prolonged execution. Finally, bandwidth utilization has to be sustainable. The GPUs have high peak bandwidth utilization to and from their onboard memory. But, for applications, such as sorting, that have a low computation to bandwidth ratio, it is of great importance to maximize coherent memory accesses.

A. GPU primitives

GPU programming exposes a different programming model and paradigm than traditional, sequential programming. That requires the programming constructs familiar to parallel supercomputer users, but often new to ordinary programmers reared on sequential machines or loosely coupled clusters [4]. New primitives and data structures, suitable for GPU execution, gained interest from research community [10]. Also, manycore architecture of the contemporary GPUs revived several old ideas, such those that emphasize parallel prefix sum (scan) operation [11] and sorting networks [12]. Sorting operation is the well-studied operation on GPUs and it is discussed in section VI.A.

There are several proprietary and open-source libraries that implement common algorithms on the GPU. Those include FFT algorithms (cuFFT), BLAS library (cuBLAS), pseudorandom-number generation (cuRAND), and CUDPP (CUDA Data Parallel Primitives) library that implements primitives like scan, sorting, and reduction [13]. Also, several projects try to provide higher level interfaces for CUDA programming. Thrust project [14] provides high-level, object-oriented API for GPU programming. The GMAC project implements an asymmetric distributed shared memory model to be used by CUDA programs [15]. That model allows CPU code to access data hosted in GPU memory.

B. *GPU applications*

Numerous papers reported speedups in various application domains. Papers [3] and [4] give a fine survey on those applications. The GPU accelerated applications are used in computational physics, computational chemistry, life sciences, medical imagining, mathematics (linear algebra, differential equations solvers), computer vision, signal and image processing and many more.

Notable scientific applications that benefited from GPU acceleration are NAMD [16] and VMD [17] molecular dynamics software, developed at University of Illinois, Urbana-Champaign, AMBER molecular mechanics software [18], and Folding@home project (protein folding) at the Stanford University [19]. Also, numerous commercial software packages are accelerated using GPUs, such as MathWorks MATLAB, Adobe Photoshop, and many more.

VI. Case study

This section outlines the experiences of the authors of this paper with the GPU accelerated applications. It shortly describes sorting on the GPU, as one of the frequently used primitives in the GPU computing, and domino-tiling, the CUDA accelerated algorithm for conforming the triangular mesh generation..

A. *Sorting on the GPUs*

Sorting operations are the important parts in modern applications, since these operations are used to optimize the search and merge activities, to produce the human-readable form of data, etc. The GPU-based sorting implementations attracted the researchers with several papers reporting significant speedups in quicksort [20], merge sort and radix sort algorithms [21]. Sorting operations are bandwidth consuming and access data in irregular patterns, so they heavily depend on efficient shared memory utilization. The authors of this paper shared a similar experience with the GPU sorting algorithms as in [22] and [23], where they conducted a performance comparison of those algorithms in the controlled test environment. The results shown in those papers suggest that the GPU implementations of the common sorting algorithms are 3 to 5 times faster compared to the same algorithms implemented on the shared memory and message passing platforms.

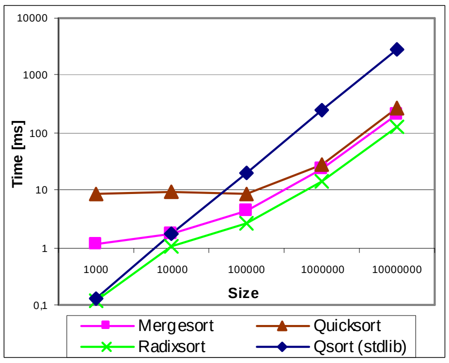

Figure 3. Performance comparison of the GPU sorting algorithms and CPU sequential (qsort) algorithm

Figure 3. shows an interesting comparison between the GPU sorting algorithms and standard C library sort function, *qsort*. Typically, the GPU implementations outperform the sequential *qsort*, except for arrays under 10k elements. The GPU kernel calls need some preparatory steps, so there is an amount of parallel overhead involved in the execution of every GPU algorithm. That time prevails in the overall execution time for smaller arrays, while it is amortized for larger arrays.

