AUTOMATYKA • 2011 • Tom 15 • Zeszyt 3

Robert Nowotniak*, Jacek Kucharski*

GPU-Based Massively Parallel Implementation of Metaheuristic Algorithms

1. Introduction

In this paper, implementation of a state-of-the-art evolutionary algorithm, Quantum-Inspired Genetic Algorithm [8, 36] (QIGA), in massively parallel environment (Graphics Processing Units [20, 22, 23]) has been presented. Contrary to many recent papers [1, 4, 10, 28] concerning parallel implementation of evolutionary algorithms, in this paper a novel approach has been taken. QIGA algorithm has been implemented entirely as a computational kernel. Parallelization of the algorithm has been performed on two levels: In a block of threads, each thread transforms a separate individual or different gene; In each block, separate populations with same or different parameters are evolved. Finally, the computations have been distributed to eight GPU devices, and over 400x speedup has been gained in comparison to sequential implementation of the algorithm in ANSI C on one Intel Core i7 2.93 GHz CPU core. Correctness of the results has been verified in statistical analysis. The presented approach can be applied to experimentation with a broad class of metaheuristics.

In recent years, programmable Graphics Processing Units have evolved into massively parallel, multithreaded and many-core environments with tremendous computational power and high memory bandwidth [5, 7, 22]. One of the leading general-purpose parallel computing architectures nowadays is NVIDIA Compute Unified Device Architecture [19, 20] (CUDA) technology. CUDA-enabled GPUs have hundreds of cores that can concurrently run thousands of computing threads. NVidia CUDA technology has been already successfully applied in a vast number of different fields; For example, linear algebra [11, 31], image processing [6], scientific simulations [9, 27, 34], finance [12, 26] and others [15, 18, 21]. It is possible by the addition of programmable stages to the rendering pipelines, which allows programmers to use powerful parallel processing on non-graphics data. To utilize tremendous processing capabilities of modern GPU units, either existing libraries can be used for common operations of selected algorithms (like matrix multiplication, linear algebra, Fourier transform etc), or one's own computational kernels can be written. In this paper, the second approach has been taken.

* Computer Engineering Department, Technical University of Lodz, Poland

In several recent papers (e.g. [1, 4, 10, 28]), successful GPU-based implementations of various metaheuristics have been presented. Usually, separate threads have been assigned to transformation and evaluation of separate individuals. However, this approach is particularly efficient for big populations only, i.e. several hounded of individuals, which is suitable only for specific optimization or search problems. In this paper, implementation of Quantum-Inspired Genetic Algorithm(QIGA) on CUDA has been presented. QIGA is a hybrid heuristic algorithm due to Han and Kim [8], drawing inspiration from both the biological evolution and unitary evolution of quantum systems. For fully comprehensive survey on Quantum-Inspired Evolutionary Algorithms, the reader is referred to a recent notable paper by Gexiang Zhang [36]. The original contribution of the current paper is limited neither to Quantum-Inspired Genetic Algorithm nor to CUDA technology only. The presented approach to parallelization of the experimentation procedure can be applied to a broad class of metaheuristics. Also, it can be implemented in similar and competitive to CUDA technologies, e.g. OpenCL [30, 32], BrookGPU [3] or Direct Compute [41].

This paper is structured as follows. In Section 2, fundamentals of CUDA architecture have been given. In Section 3, technical details of programming in CUDA C have been provided. In Section 4, the main contribution of this paper has been presented: the proposed approach to parallel implementation of the heuristic algorithm in massively parallel environment and its selected technical details. In Section 5, results of numerical experiments performed with the CUDA implementation have been given and evaluated. In Section 6, final conclusions of this paper have been drawn.

2. CUDA massively parallel architecture

This section provides fundamentals of CUDA architecture. For fully comprehensive description, the reader is referred to CUDA C Programming Guide [38] and similar resources [39, 41].

The architecture of a modern graphics card is usually organized as follows. The card can be equipped with several GPU **devices**. Each GPU device, has many **streaming multiprocessors** (SMs) with own flow control and on-chip shared memory units. Each multiprocessor has many **streaming processors** (SPs), consisting of independent arithmetic login units (ALUs). Also, each GPU device has highly effective hardware tasks schedulers. For example, nVidia GTX 295 is a dual-GPU graphic card consisting of two independent GPU devices. Each device has 30 multiprocessors, and each multiprocessor has 8 CUDA cores (SPs) with 1.2 GHz clock rate. Thus, on GTX 295 there are 2 * 30 * 8 = 480 CUDA cores and 2 * 896 = 1792 MB of global memory. Simultaneously, up to 60 different tasks can be run, and up to 480 data elements can be processed. The actual number of threads that are being run truly simultaneously rather than concurrently depends on GPU hardware and implementation details.

In CUDA, threads are grouped in **blocks**, and blocks constitutes a two-dimensional **grid**, which has been presented in Figure 1. The block size and number of threads per block affect multiprocessor occupancy, and their actual values are the programmer decision. **Warp** is a task scheduling unit, and it consists of 32 threads, each on a separate **lane**.

