WCCI 2012 IEEE World Congress on Computational Intelligence
June, 10-15, 2012 - Brisbane, Australia IEEE CEC

Multi-GPU Island-Based Genetic Algorithm for Solving the Knapsack Problem

Jiri Jaros
ANU College of Engineering & Computer Science
The Australian National University
Canberra, ACT, Australia
jiri.jaros@anu.edu.au

***Abstract*—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. The individuals are processed by CUDA warps, which enables the solution of large knapsack instances and eliminates undesirable thread divergence. The MPI interface is used to exchange genetic material among isolated islands and collect statistical data. The characteristics of the proposed GAs are investigated on a two-node cluster composed of 14 Fermi GPUs and 4 six-core Intel Xeon processors. The overall GPU performance of the proposed GA reaches 5.67 TFLOPS.**

Keywords-GPU; CUDA; MPI; GA; island model; knapsack

I. INTRODUCTION

In 1994 Becker and Sterling [1] proposed the construction of supercomputer systems through the use of off-the-shelf commodity parts and open source software. Over the ensuing year, the so called Beowulf cluster computer systems came to dominate the top 500 list [2] of most powerful systems in the world. The advantages of such systems are many, including ease of creation, administration and monitoring, and full support of many advanced programming techniques and high performance computing libraries. Interestingly, however, what was originally a major advantage of these systems, namely price and running costs, is now much less so. This is because for even a small to moderately sized cluster it is necessary to house the system in specially air-conditioned machine rooms.

Recently, tools like Compute Unified Device Architecture (CUDA) [3] and Open Compute Language (OpenCL) [4] developed in order to use Graphics Processing Units (GPUs) for general purpose computing have prompted another revolution in high-end computing, equivalent to that of the original Beowulf cluster concept. 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 [5]. Compared to a CPU, the latest GPUs are about 15 times faster than six-core Intel processors in single-precision floating point operations [6]. Stated another way, a cluster with a single GPU per node offers the equivalent performance of a 15 node CPU-only cluster. Even more interestingly, the availability of multiple PCI-Express buses, even on very low cost commodity computers, means that it is possible to construct cluster nodes with multiple GPUs. Under this scenario, a single node with multiple GPUs offers the possibility of replacing fifty or more CPU-only nodes.

On the other hand, the development tools for debugging and profiling of GPU-based applications are in their infancy. Obtaining peak performance for many real-world problems is very difficult and sometimes impossible. Moreover, only a few basic GPU libraries such as LAPACK and BLAS have so far been developed, and these are only able to utilize one GPU in a node [7]. GPU-based applications are also limited by the GPU architecture and memory model making general-purpose computing much more difficult to implement than the CPU-based ones [5].

The Genetic Algorithms (GAs) have become a widely applied optimization tool since their development by Holland in 1975 [8]. Many researchers have shown the capabilities of GAs in many real-world problems such as optimization, decomposition, design and scheduling [9]. As GAs are population-based stochastic search algorithms, they often require millions of candidate solutions to be created and evaluated. The execution time can then easily come up to the order of days or weeks [10]. The considerable advantage of GAs is their ability to be easily parallelized in many different ways. During the last two decades, many different parallel and distributed schemes have been proposed, such as island based or spatially structured GAs [11].

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). The well-known single-objective 0/1 knapsack problem [12] with 10,000 items is used as a benchmark of the CPU and GPU-based GA implementations.

II. GPU ARCHITECTURE AND CUDA TOOLKIT

As our GPU cluster is based on NVIDIA GTX 580 cards, 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

This research has been partially supported by the research grant "Natural Computing on Unconventional Platforms", GP103/10/1517, Czech Science Foundation (2010-13).

U.S. Government work not protected by U.S. copyright

a PCI-Express 2.0 bus. This bus can easily become a bottleneck as its bandwidth is only a fraction of what both GPU and CPU memories provide [13].