B. *Domino tiling*

One of the authors experienced the advantage in using CUDA to accelerate computations in the domain of computer graphics. The GPU accelerated fast conforming triangular mesh construction for rendering changeable height fields, dubbed Domino Tiling (DT) is presented in [24]. DT constructs a triangular mesh by deploying predefined mesh patterns (consisting of many triangles) so that the adjacent patterns seamlessly match, similarly to a domino game. One pattern is assigned to each height field patch (a small square portion of the height field), as an approximation of the fully accurate mesh. Mesh construction involves the visibility culling and level-of-details computation for each patch.

The speed comparison results are presented in Figure 4. The experiments were carried out on two different height fields (Puget Sound and Utah) for three different sizes, from 4K×4K points to 16K×16K points. The times given for CUDA implementation include the transfer time

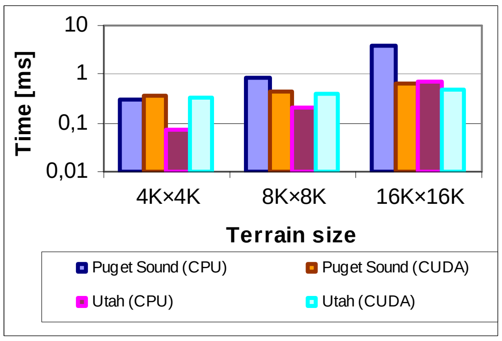

Figure 4. Performance comparison of the sequential (CPU) and CUDA (GPU) implementation of the Domino Tiling algorithm.

of the computation results, from GPU memory to CPU memory. For smaller terrain sizes, the CPU implementation is faster, especially in the case of the Utah dataset for the size 4K×4K. As the terrain size increases, the CUDA implementation takes the upper hand, and becomes significantly faster (Puget Sound dataset, 16K×16K). The reason for this effect is the inevitable overhead in invocation of several CUDA kernels and the final data transfer. The impact of the overhead diminishes as the height field grows larger. The speed comparison also shows that the CUDA implementation scales much better with the height field size than the CPU implementation. The CUDA implementation shows a sublinear increase in the computation time, while the computation time roughly quadruples in the CPU implementation.

VII. Conclusion and Future Trends

The paper presents an introduction to the general purpose GPU computing and focuses on CUDA, the most mature programming interface in the domain. Performance considerations and an overview of current application domains are given.

With the introduction of Fermi GPU architecture in 2010, NVIDIA has already taken steps to improve GPU performance and robustness. Fermi architecture relaxed some resource utilization constraints, improved flow control, introduced L2 caches, and unified GPU memory space [5]. Introduction of L2 caches largely relaxed memory coalescing constraints, since they provide hardware level support. Also, multi-GPU connection has been improved, with the support of direct memory transfers.

In the API domain, it is expected to see wider adoption of OpenCL in the programming community, since it is supported by major competitors in the field. OpenCL brings parallel computation to a broader set of devices, so there are still performance tuning considerations, but they are expected to be alleviated with maturity of OpenCL drivers for different architectures.

The fate of coprocessors in commodity computers, such as floating-point coprocessors, has been to end up into the chipset or onto the microprocessor [4]. Although the GPUs resisted that trend with constant performance improvements, potential performance increase could motivate such hybrid solutions. Hybrid CPU/GPU approaches could be seen in foreseeable future, with all major competitors announcing their new, hybrid architectures. Intel has already launched new SandyBridge architecture with integrated GPU and announced new MIC (Many Integrated Cores) architecture. AMD has the similar approach for mobile devices, called Fusion. Also, NVIDIA presented their new, research approach for heterogeneous computing, called Echelon [25].

In the future, it is expected that GPU architectures will evolve to further broaden application domains and to become more agile and better suited to handle arbitrary control flow and data access patterns. With many applications that benefited from GPU acceleration, GPU computing will become increasingly important. Since, sequential processors do not scale anymore at satisfactory rates, programmers need to use parallelism offered by the manycore architectures for an increased performance required to deliver more value to the end users.

