Computing and Informatics, Vol. 30, 2011, 1247–1257

GPGPU: HARDWARE/SOFTWARE CO-DESIGN FOR THE MASSES

Zoltán Ádám Mann

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
H-1117 Budapest, Magyar tudósok körútja 2, Hungary
e-mail: zoltan.mann@gmail.com

Communicated by Elena Gramatová

Abstract. With the recent development of high-performance graphical processing units (GPUs), capable of performing general-purpose computation (GPGPU: general-purpose computation on the GPU), a new platform is emerging. It consists of a central processing unit (CPU), which is very fast in sequential execution, and a GPU, which exhibits high degree of parallelism and thus very high performance on certain types of computations. Optimally leveraging the advantages of this platform is challenging in practice. We spotlight the analogy between GPGPU and hardware/software co-design (HSCD), a more mature design paradigm, to derive a design process for GPGPU. This process, with appropriate tool support and automation, will ease GPGPU design significantly. Identifying the challenges associated with establishing this process can serve as a roadmap for the future development of the GPGPU field.

Keywords: GPGPU, GPU computing, hardware/software co-design, design flow

1 INTRODUCTION

Graphical processing units (GPUs) are increasingly powerful and programmable. Programmability opens the possibility to use GPUs for general-purpose, i.e., non-graphical, computation that is normally carried out by the central processing unit (CPU). General-purpose computing on the GPU (GPGPU) is thus a fascinating opportunity to share the load between the CPU and the GPU in compute-intensive applications, such as matrix multiplication [15], collision detection [6], scientific simulation [9], ray tracing [26], electronic design verification [3], and genetic algorithms [8].

Despite numerous success stories, GPGPU is still far from becoming a mainstream technology. The current development of the field is driven by the competition of two GPU vendors: AMD and NVIDIA. Both vendors introduce new system generations with more features, higher performance, and improved programming facilities with a neck-breaking pace. With compatibility issues unaddressed and lacking a roadmap for the future development, the target platform for GPGPU practitioners is a moving target [25].

Our aim is to put the current development of GPGPU in a larger technological context by relating it to a now mature technology, hardware/software co-design (HSCD). We demonstrate the similarities between GPGPU and HSCD in their goals, scope, and inherent complexity. We identify concepts from HSCD that can be readily transferred to GPGPU, but also analyze basic differences that pose special challenges in the transfer of ideas. Through the analogy with HSCD, we can provide a technologically sound roadmap for the future development of GPGPU, centered around a proposed GPGPU design flow. The presented roadmap can help practitioners anticipate what is coming their way; it can help researchers in deriving the scientific challenges that need to be tackled; and it can help GPU vendors in defining focus areas for future technical innovation.

Most previous papers on GPGPU report on the experiences of implementing some specific programs on the GPU. We feel the need to complement these reports with a more abstract view. Thus, in contrast to most previous work, this paper is intentionally pitched at a higher level of abstraction.

2 GPGPU: HISTORY AND STATE OF THE ART

For many years, researchers have found it tempting to “mis-use” the resources of the GPU for general-purpose computation. In the 1990s, several isolated attempts were made that can now be seen as the precursors of modern GPGPU [16, 22, 11, 12]. The main characteristic of these approaches was the direct use of graphics APIs. That is, they had to formulate in graphical terms (such as vertices and textures) the non-graphical problems that they wanted to solve. These attempts were more of a black art than engineering practice, but they proved the general feasibility of GPGPU.

In subsequent years, GPUs became more and more programmable through shading languages of increasing level of abstraction, like the OpenGL Shading Language. The first programming languages for GPUs, in which the programmer could express general computation without using graphics terms, were introduced in 2004 [2, 21]. Since then, further programming languages with a C-like syntax have been proposed [28, 20]. The two big GPU vendors have also come up with their own programming environments: AMD offers CTM (Close To The Metal) for low-level and CAL (Compute Abstraction Layer) for high-level access, whereas NVIDIA provides the CUDA (Compute Unified Device Architecture) environment.

To allow for the special characteristics of GPUs, current GPGPU programming platforms are based on a stream-computing paradigm. A program is composed of *kernels*, each kernel processing one or more input data *streams* to create an output data stream. Technically, data streams are read from/written to the graphics card's onboard memory.