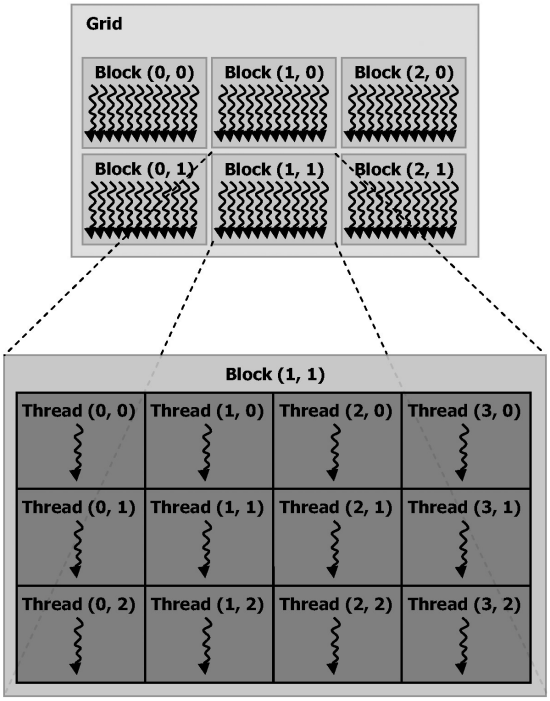

Fig. 1. Example of threads block (3×4) in the grid (2×3) on CUDA. Image source: [38]

There are several types of memory in CUDA: global, shared, local, registers, constant and texture memory. Only **registers** and **shared memory** are located on-chip in multiprocessor. Thus, access to registers and shared memory is very fast. Similarly, read-only **constant memory (64 KB)** is a cached buffer with fast access. On the contrary, access to **local** and **global memory** is not cached, thus this memory has much higher latency. **Shared memory** is a buffer shared between threads in the same block, and its capacity is 16 KB[1] per multiprocessor. What is important, precisely planned, coalesced access to the global memory is very critical for high performance of algorithms implemented on GPU. For more information on coalescence conditions, the reader is referred to [41]. **Because switching tasks is very fast due to hardware scheduler, running thousands of threads on multiprocessor concurrently mitigates latency of access to global memory. Thus, keeping high occupancy of multiprocessor is an important factor for efficient implementation on GPUs.** The function started in GPU device and executed by all threads is **computational kernel**. For software developers, CUDA runtime and driver API are provided. Programming in Runtime API is simpler, yet it provides a lower level of control of the GPU device.

[1] Modern GPU Cards with higher computation capabilities have more shared memory (up to 48 KB currently)

3. Programming in CUDA

In this section, essential information on common CUDA tools (compiler, debugger etc.) has been provided. For programming in CUDA, the following developer's kit is required:

1) **Developer Drivers** [42]
2) **CUDA Toolkit** [42] – compiler, debugger, profiler, supplementary libraries (CUBLAS, CUFFT, CUSPARCE etc.) and utilities (e.g. occupancy calculator, cuda--memcheck etc.)
3) Optionally, **CUDA Software Development Kit** [42] – several dozen examples of CUDA applications (N-body problem, linear algebra operations, image processing, fluid dynamics and others).

Runtime API in CUDA is a language extension to C with some additional qualifiers and keywords, and certain restrictions. Following function qualifiers are most important in CUDA:

- **__global__** – main function that will be executed on the GPU device (entry point)
- **__device__** – sub-function to be executed on the device (cannot be called from the host)

Variable qualifiers:

- **__shared__** – shared memory variable
- **__constant__** – constant memory variable
- **__device__** – device memory variable

Supplementary functions and built-in variables:

- **__syncthreads()** – waits until all threads in the block have reached this point
- **__threadfence()** – waits until memory accesses are visible to all threads in the device
- **__threadfence_block()** – waits until memory accesses are visible to threads in the block
- **blockDim**, **gridDim** – size of block and grid, respectively
- **blockIdx**, **threadIdx** – coordinates of current thread in grid and block, respectively

By convention, extension for CUDA source code files is .cu. To perform compilation of the file source.cu with nVidia compiler, nvcc command needs to be invoked, as follows:

```
$ nvcc -o executable source.cu
```

Additionally, to perform compilation with debug information for host and device code included, -g and -G options are required:

```
$ nvcc -g -G -o executable source.cu
```

For starting computational kernels on GPU device, a language extension <<< >>> in CUDA C has been introduced. For example, the call:

```
function1<<<dim3(3,2), dim3(10,10)>>>(arg1, arg2)
```

creates a 3×2 grid of blocks, each 10x10 threads. Each thread executes *function1* with the arguments *arg1*, *arg2*. Distinction between different threads and the data elements they are supposed to process is performed according to the thread localization in the block and grid (build-in variables blockIdx and gridIdx, respectively).

If computational kernels run longer than few seconds, headless configuration is recommended, i.e. the display should be turned off from the GPU card performing calculations completely. Otherwise, operating system routines are often likely to interfere with calculation process. To prevent freezing of the display, a watchdog in the operating systems kills the process which consumes GPU card resources constantly. Thus, if the operating system does not support disabling the watchdog, running and debugging remotely is recommended. Such functionality is provided in a commercial software, Parallel Nsight [40], and a free visual interface to debuggers, DDD (Data Display Debugger) [35]. Data Display Debugger can be started remotely, as follows:

$$\$ \text{ ssh -X } \langle host \rangle \text{ ddd —debugger cuda-gdb ./prog}$$

User interface of DDD debugger has been presented in Figure 2. At the top of the window, the computational kernel code is visible. A breakpoint has been set in the first line of the sub-function. The little arrow on the left side indicates the current line of execution. At the bottom of the window, selected debugger commands and their results are visible in the console.