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

The GTX580 processor consists of 16 independent Streaming Multiprocessors (SM), each of which is further divided into 32 CUDA cores. 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. 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 [14]:

- Thousands of threads are necessary to be executed concurrently on the GPU to hide memory latency.
- All the threads within a warp should follow the same execution path minimizing the thread divergence.
- All 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.
- 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. GPU-BASED GA DESIGN POSSIBILITIES

Recently, there have been several attempts to accelerate the genetic algorithm on the massively parallel GPU architecture. However, many researchers have taken only simple numeric benchmarks without any global data or with only a very limited data set [15], [16], [17]. This 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 for each individual in the population. 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 [18]. The fundamental prerequisite of this approach is that the fitness value evaluation takes a few orders of magnitude more time than genetic manipulation phase to overlap and hide slow PCI-Express transfers, CUDA kernels launch overhead, etc.

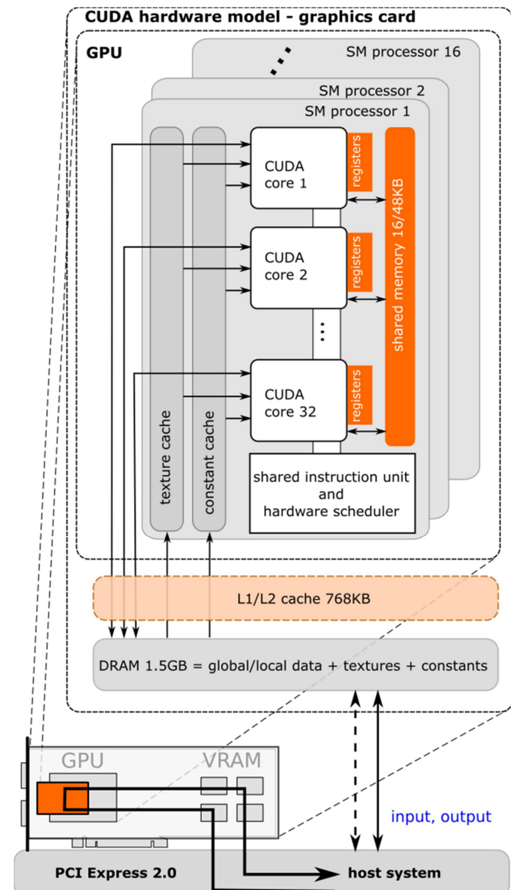

Figure 1. CUDA hardware model of NVIDIA GTX 580.

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 [15], [16], [19], others assign an individual per thread block [20]. 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 [14]. 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 [19]. 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 [21], 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 [21]. The identical GA is also implemented in C++ to compare the multi-GPU implementation with a multi-CPU one.

IV. Memory Layout of the Proposed GA

The memory layouts of the population, statistics and global data structures are carefully designed. All the host (CPU side) structures intended for host-device transfers (migration buffers, global statistics) are allocated by CUDA pinned memory routines. This allows to use the Direct Memory Access (DMA) and reach the peak PCI-Express bandwidth [13], [14].

On the other hand, the host memory is allocated using the `memalign` routine with 16B alignment when implementing the CPU-only version. This helps CPU vector units (SSE) to load chunks of data much faster and the compiler to produce more efficient code [22].

A. Population Organization

The population of GA is implemented as a C structure consisting of two one-dimensional arrays. The first array represents the genotype whereas the second one 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 of size of $N * \lceil L/32 \rceil$. As the knapsack chromosomes are based on 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 the literature [20], see Fig 2. The first one, referred to as chromosome-based, represents chromosomes as rows of a hypothetical 2D matrix implemented as a 1D array. The second layout, referred to as gene-based, is a transposed version storing corresponding genes of all chromosomes in one row.

The chromosome-based layout simplifies the individual movements in the selection and replacement phases as well as the host-device transfers necessary for the migration phase and displaying the best solution during the evolutionary process. 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

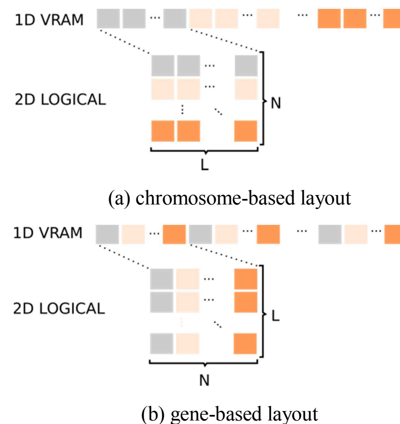

Figure 2. Different layouts of the population in GPU Video RAM.

