A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark

Jiri Jaros[1] and Petr Pospichal[2]

[1] The Australian National University, ANU College of Engineering and Computer Science, Canberra, ACT 0200, Australia
jiri.jaros@anu.edu.au
[2] Brno University of Technology, Faculty of Information Technology, Bozetechova 2, 612 66 Brno, Czech Republic
ipospichal@fit.vutbr.cz

Abstract. The paper introduces an optimized multicore CPU implementation of the genetic algorithm and compares its performance with a fine-tuned GPU version. The main goal is to show the true performance relation between modern CPUs and GPUs and eradicate some of myths surrounding GPU performance. It is essential for the evolutionary community to provide the same conditions and designer effort to both implementations when benchmarking CPUs and GPUs. Here we show the performance comparison supported by architecture characteristics narrowing the performance gain of GPUs.

Keywords: GPU, multicore CPU, knapsack, performance comparison, CUDA.

1 Introduction

The Genetic Algorithms (GAs) have become a widely applied optimization tool since developed by Holland in 1975 [4]. Many researchers have shown GA abilities in real-world problems such as optimization, decomposition, scheduling and design. As the genetic algorithms are population based stochastic search algorithms, they often require hundreds of thousands test solutions to be created and evaluated.

One of the advantages of the genetic algorithms is their ability to be easily parallelized. During the last two decades, plenty of different parallel implementations have been proposed, such as island based or spatially structured GAs [19].

The trend over last few years has been to utilize Graphics Processing Units (GPUs) as general purpose co-processors. Although originally designed for rasterization and the game industry, their raw arithmetic power has attracted a lot of research [6], [15].

The evolutionary community has adopted this trend relatively quickly and a lot of papers have been presented in this area, collected e.g. at www.gpgpgpu.com. However, a developer experienced in computer architectures can shortly see that there is something amiss in the state of GA. Most of the papers compare the speedup of the GPU implementation against a sequential version, moreover, mostly implemented in the simplest possible way [8], [13], [16]. This is an exact contradiction with the way

C. Di Chio et al. (Eds.): EvoApplications 2012, LNCS 7248, pp. 426–435, 2012.

the speedup of the parallel processing is defined. G.A. Amdahl in 1967 stated the speedup as the performance of the parallel (GPU) version against the performance of the best known sequential version (an SSE/AVX multi-thread CPU one in the 21st century) [1]. What meaning would it make to accelerate the BubbleSort algorithm on GPUs and compare it with a sequential CPU one both with $O(n^2)$ when a parallel QuickSort with $O(n \log n)$ could be employed?

The main goal of this paper is to show a proper CPU and GPU implementation of the GA, written the ground up taking into account each the architectures features. This paper puts the reached speedups into relation with the architecture performance and discusses the validity of the results. We will simply convince ourselves that there is no way to reach speedups in order of 100 and beyond [7].

The well-known single-objective 0/1 knapsack problem is used as a benchmark. It is defined as follows: given a set of items (L), each with a weight $w[i]$ and a price $p[i]$, with $i = 1,..,L$. The goal is to pick such items that maximize the price of the knapsack and do not excess the weight limit (C) [18]. The solutions that break this limit are penalized according to amount of overweight and the peak price-weight item ratio.

As target architectures we have chosen the leading GPU and CPU on the market, namely the NVIDIA GTX 580 and the Intel Xeon X5650. It does not make any sense for the high performance computing community to compare with desktop CPUs providing that real-word problems require to be run on servers for a long time period.

2 Memory Layout of the GA

The section describes the memory layouts of the population, statistics and global data structures. It is crucial to allocate all host structures intended for host-device transfers by CUDA pinned memory routines. This makes it possible to use the Direct Memory Access (DMA) and reach the peak PCI-Express performance [11]. On the other hand, the host memory should be allocated using the `memalign` routine with 16B alignment when implementing the CPU only version. This helps CPU vector units to load chunks of data much faster and the compiler to produce more efficient code.