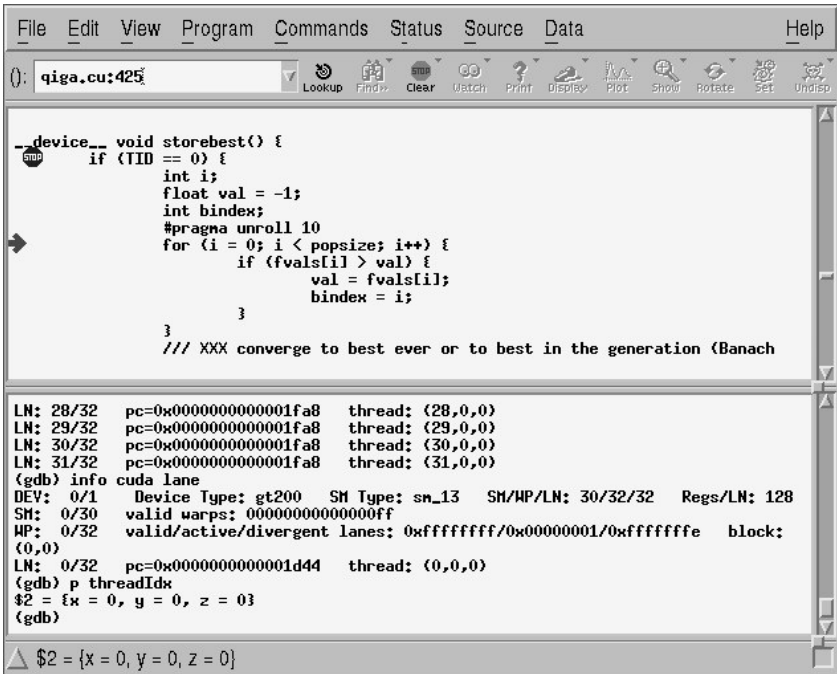

Fig. 2. Visual debugging of remote CUDA computational kernel in Data Display Debugger

Another useful tool from CUDA Toolkit is cuda-memcheck, which can detect incorrect or misaligned memory addressing. cuda-memcheck can be started as:

```
$ cuda-memcheck ./prog
```

Unfortunately, in some situations, problems that exist in normal mode, does not exist if the program is being run in the debugger. Moreover, sometimes a program runs correctly only if breakpoints have been set on certain lines, and such situation is an obvious symptom of race conditions. Since CUDA Toolkit 2.3, cuPrintf() function is also provided, but it is rather useless for serious debugging. cuda-gdb is based on GDB, GNU Debugger [29], and its details are described in [41]. The basic commands of the debugger are common for GDB and CUDA-GDB:

- **run** – starts the debugged program, arguments for the program can be provided
- **continue** – continues the program being debugged, after signal or breakpoint
- **break** – sets breakpoint at specified line or function
- **step** – steps the program until it reaches a different source line
- **list** – lists specified function or line
- **info break** – presents status of user-defined breakpoints

Selected CUDA-GDB specific commands are as follow:

- **info cuda** – presents information about the current CUDA activities
- **cuda** – generic command for CUDA-specific subcommands, for example:
  - **device** – presents or selects the current GPU device
  - **sm** – presents or selects the current multiprocessor
  - **grid** – presents information on the current grid
  - **block** – presents information on the current block

4. Implementation of metaheuristics on CUDA

In this section, successful implementation of Quantum-Inspired Genetic Algorithm [8] in CUDA has been presented and the implementation details have been provided. The authors of the present paper encountered some difficulties implementing the algorithm as a computational kernel. The problems and their solutions has been discussed in this section.

Because of stochastic nature of many contemporary metaheuristics, reliable evaluation of the heuristic algorithm performance requires independent executions of the algorithm at least several dozen times. Thus, many instances of the algorithm can be run in parallel, and their results are statistically analysed afterwards. Between separate experiments neither synchronization nor communication is required, which makes the experimentation procedure embarrassingly parallel. In population-based metaheuristics, like evolutionary algorithms or swarm intelligence techniques, selected stages of the algorithm can be also performed in parallel. For example, each thread can evaluate the fitness of different individuals. This level of parallelization has finer granularity, as some genetic operators may involve processing the whole population. Moreover, sequential execution of subsequent

stages of the evolutionary algorithm is strictly necessary. Successful applications of this type of parallelization in evolutionary algorithms have been already reported in many recent papers [1, 4, 10, 13, 14, 33]. **In our approach, parallelization has been performed on two levels: In a block of threads, each thread transforms a separate individual or different gene; In each block, a separate experiment with different population is conducted.** It has been illustrated in Figure 3. If evaluation of the fitness function does not involve processing large amounts of data, essential data structures can be often stored entirely in the very fast shared memory (on-chip memory in Streaming Multiprocessors). This makes the whole experimentation procedure feasible for efficient implementation on CUDA. Moreover, due to embarrassingly parallel nature of the procedure, the speedup scales linearly to the number of multiprocessors and GPU devices.

