CUDA BASED ENHANCED DIFFERENTIAL EVOLUTION: A COMPUTATIONAL ANALYSIS

Donald Davendra and Jan Gaura,
Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VSB-Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba
Czech Republic.
Email: {donald.davendra,jan.gaura}@vsb.cz

Magdalena Bialic-Davendra,
Tomas Bata University in Zlin,
Faculty of Management and Economics,
Nam T.G. Masaryka 5555, 760 01 Zlin,
Czech Republic.
Email: bialic@fame.utb.cz

Roman Senkerik,
Tomas Bata University in Zlin,
Faculty of Applied Informatics,
Nam T.G. Masaryka 5555, 760 01 Zlin,
Czech Republic.
Email: senkerik@fai.utb.cz

KEYWORDS

Differential evolution, flowshop scheduling, CUDA

ABSTRACT

General purpose graphic programming unit (GPGPU) programming is a novel approach for solving parallel variable independent problems. The graphic processor core (GPU) gives the possibility to use multiple blocks, each of which contains hundreds of threads. Each of these threads can be visualized as a core onto itself, and tasks can be simultaneously sent to all the threads for parallel evaluations. This research explores the advantages of applying a evolutionary algorithm (EA) on the GPU in terms of computational speedups. Enhanced Differential Evolution (EDE) is applied to the generic permutative flowshop scheduling (PFSS) problem both using the central processing unit (CPU) and the GPU, and the results in terms of execution time is compared.

INTRODUCTION

During the later part of the past decade, a novel trend emerged where programmers started using the Graphics Processing Unit (GPU) for programming not graphic applications which usually was in the preview of the Central Processing Unit (CPU). The reasoning behind such a move was the possibility to achieving speedups of magnitude compared to optimized CPU implementations.

GPU's have evolved into fast, highly multi-threaded processors, with hundreds of cores and thousands of concurrent threads. These threads which can be invoked simultaneously, provide an excellent platform for parallel execution. A GPU is optimal when a problem has to be executed many times, can be isolated as a function and works independently on different data.

One of the most challenging and computational demanding problems in engineering are the NP-Hard problems. These problems are computationally intractable, and often require the use of optimization algorithms. This research attempts to solve the challenging flowshop scheduling (FSS) problem using a novel Enhanced Differential Evolution (EDE) algorithm utilizing GPU programming.

One of the most widespread programming architectures is the Compute Unified Device Architecture (CUDA) of Nvidia (NVIDIA, 2012). A number of research has been conducted on GPU programming involving evolutionary algorithms and these two architectures. Tabu Search has been used for the evaluating the FSS problem using CUDA by Czapinski and Barnes (2011). Genetic Algorithms (GA) has been been used to solve the traveling salesman problem by Chen et al. (2011), whereas a parallel GA approach has been done by Pospichal et al. (2010). The particle swarm algorithm has also been modified to be used by CUDA Mussi et al. (2011). More interestingly Genetic Programming has also found a niche in GPU programming (Robilliard et al., 2009).

This research utilizes the Nvidia CUDA framework for GPU computation. The enhanced Differential Evolution (EDE) (Davendra and Onwubolu, 2009) is modified to the GPU framework and execution time for both the GPU and CPU variants are compared.

This paper follows the following structure. Section 1 outlines the CUDA framework and syntax. Section 2 describes Differential Evolution (DE) and the EDE algorithms. The problem attempted in this research; flow shop scheduling is given in Section 3. Section 4 describes the code design on the GPU, whereas the experimentation and analysis (Section 5) compares the obtained results. The paper is concluded in Section 6.

Proceedings 26th European Conference on Modelling and Simulation ©ECMS Klaus G. Troitzsch, Michael Möhring, Ulf Lotzmann (Editors)
ISBN: 978-0-9564944-4-3 / ISBN: 978-0-9564944-5-0 (CD)

1 CUDA

The Compute Unified Device Architecture (CUDA) is a propriety parallel computing architecture developed by Nvidia Corporation and released in November 2006. The main objective for this was to introduce general programming to the GPU, in the effort to scale up raw processing power.

The CUDA API (both low level and high level) provides a platform for accessing the Nvidia GPU for processing. This allowed a programmer to bypass the traditional OpenGL or Direct3D techniques which were needed to program the chips. Additionally, and most importantly, C language can now be used to program for CUDA, through the PathScale Open64 C compiler. This has essentially introduced CUDA for mainstream programmers (Sanders and E.Kandrot, 2010).