The strength of GPUs is the number of instructions executed per second, much more than latency, i.e., the time to process one instruction. This is in contrast with traditional CPUs, which offer lower latency at the cost of fewer instructions per second. GPUs are optimized for running the same sequence of instructions on a large number of data items in parallel. This is known as the SPMD (Single Program Multiple Data) paradigm. With today's GPGPU platforms, it is possible for the program to take different paths for different inputs, but this incurs substantial overhead [25]. GPUs excel when the application provides sufficient data-parallelism to leverage the hardware's high degree of parallelism.

Some parts of a program[1] can be more efficiently carried out by the GPU, whereas others may be more suitable for a traditional, CPU-based implementation. Typically, compute-intensive tasks with low variance in control flow are good candidates for a GPU implementation; for the other tasks, the CPU implementation may be more suitable. Therefore, it is often beneficial to partition the application between the two available processors. This also implies that communication between the CPU and the GPU is necessary. Technically, this involves data transfer between the system's main memory and video memory. This may lead to a significant time penalty, which can have a negative impact on the performance of the whole system [29]. Minimization of this overhead is one more aspect to consider when partitioning the tasks between CPU and GPU.

For specific problem domains, first attempts to automate the partitioning between GPU and CPU have already been proposed [13, 24].

3 THE HSCD PERSPECTIVE

In the early 1990s, hardware/software co-design (HSCD) emerged as an approach to combine the advantages of fast but expensive special-purpose hardware with cheap but slow software-based solutions [5, 7, 18, 4, 23, 30]. The idea is to combine, in a single design, both special-purpose circuits and a general-purpose processor, on which the appropriate software can run. HSCD targets mainly embedded systems with strict constraints on performance, area, and energy consumption. By carefully partitioning tasks between hardware and software, an appropriate trade-off between the conflicting requirements can be found. During partitioning, it has to be taken into account which tasks are more suitable for a hardware implementation and which ones for a software implementation. Moreover, the communication overhead between hardware and software also has to be taken into account [14, 17, 19, 1].

[1] The program parts of interest will be called *tasks* in the rest of the paper. Tasks can be of different granularity, see later.

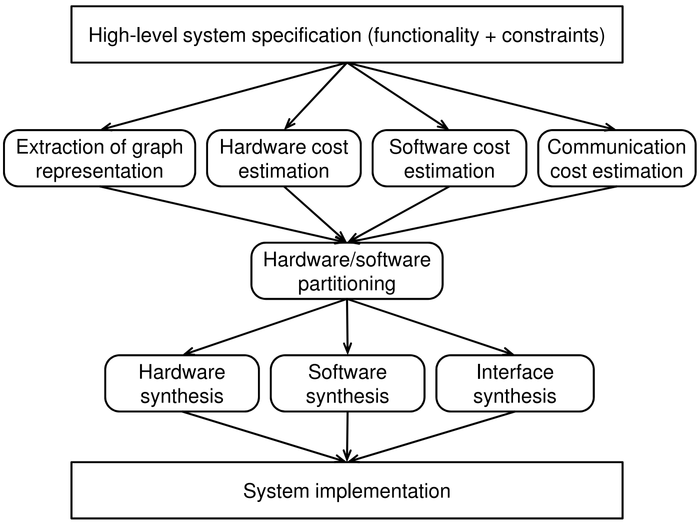
High-level system specification (functionality + constraints)
Extraction of graph representation
Hardware cost estimation
Software cost estimation
Communication cost estimation
Hardware/software partitioning
Hardware synthesis
Software synthesis
Interface synthesis
System implementation

Fig. 1. Hardware/software co-design process

A possible HSCD design flow is depicted in Figure 1. It starts with a high-level specification, defining the *functionality* of the system. At this stage, non-functional requirements such as performance or area constraints are specified in a *declarative* way, without explicitly defining *how* the system should be implemented. The implementation is determined in subsequent steps. For partitioning, a graph representation is extracted from the functional specification, in which nodes represent tasks and edges represent dependency and communication between tasks. Each node is associated with one or more hardware cost (e.g. required area on the chip), which is only relevant if the given task is implemented in hardware. Likewise, each node is associated with one or more software cost (e.g. running time on the processor). Finally, each edge is associated with a communication cost, relating to the amount of communication between the given pair of tasks. These costs are determined based on the high-level system specification, using static analysis, high-level synthesis, profiling etc. Based on the extracted information, the system is partitioned by mapping each task to either hardware or software. Afterwards, the hardware and software implementation of the tasks are synthesized based on the partitioning decisions. Communication interfaces are synthesized to allow the seamless communication between tasks in hardware and tasks in software. The result is a fully synthesized and optimized system implementing the original specification and obeying the given constraints.