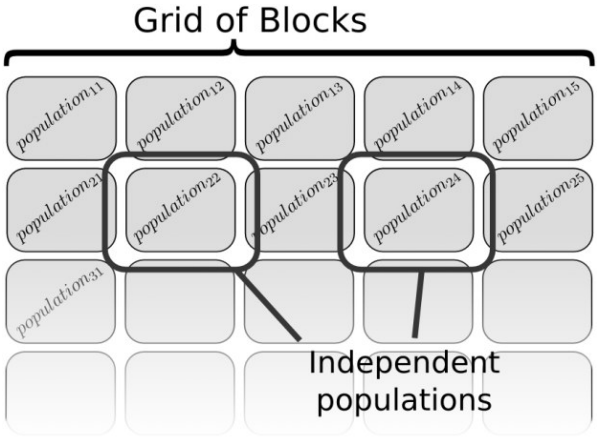

Fig. 3. Proposed approach to parallelization of the experimentation procedure. Computational threads in each block transform different individuals or different genes

Another important aspect of reliable experimentation with metaheuristics concerns a high quality pseudo random numbers generator (PRNG) with certain statistical properties [17]. Thus, generation of random numbers needs a special attention. One of highly regarded pseudo random numbers generators is Mersenne Twister (MT) algorithm, available in most modern programming languages. Let us assume that this PRNG is a reference for our considerations. During the implementation, two possibilities of providing random numbers to evolutionary algorithm running on GPU streaming multiprocessor have been considered:

1) Before running the computational kernel, random numbers can be calculated with arbitrary PRNG on CPU, and then copied to the device global memory. Thus, on the GPU device, no random numbers are actually generated. Instead, they are just read consequently from the global memory buffer.
2) Generation of random numbers directly in GPU device computational kernel. In this situation, threads running on GPU generate random numbers, when they are required.

Both possibilities have their drawbacks, and it needs some attention. Significant statistical correlation between random numbers in different threads is completely inacceptable for reliable experimentation. Thus, each thread needs its own source of random numbers, and the first possibility is calculation of random numbers on CPU and writing them into global memory of GPU in the beginning of the whole experiment. However, simplicity and efficiency of this approach are more apparent than real. For example, if there are 500 populations of 10 individuals, each represented with 256 genes, evolving for 500 generations, it will require a buffer of approximately 256*10*500*500 = 640 000 000 random numbers, and thus over 2 GB of global memory, assuming that a random number is represented with 4 bytes-long float. If the GPU device provides that much global memory, constant and uncoalesced access to this amount of memory has considerably high latency. Obviously, it would be a bottleneck of the implementation. Moreover, indexing of the calculated random numbers table needs to be performed very carefully with respect to the population and generation number, thread localization in the grid and operation number. Whenever the evolutionary algorithm is extended with an operation that requires another random number (e.g. additional stochastic genetic operator), it needs to be taken into account, which brings additional complexity.

On the contrary, random numbers can be also generated directly in device kernels. However, to represent a state of Mersenne Twister generator, 624*4 = 2496 bytes are required [17]. If there are several dozen thousands of threads, over 624*4*100 000 = 200 MB is required. Calculating random numbers directly on that amount of global memory would be also a bottleneck. Fortunately, there are other random numbers generators that are favourable for implementation on GPU. In CUDA Toolkit 3.2, a new library, CURAND [37], has been introduced, which provides effective generation of random numbers on GPU devices. In CURAND, Sobol [2] quasi-random and XORWOW [16, 24] pseudo-random routines are implemented. XORWOW algorithm is a member of the xor-shift family of pseudorandom number generators, and it is much more appropriate for running on GPU device than Mersenne Twister. XORWOW requires only 40 bytes to represent state of the generator. For the example presented in the previous paragraph, it requires only 256*500*40 = 512 KB of global memory. Disadvantage of this solution are worse statistical properties in comparison to Mersenne Twister. Also, results of the stochastic algorithm based on XORWoW cannot be compared directly to the results obtained from MT-based implementation of the algorithm on CPU. Only statistical comparison is possible in this case.

In our research, Quantum-Inspired Genetic Algorithm [8] has been selected for parallel implementation in CUDA. QIGA is a hybrid heuristic algorithm, drawing inspiration from both, the biological evolution and unitary evolution of quantum systems. Concepts such as qubits, observations and superposition of states are involved in different stages of the algorithm. In QIGA, genes are modelled upon the concept of qubits, which brings an additional

element of randomness and a "new dimension" into the algorithm. The *qubit* is a basic unit of quantum information. It is a normalised vector in a two-dimensional Hilbert space spanned by the base vectors $|0\rangle$ and $|1\rangle$, as given in equation (1):

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

where: $\alpha, \beta \in \mathbf{C}, |0\rangle = [1 \quad 0]^T, |1\rangle = [0 \quad 1]^T$ and $|\alpha|^2 + |\beta^2| = 1$.