The general outline of CUDA is given in Fig 1. The CPU is able to communicate with the GPU using the PCIe bus, and therefore the communication speed is limited by the bus speed. A GPU itself has a number of attributes. The GPU is made up of a number of *blocks*. Each *block* is subsequently divided into a number of *threads*. Each block has its own *shared memory* and *registers*, which can be accessed by all the threads residing in that particular *block*. All *blocks* can access the *global memory* and the *shared memory* of the GPU. The minimum applicable amount of *blocks* available is 65,535, and each *block* has 512 *threads* each. CUDA processing cores are referred to as *kernels*, which are launched from the CPU. *Kernels* can be made up of any combinations of *blocks* and *threads*, depending on the application (Kirk and Hwu., 2010).

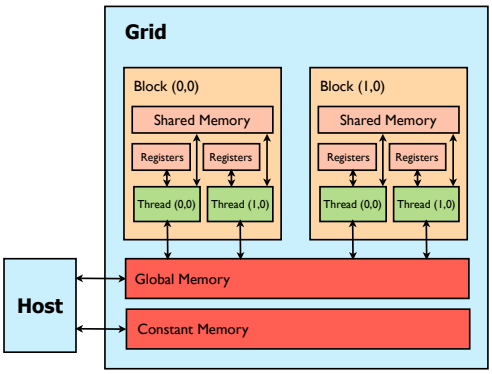

Figure 1: CUDA outline.

There are many types of memory, differing in size, visibility, access time and whether it is cached and writeable. Below is the description of each memory type, its capabilities and purpose.

1. *Global memory* is used for communication between host and device, therefore it is accessible from blocks (directly), and CPU (through API). It is the largest memory space available on GPU, but it is the slowest at the same time.

2. *Shared memory* is a very fast on-chip memory, available for both reading and writing, but only accessible by threads within the same block. Unfortunately, it is also small with a maximum of 16,384 bytes per block.

3. *Constant memory* is accessible as global memory, but it is cached in the case of a cache miss, a read operation takes the same time as reading from global memory, otherwise it is much faster.

4. *Registers* are the fastest on chip memory used for threads automatic variables. The number of 32-bit registers on each multiprocessor is limited up to 16,384, therefore if each thread requires too many registers, the number of blocks executed on each multiprocessor is reduced.

5. *Local memory* is a local per thread memory, used for large automatic variables (e.g. arrays or large structures) that will not fit in registers. It is not cached, and is as slow as global memory.

1.1 CUDA syntax

A brief outline of the CUDA syntax is presented. Memory allocation of dynamic variables is given as: `cudaMalloc((void**)&GPUvarible,(size)*sizeof(type));`, where GPUvariable is the name, type is the datatype and size refers to the total size being allocated.

Data is passed from the CPU to the kernels in the GPU using `cudaMemcpy(GPUvariable,CPUvariable,(size)*sizeof(type),#cudaMemcpyHostToDevice);`. This copies the data from the CPUvariable to the GPUvariable.

The kernel is launched using the following syntax `kernel<<<dim3 grid,dim3 block>>>(...)`, where `dim3` is the built in device variable. `dim3 gridDim` refers to the number of dimension of the grid in blocks whereas `dim3 blockDim` is the dimension of the blocks in threads. Within the kernel, the block index is given by `dim3 blockIdx` and the thread index is given by `dim3 threadIdx`.

After execution on the GPU kernels, the data is read back to the CPU as `cudaMemcpy(CPUvariable,GPUvariable,(size)*sizeof(type),cudaMemcpyDeviceToHost);`. This commands copies the data from the GPUvariable to the CPUvariable. The CUDA procedural outline is shown in Fig 2

2 DIFFERENTIAL EVOLUTION

Developed by Price and Storn (Price, 1999), Differential Evolution (DE) algorithm is a very robust and efficient approach to solve continuous optimization prob-

```
Memory Allocation
    int *GPUvariable
    cudaMalloc((void**)&GPUvariable,(size)*sizeof(type));

Kernel Parameter Passing
    cudaMemcpy(GPUvariable,CPUvaribale,(size)*sizeof(type),
                cudaMemcpyHostToDevice);

GPU kernel Allocation
    dim3 grid, block;

Execution
    kernal<<<dim3 grid,dim3 block>>>(GPUvariable);

Read Back
    cudaMemcpy(CPUvariable,GPUvaribale,(size)*sizeof(type),
                cudaMemcpyDeviceToHost);

Release Memory
    cudaFree(GPUvariable);
```

Figure 2: Generic CUDA template

lems. One of the core features of DE is that it uses a vector perturbation methodology for crossover.