As a result of intensive research in the last 15 years, now all these steps are well understood. In particular, design automation support exists for all the steps. Today, HSCD is a mature and mainstream approach in embedded system design.

HSCD	GPGPU	Explanation
Software implementation of a task	Implementation of the task on the CPU	In both cases, the task is implemented in a traditional, CPU-based way
Hardware implementation of a task	Implementation of the task on the GPU	In both cases, the task is implemented on a non-CPU-like platform, offering a high degree of parallelism, thus potentially – if the task is suitable – offering a significant speedup over the CPU-based implementation
Hardware/software communication overhead	CPU/GPU communication overhead	In both cases, the communication between the two parts of the system incurs a non-negligible time penalty
Hardware/software partitioning	Deciding which tasks to assign to the CPU and which ones to the GPU	In both cases, it is a crucial decision which tasks to implement in which part of the system. The decision must take into account the different cost factors associated with the implementation options for each task, as well as the incurred communication overhead
Hardware area constraint	Available parallel hardware resources of the GPU	In both cases, the speed-up achievable through moving tasks from the CPU to the other implementation option is constrained by the available resources
Performance optimization of the whole hardware/software system	Performance optimization of the whole GPU/CPU system	In both cases, it is the overall system performance that needs to be optimized

Table 1. Analogies between hardware/software co-design and GPGPU

4 CONSEQUENCES FOR GPGPU

As can be seen from Table 1, HSCD and GPGPU are analogous, at least at a sufficiently high level of abstraction. This makes it possible to transfer some of the results of the HSCD community to the field of GPGPU:

- Typically, most of the runtime of a program is spent in some relatively small loops. These loops, or parts of them, are the candidates for acceleration in hardware/GPU.
- With appropriate algorithms (e.g., genetic algorithms, integer linear programming etc.), partitioning can be automated. A carefully implemented partitioning

algorithm can outperform the human expert concerning the quality of the found solution, and needs only a fraction of the time that human experts need to tackle the problem.

- Just like with hardware/software partitioning, the decision of what to put into the GPU can be made at different levels of granularity. One extreme is to decide for each instruction, on which processor it should run. Alternatively, the partitioning decisions can be made on the level of basic blocks, functions, objects, components etc., and a mixture of these granularity levels is also possible. The chosen level of granularity has significant impact on the effectiveness but also on the hardness of partitioning: fine-grain partitioning decisions might result in the best resource utilization; however, they might also lead to high overhead in terms of communication and management and increase the search space for partitioning. For more details on how the optimal granularity can be found in the context of HSCD, see [10] and references therein.
- For both HSCD and GPGPU, it is beneficial if there is little variance in the control flow of the application. This way, partitioning decisions can be made a priori with high confidence. Otherwise, dynamic re-partitioning might be required on the fly. This is possible in GPGPU, just like with some reconfigurable HSCD platforms [27].
- The "glue code" responsible for establishing the technical context of the GPU-CPU collaboration – communication, scheduling, and memory management issues – can be implemented in an application-independent manner, thus fostering reuse. Such code can be made available as a library and linked to the application.
- The proliferation of HSCD was significantly supported by the enhancement of the capabilities of the underlying hardware platforms, e.g. field-programmable gate arrays and synthesizable processor cores. Currently, a similar trend can be observed in the transformation of GPU data paths (e.g., architectural transformation, programmability of the GPU pipeline, support for double-precision computation etc.), see [25].

Of course, no analogy is perfect. In order to fully understand the implications and limitations of this analogy, it is vital to also look at the differences between HSCD and GPGPU, as they are the challenges in transferring ideas from HSCD to GPGPU:

- In HSCD, moving a function from software to hardware will usually accelerate it. So, if the goal is to optimize overall performance and if there were no other constraints, then the optimum would be to implement everything in hardware. It is due to constraints on cost and/or size that this solution is not applicable. In contrast, in GPGPU it is not necessarily optimal to put everything into the GPU. In fact, moving something from the CPU to the GPU might not accelerate it at all, if the given function does not fit well to the highly data parallel nature of the GPU.