Observation of the qubit $|\Psi\rangle$ yields a value 0 with probability $|\alpha|^2$ and value 1 with probability $|\beta|^2$. Entire solutions in QIGA are represented as *binary quantum chromosomes*, encoded as:

$$q = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_m \\ \beta_1 & \beta_2 & \cdots & \beta_m \end{bmatrix} \tag{2}$$

where each column corresponds to binary quantum gene $|\Psi\rangle_1, ..., |\Psi\rangle_m$. During the phenotype creation, states of all genes in quantum chromosomes are *observed*, i.e. the search space is sampled with respect to the probability distribution encoded in the quantum chromosomes. The genetic operators applied in the algorithm are based on *quantum rotation gates,* which rotate state vectors in the quantum gene state space. Full pseudo-code of Quantum-Inspired Genetic Algorithm has been presented in Listing 1.

```
procedure QIGA
begin
    t←0
    initialize Q(0)
    make P(0) by observing Q(0)
    evaluate P(0)
    store the best solution among P(0)
    while not termination-criterion do
        t←t+1
        make P(t) by observing Q(t-1) population
        evaluate P(t)
        update Q(t) using quantum gates U(θt)
        store the best solution among P(t)
    end while
end
```

Listing 1. Pseudo-code of Quantum-Inspired Genetic Algorithm

Let us denote as *popsize,* the size of quantum population Q, as *chromlen,* the length of quantum chromosome, and as *MAXGEN*, the maximum number of generations. Our proposal of the main data structures organization is as follows:

- **In shared memory** – quantum population Q (sizeof(float) * *chromlen* * *popsize*) observed population P (sizeof(char) * *chromlen* * *popsize*)
  Fitness values of individuals *fvals* (sizeof(float) * *popsize*)
- **In global memory** – PRNG states only (sizeof(curandState) * *chromlen* * *popsize*)
- **In constant memory –** read-only data (e.g. data required for fitness evaluation)

Simplified code of the main kernel function has been presented below. As it is easy to see, it corresponds directly to the pseudo-code of QIGA.

```
__global__ void qiga(char *BESTgmem, float *FITNESSgmem, curandState
*rngStates) {

      int t = 0; // generation number

      // fitness of the best individual in population
      bestval = -1; // declared in shared memory

      initialize();
      __syncthreads();
      observe(rngStates);
      __syncthreads();
      repair();
      __syncthreads();
      evaluate();
      __syncthreads();
      storebest();

      while (t < MAXGEN) {
        t++;
        __syncthreads();
        observe(rngStates);
        __syncthreads();
        repair();
        __syncthreads();
        evaluate();
        __syncthreads();
        update();
        __syncthreads();
        storebest();
        __syncthreads();
      }

      __syncthreads();
      if (threadIdx.x == 0) {
        // copy the evolution result to global memory buffer
        FITNESSgmem[gridDim.x * blockIdx.y + blockIdx.x] = bestval;
        // ...
      }
   }
```

In the beginning, bestval (per block variable, declared in the shared memory) is initialized with a negative value. Then, MAXGEN generations of the quantum population Q are evolved. Eventually, the first thread in a block (i.e. threadIdx.x == 0) is designated to write the fitness of the best individual to the global memory buffer FITNESSgmem.

Proper synchronization of threads is extremely critical. Therefore, __syncthreads() is called after each subsequent stage of the algorithm. If there had been a lack of some __syncthreads() calls, execution of the program would have been terribly wrong. For example, some threads would have started evaluating individuals (`evaluate()` call), while other threads would not have finished modifying them (`repair()` call). Possible concurrent evaluation and modification of the same data structures would have resulted in wrong outcomes of the algorithm eventually. The wrong synchronization of threads is often hard to detect, if a separate reliable implementation of the algorithm and a results comparison procedure are not at the programmer's disposal.

The presented `qiga` computational kernel can be started with the call:

```
qiga<<<dim3(64, 10), 250>>>(d_best, d_fit, rngStates);
```

For example, this call creates a grid of 64x10 = 640 blocks (separate populations), 250 threads in each block. Consequently, 640*250 = 160 000 threads are started on GPU concurrently, conducting evolution of 640 populations in parallel. d_best, d_fit and rngStates are pointers to memory buffers in the device global memory, and memory for the buffers must be allocated in advance, before running the kernel.

5. Experimental results

Firstly, Quantum-Inspired Genetic Algorithm has been implemented as a typical sequential program in ANSI C running on CPU for comparison (Mersenne Twister PRNG). To verify the correctness of the CPU implementation, selected results presented in [8] have been reproduced. Secondly, the authors have created new implementation in CUDA C Runtime API (XORWOW PRNG), and statistical significance of its outcomes has been compared to the expected results in statistical analysis. The performance comparison of the two implementations is the main experimental contribution of this article.

In [8], knapsack problem has been used as an underlying combinatorial optimization problem. Strongly correlated set of data has been generated, i.e. hard version of the problem where precious items are heavy:

$$\begin{cases} w_i = uniformlyrandom[1,10) \\ p_i = w_i + 5 \end{cases} \tag{3}$$

where w_i denotes weight of the i-th item, and p_i denotes profit of the item. Experiments with 250 items and 10 quantum individuals evolving for 500 generations have been conducted.

