Heinrich C. Mayr, Martin Pinzger (Hrsg.): INFORMATIK 2016,
Lecture Notes in Informatics (LNI), Gesellschaft fr Informatik, Bonn 2016 2163

Overview of the current state of research on parallelisation of evolutionary algorithms on graphic cards

Paul Jähne[1]

Abstract: Evolutionary algorithms (EA) have a lot of potential for parallelisation, which can be used by graphics processing units (GPU). They present an available, cheap and energy-efficient alternative to computer clusters. There are already several publications on using GPUs for EAs. This paper presents selected publications and discusses important implementation details of them. Hence recommendations are derived on how to efficiently implement EAs on GPUs. Thereby it's important to take the architecture of GPUs into consideration. Furthermore it's shown that GPUs can only be used profitably when a certain problem complexity is reached. In addition the speed-up to expect is critically scrutinised and it's explained, that it's impossible to reach a speed-up of 100 or even more by means of a fair comparison of GPU and CPU implementations.

Keywords: meta-paper, evolutionary algorithm, GPGPU, CUDA

1 Motivation

Often there are no efficient algorithms known for difficult problems or even impossible to construct due to their complexity. Then only general search methods remain to get a good approximate solution within acceptable time. Evolutionary algorithms (EA) are one option to construct such domain independent searches. Yet also these reach their limits at a certain problem size. Fortunately they have big potential for parallelisation. This can be utilised through the use of multiple processors (CPU) or computers. However clusters out of several computers are harder to use and maintain and also not generally available.

An alternative to this is the use of graphics processing units (GPU). They evolved from specialised graphic accelerators to flexible, highly parallel coprocessors and are available in every modern computer. Therefore they were recently used for many applications aside from graphics processing with partly remarkable results [Ow05][Nv16]. Since EAs have a lot of parallelisation potential, there were also efforts to use graphic cards in this domain. This paper presents selected works and gives an overview of the different approaches.

2 Evolutionary Algorithms

Evolutionary algorithms are powerful, domain independent search methods derived from the evolutionary theory. Their search strategy is based on stochastic methods. Because of this there is no guarantee to find the optimal solution, but in most cases sufficient approximate solutions are found. The process is as follows. At the beginning a random population

[1] Hochschule für Technik, Wirtschaft und Kultur, Fakultät Informatik, Mathematik und Naturwissenschaften, Karl-Liebknecht-Straße 132, 04277 Leipzig, paul.jaehne@gmx.de

is generated. This consists of possible solutions for the given problem, referred to as individuals. Based on this initial population new generations are created by repeating evaluation, selection, crossover and mutation. This process iterates until a sufficient solution is found or if the maximum number of generations is exceeded. The evaluation assigns a fitness value to all individuals, which describes the quality of the solution. Thereby a search direction is given. The selection chooses promising individuals for the next generation based on their fitness value. The crossover uses the chosen individuals to create new ones through mixing. The mutation alters some of the resulting individuals to retain the populations diversity and open up new areas of the search space.

Based on this principle the four following classical variants were developed: genetic algorithms (GA), evolutionary strategies (ES), genetic programming (GP) and evolutionary programming (EP). They differ in the type of individuals and the used operators. Exact differentiations are difficult, as the approaches are interacting and influencing each other [WW09]. All have in common, that calculation effort increases with increasing number of individuals, generations and operators. Yet the operators work only with few individuals at the same time and the evaluation can be executed independently for each individual. Therefore EAs have a lot of potential for parallelisation.

3 Programming graphic cards

Through the popularity of 3D consumer applications like video games a market for affordable hardware accelerators was established. They implemented parts of the rendering pipeline of application programming interfaces (API) like Open Graphics Library (OpenGL) and DirectX. Graphics calculations need to do a series of operations for every pixel on the screen. Therefore large computing power is required, but each pixel can be calculated independently in parallel. Thus graphic cards evolved into powerful and highly parallel coprocessors, while their price remained in the consumer segment [SK10].

This development attracted the attention of some scientists. Since GPUs calculate colour values out of textures, objects and additional information, which are specified by the programmer, any data is possible. Only a mapping between the problem, it's solution algorithm and graphics calculation is required and the result has to be interpreted accordingly. Therefore GPUs could be used for non-graphic computations. This is called general-purpose computing on graphics processing units (GPGPU) [SK10]. Yet the only way to interact with a GPU were graphics APIs in which the actual calculation has to be packaged.