- In HSCD, partitioning is based on *functionality*: some parts of the code are mapped to hardware, others to software. This is necessary in GPGPU as well. However, since GPGPU is inherently a platform for highly data parallel applications, partitioning *data* between CPU and GPU is also vital. An example of data partitioning is the GPGPU implementation of 2D FFT calculation as described in [24]. 2D FFT calculation involves a high number of 1D FFT calculations on different columns and rows of the 2D matrix. Performance can be optimized by appropriately distributing the 1D FFT calculations between the GPU and the CPU. That is, the functionality carried out by the two processors is the same, but the data are partitioned. In general, partitioning both functionality and data is possible and should be exploited.
- A more technical, yet important difference is the maturity of synthesis tools. When HSCD appeared, tools were already available for hardware synthesis (high-level synthesis, silicon compilation, design simulation and verification techniques etc.) that could be built upon. Today, synthesizing code for GPU is still in its infantry.

Based on the presented analogy, the idea of a *GPU/CPU co-design process* emerges, as shown in Figure 2. The structure of the process is largely the same as the previously presented HSCD process, with differences in the details.

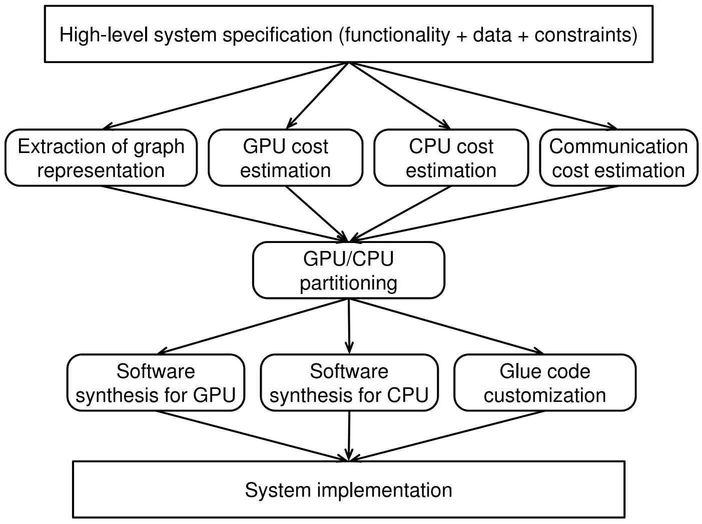
High-level system specification (functionality + data + constraints)
Extraction of graph representation
GPU cost estimation
CPU cost estimation
Communication cost estimation
GPU/CPU partitioning
Software synthesis for GPU
Software synthesis for CPU
Glue code customization
System implementation

Fig. 2. GPU/CPU co-design process

The system to implement is given in form of a high-level specification, describing functionality and the structure and amount of data to be processed. At this stage, it is not decided yet which tasks will be implemented on the GPU and which ones on the CPU. These decisions will be made later in the partitioning step. For partitioning, the necessary data have to be extracted: dependency and data flow between the tasks, as well as the associated costs. For each task, it has to be determined how long it would take to execute that task on the GPU and on the CPU, respectively, as well as how much it would add to the load of the processors. For each pair of communicating tasks, the amount of transferred data is determined. For these cost estimations, static analysis and simulation runs can be used. Then, based on the extracted data, the partitioning decisions can be made, by mapping each task to either the GPU or the CPU. Partitioning must also include preliminary scheduling information about the schedule of the tasks and data transfers. Afterwards, each task is synthesized to make it executable on the GPU or CPU, respectively, according to the partitioning decision. In order to allow for communication between tasks on the GPU and the CPU, the appropriate communication routines are also parameterized and linked to the system. The result is an optimized and fully synthesized system consisting of tasks on the GPU and the CPU.

In a couple of years, the steps of this process could also be automated using smart optimization techniques. This would bring tremendous benefits for GPGPU:

- Today, adapting a program for GPGPU and optimizing which tasks should be implemented on the GPU are carried out manually. This is a long and tedious process, which could be significantly shortened with the appropriate tools.
- Automated synthesis of GPU code and communication routines would add a lot to the quality of the produced code by reducing the probability of inserting errors.
- Automated partitioning can be superior to human judgement, especially in the case of fine-grained (e.g., instruction-level) partitioning that implies a huge problem space.
- The high-level system specification lets designers focus on application design instead of low-level implementation details, thus boosting design productivity.
- Using a high-level functional system specification enhances portability between different GPUs, even of different vendors.