2.1 Population Organization

The population of GA has been implemented as a C structure consisting of two one-dimensional arrays. The first array represents the genotype while the second one represents fitness values. Assuming the size of the chromosome is L and the size of the population is N, the genotype is defined as an array[$N*L/32$]. As the knapsack chromosomes are based on the binary encoding, 32 items are packed into a single integer. This rapidly reduces the memory requirements as well as accelerates genetic manipulations employing logical bitwise operations. The fitness value array has the size of N.

Two different layouts of genotype can be found in literature [16]. The first one, referred to as chromosome-based, represents chromosomes as rows of a hypothetical

2D matrix implemented as a 1D array whilst the second one, referred to as gene-based, is a transposed version storing all genes with the same index in one row.

The chromosome-based layout simplifies the chromosome transfers in the selection and replacement phases as well as the host-device transfers necessary for displaying the best solution during the evolutionary process and in more advanced island-base models. In this case, multiple CUDA threads work on one chromosome to evaluate its fitness value. This layout should be preferred also for the CPU implementation in order to preserve data locality and enable the CPU to store chromosomes in the L1 cache and exploit modern prefetch techniques.

On the other hand, the gene-based representation allows working with multiple chromosomes at a time utilizing the SIMD/SIMT nature of CPUs and GPUs assuming there are no dependencies between chromosomes. However, evaluating multiple chromosomes at a time tends to run out of other resources such as registers, cache, shared memories, etc.

Taking into account architecture characteristics, the only thing that matters is to allow threads inside a warp to work on neighbor elements. Different warps can access different memory areas with only a small or no penalization. The chromosome-based layout seems to be the most promising layout enabling the warp to work with the genes of one chromosome, especially, if it is necessary for the fitness evaluation to read genes multiple times. The different warps can simply operate on different chromosomes. This reaches the best SIMD (SSE, CUDA) performance while reducing registers, share memory, and cache requirements. For this reason, the chromosome-based layout is used for both CPU and GPU.

2.2 GA Parameters Storage

A C data structure has been created to accommodate all the control parameters of the GA. Such parameters include the population and chromosome size, the crossover and mutation ratio, the statistics collecting interval, the total number of evaluated generations etc. Once filled in with command line parameters, the structure is copied to the GPU constant memory. This simplifies CUDA kernel invocations and saves memory bandwidth according to the CUDA C best practice guide [11].

2.3 Knapsack Global Data Storage

The knapsack global data structure describes the benchmark listing the price and weight for all items possible included in the knapsack. The structure also maintains the capacity of the knapsack and the item with the maximum price/weight rate. The prices and weights are stored in two separate 1D arrays. The benefit over an array of structures is data locality as all the threads first read prices and only then the weights.

The best memory area where to place this structure may seem to be the constant memory. Unfortunately, this area is too small to accommodate real-world benchmarks. Its capacity of 64KB allows solving problems up to 4K items. On the other hand, introducing L2 caches and a load uniform (`LDU`) instruction in Fermi cards [11] makes the benefits of constant memory negligible supposing all threads within a warp

accesses the same memory location. As the result, the global data are stored in main GPU memory. The problem size (the chromosome size in bits) is always padded to a multiple of 1024 to prevent uncoalesced accesses.

3 Genetic Algorithm Routines

This section goes through the evolution process and comments on the genetic manipulation phase, fitness function evaluation, replacement mechanism and statistics collection. Each phase is implemented as an independent CUDA kernel to put global synchronization between each phase. The source codes can be downloaded from [5].

All the kernels of the GA has been carefully designed and optimized to exploit the hidden potential of modern GPUs and CPUs. It is essential for a good GPU implementation to avoid the thread divergence and to coalesce all memory accesses to minimize the required memory bandwidth. Thus, the key terms here are the **warp** and the **warp size** [11] . In order to write a good CPU implementation, we have to meet exactly the same restrictions. The warp size is now reduced to SSE or AVX width and coalescing corresponds to L1 (L2, L3) cache line accesses while GPU shared memory can be directly seen as the L1 cache.

As the main principles are the same, the CPU implementation follows the GPU one adding only an outer-most `for` cycle and the OpenMP `pragma omp parallel for` sections [2] to utilize all available CPU cores and simulate GPU execution.