In the year 2004 Brook[2] was a first attempt to solve these issues and to make GPGPU accessible for more people. Brook is an extension for the programming language C. It abstracts the graphics APIs and presents the programmer an environment, which he's used to from the CPU. Still various constraints existed on the GPU. These were resolved through architectural evolution of the GPU. In addition in the year 2007 CUDA[3] and Open Computing Language[4] (OpenCL) were released. These are low-level APIs specifically designed

[2] http://graphics.stanford.edu/projects/brookgpu/

[3] https://developer.nvidia.com/content/cuda-10

[4] https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification

for GPGPU. An high-level alternative emerged in 2011 with OpenACC[5]. OpenACC uses compiler directives to annotate parallelisable code segments for the GPU. These are processed by the compiler. So there is no need to rewrite existing code and the programmer doesn't has to bother with details. Also Open Multi-Processing[6] (OpenMP) got support for coprocessors in 2013 with version 4.0. OpenMP is as well based on compiler directives, but was originally developed for multiprocessor systems with shared memory.

The rise of language extensions and the related simplification increased the adoption of GPGPU. The development of systems with coprocessors in the 500 most powerful computer systems, which are listed in TOP500, proves this. The development over time of systems using coprocessors in this list can be seen in figure 1.

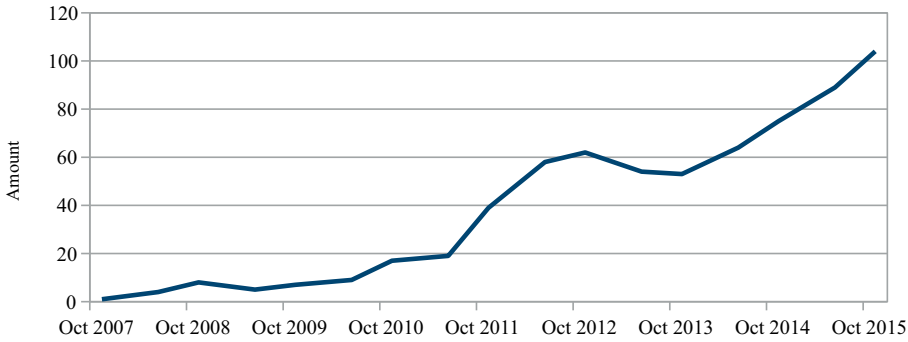

Fig. 1: Temporal development of the amount of systems using coprocessors in TOP500 as per [TO15]

The main limitations for systems in the TOP500 are space and energy consumption [Be08]. This is where graphic cards provide advantages compared to CPUs besides there computing power. Table 1 shows important features of a high-end server CPU and GPU with comparable release dates. It can be seen that the GPU has the eightfold computing power at twice the power consumption. Thus it has the fourfold efficiency. To compensate for the differences in price, power consumption and space, one could compare a graphic card with two such CPUs. Then a fourfold advantage in computing power remains.

Feature	Intel Xeon E5 2697 v2[a]	Nvidia Tesla K40[b]
Release date	September 10, 2013	November 18, 2013
Manufacturing process	22 nm	28 nm
Release price	2614 $	5499 $
Power dissipation	130 W	235 W
Computing power[c]	0.52 Tflop/s	4.29 Tflop/s
Processing power[c]/Watt	3.98 Gflop/Ws	18.3 Gflop/Ws

[a] http://ark.intel.com/products/75283
[b] http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05-en.pdf
[c] For calculations in single precision

Tab. 1: Exemplary comparison of important features of CPUs and GPUs

[5] http://www.nvidia.co.uk/object/openacc-parallel-computing-standard-20111114-uk.html
[6] http://openmp.org/wp/openmp-40-api-released/

4 Early days

Until 2007 there were no programming languages or frameworks specifically designed for GPGPU. So graphic APIs were the only way to interact with a GPU. Because they were designed for graphic applications, the calculations had to be transformed into graphic calculations to satisfy there input and output format. Therefore several limitations existed like a maximum number of arrays and elements within these through the size and number of textures. Due to such restrictions GPUs were difficult to use for general computation. But when the high initial hurdles were overcome one could get a considerable speed-up compared to a CPU implementation, which showed the potential of GPUs [Ow05]. Hereafter are several papers from that period presented.