In Han's implementation (Visual C++ 6.0, Pentium-III 500MHz), about 0.724 evolutions per second is performed (cf. Tab. 2. In [8]). In our CPU implementation (ANSI C, Intel Core i7, 2.93GHz), about 7.324 evolutions per second is performed. In the GPU implementation (nVidia CUDA C, GTX-295), evolving independent populations in the grid of size 50×20, about 882.7 evolutions per second is performed. **Thus, the speedup gained on GTX-295 is about 120× in comparison to the sequential implementation.** Performance comparison has been presented in Figure 4. In the measurement, only the qiga computational kernel execution time has been taken into account. Device initialization (`cudaSetDevice()` call) takes also few seconds, but this delay is constant and one-time, thus it has not been accounted in the measurement.

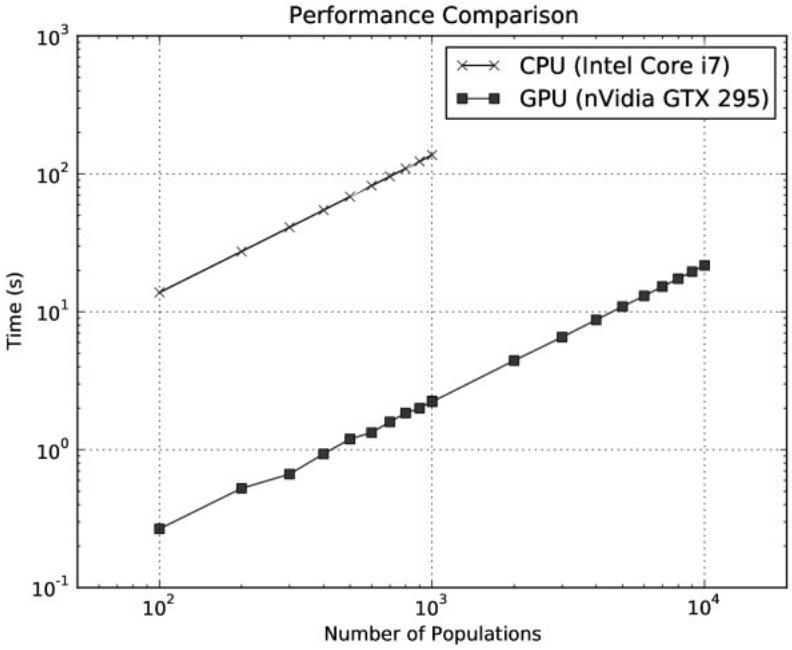

Fig. 4. Performance Comparison (Intel Core i7 CPU vs dual-GPU GTX 295)
On the left, linear scaling of the axes. On the right, log scaling of the axes

Due to stochastic nature of the evolutionary algorithm and different PRNGs used in CPU and GPU implementations, correctness of the GPU implementation has been verified with statistical analysis in comparison to sequential implementation in ANSI C. Evolution of 30000 populations has been performed on CPU and GPU. It took 69 minutes and 34 seconds, respectively. Box plot and histogram of the results are given in Figures 5 and 6. Because of different random numbers generators and their different statistical properties, the results comparison fail to pass strict statistical tests, but essential measurements of variability and diversity (Tab. 1) confirms that the implementations are consistent with each other with high certainty.

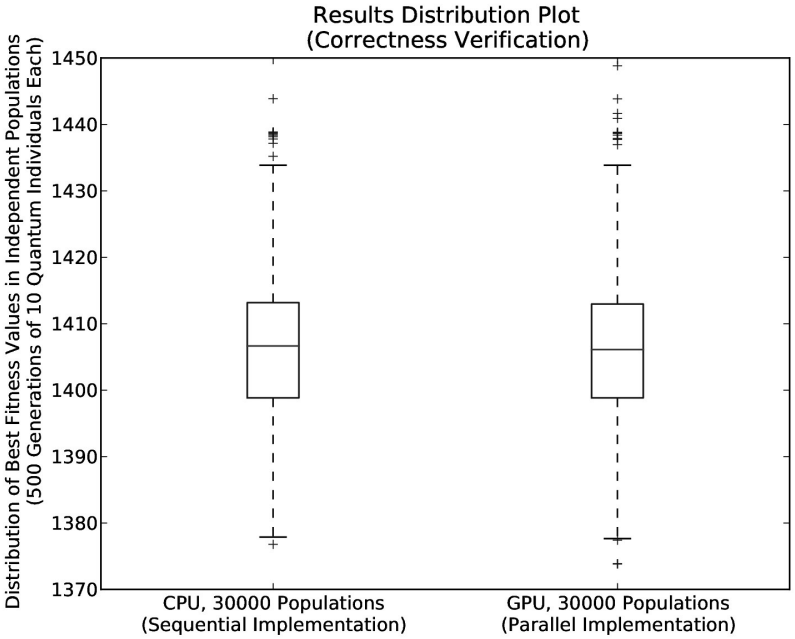

Fig. 5. Correctness verification. The ends of the whiskers represent the lowest and the highest result still within 1.5 interquartile range (IQR)