3.1 Random Number Generation

As genetic algorithms are stochastic search processes, random numbers are extensively used throughout them. CUDA does not provide any support for on the fly generation of a random number by a thread because of many associated synchronization issues. The only way is to generate a predefined number of random numbers in a separate kernel [10]. Fortunately, a stateless pseudo-random number generator has recently been published based on hash functions [14]. This generator is implemented in C++, CUDA and OpenCL. The generator has been proven to be crush resistant with the period of 2^{128}. The generator is three times faster than the standard C `rand` function and more than 10x faster than the CUDA `cuRand` generator [12], [14].

3.2 Genetic Manipulation Phase

The genetic manipulation phase creates new individuals performing the binary tournament selection on the parent population and exchanging genetic material of two parents using uniform crossover with a predefined probability. Every gene of the offspring is mutated by the bit-flip mutation and stored in the offspring population.

The key for the efficient implementation of the genetic manipulation kernel is a low divergence and enough data to utilize all the CUDA cores. Each CUDA block is organized as two dimensional. The x dimension corresponds to the genes of a single chromosome while the y dimension corresponds to different chromosomes. The size

of the x dimension meets the warp size of 32 to prevent lots of divergence within a warp. The size of y dimension of 8 is chosen based on the assumption that 256 threads per block is enough [15].

The entire grid is organized in 2D with the x size of 1, and the y size corresponding to the offspring population size divided by the double of the y block size (two offspring are produced at once). Since the x grid dimension is exactly the 1, the warps process the individuals in multiple rounds.

The selection is performed by a single thread in a warp. Based on the fitness values, two parents are selected by the tournament and their indices within the parent population are stored in shared memory.

Now, each warp reads two parents in chunks of 32 integer components (one integer per thread). As binary encoding enables 32 genes to be packed into a single integer, the warp effectively reads 1024 binary genes at once. Since this GA implementation is intended for use with very large knapsack instances, uniform crossover is implemented to allow better mixing of genetic material. Each thread first generates a 32b random number serving as the crossover mask. Next, logic bitwise operations are used to crossover the 32b genes. This removes all conditional code from the crossover except testing of the condition whether or not to do the crossover at all. This condition does not introduce any thread divergence as it is evaluated in the same way for the whole warp.

Mutation is performed in a similar way. Each thread generates 32 random numbers and sets the bit of the mask to 1 if the random number falls into the mutation probability interval. After that, the bitwise `xor` operation is performed on the mask and the offspring. This is done for both the offspring. Finally the warp writes the chromosome chunk to the offspring population and starts reading the next chunk.

3.3 Fitness Function Evaluation

The fitness function evaluation kernel follows the same grid and block decomposition as the genetic manipulation kernel. Evaluating more chromosomes at a time allows the GPU to reuse the matching chunk of global data and saves memory bandwidth.

Every warp processes one chromosome in multiple rounds handling a single 32b chunk at a time. In every round, the first warp of the block transfers the prices and weights of 32 items into shared memory employing coalesced memory accesses. After the barrier synchronization, every warp can read the knapsack data directly from shared memory. Now, every warp loads a single 32b chunk into shared memory. As all the threads within a warp access the same memory location (one integer), the L2 GPU cache is exploited. Every thread masks out an appropriate bit of the 32b chunk, multiplies it with the item price and weight, and stores the partial results into shared memory. When the entire chromosome has been processed, the partial prices and weights of the items placed in the knapsack have to be reduced to a single value. Since the chromosome is treated by a single warp a barrier-free parallel reduction can be employed. Finally, a single warp thread checks the total capacity of all the items and if the capacity has been exceeded, the fitness is penalized. Finally, the fitness value is stored in the global memory.

The CPU implementation evaluates chromosomes one by one provided that the global data can be easily stored in L3 cache. The evaluation process is distributed over multiple cores using OpenMP. The evaluation can be carried out immediately after a new offspring has been created which results in the chromosome being evaluated stored in L1 cache. This might also be possible for the GPU implementation, however, the kernel would run out of registers and shared memory resulting in poor GPU occupation and low performance.

3.4 Replacement Phase

The replacement phase employs the binary tournament over the parents and offspring to create the new parent population. The kernel and block decompositions are the same as in the previous phases. The only modification is that the kernel dimensions are derived from the parent population size.