Chitty presents in [Ch07] an implementation of genetic programming with OpenGL and C for Graphics (Cg). Therein the fitness evaluation was offloaded to the GPU. GP isn't based on the principle Single Instruction, Multiple Data (SIMD) like the GPU, but rather on Multiple Instruction, Single Data (MISD). The problem is solved by running the fitness function on different example data in parallel. For comparison an Intel Pentium 4 and a Nvidia GeForce 6400 Go is used. The results show that the CPU is faster for small Problems. Beginning with 200 samples the GPU outperforms the CPU. With a sufficiently large data set and complex examples a speed-up between 12 and 30 is reached.

Fok, Wong and Wong implement evolutionary programming on the GPU in [FWW07]. They chose EP, because crossover is not used therein and thus it is easier to implement. Random number generation and selection is done on the CPU and the results are provided as a texture to the GPU. The used operators are tournament selection and Cauchy mutation. The selection of individuals isn't done by sorting, but by searching the median of the tournament victories. Subsequently the population is traversed and all individuals with greater or equal amount of wins are chosen. For comparison an Intel Pentium 4 and a Nvidia GeForce 6800 Ultra is used. The GPU implementation is up to five times faster for different test functions with more than 800 individuals. The runtime increase is sublinear. This is an indication for underutilisation of the GPU. By increasing the problem size more computing units can be used. Hence the increase in runtime is sublinear.

Wong and Wong present a genetic algorithm based on [FWW07] in [WW09] which is executed on the GPU except for the random number generation. The used operators are tournament selection, single point crossover and Cauchy mutation. Different functions are used to test on an AMD Athlon 64 and a Nvidia GeForce 6800 Ultra. The GPU implementation is up to five times faster depending on complexity of the function and number of individuals.

Li et al. present in [Li07] a genetic algorithm, which runs completely on the graphic card. The used operators are single point crossover, single point mutation and local roulette wheel selection. Different test functions are used for comparison. The tests are carried out on an Intel Pentium 4 and a Nvidia GeForce 6800 LE. The GPU implementation reaches a speed-up up to 74 for large population sizes and complex functions. Also for small populations a small speed-up is achieved.

5 Modern era

The emergence of frameworks, language extensions and development tools specifically designed for GPGPU lowered the initial hurdles and therefore made it accessible to a wider audience. Furthermore graphic cards developed from fixed-function devices into flexible, highly parallel coprocessors, which resolved several limitations. This facilitated the implementation of more complex algorithms.

5.1 Fitness evaluation on the GPU

The simplest use of graphic cards is the implementation of the fitness evaluation on the GPU. This can be done entirely independent for each individual and is therefore suited for parallelisation according to the SIMD architecture of GPUs. An easy implementation can be seen in listing 1. There is an easy and general pattern for parallelisation of loops shown. `evaluateKernel` is a GPU function, which is also called kernel. Additionally the arrays `population` and `fitness` have to be transferred to the GPU for processing and back afterwards. This is not included in the following example.

```
C:
...
  for (int i = 0; i < populationLength; i++) {
    fitness[i] = evaluate(population[i]);
  }
...

CUDA:
__global__ void evaluateKernel(float *population,
    float *fitness, int populationLength) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
      i < populationLength; i += blockDim.x *
          gridDim.x) {
    fitness[i] = evaluate(population[i]);
  }
}
...
  evaluateKernel<<<BLOCKS, THREADS>>>(population,
    fitness, populationlength);
...
```

List. 1: Parallel fitness evaluation in CUDA

The described method is easy to implement and can achieve a reasonable speed-up with sufficient individuals and complex fitness functions. If these conditions aren't fulfilled, the GPU isn't fully utilised or the data transfer takes more time, than what is saved through parallel execution.

An example for this strategy present Cavouti et al. in [Ca12]. They extend an existing implementation of a genetic algorithm for machine learning with CUDA and offload the creation of the initial population and the fitness evaluation to the GPU. For comparison they used an Intel Core i7-2630QM and a Nvidia GeForce GT540M. The classification results are the same, but the GPU reaches a speed-up of 75. However the CPU implementation only used one core out of four. Therefore the comparison is unfair in favour of the GPU and the achieved speed-up should be divided by four.

