Multi-GPU Island-Based Genetic Algorithm

Namrata Mahakalkar & A. R. Mahajan

Department of Computer Science and Engineering, P.I.E.T, Nagpur

Abstract – Genetic algorithms are effective in solving many optimization tasks. However, the long execution time associated with it prevents its use in many domains. In this paper, we propose a new approach for parallel implementation of genetic algorithm on graphics processing units (GPUs) using CUDA programming model. This paper introduces a novel implementation of the genetic algorithm exploiting a multi-GPU cluster. The proposed implementation employs an island-based genetic algorithm where every GPU evolves a single island. Recently there has been a growing interest in developing parallel algorithms using graphic processing units (GPU) also referred as GPU computation. Advances in the video gaming industry have led to the production of low-cost, high performance graphics processing units that possess more memory bandwidth and computational capability than central processing units (CPUs). As GPUs are available in personal computers, and they are easy to use and manage through several GPU programming languages, graphics engines are being adopted widely in scientific computing.

Keywords – GPU; CUDA; GA; island model

I. INTRODUCTION

The hardware developments from recent years have made multi-core architectures a common place in the industry. Taking advantage of such platforms is now the issue we are facing.

Parallel and distributed versions of the genetic algorithms (GAs) are popular and diverse. The simplest parallel models are function-based where the evaluation of the fitness function is distributed among the processes [5]. The parallel models in GA are population-based where the population itself is distributed and migration takes place. Since they resemble the 'island model' in population genetics also called islands. In Island model periodic migration of individuals between the sub-populations is done.

Genetic Algorithms (GAs) are a set of evolutionary algorithms, powerful and effective in solving search and optimization tasks. Such algorithms are inspired by the process of biological evolution. Genetic algorithms employ natural selection, crossover, mutation and survival of fittest to find the fittest solution in a search space represented by a population of chromosomes, where each chromosome represents one possible solution to the optimization problem [1].

Genetic algorithm starts with selecting random points in search space, representing them as chromosomes and building an initial population. The initial population is evaluated using a fitness function to suggest how fit a chromosome is to represent the solution. A fitness-based or uniform selection is carried out to select parent chromosomes to undergo crossover and produce offsprings,

In this paper, we present a generic framework for Genetic Algorithms accelerated by the modern Graphics Processing Units (GPUs), inspired by GALib. Such a framework not only provides a platform for fast execution but encourages experiments in new domains and with novel approaches involving huge population sizes which was limited due to impractical execution times. The key distinction of the approach is the effort to go beyond chromosome level parallelism whenever possible and utilize the massively multithreaded model of GPUs to its fullest. Our approach is implemented using Nvidia's CUDA programming model (NVIDIA [4]), but with enough isolation from the user program so that users need not be proficient in CUDA programming. We implement the original genetic algorithms and achieve a speed up of about 1500 on large problems using the massively multithreaded model of the GPU as exposed by CUDA.

Compute Unified Device Architecture (CUDA) [3] and Open Compute Language (OpenCL) [2] tools are developed in order to use Graphics Processing Units (GPUs) for general purpose computing have prompted another revolution in high-end computing. Although these chips were designed to accelerate rasterisation of graphic primitives, their raw computing performance has attracted a lot of researchers to use them as acceleration units for many scientific applications [6]. Compared to a CPU, the latest GPUs are about 15 times faster than six-core Intel processors in single-precision floating point operations [7]. A cluster with a single GPU per node offers the equivalent performance of a 15 node CPU-

only cluster. The goal of this paper is to utilize a cluster of NVIDIA GPUs to accelerate the GA and properly compare the execution time with a CPU cluster. In order to utilize multiple GPUs, we propose an island based GA where a single island is completely evolved on a single GPU. All necessary inter-island data transfers such as migration of individuals and global statistics collection are performed by means of message passing routines (OpenMPI).

II. GPU ARCHITECTURE

Our GPU cluster is based on NVIDIA GTX 580 cards, so we implemented the algorithms in CUDA 4.0 [3]. The CUDA hardware model of the GTX 580 is shown in Fig. 1. The graphics card is divided into a graphics chip (GPU) and 1.5GB of main memory. Main memory, acting as an interface between the host CPU and GPU, is connected to the host system using a PCI-Express 2.0 bus. The bandwidth of the bus is only a fraction of what both GPU and CPU memories provide, so it can become a bottleneck [8].