In order to achieve these benefits, a number of challenges need to be addressed:

- Development of a high-level specification language, from which both CPU code and GPU code can be synthesized.
- Development of appropriate analysis and simulation techniques to quickly and accurately predict the characteristics of the implementation of a task on the GPU. In particular, performance and processor load are of interest.
- Development of appropriate GPU/CPU partitioning algorithms to find the best trade-off between the two implementation options.
- Defining the best granularity for partitioning between GPU and CPU.
- Development of synthesis techniques to automatically convert a task from a high-level specification to an optimized GPU-based implementation.

5 CONCLUSIONS

We presented analogies between GPGPU and HSCD, in order to derive, based on a HSCD process, a possible future design process for GPGPU applications. We identified the main steps of such a GPGPU design process, the advantages associated with the approach, and the challenges that need to be tackled to make this reality.

Of course, the GPGPU process that we presented is certainly not the only possibility: there will be differences in scope, aims, and realization details. However, we believe that the presented process is a good basis to interpret GPGPU progress, providing a sound roadmap for the future development for practitioners, researchers, and vendors alike. The next step will be to elaborate on the presented challenges, each of which will require a substantial amount of future research.

Acknowledgements

This work was partially supported by the Hungarian National Research Fund and by the National Office for Research and Technology (Grant Number OTKA 67651), and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

REFERENCES

[1] Arató, P.—Mann, Z. Á.—Orbán, A.: Algorithmic Aspects of Hardware/Software Partitioning. ACM Transactions on Design Automation of Electronic Systems, Vol. 10, 2005, No. 1, pp. 136–156.

[2] Buck, I.—Foley, T.—Horn, D.—Sugerman, J.—Fatahalian, K.—Houston, M.—Hanrahan, P.: Brook for GPUs: Stream Computing on Graphics Hardware. In ACM SIGGRAPH '04 International Conference on Computer Graphics and Interactive Techniques 2004, pp. 777–786.

[3] Chatterjee, D.—Deorio, A.—Bertacco, V.: GCS: High-Performance Gatelevel Simulation with GP-GPUs. In Design Automation and Test in Europe (DATE) 2009.

[4] Dick, R. P.—Jha, N. K.: MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Co-Synthesis of Hierarchical Heterogeneous Distributed Embedded Systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, 1998, No. 10, pp. 920–935.

[5] Ernst, R.—Henkel, J.—Benner, T.: Hardware/Software Cosynthesis for Microcontrollers. IEEE Design and Test of Computers, Vol. 10, 1993, No. 4, pp. 64–75.

[6] Govindaraju, N. K.—Redon, S.—Lin, M. C.—Manocha, D.: CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware. In HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware 2003, pp. 25–32.

[7] Gupta, R. K.—de Micheli, R.: Hardware-Software Cosynthesis for Digital Systems. IEEE Design & Test of Computers, Vol. 10, 1993, No. 3, pp. 29–41.

[8] Harding, S.—Banzhaf, W.: Fast Genetic Programming on GPUs. In Proceedings of the 10th European Conference on Genetic Programming 2007, pp. 90–101.

[9] Harris, M. J.—Baxter, W. V.—Scheuermann, T.—Lastra, A.: Simulation of Cloud Dynamics on Graphics Hardware. In HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware 2003, pp. 92–101.

[10] Henkel, J.—Ernst, R.: An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity That Is Driven by High-Level Estimation Techniques. IEEE Transaction on VLSI Systems, Vol. 9, 2001, No. 2, pp. 273–289.

[11] Hoff, K. E.—Keyser, J.—Lin, M.—Manocha, D.—Culver, T.: Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques 1999, pp. 277–286.

[12] Hopf, M.—Ertl, T.: Accelerating 3D Convolution Using Graphics Hardware (Case Study). In VIS '99: Proceedings of the conference on Visualization '99, pp. 471–474.