If one population isn't enough to keep the graphic card busy, parallel evaluation of multiple populations through streams is an option to provide enough work. Streams are one way to do different tasks in parallel on the GPU. All commands placed into one stream are executed sequentially while different streams run in parallel. So it is possible to run multiple small kernels on the GPU or to interleave copy processes with computations. Most CUDA functions provide an additional stream parameter for this purpose. An example for a kernel launch can be seen in listing 2.

```
...
  cudaStream_t stream1;
  cudaStreamCreate(&stream1);
  evaluateKernel<<<BLOCKS, THREADS, 0, stream1>>>(
      population, fitness, populationlength);
  cudaStreamDestroy(&stream1);
...
```

List. 2: Use of CUDA-streams on kernel call

The parallel evaluation of several small populations is used by Robilliard, Marion-Poty and Fonlupt in [RMPF08]. They notice that many typical problems are to small to keep a GPU busy. Therefore multiple small populations are evaluated in parallel. They don't use the previously shown approach with streams, but an explicit implementation within the kernel. Thereby the fitness evaluation is implemented on the GPU. The remaining operations are executed on the CPU with the ECJ[7] library. The implementation is compared to a serial one using the ECJ library for all parts. The tests show a speed-up between 8 and 80 in the evaluation phase. This leads to a reduction of the overall runtime by a factor between 5 and 45. No speed-up is achieved for examples with many conditionals in the fitness function. This is due to the SIMD architecture of GPUs, which is unsuitable for branching.

5.2 Complete GPU-implementation

Even though the parallelisation of fitness evaluation on the GPU is profitable under certain circumstances, it is limited by the bandwidth of the interconnection between CPU and GPU, because the population has to be copied back and forth for each evaluation. One solution for this problem is a complete GPU implementation, which eliminates the need for frequent copies. Even if individual steps are slower on the GPU, the overall speed-up can

[7] http://cs.gmu.edu/~eclab/projects/ecj/

be increased by removing the slow transfers. However a complete GPU implementation is more complex, because the algorithm has to be fitted to the GPU architecture, to use the full potential.

Debattisti et al. present a complete implementation of a genetic algorithm in CUDA in [De09]. The test problem is maximizing the number of bits set to one, which isn't computationally demanding. Therefore it is limited by the available bandwidth. The operators are tournament selection, two-point crossover and mutation via bitwise exclusive OR with a template. Random numbers are generated with the Mersenne Twister implementation out of the CUDA SDK examples. The implementation runs on a Nvidia GeForce 8800 GT and is compared with a sequential implementation, created with TinyGA on an Intel Core2Duo. A speed-up of 20 is reached for larger populations and individuals.

A critical examination to parallelisation of evolutionary algorithms on GPUs can be found in [JP12] by Jaroš and Pospíchal. They notice, that comparisons of GPU and CPU implementations often use less optimised or sequential versions for the CPU, although they have multiple cores available. Because of this they present and compare one implementation specifically tuned for each architecture. This means using vector instructions and multiple cores on the CPU and efficient memory access and low branching on the GPU. The paper describes the memory organisation and all operators in detail. The operators are tournament selection, uniform crossover and mutation via one bit flip. Additionally several statistics are gathered. Random numbers are generated via an algorithm based on hash functions from Salomon et al. published in [Sa11]. Both implementations are available on GitHub[8]. Large instances of the knapsack problem (10,000 items, 12,000 individuals) are used for comparison, to achieve maximal utilisation. They use an Intel Xeon X5650 and a Nvidia GTX 580 for their tests. The tests and there parameters are also described in detail. They additionally implement the same algorithm with the GAlib[9] library, which is often used for comparison in other papers. The results show that the GPU implementation reaches a speed-up of 375 compared to GAlib. The speed-up was reduced to 221 when GAlib is compiled with optimisations. The GPU reached a speed-up of 68 in comparison to the proposed CPU implementation with one thread. This value corresponds to the speed-up values reported in other papers, which as well use only one core. The speed-up decreases to 12 if all cores are used. This complies with the ratio of the theoretical peak performance for the CPU and GPU. Compared to two CPUs a speed-up of 6 is reached. Additionally the cache hit ratio is analysed. GAlib reaches a hit ratio of 18 % and the proposed CPU implementation achieves 98 %. This shows how well they utilise the CPU architecture. Overall the paper shows that a GPU can't achieve a speed-up of 100 or even more, if an equally optimised CPU implementation is considered.