Hiding this latency is very important for keeping GPU executions units busy.GPU main memory is optimized for block transactions and stream processing providing very high bandwidth but also high latency. The GTX 580 offers 768KB of fast onchip L2 cache to allow reordering of the memory requests and on-chip shared memory, and large register fields so that the working data is close to execution units as possible.

SMs are based on the Single Instruction, Multiple Thread (SIMT) concept allowing them to execute exactly the same instruction over a batch of 32 consecutive threads (referred to as a warp) at a time. This concept dramatically reduces the control logic of SMs, but on the other hand, dictates strict rules on thread cooperation and branching. The GTX580 processor consists of 16 independent Streaming Multiprocessors (SM), each of which is further divided into 32 CUDA cores. A few consecutive warps form a thread block that is the smallest resource allocation unit per SM.

In order to fully exploit the potential of a given GPU, a few concepts must be kept in mind [9] that thousands of threads are necessary to be executed concurrently on the GPU to hide memory latency and all the threads within a warp should follow the same execution path minimizing the thread divergence. The memory requests within a warp should be coalesced reading data from consecutive addresses. Synchronization and/or communication among threads can be done quickly only within a thread block.

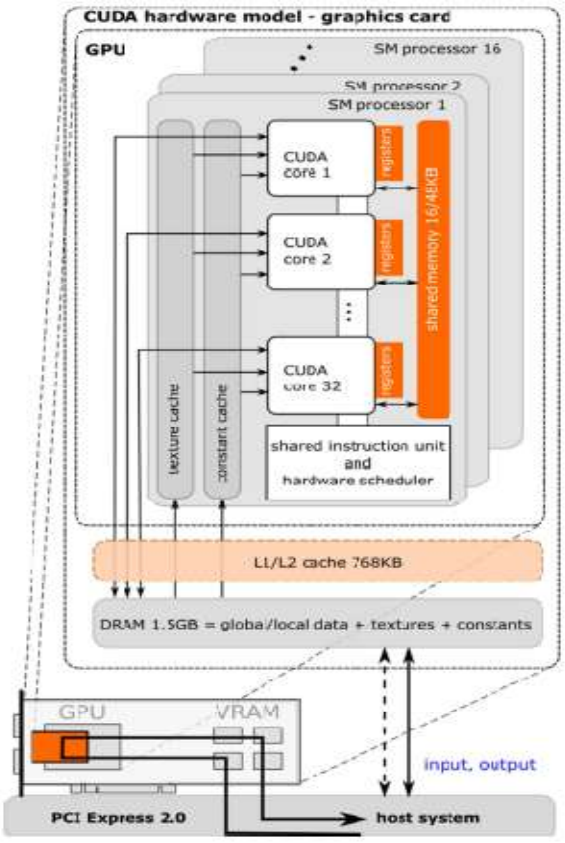

Fig. 1 : CUDA hardware model of NVIDIA GTX 580

Even the working data set should be partitioned to fit on-chip shared memory to minimize main memory accesses. Data transfers between CPU and GPU memories can easily become a bottleneck given the low PCI-Express bandwidth.

III. GA DESIGN POSSIBILITIES

There have been several attempts to accelerate the genetic algorithm on the massively parallel GPU architecture. Many researchers have taken only simple numeric benchmarks without any global data or with only a very limited data set [10], [11]. It is in contradiction to the real world problems, where big simulations have to be carried out and the fitness evaluation is often the most time-consuming operation.

The individual creation and fitness evaluations can be performed independently. A master-slave GA can be used here, i.e. candidate solutions are evolved on the CPU side and transferred to a GPU or GPUs only for evaluation. The fitness value evaluation takes a few orders of magnitude more time than genetic manipulation phase to overlap and hide slow PCI-Express transfers and CUDA kernels launch overhead is the fundamental prerequisite. etc. Figure 1. CUDA hardware model of NVIDIA GTX 580. In cases where the fitness evaluation takes a comparable time to the rest of GA, it is usually better to execute the entire GA on the GPU. Recently, a few papers have investigated this