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.

The key to reach peak GPU performance is to allow threads within a warp to work on neighbour elements and avoid control flow divergence. Different warps can access different memory areas with only a small or no penalization. The chromosome-based layout appears to be the most promising layout enabling the warp to work with the genes of one chromosome, especially if it is necessary for the benchmark fitness evaluation to read genes multiple times. The different warps can simply operate on different chromosomes. This reaches the best SIMD/SIMT performance while reducing registers, shared memory, and cache requirements. For this reason, the chromosome-based layout is used in this work.

B. GA Parameters Storage

The GA control parameters are maintained by a C structure. Such parameters include the population and chromosome size, the crossover and mutation rates, the statistic collection and migration 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 [14].

C. 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 an item with the maximum price/weight rate used for penalization. The prices and weights are stored in two 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 4000 items. On the other

hand, introducing L2 caches and a load uniform instruction in Fermi cards makes the benefits of constant memory negligible [3]. As a 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 not coalesced memory accesses while working with chromosomes.

V. MULTI-GPU GENETIC ALGORITHM IMPLEMENTATION

This subsection goes through the GA and describes the genetic manipulation phase, fitness evaluation, replacement mechanism, migration phase, and statistics collection. 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 [14]. 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 [21] to distribute genetic manipulation and evaluation and the loop iterations (individuals) over available CPU cores [23].

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 [24]. Fortunately, a stateless pseudo-random number based on a hash function generator has recently been published [25]. 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 [25], [7].