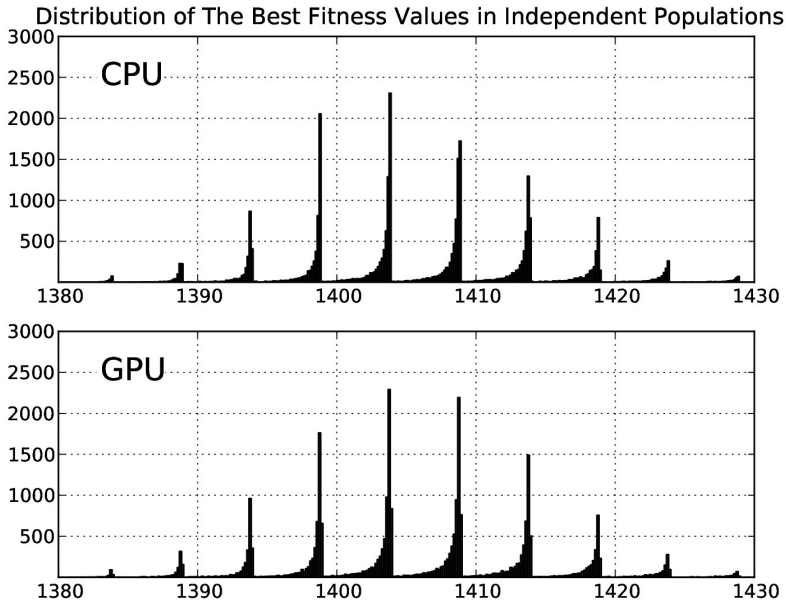

Fig. 6. Correctness verification (histogram of 30000 evolutions)

Table 1
Results of comparison (correctness verification)

	Median	Average	Variance
CPU	1406.659424	1406.25894801	74.70038777
GPU	1406.108887	1406.09914151	74.67657533

Finally, an experiment has been conducted with distributed calculations on eight GPU devices (4 × Tesla T10 GPU, GTX 285, dual-GPU GTX 295 and Tesla C2070 GPU). **On this configuration, the speedup gained was over 400x**. The total number of available streaming multiprocessors is very important in this approach. For distributed calculations on several remote nodes, the authors of the present paper finds dsh [44], a distributed shell, particularly useful. Other possibilities of distributing the calculations include rCUDA [43], a framework which enables the concurrent usage of CUDA-compatible devices remotely.

6. Conclusions

In this paper, successful implementation of Quantum-Inspired Genetic Algorithm in massively parallel environment (CUDA technology) has been presented and implementation details have been provided. The proposed approach to parallelization is twofold: In a block of threads, each thread transforms a separate individual or different gene; In each block, evolution of a separate population with same or different parameters is conducted. This approach can be applied to experimentation with any heuristic algorithm, and also it can be implemented in any similar to CUDA technology. Correctness of the results has been verified in statistical analysis. On one nVidia GTX-295 card, about 120× speedup has been gained in comparison to sequential implementation of the algorithm. On eight GPU devices, over 400× speedup has been gained. This result can be further improved by running the algorithm on more GPU cards. Due to embarrassingly parallel nature of the experimentation procedure, the speedup scales linearly to the number of multiprocessors and GPU devices. Possibly, further speedup improvement could be gained in OpenCL [30, 32] on Radeon GPU cards, which were not available to the authors during this work. What is more important, the speedup gained allows efficient meta-optimization [25] of modern metaheuristics, which is the most exciting path for possible future research.

Writing advanced computational kernels in CUDA that are both effective and correct requires programmer's special diligence. In massively parallel programming, numerous nontrivial issues exist the programmer must be fully aware of, be able to detect, analyze and to prevent them. Random factors present in many heuristic algorithms bring additional complexity. **The common tricky issues that always need special attention include precisely**

planned global memory access, proper synchronization, handling of critical sections, avoiding possible race conditions and deadlocks. In special situations, also avoiding possible threads resources starvation. Thus, some aspects of programming in massively parallel environment is hardly comparable to sequential programming. Moreover, according to the authors' experience, some tools and features are not very mature yet in comparison to modern compilers and development environments

Acknowledgement

The authors are grateful to Piotr Kowalski (Technical University of Lodz) for several discussions. Robert Nowotniak, a co-author of the present paper, is a scholarship holder of project entitled „Innovative education ..." supported by European Social Fund.

References

[1] Banzhaf W., Harding S., *Accelerating evolutionary computation with graphics processing units.* Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, 2009.

[2] Bratley P., Fox B.L., *ALGORITHM 659: implementing Sobol's quasirandom sequence generator.* ACM Transactions on Mathematical Software (TOMS), ACM, 14, 1988, 88–100.

[3] Buck I., Foley T., Horn D., Sugerman J., Fatahalian K., Houston M., Hanrahan P., *Brook for GPUs: stream computing on graphics hardware.* ACM Transactions on Graphics (TOG), 2004.

[4] Cabido R., Montemayor A.S., Pantrigo J.J., *High performance memetic algorithm particle filter for multiple object tracking on modern GPUs.* Soft Computing-A Fusion of Foundations, Methodologies and Applications, Springer, 1–14.

[5] Fernando R. (ed.), *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[6] Fialka O., Cadik M., *FFT and convolution performance in image filtering on GPU.* Tenth International Conference on Information Visualization, 2006. IV 2006.

[7] Fok K.L., Wong T.T., Wong M.L., *Evolutionary computing on consumer graphics hardware.* Intelligent Systems, IEEE, 22, 2007, 69–78.