possibility. The key here is the distribution of the individuals over SMs. Some approaches assign an individual per thread [10], [11], others assign an individual per thread block. Both approaches have their limits. Assigning a single individual per thread always leads to thousands of individuals per GPU. This is counterproductive for some kind of evolutionary algorithms that work with small populations. The bigger problem arising from this is a per block resource limitation introduced by CUDA [9]. An SM can accommodate up to 1536 threads which gives us 21 registers and 32B of shared memory per thread. This makes it very difficult to implement complex fitness functions and deal with long chromosomes. On the other hand, assigning an individual per thread block requires really long chromosomes to give employment to all the threads, or the genes have to be read multiple times to perform a complex simulation. The other approach employs an SM-based island model storing an island in shared memory[12]. Although very popular, this model does not scale at all, and can be used only for a low dimension numerical optimization as the product of the individual length and population size has to fit SM shared memory constrains (e.g. 128 individuals, 96 float genes long, no other working or temporary data) As far as the author knows, nobody has attempted to use per warp assignment although it seems be a sweet spot. Warp granularity requires a reasonable number of individuals and using appropriate mapping does not limit the size of individual. Moreover, it virtually eliminates the thread divergence. In this paper, a multi-GPU island-based model based on warp granularity is proposed. The population of this GA is distributed over multiple GPUs. Every GPU is controlled by a single MPI process [13], and entirely evolves a single island using a steady-state approach with elitism, uniform crossover, bit flip mutation, and tournament selection and replacement. Migration of individuals occurs after a predefined number of generations exchanging the best local solution and an optional number of randomly selected individuals. The tournament selection is used to pick emigrants and incorporate immigrants. The migration is performed along the unidirectional ring topology where only adjacent nodes can exchange individuals [9]. All the data exchanges are implemented by means of MPI [13]. The identical GA is also implemented in C++ to compare the multi-GPU implementation with a multi-CPU one.

IV. MULTI-GPU GENETIC ALGORITHM IMPLEMENTATION

The genetic manipulation phase, fitness evaluation, replacement mechanism, migration phase, and statistics collection phases have been described in this section. Each phase is implemented as an independent CUDA kernel to put the necessary global synchronization between each phase.

All the CUDA kernels are optimized to exploit the hidden potential of modern GPUs and CPUs. For a good GPU implementation, it is essential to avoid thread divergence and coalesce all memory accesses to maximize GPU utilization and minimize required memory bandwidth. The key terms here are the warp and the warp size . In order to write a good CPU implementation, we have to meet exactly the same restrictions. The warp size is now reduced to SSE (AVX) width and GPU shared memory can be imagined as CPU cache memory.

As the main principles are the same for both CPU and GPU, the CPU implementation follows the GPU one. Besides the island-based CPU implementation, a single population multithread GA was developed to compare the performance of a single GPU with a multicore CPU. The multithreaded version only adds simple OpenMP pragma clauses [13] to distribute genetic manipulation and evaluation and the loop iterations (individuals) over available CPU cores.

A. Random Number Generation

As GAs 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 [3]. Fortunately, a stateless pseudo-random number based on a hash function generator has recently been published [16]. 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 generator [14].

B. Genetic Manipulation Phase

The genetic manipulation process creates new individuals by performing the binary tournament selection on the parent population, exchanging genetic material of two parents using uniform crossover, applying a bit-flip mutation, and storing them in the offspring population. All CUDA thread blocks are organized as two dimensional. The *x* dimension corresponds to the genes within chromosomes 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 warps. The size of the *y* dimension is chosen as 8 based on the assumption that 256 threads per block is an appropriate block granularity [15]. Thus, 8 independent warps of 32 threads run simultaneously. The entire CUDA 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* thread block size (two offspring are produced at once). Since the *x* grid dimension is exactly 1, the warps process the individuals in multiple rounds. The grid can be seen as a parallel pipeline processing a piece of multiple individuals at once.

Every warp is responsible for generating two offspring. 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 a shared memory array. As only a single warp works on an individual and all the threads are implicitly synchronous within a warp, there is no need to use a barrier. Setting the indices array as volatile rules out any read/write conflict. This prevents independent warps from waiting for each other and allows better memory latency hiding. Once the parents have been selected the crossover phase starts. Every 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 (1). It 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

```
Child_1 = (~Mask & Parent_1) | ( Mask & Parent_2)
Child_2 = (Mask & Parent_1) | (~Mask & Parent_2)
```

(1)

Mutation is performed in a similar way. Every thread generates 32 random numbers and sets the *i*-th bit of the mask to one if the *i*-th random number falls into the mutation probability interval as shown in (2). After that, the bitwise xor operation is performed on the mask and the offspring. This is done for both offspring. Finally the warp writes the chromosome chunk to the offspring population and starts reading the next chunk.

```
Child_1 ^= (RandomValue[i] < MutationPst) << i
```