5.2.1 Cellular EA

In contrast to standard EAs cellular EAs use operators, which work locally. This means, that individuals are arranged in some structure and the operators apply to a certain neigh-

[8] https://github.com/jarosjir/GPU-GA-Knapsack und https://github.com/jarosjir/MPI-GA-Knapsack
[9] http://lancet.mit.edu/ga/

bourhood within this structure. Costly global communication and dependencies between threads are thereby reduced. An example is the Moore neighbourhood, where the eight surrounding cells in a 2D grid are considered as neighbours. A schematic representation of the field of action for the Moore neighbourhood can be seen in figure 2.

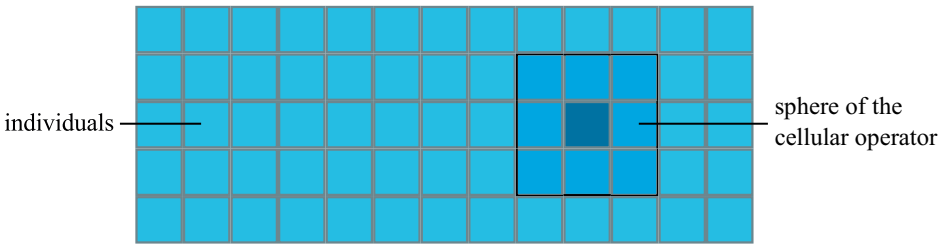

Fig. 2: Schematic presentation of the Moore neighbourhood for a cellular operator

An implementation can be found in [LL06] by Luo and Liu. They develop a genetic algorithm to solve the 3-SAT problem. The individuals are arranged in a 2-dimensional toroidal grid and the Moore neighbourhood is used. This grid structure avoids borders where individuals have fewer neighbours. They use an Intel Pentium 4 and a Nvidia GeForce 6200 for comparison. The results are comparable and the GPU reaches a speed-up of 5.

5.2.2 Island model

The island model tries as well to reduce the amount of global communication. This is also accomplished by the use of neighbourhood relationships. Separated subpopulations are used for this purpose. They only exchange few individuals in set intervals by the additional migration operator. This can lead to better solutions because the different populations are more likely to not get stuck at the same local optimum. Then again arbitrary models can be used within one island. Figure 3 shows a schematic representation.

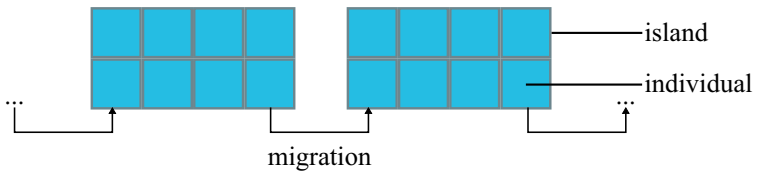

Fig. 3: Schematic presentation of the island model with migration

Pospíchal, Jaroš and Schwarz present in [PJ09], [PJS10] and [PSJ10] a genetic algorithm based on the island model. Therein each thread is assigned to one individual and a thread block corresponds to an island. Therefore the fast shared memory and synchronisation, which are only available within one block, can be used for each subpopulation. A migration is performed between blocks in intervals. The best individuals of a subpopulation replace the worst. This exchange is done asynchronously in a circle. Therefore a set interval isn't guaranteed, due to different thread blocks that run independently. Though this is no big issue as EAs are stochastic algorithms. The implementation is compared with a sequential

one of the same algorithm. The algorithms are compared on an Intel i7-920 and Nvidia GeForce 8800 GTX, GTX 260 and GTX 285 for different test functions and the knapsack problem. The results are speed-ups up to 1000 for a sufficient number of individuals and generations. As mentioned earlier such values are unrealistic and caused by less optimised CPU implementations. The island model converges faster then a standard algorithm, while the quality of results stays the same. Additionally the compiler option `-use_fast_math` is tested. This instructs the compiler to replace mathematical operations with faster but less precise ones. The runtime can be reduced further without deterioration of the solution quality.

5.2.3 Multi-GPU

Using multiple GPUs is the logical continuation if the processing power of one GPU isn't enough. Multi-GPU configurations are still feasible for consumer computers and become more common. This use case is also addressed by modern APIs. When multiple GPUs in one computer should be used, the CUDA function `cudaSetDevice` can be called. Afterwards all commands use the specified device.