References

[1] Kirk, D. B., Hwu, W. M., „Programming Massively Parallel Processors: A Hands-on Approach“, Morgan Kaufmann, 2010..

[2] „Vertex shaders: A Facet of the nfiniteFX Engine “, 2012., http://www.nvidia.com/object/feature_vertexshader.html

[3] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., Purcell, T. J., „A Survey of General-Purpose Computation on Graphics Hardware“, *Computer Graphics Forum*, vol. 26, No. 1, 2007., pp. 80–113

[4] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C., “GPU Computing”, *Proceedings of the IEEE*, vol. 96, No. 5., 2008., pp. 879–899

[5] Nickolls, J., Dally, W.J., „GPU Computing era“, *IEEE Micro*, vol. 30, No. 2, 2010., pp. 56–69

[6] „A Brief History of General Purpose (GPGPU) Computing“, AMD Corporation, 2009., http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/gpgpu-history.asp

[7] „Nvidia CUDA C Best Practices Guide“, version 4.0, NVIDIA Corporaton, 2011.

[8] Sanders, J., Kandrot, E., „CUDA by Example: An Introduction to General-Purpose GPU Programming“, Addison-Wesley, 2010.

[9] „NVIDIA CUDA C Programming Guide“, version 4.0, NVIDIA Corporaton, 2011.

[10] Cederman, D., „Concurrent Algorithms and Data Structures for Many-Core Processors“, Chalmers University of Technology, Göteborg, 2011.

[11] Blelloch, G. E., „Prefix Sums and Their Applications”, *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1990., pp. 35–60

[12] Batcher, K., E., „Sorting networks and their applications“, *Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968.

[13] Project „CUDPP – CUDA Data Parallel Primitives Library 2.0“, 2011., http://code.google.com/p/cudpp/

[14] Project „Thrust 1.5.1“, 2011., http://code.google.com/p/thrust/

[15] Gelado, I., Cabezas, J., Navarro, N., Stone, J. E., Patel, S., Hwu, W. W., „An asymmetric distributed shared memory model for heterogeneous parallel systems“, *ACM SIGARCH Computer Architecture News - ASPLOS '10*, vol. 38, No. 1, 2010.

[16] „NAMD - Scalable molecular dynamics“ project, 2011., http://www.ks.uiuc.edu/Research/namd/,

[17] „VMD - Visual molecular dynamics“ project, 2011., http://www.ks.uiuc.edu/Research/vmd/

[18] „AMBER - Assisted Model Building with Energy Refinement” project, 2011., http://ambermd.org/

[19] “Folding@home” project, 2011., http://folding.stanford.edu/

[20] Cederman, D., Tsigas, P., „A Practical Quicksort Algorithm for Graphics Processors“, *Technical Report 2008-01*, Chalmers University of Technology, 2008.

[21] Satish, N., Harris, M., Garland, M., „Designing Efficient Sorting Algorithms for Manycore GPUs“, *NVIDIA Technical Report NVR-2008-001*, 2008.

[22] Mišić, M., Tomašević, M., „Analysis of parallel sorting algorithms on different parallel platforms“, *Proceedings of the ACACES 2011*, Italy, 2011., pp. 95-98.

[23] Mišić, M., Tomašević, M., „Data Sorting Using Graphics Processing Units“, *Proceedings of 19th Telecommunications Forum - TELFOR 2011*, 2011., pp. 1446-1449. (in Serbian)

[24] Đurđević, Đ., Tartalja, I., "Domino tiling: a new method of real-time conforming mesh construction for rendering changeable height fields", *Journal of Computer Science and Tecnology*, vol. 26, No. 6, 2011., pp. 971-987.

[25] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., Glasco, D., „GPUs and the Future of Parallel Computing”, *IEEE Micro*, vol. 31, No. 5, 2011., pp. 7-17.