(2)

C. 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 enables the reuse of matching chunks 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 thread block transfers the price and weight values of 32 items into shared memory employing coalesced memory accesses. Barrier synchronization is necessary because of sharing data among multiple warps (the entire thread block). Next, every warp loads a single 32-bit chromosome chunk into shared memory. As all the threads within a warp access the same memory location (single integer), L2 GPU cache is exploited. Now, every thread masks out an appropriate bit of the chromosome chunk, multiplies it with the corresponding item price and weight, and add both results to the private partial sums stored in shared memory. When the entire chromosome has been processed, the partial prices and weights are reduced to single values. Since the chromosome is treated by a single warp, a barrier-free parallel reduction can be employed. After that, the first thread of the warp checks the total weight against the knapsack capacity and if the capacity has been exceeded, the fitness value (total price) is penalized. Finally, the fitness value is stored in the global memory by this thread. The CPU implementation evaluates chromosomes one by one. The fitness evaluation can be carried out immediately after the new offspring has been created which results in the chromosome evaluated 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 occupancy and low performance.

D. Replacement Phase

The replacement phase employs the binary tournament over the parents and offspring to create the new parent population. The kernel and blocks 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 only restricts the thread divergence to the random number generation phase. This replacement schema also abides by the elitism because it is not possible to override the best individual by a worse one.

E. Migration Phase

The migration process enables distributed islands to exchange good individuals among them. The migration scheme is based on the unidirectional ring topology where every island sends its migrants to the adjacent island with a higher index and receives migrants from the island with a lower index. The migration phase consists of three stages:

1) Selection of Emigrants:

First, a CUDA kernel is called to select a predefined number of migrating individuals and put them into a new population placed in GPU main memory. The same selection mechanism as in the case of the genetic manipulation phase is employed. In order to ensure that the best individual has also been selected, the first warp always selects this one and put it to the migration population.

2) Transferring Migrants to Another Island:

After the migrants have been selected, it is necessary to download them to the CPU memory. This is done by means of two PCEExpress transfers; first one

for individual genomes and the second one for their fitness values. After that, two OpenMPI non-blocking send routines are employed to dispatch the migrants. Concurrently, two non-blocking OpenMPI receive routines have been waiting for the migrants from an adjacent island. Note that with the upcoming version of OpenMPI and CUDA, it will be possible to skip CPU-GPU transfers and transfer the data directly from/to GPU memory [18].

3) Incorporation of Immigrants:

After new immigrants have been received, they are stored in a CPU memory buffer. First, these individuals have to be uploaded to a GPU using two PCI-Express transfers. After that, a kernel merging immigrants and the primary island population is invoked.

Every warp processes a single immigrant and compares it with a randomly selected one from the primary population. This approach gives every immigrant an opportunity to get into the primary island population. A problem arises if two warps picked the same individual to be replaced. This leads to the racing condition and data inconsistency. In order to prevent this, a critical section has to be entered before writing anything into the primary population. Thus, the first thread of the warp selects an individual from the primary population, locks this individual and saves its index into shared memory.If the immigrant has a higher fitness value, the entire warp is used for genome replacement. Finally, the first thread unlocks the lock. If there is another warp trying to acquire this individual, it will succeed after the previous replacement has been finished. As better solutions always replace worse ones, the elitism is guaranteed.

F. Global Statistics Collection Phase

The last component of the genetic algorithm is a module collecting the global statistics over all the islands. This module maintains the best solution found so far, and the basic statistics such as the highest, lowest and average fitness value over all the islands, the standard deviation of the fitness values, and index of the best island. The statistics collection can be divided into two phases. First, local island statistical data are collected and then a global gathering process over all the islands is carried out. The local statistics collection phase first initializes an auxiliary structure on the GPU and then launches a data collection kernel. The kernel is divided into twice as many blocks as the GPU has SM processors. Each block is decomposed into 256 threads based on the practice published in [15]. After the kernel invocation, the chunks of fitness values array are distributed over the blocks. Each thread processes as many fitness values as necessary and stores its partial results into shared memory.After completion, the statistical data as well as the best individual are downloaded to CPU main memory. These data are packed and sent off to the master process (island with index 0). The master process collects local island statistics and the best solutions using two MPI gather routine, merge them together and store them into a log file.