Every warp compares a randomly picked offspring with the parent laying on the index calculated from the *y* index of the warp in the grid. If the offspring fitness value is higher than the parent one, the entire warp is used to replace the parent by the offspring. This restricts the thread divergence to the random number generation phase.

3.5 Statistics Collection

The last component of the genetic algorithm is the class collecting necessary statistics about the evolutionary process. It maintains the best solution found so far, and handing them over the CPU for saving into a log file.

The statistics collection consists of a kernel and statistics structure initialization. The GPU statistic structure maintains the highest and lowest fitness values over the population as well as the sum and the sum-of-squares over of fitness values. The last two values are necessary for calculating the average fitness value and the standard deviation. The last value is the index of the best individual.

The kernel is divided into twice as many blocks as the GPU has stream processors. Each block is decomposed into 256 threads based on the practice published in [15]. After the kernel invocation, the chunks of fitness values are distributed over the blocks. Each thread processes as many fitness values as necessary and stores the partial results into shared memory. After the barrier synchronization, the reductions over highest, lowest and two sum values are carried out. Finally, the first thread of each block uses a global memory lock to modify the global statistics.

After completion, the statistics structure is downloaded to host memory to compute average value and the standard deviation over the fitness values. Finally, the best solution is downloaded from GPU based on the index stored in the statistics structure.

The CPU implementation of the statistics collection has been left in a sequential form because the overhead of parallel execution would exceed the execution time provided by parallel processing.

4 Experimental Comparison of CPU and GPU Implementations

The goal of the experiments is to compare an optimized multicore CPU implementation with a well-designed GPU version and provide some insight into realistically achievable speedups. All the experiments were carried out on a dual Intel Xeon X5650 server equipped with a single NVIDIA GTX 580 running Ubuntu 10.04 LTS.

The knapsack benchmark with 10,000 items and a population size of 12,000 individuals were used. We chose such a big benchmark and large population to show the most optimistic results. The smaller the benchmark and population are, the slower a GPU will be compared to a multicore CPU. This is given by the massively parallel architecture of modern GPUs. Six thousand new individuals are created and evaluated every generation. The genetic algorithm works with tournament selections and replacement, a crossover ratio of 0.7, and a mutation ratio of 0.01. The statistics are collected after every generation. All the proposed codes were compiled using GNU C++ with the highest optimization level, SSE 4.2 support, the OpenMP library [2] and the CUDA 4.0 developer kit [11].

As the reference, we chose the GALib library [20] adopted by a lot of scientists. GALib is a comprehensive rapid prototyping library for evolutionary algorithms, however, the last version comes from 1997. Because of its age, the library cannot benefit from vector units (SSE/AVX) or multiple cores.

In order to validate the optimization abilities of the proposed implementations, we carried out 30 independent runs. The average highest fitness values reached after 100 generations as well as the standard deviation are plotted in Fig. 1. Although there is a statistically significant difference among the implementations, the practical impact on the result quality is negligible (lower than 0.1%).

The performance results are revealing. As a lot of researchers do not pay enough attention to the CPU implementation, the GALib is often compiled under default conditions without any optimization and with debugging support enabled. This degrades the performance (GALib-D) to 375 times slower than the GTX 580. Just a trivial modification of the GALib makefile (turning on appropriate compiler optimizations) can bring a huge performance gain for free. The GALib-O (Optimized) version is 221 times slower than the GPU (see Fig. 2).

Implementing the CPU version carefully rapidly decreases the execution time. The single thread (1T) implementation is 68 times slower than the GPU. This is similar to the speedup reported in many other studies, e.g. [17]. However, parallelization of the 1T version is trivial. It is only necessary to put three OpenMP pragmas in the entire code. The impact on the performance is significant! Running the GA on a single six-core processor reduces the speedup to 11.82 (5.78 faster than 1T). As common HPC servers are equipped with multiple CPU sockets, the dual Xeon5650 server takes only 6 times longer time to perform the task. This is appreciably different to the results reported speedups to 800x, 1000x, 2072x in [13], [16], [8], respectively.