Each solution is visualized as a vector in search space. A new vector is created by the combination of four unique vectors. A schematic of DE is given in Fig. 3.

1. Input: $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds: $\vec{x}^{(lo)}, \vec{x}^{(hi)}$.
2. While $G < G_{\max}$
   - 3. Mutate and recombine:
     - 3.1 $r_1, r_2, r_3, r_4 \in \{1,2,....,P_{size}\}$, randomly selected from each cluster
     - 3.2 $j_{rand} \in \{1,2,...,D\}$, randomly selected once each i
     - 3.3 $\forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{best,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G} - x_{j,r_3,G} - x_{j,r_4,G}) \\ \quad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} \quad \text{otherwise} \end{cases}$
   - 4. Select Criteria

   $\forall i \leq NP$
3. $G = G + 1$

Figure 3: DE selection

2.1 Enhanced Differential Evolution outline

Enhanced Differential Evolution (EDE) (Davendra and Onwubolu, 2007), heuristic is an extension of the canonical DE developed for the task of permutative based combinatorial optimization. The basic outline is given in Fig 4.

EDE operates on a permutative set of individuals. These individuals are transformed to a real number using the forward transformation. The DE strategy can then be applied to the real domain values. Once the DE crossover and mutation routines have finished, the resulting vector is transformed back into the permutative individual using the backward transformation. The new trial individual is then randomly repaired, and its objective function calculated. The local search is utilized when stagnation is detected in the population. The detailed description of the EDE approach is given in Davendra and Onwubolu (2009).

1. **Initial Phase**
   (a) *Population Generation*: An initial number of discrete trial solutions are generated for the initial population.
2. **Conversion**
   (a) *Forward transformation*: This conversion schema transforms the parent solution into the required continuous solution.
   (b) *DE Strategy*: The DE strategy transforms the parent solution into the child solution using its inbuilt crossover and mutation schemas.
   (c) *Backward Transformation*: This conversion schema transforms the continuous child solution into a discrete solution.
3. **Mutation**
   (a) *Relative Mutation Schema*: Formulates the child solution into the discrete solution of unique values.
4. ***Local Search**
   (a) *Local Search*: 2 Opt local search is used to explore the neighborhood of the solution.

Figure 4: EDE outline

3 PERMUTATIVE FLOWSHOP SCHEDULING PROBLEM

In many manufacturing and assembly facilities, a number of operations have to be done on every job. Often these operations have to be done on all the jobs in the same order implying the jobs have to follow the same route. The machines are assumed to be set up in series and the environment is referred to as a *flow shop* (Pinedo, 1995).

Flow Shop *Fm*: There are *m* machines in series. Each job has be processed in each one of the *m* machines. All the jobs have to follow the same route (i.e., they have to processed on Machine 1, and then on Machine 2, etc). After completing on one machine, a job joins the queue at the next machine. Usually all jobs are assumed to operate under the *First In First Out (FIFO)* discipline - that is a job cannot "pass" another while waiting in a queue. Under this effect the environment is refereed to as a *permutative* flow shop. the general syntax of this problem as described in the triplet format $\alpha|\beta|\gamma$, is given as

$$Fm\,|Perm\,|C_{\max}$$

The first field denotes the problem being solved, the second field the type of problem (in this case permutative) and the last field denotes the objective being under investigation, which is the makespan (total time taken to

complete the job).

Stating these problem descriptions more elaborately, the minimization of completion time (makespan) for a flow shop schedule is equivalent to minimizing the objective function $\Im$:

$$\Im = \sum_{j=1}^{n} C_{m,j} \tag{1}$$

s.t.

$$C_{i,j} = \max\left(C_{i-1,j}, C_{i,j-1}\right) + P_{i,j} \tag{2}$$

where, $C_{m,j}$ = the completion time of job *j*, $C_{i,j} = k$ (any given value), $C_{i,j} = \sum_{k=1}^{j} C_{1,k}$; $C_{i,j} = \sum_{k=1}^{j} C_{k,1}$ machine number, *j* job in sequence, $P_{i,j}$ processing time of job *j* on machine *i*. For a given sequence, the mean flow time, $MFT = \frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}$, while the condition for tardiness is $c_{m,j} > d_j$. The constraint of Equation 2 applies to these two problem descriptions.