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 as outlined in Fig. 3. 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 [26]. 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, see (1). 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.

```
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.

Fig. 3 illustrates the kernel structure. 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

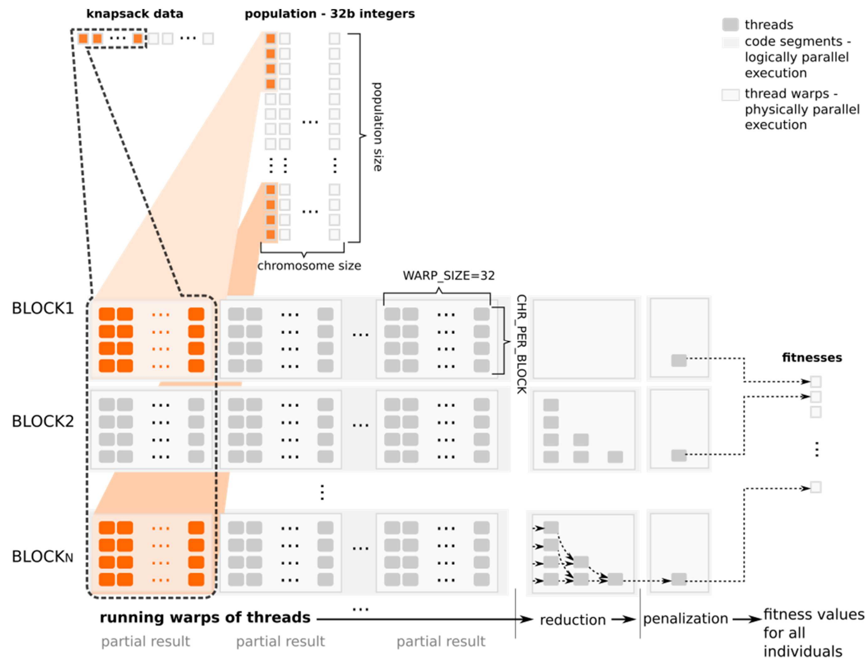

Figure 3. Design of the knapsack fitness function executed by the GPU. The technique is repeated in other GPU kernels.

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 PCE-Express transfers; first one for individual genomes and the second one for their fitness values. After that, two OpenMPI non-blocking send [21] 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 [27].

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. The critical section is guarded by a vector lock where every individual in the primary population has its own entry.

Thus, the first thread of the warp selects an individual from the primary population, locks this individual and saves its index into shared memory. There is no need for synchronization here, because of the SIMT nature of warps. Now, all the warp threads read this index and make the decision whether or not to replace the individual by the corresponding immigrant. 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 [26]. 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 the barrier synchronization, the reductions over statistical data within a thread block are carried out. Finally, the first thread of each thread block uses a global memory lock to update the local island statistics.

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, merges them together and stores them into a log file.

VI. Experimental Results

All the proposed algorithms were implemented and tested on two TYAN servers [28]. Both servers are equipped with two six-core Intel Xeon X5650 processors at 2.6 GHz, 24GB RAM memory, 7 NVIDIA GTX 580 cards, 40Gb infiniband network interface, and running Ubuntu 10.04 server operating system.

A knapsack instance with 10,000 items was chosen as a test case simulating a real-world problem with a reasonable large global data set. GA control parameters were experimentally set as follows: the crossover probability of 0.7, the gene mutation probability of 0.001, the migration interval of 100 generations, 10% of individuals migrate including the best individual, and 50% of old population is replaced. The population size varied from 128 to 2048 individuals per island. Different GPU configurations were tested. Six and twelve GPU islands were used to have an analogy to a single six-core CPU and two six-core CPUs in a server, respectively. Seven and fourteen GPUs represent the maximum configuration of a single or both servers, respectively. All results shown here represent averaged values after 100k generations over 30 independent runs and 95% confidence intervals.

Fig. 4 shows the quality of the global best solution evolved after 100k generations. The curves show the improvement of the solution quality when increasing the island population size and evolving a higher number of islands. A significant quality leap can be observed between 1 and 14 islands. A considerable improvement is also present between 1 and 7, as well as, 7 and 14 islands. Although differences in solution quality using 6 and 7, or, 12 and 14 GPUs are not very high, they are still statistically significant (based on 95% confidence interval). As mentioned in [15], distribution of a large population over smaller islands leads to slight quality drops. These drops are mainly present for small island sizes (e.g. compare 1 GPU with 1024 individuals and 7 GPUs with 128 ones).