[13] Joselli, M.—Zamith, M.—Clua, E.—Montenegro, A.—Conci, A.—Leal-Toledo, R.—Valente, L.—Feijo, B.-0-D'Ornellas, M.—Pozzer, C.: Automatic Dynamic Task Distribution Between CPU and GPU for Real-Time Systems. In 11th IEEE International Conference on Computational Science and Engineering 2008, pp. 48–55.

[14] Kalavade, A.—Lee, E. A.: The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling and Implementation-Bin Selection. Design Automation for Embedded Systems, Vol. 2, 1997, No. 2, pp. 125–164.

[15] Scott Larsen, E.—McAllister, D.: Fast Matrix Multiplies Using Graphics Hardware. In Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, p. 55.

[16] Lengyel, J.—Reichert, M.—Donald, B. R.—Greenberg, D. P.: Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In SIGGRAPH '90: Proceedings of the 17th annual conference on computer graphics and interactive techniques 1990, pp. 327–335.

[17] Lopez-Vallejo, M.—Lopez, J. C.: On the Hardware-Software Partitioning Problem: System Modeling and Partitioning Techniques. ACM Transactions on Design Automation of Electronic Systems, Vol. 8, 2003, No. 3, pp. 269–297.

[18] Madsen, J.—Grode, J.—Knudsen, P. V.—Petersen, M. E.—Haxthausen, A.: LYCOS: The Lyngby Co-Synthesis System. Design Automation for Embedded Systems, Vol. 2, 1997, No. 2, pp. 195–236.

[19] Mann, Z. Á.: Partitioning Algorithms for Hardware/Software Co-Design. Ph. D. thesis, Budapest University of Technology and Economics 2004.

[20] McCool, M.: Data-Parallel Programming on the Cell BE and the GPU Using the Rapid-Mind Development Platform. In GSPx Multicore Application Conference 2006.

[21] McCool, M.—Du Troit, S.—Popa, T.—Chan, B.—Moule, K.: Shader Algebra. ACM Transactions on Graphics, Vol. 23, 2004, No. 3, pp. 787–795, 2004.

[22] Myszkowski, K.—Okunev, O. G.—Kunii, T. L.: Fast Collision Detection Between Complex Solids Using Rasterizing Graphics Hardware. Computer, Vol. 11, 1995, No. 9, pp. 497–511.

[23] Niemann, R.: Hardware/Software Co-Design for Data Flow Dominated Embedded Systems. Kluwer Academic Publishers 1998.

[24] Ogata, Y.—Endo, T.—Maruyama, N.—Matsuoka, S.: An Efficient, Model-Based CPUGPU Heterogeneous FFT Library. In IEEE International Symposium on Parallel and Distributed Processing 2008, pp. 1–10.

[25] Owens, J. D.—Houston, M.—Luebke, D.—Green, S.—Stone, J. E.—Phillips, J. C.: GPU Computing. Proceedings of the IEEE, Vol. 96, 2008, No. 5, pp. 879–899.

[26] Purcell, T. J.—Buck, I.—Mark, W. R.—Hanrahan, P.: Ray Tracing on Programmable Graphics Hardware. In SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, p. 268.

[27] Stitt, G.—Lysecky, R.—Vahid, F.: Dynamic Hardware/Software Partitioning: A first approach. In Proceedings of DAC 2003.

[28] Tarditi, D.—Puri, S.—Oglesby, J.: Accelerator: Using Data-Parallelism to Program GPUs for General-Purpose Use. In 12th International Conference on Architectural Support for Programming Languages and Operating Systems 2006, pp. 325–335.

[29] Wang, L.—Huang, Y.—Chen, X.—Zhang, C.: Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment. In International Conference on Computer Science and Information Technology 2008, pp. 228–232.

[30] Wolf, W.: A Decade of Hardware/Software Codesign. Computer, Vol. 36, 2003, No. 4, pp. 38–43.

Zoltán Ádám Mann received the M. Sc. and Ph. D. degrees in computer science from Budapest University of Technology and Economics in 2001 and 2005, respectively. He also received an M. Sc. degree in mathematics from Eötvös Loránd University in 2004. Currently, he is an Associate Professor at the Department of Computer Science and Information Theory, Budapest University of Technology and Economics. In addition, he works as Principal Consultant at Capgemini. His main research interest is in the application of combinatorial algorithms to computer engineering problems.