The communication between GPUs in one computer can take place directly via the Peripheral Component Interconnect Express (PCIe) bus. The transfer bandwidth between GPUs is limited by PCIe bandwidth, which is significantly slower, than memory access to GPU main memory. If multiple computers are needed, the communication between them can be done via the Message Passing Interface (MPI). Thereby communication costs are further increased, because the data has to be copied from the GPU to the main memory of the computer, then transferred to another computer and copied into memory of the other GPU.

Jaroš presents in [Ja12] an implementation of a genetic algorithm for multiple GPUs. Different implementation options are discussed at the beginning. Mapping individuals to threads is restricted by the size of the individuals and therefore only applicable for small problems. A thread block per individual is only useful for large individuals. Using thread groups is proposed as a middle course. The individuals are divided into subpopulations and each GPU is assigned to one such island. The used operators are uniform crossover, tournament selection and mutation via one bit flip. The migration between islands is done in a ring topology. The communication between GPUs is done via MPI. Furthermore the memory organisation is described in detail. Two different memory layouts for the population are discussed. These are the gene- and chromosome-based layouts. The gene-based structure stores the same genes of the different individuals in succession. The chromosome based structure stores individuals continuously. This is favoured because it preserves data locality and leads to better cache utilisation. Figure 4 schematically represents these layouts. The same algorithm is also implemented for the CPU. The knapsack problem with 10,000 items is used to compare both implementations. The hardware used are two servers each with two Intel Xeon X5650 and seven Nvidia GTX 580. Both implementations scale well for sufficiently large islands. The graphic cards need bigger islands to be fully utilised. The fourteen GPUs reach a speed-up of 35 when compared to the four CPUs and 194 com-

pared to one CPU. Scaled down to one GPU this results in a speed-up of 9 and 14 times respectively.

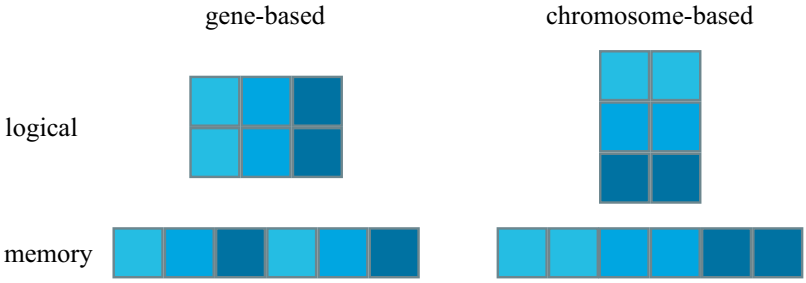

Fig. 4: Schematic presentation of the gene- and chromosome-based memory layout

6 Summary

This paper presents different approaches to performing evolutionary algorithms on graphic cards. It is shown that a complete GPU implementation is possible and sensible. The best implementation can differ depending on structure and complexity of the problem. In general the following recommendations can be given. The algorithm should be implemented completely on the GPU to avoid slow transfer between CPU and GPU memory. Also the architectural features should be considered. This includes the avoidance of branching and global communication, which can be achieved via cellular operators or an island model. These methods are also able to increase the quality of results. The chromosome based memory layout is recommended to increase locality and therefore makes better use of caches.

It's possible to reduce the runtime by an order of magnitude for sufficiently large problems, if a fair comparison between a GPU and a CPU implementation is considered. This corresponds to the ratio of theoretical peak performance of CPUs and GPUs. Therefore speed-ups of 100 or even more are unrealistic and result from comparisons with less optimised CPU implementations. Common mistakes are the use of only one core on multi-core processors or bad access patterns, which reduce the effectiveness of caches.

It's noteworthy that nearly all publications use CUDA for their implementation since it was released. No information could be found on the use of newer frameworks like OpenACC or OpenMP 4.0. This is because of the lack of freely available compilers supporting these standards. At present only commercial compilers offer useful support for these standards. However the GNU Compiler Collection aims to support these too[10].

Overall it can be seen, that only a few publications release their source code. This compromises the traceability and impedes further use or improvement. It also makes more detailed analysis and comparison impossible.