The value of the makespan under a given permutation schedule can also be computed by determining the *critical path* in a directed graph corresponding to the schedule.

For a given sequence $j_1, .., j_n$, the graph is constructed as follows: For each operation of a specific job j_k on a specific machine *i*, there is a node (i, j_k) with the *processing time* for that job on that machine. Node (i, j_k), $i = 1, ..., m-1$ and $k = 1, ..., n-1$, has arcs going to nodes $(i+1, j_k)$ and (i, j_{k+1}). Nodes corresponding to machine *m* have only one outgoing arc, as do the nodes in job j_n. Node (m, j_n), has no outgoing arcs as it is the terminating node and the total weight of the path from first to last node is the makespan for that particular schedule (Pinedo, 1995). A schematic is given in Fig 5.

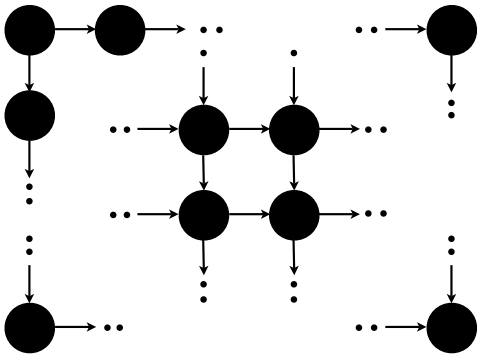

Figure 5: Directed graph representation for $Fm\,|Perm\,|C_{\max}$

The pseudocode of the routine as coded in the GPU kernel is given in Fig 6.

Algorithm for Flow shop scheduling on the GPU

Assume a problem of size n, and a schedule given as $X = \{x_1, .., x_n\}$. Assume the problem matrix as R, which is of size *n* by *m* (job and machine) and the solution array as Y of size *m*.

1. For $i = 1, 2, ..., n$ do the following:
   (a) For $j = 1, 2, ..., m$ do the following:
      i. **IF** *i* = 1
         A. $Y_i = \sum_{i=1}^{i-1} Y_i + R_{X_i,j}$
      ii. **ELSE**
         A. **IF** *i* = 1
            $Y_1 = Y_1 + R_{X_1,1}$
         B. **ELSE**
            $Y_j = \max(Y_j, Y_{j-1}) + R_{X_i,j}$
2. Output Y_m as the objective function.

Figure 6: Pseudocode for Flow shop scheduling

4 CODE DESIGN ON THE GPU

In an evolutionary algorithm, the most time and processor consuming task is the objective function calculation. This amounts to almost 80% of the execution time (Czapinski and Barnes, 2011). Therefore it is logical to utilize the GPU for this task, providing that it can be parallelized.

The major drawback for such a approach is the allocation of memory on the GPU. By rule, the GPU cannot allocate memory dynamically, therefore all dynamically allocated memory has to be assigned in the CPU. This makes all these memories as global memory, which is the slowest. Also, sufficient memory has to be allocated for the calculation itself.

This research uses the GPU only for objective function calculation. The outline is given in Fig 7. A number of parameters have to be passed to the GPU as given in Table 1. These include three individual items; Machine size (*m*), Job size (*n*) and Population size (*p*). The population, problem data and calculation matrix also have to be passed to the GPU. The calculation matrix is required for the calculation of the makespan.