V. RESULTS

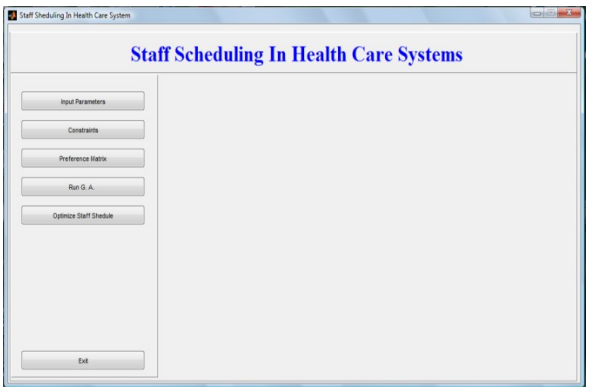

Fig. 1: GUI Window for NSP

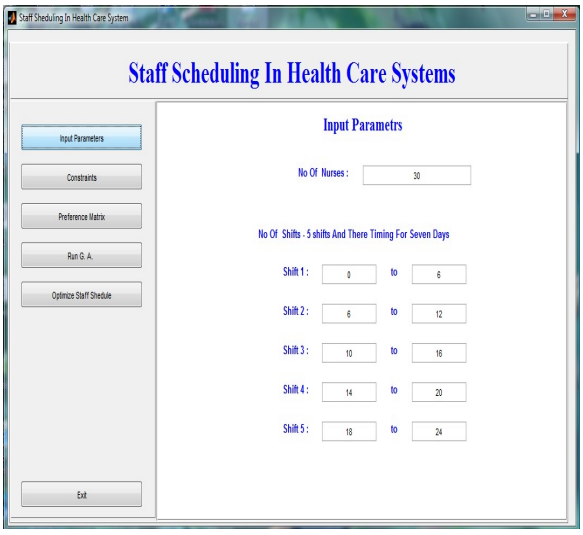

Fig. 2: GUI Windows when input parameter button is clicked

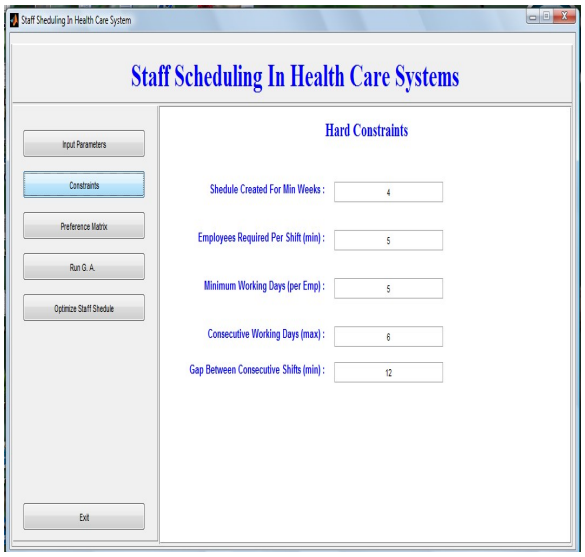

Fig. 3: GUI Windows when constraints button is clicked

Similarly, different buttons are clicked and the output is viewed in the GUI window according to the requirements of the user. GUI helps in reducing the time involved for solving mathematical expressions and gives a perfect pattern for presenting the problem along with the solution.