The reason for such a big difference in the CPU codes is shown in Table 1. Three different CPU implementations were investigated using the PAPI performance counter library [9]. The key to the fast CPU code is to utilize cache memories properly.

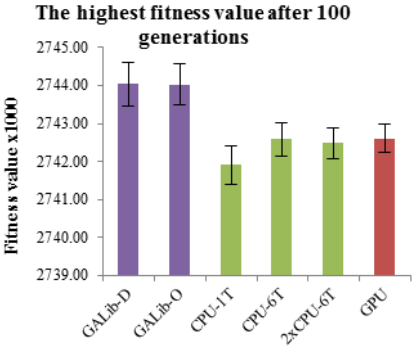

Fig. 1. Solution fitness value after 100 generations achieved by different implementations

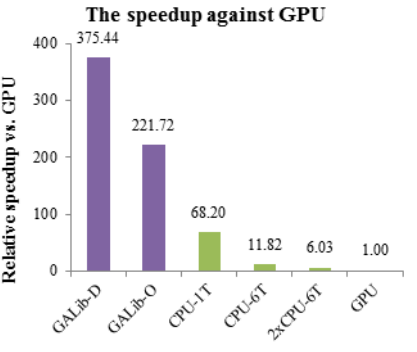

Fig. 2. Speedup comparisons of GPU against different CPU implementations

The first line of the table clearly shows an awful L2 cache hit of the GALib. The problem lays in the way genotype and phenotype are organized, and the immense number of copy constructors employed virtually everywhere. Moreover, from the number of L2 cache accesses we can deduce, the L1 cache is also not exploited (caused by accesses with stride, e.g., when calculating the statistics in GALib). The custom CPU implementation reduces L2 access by factor of 26 (in case of the single thread implementation). The 12 thread implementation further benefits from data distribution over multiple L2 caches. In contrast, the L3 cache hit ratio seems to be better in the case of GALib. However, the number of L3 accesses of GALib is higher by 3 orders which leads to an enormous number of ALU stalls. On the other hand, interpreting the number of cache accesses in the table, the optimized CPU version spends more than 99.99% of time working within the L1 cache.

All these inefficiencies are projected to the CPU performance. GALib only reaches 1,099 MIPS (Million Instructions Per Second). On the other hand, the same six core Xeon running the optimized code reaches about 33,600 MIPS and the entire server can touch up to 67,248 MIPS. We have measured fixed point arithmetic instead of floating point FLOPs here because of the nature of GA encoding and the knapsack prices and weights being encoded as integers.

Given that fixed-point SIMD instructions are nearly as fast as single precision floating-point instructions on the CPU, we can calculate the efficiency of the CPU code. The dual Intel Xeon 5650 server reaches of 118 GFLOPs in the Intel optimized LINPACK benchmark. The overall efficiency of the CPU code is thus about 56%, which is pretty good. As the GPU is about 6 times faster, the peak GPU performance attacks 405 GFLOPS. Although this is roughly one fourth of the theoretical peak performance of the GTX580, it represents a very good result [7]. It should be clear we can never approach the peak performance because of many parallel reductions in the fitness function evaluation and statistics kernel, necessary synchronization and single thread operations inside the genetic manipulation, handing data over to the CPU to store statistics, and many other issues. The peak performance would require all the CUDA cores working in every clock cycle, which is not possible for such operations.

Table 1. PAPI performance counters profililng data of 100k knapack and 100 generations using GALib and the custom implementation on dual Intel Xeon X5650

	GALib-O	CPU-1T	2xCPU-6T
L2 cache hit	18.01%	98.81%	96.82%
L2 accesses	21 267M	800M	24M
L3 cache hit	99.04%	57.07%	81.3%
L3 accesses	17 278M	8M	12M
MIPS	1 099	5 392	67 248
Execution time	165.25s	48.52s	3.81s

In order to compare the 405 GFLOPs of this implementation with other CUDA applications, consider these values measured by SHOC benchmark [3] in single precision: FFT = 213 GFLOPS, GEMM = 529 GFLOPS, parallel reduction = 93 GFLOPS, parallel sort = 2 GFLOPS.