As most of the data is assigned as global memory, and only population size of 100 is used for all simulations, the kernels allocated are to only blocks. All internal memory allocation (loop indexes etc) have been allocated as shared memory. At the first initialization, all the data is

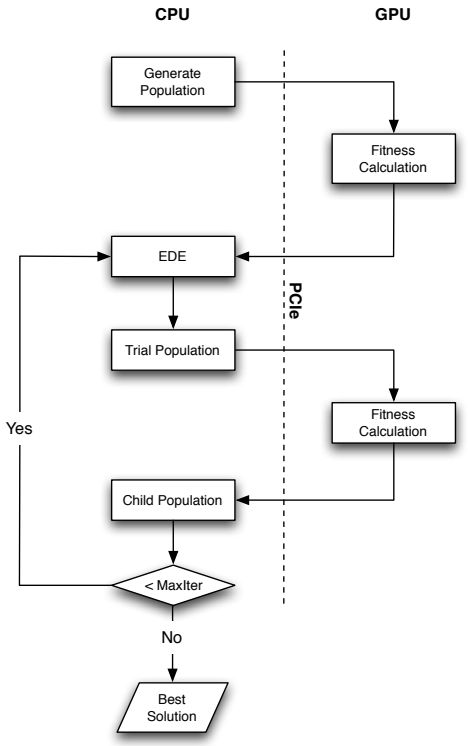

Figure 7: Conceptual outline.

Table 1: Parameters passed to the GPU

Parameters	**Type**	**Size**
Machine size (m)	int	sizeof (int)
Job size (n)	int	sizeof (int)
Population size (p)	int	sizeof (int)
Problem data	int	$(m \cdot n) \cdot$ sizeof (int)
Population	int	$(p \cdot n) \cdot$ sizeof (int)
Calculation matrix	int	$(p \cdot m) \cdot$ sizeof (int)
Objective function	int	$p \cdot$ sizeof (int)

passed to the GPU. After execution of the GPU, only the population and objective function matrix is returned to the CPU. Thereafter, within the generation loop, only the population and objective function matrix are passed to and from the GPU.

The experiment involves the embedding of the 2-opt local search on each trial solution. Therefore the complexity of this routine on each kernel is a minimum of $O\left(n^2\right)$.

5 EXPERIMENTATION

The main objective of this research is to validate the application of using a GPU for general purpose programming. In terms of performance measurement of any algorithm, the execution time is the key indicator, as all other measurement criteria are tied to it.

Table 2: Parameters passed to the GPU

Parameters	**Value**
Population	100
Generations	100
CR	0.9
F	0.5
Strategy	DE/best/2/bin

Table 3: Processing time for CUDA and CPU

Instance	**CUDA**	**CPU**	Δ_{avg}
20 x 5	**1.72**	26.84	1460.46
20 x 10	**2.53**	63.05	2392.09
20 x 20	**4.16**	111.85	2588.7
50 x 5	**19.87**	209.46	954.15
50 x 10	**28.87**	1483.19	5037.47
50 x 20	**50.20**	3483.48	6839.2
100 x 5	**151.31**	4600.52	2940.45
100 x 10	**246.75**	5988.96	2327.14
100 x 20	**387.26**	7545.46	1848.42
200 x 10	**2052.62**	15548.67	657.48
200 x 20	**3237.77**	25572.84	689.82
500 x 20	**8783.75**	44682.3	408.69
Average	**1247.56**	9109.69	2345.34

The experiment design was to apply the EDE code utilizing the local search on each fitness calculation, firstly on the CPU and then on the GPU and measure the time taken to complete the experimentation with fixed operating parameters.

The operating parameters for EDE is given in Table 2.

The operating system utilized for this experiment was a Nvidia Tesla C2050 graphics processing unit. It has 1 GPU core with 3072MB RAM memory, 575MHz core clock speed, version 2 of the CUDA core architecture and supports double precision. This hardware is part of the Media Research Lab (MRL) at the Technical University of Ostrava (MRL, 2012).

The experimentation results are given in Table 3. The average improvement Δ_{avg} of the GPU executed EDE over the CPU version is calculated as given in Equation 3.

$$\Delta_{avg} = \frac{(CPU - GPU) \times 100}{CPU} \tag{3}$$

From the results, it can be easily established that the CUDA based EDE is a faster executing algorithm, however the scale of improvement is quite substantial for medium and larger sized problems. For each problem set, there is a significant improvement for the GPU execution time. The total average improvement of the GPU is 2345.34 over the CPU. The average execution time is 1247.56 sec on the GPU compared to 9109.69 on the CPU. Therefore, we can conclude that the execution is

on a magnitude of eight times faster on the GPU. The results are displayed in Fig 8.