[10] https://gcc.gnu.org/wiki/OpenACC

References

[Be08] Bergman, Keren; Borkar, Shekhar; Campbell, Dan; Carlson, William; Dally, William; Denneau, Monty; Franzon, Paul; Harrod, William; Hiller, Jon; Karp, Sherman; Keckler, Stephen; Klein, Dean; Lucas, Robert; Richards, Mark; Scarpelli, Al; Scott, Steven; Snavely, Allan; Sterling, Thomas; Williams, R. Stanley; Yelick, Katherine: , ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf, 2008.

[Ca12] Cavuoti, Stefano; Garofalo, Mauro; Brescia, Massimo; Pescapâe, Antonio; Longo, Giuseppe; Ventre, Giogio: Genetic Algorithm Modeling with GPU Parallel Computing Technology. Neural Nets and Surroundings, Proceedings of 22nd Italian Workshop on Neural Nets, WIRN 2012, 2012.

[Ch07] Chitty, Darren M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation. volume 2, ACM Press, London, pp. 1566–1573, 2007.

[De09] Debattisti, Stefano; Marlat, Nicola; Mussi, Luca; Cagnoni, Stefano: Implementation of a Simple Genetic Algorithm within the CUDA Architecture. GPUs for Genetic and Evolutionary Computation Competition at 2009 Genetic and Evolutionary Computation Conference, 2009.

[FWW07] Fok, Ka-Ling; Wong, Man-Leung; Wong, Tien-Tsin: Evolutionary Computing on Consumer-Level Graphics Hardware. In: IEEE Intelligent Systems. volume 22, pp. 69–78, 2007.

[Ja12] Jaroš, Jiří: Multi-GPU island-based genetic algorithm for solving the knapsack problem. In: WCCI 2012 IEEE World Congress on Computational Intelligence. IEEE, pp. 1–8, 2012.

[JP12] Jaroš, Jiří; Pospíchal, Petr: A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark. Lecture Notes in Computer Science, 2012(7248):426–435, 2012.

[Li07] Li, Jian-Ming; Wang, Xiao-Jing; He, Rong-Sheng; Chi, Zhong-Xian: An Efficient Fine-grained Parallel Genetic Algorithm Based on GPU-Accelerated. In: 2007 IFIP International Conference on Network and Parallel Computing Workshops. IEEE, Liaoning, pp. 855–862, 2007.

[LL06] Luo, Zhongwen; Liu, Hongzhi: Cellular Genetic Algorithms and Local Search for 3-SAT problem on Graphic Hardware. In: 2006 IEEE Congress on Evolutionary Computation. IEEE, Vancouver, pp. 2988–2992, 2006.

[Nv16] GPU applications. http://www.nvidia.com/object/gpu-applications.html.

[Ow05] Owens, John D.; Luebke, David; Govindaraju, Naga; Harris, Mark; Krger, Jens; Lefohn, Aaron E.; Purcell, Timothy J.: A Survey of General-Purpose Computation on Graphics Hardware. In: Eurographics 2005, State of the Art Reports. pp. 21–51, 8 2005.

[PJ09] Pospíchal, Petr; Jaroš, Jiří: GPU-based Acceleration of the Genetic Algorithm. Genetic and evolutionary computation conference, 2009.

[PJS10] Pospíchal, Petr; Jaroš, Jiří; Schwarz, Josef: Parallel Genetic Algorithm on the CUDA Architecture. In: In Applications of Evolutionary Computation, LNCS 6024. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 442–451, 2010.

[PSJ10] Pospíchal, Petr; Schwarz, Josef; Jaroš, Jiří: Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. In: 16th International Conference on Soft Computing MENDEL 2010. Brno University of Technology, pp. 64–70, 2010.

[RMPF08] Robilliard, Denis; Marion-Poty, Virginie; Fonlupt, Cyril: Population Parallel GP on the G80 GPU. In: Genetic Programming, pp. 98–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Sa11] Salmon, John K.; Moraes, Mark A.; Dror, Ron O.; Shaw, David E.: Parallel Random Numbers: As Easy As 1, 2, 3. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, ACM, New York, pp. 16:1–16:12, 2011.

[SK10] Sanders, Jason; Kandrot, Edward: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, Boston, 2010.

[TO15] Coprozessoren TOP500. http://www.top500.org/statistics/list/.

[WW09] Wong, Man-Leung; Wong, Tien-Tsin: Implementation of Parallel Genetic Algorithms on Graphics Processing Units. In: Intelligent and Evolutionary Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 197–216, 2009.