Fig. 5 shows the execution time of the proposed multi-GPU island based GA. All the curves represent different numbers of islands merged together. This implies negligible overhead of the inter-GPU migration process that is detailed in Fig. 7. The most important observation is that the execution time remains constant for island sizes up to 512. Such small islands cannot saturate these GPUs. Assuming 50% of individuals are created every generation, and a warp processes a single individual, the GTX 580 needs at least 256 warps (32 blocks) to be saturated. This exactly meets the necessity of running twice as many blocks as there are SMs in a given GPU published in [26]. Beyond 512 individuals per island, the execution time starts to grow linearly.

Fig. 6 reveals the overhead of the migration process including selection and incorporation of migrants, PCI-Express transfers and OpenMPI communication routines. With the island size of 2048, the overhead is almost negligible. This is given by appropriate migration parameters as well as the 40Gb infiniband interconnection and fast memory transfers. A bigger difference is believed to appear using the 1Gb Ethernet interconnection with more frequent migration.

Fig. 7 compares the execution time of a single island GA evolved using 6 and 12 CPU threads and GTX580 with respect to a single thread CPU implementation. CPU implementations show speedup corresponding to the level of parallelism (5.67 and 11.32 for 6 and 12 threads, respectively). The reported speedups are slightly lower than the theoretical values, limited by necessary synchronization and the NUMA architecture [28]. The single island GPU implementation reaches a peak speedup of 56.3 compared to a single thread CPU. Compared to 6 and

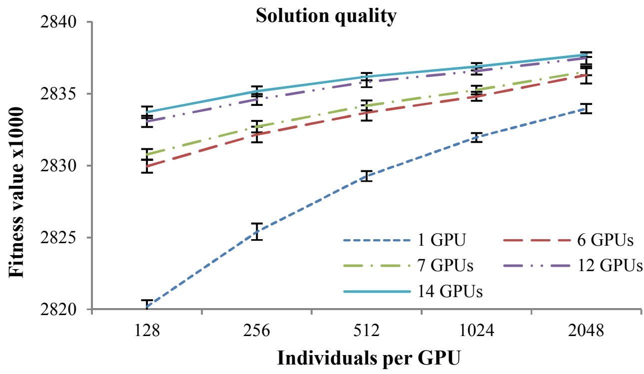

Figure 4. Average best fitness values after 100k generations evolved using 1 to 14 GPUs with the island sizes from 128 to 2048.

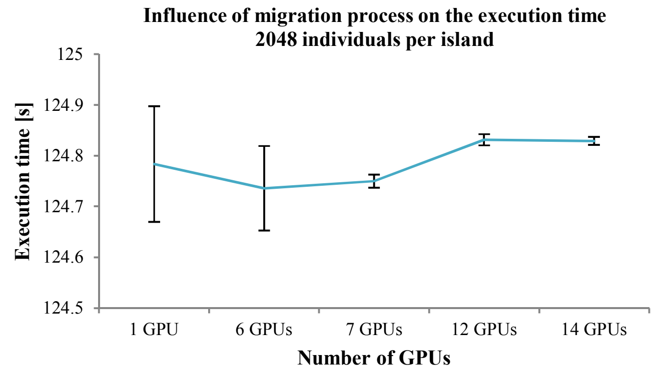

Figure 6. Presentation on negligible overhead of the inter-island communication for different number of island with 2048 individuals.

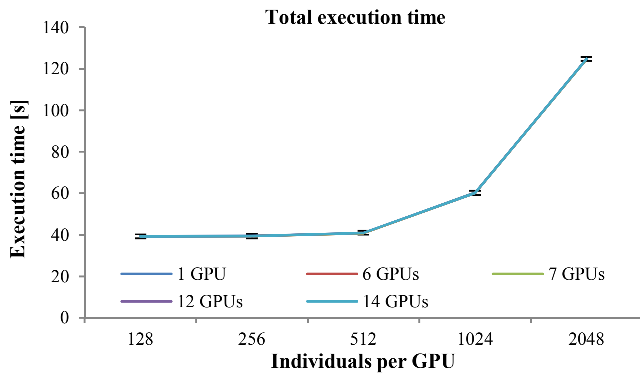

Figure 5. Total execution times of the evolution on 1 to 14 GPUs with the island sizes from 128 to 2048.

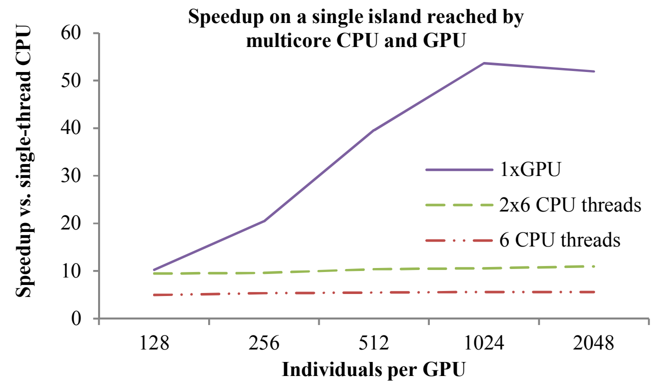

Figure 7. Comparison of relative speedup when evolving a single island using six and twelve CPU threads and a single GPU vs. a single CPU thread.

12 CPU threads, a single GPU reaches speedups of 9.45 and 4.73, respectively. These values roughly correspond to the theoretical peak performance ratio of these architectures. The proposed GA implementation was also compared with the well-known GALib library [29]. As GALib does not exploit multi-core CPUs, sequential implementations were compared only. The proposed implementation outperformed the GALib by 3.25 times because of exploiting better memory layout, avoiding unnecessary data movement and supporting Intel SSE 4.1 extension.

Table I summarizes the reached speedup of the island-based GA running over multiple CPU cores or GPUs with respect to a single thread CPU implementation. The speedup is computed as a multiple of time a sequential CPU implementation would need to simulate the evolution of all the islands. Table I is basically included because of authors who compare their GA implementations against a sequential one, which is not fair to CPU and results in unrealistic speedups as explained in [6].

Table II takes joint CPU computation power of both servers as a baseline to compares CPU and GPU side of the cluster. It is very important to observe 4 six-core CPUs can simply beat a single GPU for small populations. Moreover, a benchmark instance of 10000 items was used here. As 1024 items pose the smallest instance fully utilizing a warp and exploiting the potential of Fermi GPU [14], CPUs evolve smaller knapsacks much faster while GPU performance stagnates. On the other hand, for large knapsack instances and higher islands sizes necessary for their successful optimization, 14 GPUs can outperform four six-core CPUs by a speedup factor of 35.