[8] Han K.H., Kim J.H., *Genetic quantum algorithm and its application to combinatorial optimization problem.* Proceedings of the 2000 Congress on Evolutionary Computation, 2000.

[9] Harris M.J., *Fast fluid dynamics simulation on the GPU.* GPU Gems, Citeseer, 1, 2004, 637–665.

[10] Krüger F., Maitre O., Jiménez S., Baumes L., Collet P., *Speedups between ×70 and ×120 for a generic local search (memetic) algorithm on a single GPGPU chip.* Applications of Evolutionary Computation, Springer, 2010, 501–511.

[11] Krüger J., Westermann R., *Linear algebra operators for GPU implementation of numerical algorithms*. ACM SIGGRAPH 2005 Courses, 2005.

[12] Lee M., Jeon J., Bae J., Jang H. S., *Parallel implementation of a financial application on a GPU.* Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, 2009.

[13] Li J., Zhang L., Liu L., *A parallel immune algorithm based on fine-grained model with GPU-acceleration.* Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009.

[14] Li J.M., Wang X.J., He R.S., Chi Z.X.: *An efficient fine-grained parallel genetic algorithm based on gpu-accelerated*. IEEE Computer Society, 2007.
[15] Manavski S., Valle G., *CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment*. BMC Bioinformatics, BioMed Central Ltd, 9, 2008, S10.
[16] Marsaglia G., *Xorshift rngs.* Journal of Statistical Software, 8, 2003, 1–6.
[17] Matsumoto M., Nishimura T., *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.* ACM Transactions on Modeling and Computer Simulation (TOMACS), ACM, 8, 1998, 3–30.
[18] Moreland K., Angel E., *The FFT on a GPU.* Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2003.
[19] Nickolls J., Buck I., Garland M., Skadron K., *Scalable parallel programming with CUDA.* Queue, ACM, 6, 2008, 40–53.
[20] Nvidia: *Compute Unified Device Architecture Programming Guide.* NVIDIA, Santa Clara, CA, 2007.
[21] Oh K.S., Jung K., *GPU implementation of neural networks.* Pattern Recognition, Elsevier, 37, 2004, 1311–1314.
[22] Owens J., *GPU architecture overview.* ACM SIGGRAPH, 2007.
[23] 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, 2007.
[24] Panneton F., L'ecuyer P., *On the xorshift random number generators.* ACM Transactions on Modeling and Computer Simulation (TOMACS), ACM, 15, 2005, 346–361.
[25] Pedersen M.E.H., *Tuning & Simplifying Heuristical Optimization.* University of Southampton, School of Engineering Sciences, 2010.
[26] Preis T., Virnau P., Paul W., Schneider J.J., *Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets.* New Journal of Physics, IOP Publishing, 11, 2009, 093024.
[27] Preis T., Virnau P., Paul W., Schneider J.J., *GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model.* Journal of Computational Physics, Elsevier, 228, 2009, 4468–4477.
[28] Robilliard D., Marion-Poty V., Fonlupt C., *Population parallel GP on the G80 GPU.* Genetic Programming, Springer, 2008, 98–109.
[29] Stallman R., Pesch R. H., Shebs S. (ed.), *Debugging with GDB: The GNU source-level debugger.* Free Software Foundation, 1995.
[30] Stone J.E., Gohara D., Shi G., *OpenCL: A parallel programming standard for heterogeneous computing systems.* Computing in Science and Engineering, 12, 2010, 66–73.
[31] Tomov S., Dongarra J., Baboulin M., *Towards dense linear algebra for hybrid GPU accelerated manycore systems.* Parallel Computing, Elsevier, 36, 2010, 232–240.
[32] Tsuchiyama R., Nakamura T., Iizuka T., Asahara A., Miki S., *The OpenCL Programming Book.* Group, Fixstars Corporation, 2009.
[33] Wong M.L., *Parallel multi-objective evolutionary algorithms on graphics processing units.* Proc. of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, 2009.
[34] Yang J., Wang Y., Chen Y., *GPU accelerated molecular dynamics simulation of thermal conductivities.* Journal of Computational Physics, Elsevier, 221, 2007, 799–804.
[35] Zeller A., Lütkehaus D., *DDD – a free graphical front-end for UNIX debuggers.* ACM Sigplan Notices, ACM, 31, 1996, 22–27.
[36] Zhang G., *Quantum-inspired evolutionary algorithms: a survey and empirical study*. Journal of Heuristics, Springer, 2010, 1–49.

[37] http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CURAND_Library.pdf [2011-05-31].
[38] http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf [2011-05-31].
[39] http://www.lunarc.lu.se/Documents/nvidia-workshop/files/tutorial/CUDA_C_QuickRef.pdf [2011-05-31].
[40] http://developer.nvidia.com/nvidia-parallel-nsight [2011-05-31].
[41] http://developer.nvidia.com/nvidia-gpu-computing-documentation [2011-05-31].
[42] http://developer.nvidia.com/cuda-toolkit-40 [2011-05-31].
[43] http://www.hpca.uji.es/rCUDA [2011-05-31].
[44] http://www.netfort.gr.jp/~dancer/software/dsh.html.en [2011-05-31].