5 Conclusions

This paper points out the way many authors presents the speedups of the GPU implementation of the genetic algorithm against the CPU version. A lot of papers have used only a single thread implementation, [8], [13], [16], [17]. Such authors should have immediately divided their speedups by the factor of 6 at least. Do not forget there is nothing like a single thread CPU on the market any more. As we always need to perform multiple trials to produce good results, we can run as many trials as physical cores with negligible impact on the performance. The trials are embarrassingly parallel. The reason why some authors did not do so might lie in a foolish hunt for the highest speedup, or an attempt to hide the fact the performance gain by a GPU would have been so low that it would not have justified the amount of effort put it into.

The best GPU on the market has the peak performance of 1.5 TFLOPS while a typical server processor reaches the peak of 60 GFLOPs. Confronted with these architecture limits, it is not possible to report speedups of more than 100. Such speed-ups only show that CPU implementation is not well optimized. Fair comparison is to say how fast the implementation is in terms of GFLOPS, and what fraction of the peak performance has been achieved.

Modern GPUs have amazing computational power, and it is worth it porting computationally expensive applications such as evolutionary algorithms onto them. However, we must be careful about making the performance comparisons. We have clearly shown that a carefully implemented CPU version can be up to 30 times faster than a single thread default-compiled GALib. We have also shown that realistically a single NVIDIA GTX580 can outperform an Intel Xeon X5650 by a factor of around 12 while reaching an execution efficiency of 26% and performance of 405 GFLOPS. The proposed CPU and GPU implementations have been released as open source at [5].

Acknowledgement. This research has been partially supported by the research grant "Natural Computing on Unconventional Platforms", GP103/10/1517, Czech Science Foundation (2010-13), and the research plan "Security-oriented research in information technology", MSM 0021630528 (2007-13).

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 1820 1967 Spring Joint Computer Conference, vol. 23(4), pp. 483–485 (1967)
2. Chandra, R., Dagum, L., Kohr, D., et al.: Parallel programming in OpenMP. Morgan Kaufmann (2001)
3. Danalis, A., Marin, G., et al.: The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite Categories and Subject Descriptors. In: Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors, GPGPU 2010 (2010)
4. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press (1975)
5. Jaros, J.: Jiri Jaros's software website, `http://www.fit.vutbr.cz/~jarosjir/prods.php.en`
6. Kirk, D.B., Hwu, W.-M.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
7. Lee, V.W., Hammarlund, P., Singhal, R., et al.: Debunking the 100X GPU vs. CPU myth. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA 2010, p. 451. ACM Press, New York (2010)
8. Luong, T.V.: GPU-based Island Model for Evolutionary Algorithms. Evaluation, 1089–1096 (2010)
9. Malony, A.D., Biersdorff, S., Shende, S., et al.: Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. Performance Computing
10. NVIDIA: CUDA Toolkit 4. 0 CURAND Guide (2011)
11. NVIDIA: Cuda c best practices guide (2011)
12. NVIDIA: Math Library Performance CUDA Math Libraries (2011)
13. Pospichal, P., Schwarz, J., Jaros, J.: Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In: 16th International Conference on Soft Computing MENDEL, pp. 64–70. Brno University of Technology, Brno (2010)
14. Salmon, J.K., Moraes, M.A., Dror, R.O., Shaw, D.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 2011, pp. 16:1–16:12. ACM Press, New York (2011)
15. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley (2010)
16. Shah, R., Narayanan, P., Kothapalli, K.: GPU-Accelerated Genetic Algorithms, cvit.iiit.ac.in
17. Simonsen, M., Pedersen, C.N.S., Christensen, M.H.: GPU-Accelerated High-Accuracy Molecular Docking using Guided Differential Evolution. In: Proceedings of the Genetic and Evolutionary Computation Confernce GECCO 2011. ACM Press (2011)
18. Simões, A., Costa, E.: An Evolutionary Approach to the Zero / One Knapsack Problem Testing Ideas from Biology. In: The Fifth International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA 2001), April 22-25 (2001)
19. Tomassini, M.: Spatially Structured Evolutionary Algorithms. Springer, Heidelberg (2005)
20. Wall, M.: GAlib: A C ++ Library of Genetic Algorithm Components. Statistics (August 1996)