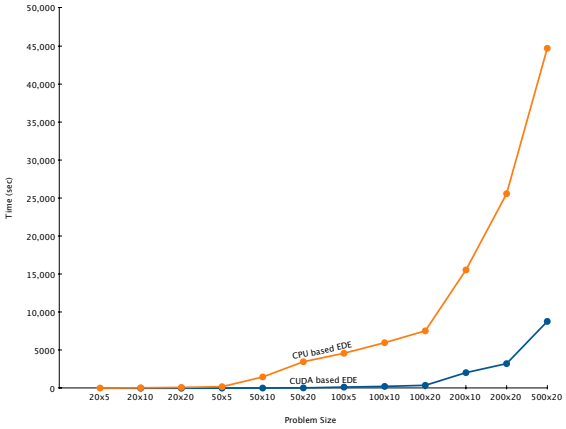

Figure 8: Graph of CUDA vs CPU processing time.

6 CONCLUSION

With the advent of faster processors, it becomes feasible to improve the structure of algorithms in order to harness this new power. With the application of GPGPU based applications gaining a foothold in computational analysis, this research aims to answer the question as to weather it is feasible to spend the necessary resources to convert algorithms to a GPU framework.

From the results obtained during this research, it is very clear that CUDA based EDE is a vast improvement over the CPU based variant in terms of execution time. The improvement is quite substantial, which in turn gives the scope of further optimization of the algorithm based on memory management.

The major fallibility of using the GPU is the time taken to transfer the data using the PCIe bus, which has a fixed bus speed. The next iteration of this research is the porting of the EDE routines to the GPU itself, thus minimizing the transfer time.

REFERENCES

Chen, S., Davis, S., Jiang, H., and Novobilski., A. (2011). Cuda-based genetic algorithm on traveling salesman problem. *Computer and Information Science*, 364:241–252.

Czapinski, M. and Barnes, S. (2011). Tabu search with two approaches to parallel flowshop evaluation on cuda platform. *Journal of Parallel and Distributed Computing*, 71:802–811.

Davendra, D. and Onwubolu, G. (2007). Flow shop scheduling using enhanced differential evolution. In *Proc.21 European Conference on Modeling and Simulation*, pages 259–264, Prague, Czech Rep.

Davendra, D. and Onwubolu, G. (2009). Forward/backward transformation approach. In Onwubolu, G. and D.Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*. Springer, Germany.

Kirk, D. and Hwu., W., editors (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann., New York.

MRL (2012). Media research lab. `http://mrl.cs.vsb.cz/`.

Mussi, L., Daolio, F., and S.Cagnoni (2011). Evaluation of parallel particle swarm optimization algorithms within the cudatm architecture. *Information Sciences*, 181:4642 – 4657.

NVIDIA (2012). Cuda webpage. `http://www.nvidia.com/object/cuda$_-$home$_-$new.html`.

Pinedo, M. (1995). *Scheduling: theory, algorithms and systems*. Prentice Hall, New Jersey.

Pospichal, P., Jaros, J., and Schwarz, J. (2010). Parallel genetic algorithm on the cuda architecture. In et.al, D. C. C., editor, *Applications of Evolutionary Computation: Lecture Notes in Computer Science*. Springer, Germany.

Price, K. (1999). An introduction to differential evolution. In Corne, D., Dorigo, M., and Glover, F., editors, *New ideas in Optimisation*. McGraw Hill, UK.

Robilliard, D., Marion-Poty, V., and Fonlupt, C. (2009). Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines.*, 10:447 – 471.

Sanders, S. and E.Kandrot (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, USA.

AUTHOR BIOGRAPHIES

DONALD DAVENDRA is an Assistant Professor of Computing Science at the Technical University of Ostrava. He has a Ph.D. in Technical Cybernetics from the Tomas Bata University in Zlin. His email address is `donald.davendra@vsb.cz`.

JAN GAURA is an Assistant Professor of Computing Science at the Technical University of Ostrava with a research background is in digital image processing. His email address is `jan.gaura@vsb.cz`.

MAGDALENA BIALIC-DAVENDRA is a postdoctoral researcher at the Center of Applied Economic Research at the Tomas Bata University in Zlin from where she has a Ph.D. in Finance. Her email address is `bialic@fame.utb.cz`.

ROMAN SENKERIK is an Assistant Professor of Informatics at the Faculty of Applied Informatics, Tomas Bata University in Zlin. His email address is `senkerik@fai.utb.cz`.