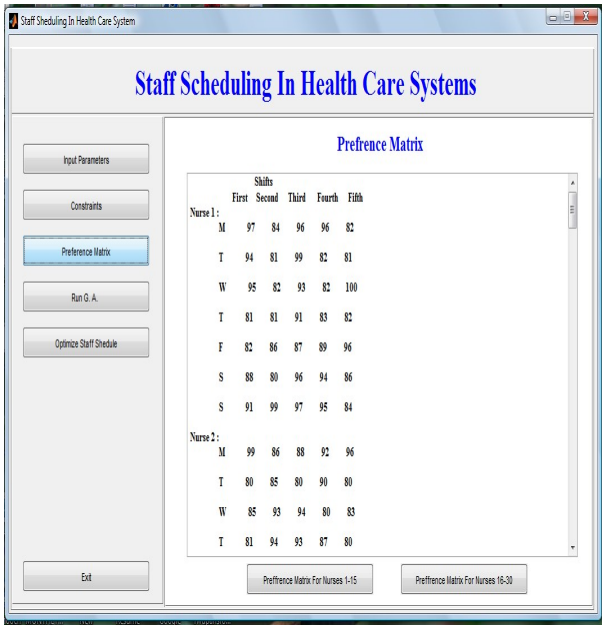

Fig. 4: GUI windows when preference matrix button is clicked

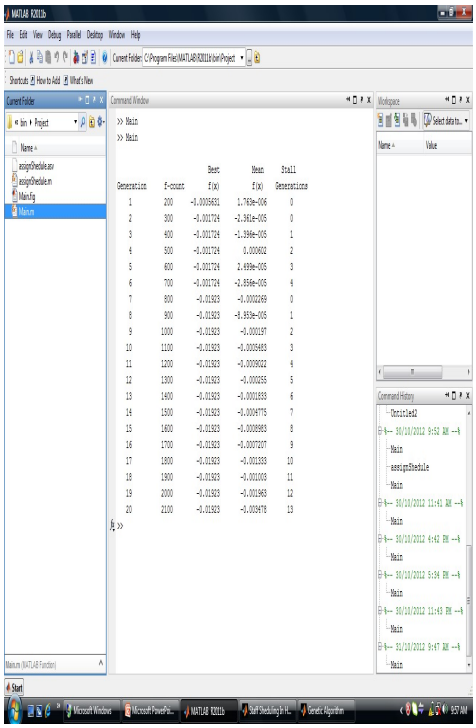

Fig. 5 : GUI when Run GA button is clicked

VI. CONCLUSIONS

A new implementation of island based genetic algorithm exploiting a multi-GPU cluster is proposed in this paper. Every island is evolved on a single GPU where the individuals are assigned one per warp that reduces the thread divergence below 0.5% and does not restrict the individual or population size. The speedup is increased. The overall performance reached by all GPUs is very good. The results show that an optimized result is obtained while applying GA till the stall generation gets constant.

VII. REFERENCES

[1] J. Holland. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT Press, Cambridge Mass., 1st MIT press ed. edition, 1992.

[2] Krhonos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems," 2012.[Online]:Available: http://www.khronos.org/opencl/. [Accessed: 10-Jan-2012]

[3] NVIDIA, "CUDA Toolkit 4.0," 2011. [Online]. Available: http://developer.nvidia.com/cuda-toolkit-40. [Accessed: 10-Jan-2012].

[4] NVIDIA. NVIDIA CUDA Programming Guide Version3.0. NVIDIA Corporation, 2010.

[5] Y. Sato, M. Namiki, and M. Sato, "Acceleration of genetic algorithms for Sudoku solution on many-core processors", in Proceeding of the Genetic and Evolutionary Computation Conference, pp. 407-414, Dublin, IrelandOR, 2011.

[6] D. B. Kirk and W.-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann,, 2010, p. 280.

[7] V. W. Lee et al., "Debunking the 100X GPU vs. CPU myth," in Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10, 2010, p. 451.

[8] J. Jaros, B. E. Treeby, and A. P. Rendell, "Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations," in Proceedings of Australasian Symposium on Parallel and Distributed Computing (AusPDC), 2012, p. 10.

[9] NVIDIA, "Cuda c best practices guide," 2011.

[10] L. Zheng, Y. Lu, M. Ding, and Y. Shen, "Architecture-based Performance Evaluation of Genetic Algorithms on Multi/Many-core Systems," Science and Engineering, pp. 321-334, Aug. 2011.

[11] P. Vidal and E. Alba, "A multi-GPU implementation of a Cellular Genetic Algorithm," in IEEE Congress on Evolutionary Computation, 2010, pp. 1-7.

[12] P. Pospichal, J. Schwarz, and J. Jaros, "Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu," in 16th International Conference on Soft Computing MENDEL, 2010, no. 1, pp. 64–70.

[13] Indiana University, "OpenMPI: Open Source High Performance Computing," 2012. [Online]. Available: http://www.open-mpi.org/. [Accessed: 09-Jan-2012].

[14] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel Random Numbers: As Easy as 1, 2, 3," in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11, 2011, pp. 16:1-16:12.

[15] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, 2010, p. 279.

[16] N. Corporation, "NVIDIA GPUDirect TM Technology NVIDIA GPUDirect TM : Eliminating CPU Overhead," 2011.

[17] Jiri Jaros ,"Multi-GPU Island-Based Genetic Algorithm for Solving the Knapsack Problem" ,in WCCI IEEE World Congress on Computational Intelligence, - Brisbane, Australia,June, 10-15, 2012.

◈◈◈