In order to assess the FLOP performance of the proposed implementations, the PAPI performance counters were used [30]. The peak performance of a single GPU reached 405 GFLOPS and the total GPU cluster performance 5.67 TFLOPS. Compared to a synthetic SHOC benchmark [31], this accounts for about 26%. In contrast to SHOC, parallel GAs require a lot of synchronization, reduction and single thread work, and data exchange among islands that leads to GPU stalls.

VII. CONCLUSIONS

This paper has proposed a new implementation of island-based genetic algorithm exploiting a multi-GPU cluster. 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.

Unlike other related works solving only low dimensional numerical problems [15], or very small combinatorial problems [19], a benchmark requiring very long chromosomes of 10,000 bits and a considerably large global data set was used here. The computational power of 14 NVIDIA GTX580 cards against 4 six-core Intel Xeon CPUs was compared. The speedup reached by fourteen GPUs reaches 35, 194, and 781 compared to 4 CPUs, 1 CPU and a single thread, respectively (Table I and II).

TABLE I. SPEEDUP OF ISLAND-BASED CPU AND GPU IMPLEMENTATIONS WITH RESPECT TO A SEQUENTIAL CPU IMPLEMENTATION.

vs. single CPU core	**Island size**				
	128	**256**	**512**	**1024**	**2048**
1 CPU	1.00	1.00	1.00	1.00	1.00
6 CPUs	5.55	5.55	5.55	5.55	5.67
12 CPUs	11.12	11.13	11.14	11.14	11.32
24 CPUs	22.18	21.93	22.21	22.26	22.42
1 GPU	10.70	21.31	41.13	55.78	54.49
6 GPUs	64.38	128.22	247.02	335.38	327.09
7 GPUs	75.10	149.62	287.85	391.30	381.56
12 GPUs	128.76	256.40	493.09	670.04	653.68
14 GPUs	150.21	299.17	573.96	781.78	762.64

TABLE II. GPU SPEEDUP COMPARED TO FOUR SIX-CORE PROCESSORS

vs. 24 CPU cores	**Island size**				
	128	**256**	**512**	**1024**	**2048**
1 GPU	0.48	0.97	1.85	2.51	2.43
6 GPUs	2.90	5.85	11.12	15.06	14.59
7 GPUs	3.39	6.82	12.96	17.58	17.02
12 GPUs	5.81	11.69	22.20	30.10	29.16
14 GPUs	6.77	13.64	25.85	35.11	34.02

The overall performance reached by all GPUs in terms of FLOPS amount to 5.67 TFLOPs, which corresponds to 26% of the theoretical cluster performance. Given that the GA process requires a lot of synchronization, parallel reductions, and data movement during migration, this number poses a very good result when compared with other GPU applications [6]. The source codes of the proposed implementations have been released under GPU licence at [32].

ACKNOWLEDGEMENT

The author would like to thank Bradley E. Treeby, Josef Schwarz and Petr Pospichal for useful discussion.

REFERENCES

[1] T. Sterling, D. J. Becker, C. Park, J. E. Dorband, U. A. Ranawake, and C. V. Packer, “BEOWULF : A Parallel Workstation for Scientific Computation,” in *Proceedings of the 24th International Conference on Parallel Processing*, 1995, pp. 11-14.

[2] “TOP500 Supercomputer sites.” [Online]. Available: http://www.top500.org/. [Accessed: 09-Jan-2012].

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

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

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

[6] 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.

[7] NVIDIA, “Math Library Performance CUDA Math Libraries,” 2011.

[8] J. H. Holland, *Adaptation in Natural and Artificial Systems*, vol. Ann Arbor, no. 53. University of Michigan Press, 1975, p. 211.

[9] E. Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 2000, p. 187.

[10] J. Jaros, *Evolutionary Design of Collective Communications on Wormhole Networks*. Brno: Publishing house of Brno University of Technology VUTIUM, 2010, p. 183.

[11] M. Tomassini, *Spatially Structured Evolutionary Algorithms*. Springer, 2005, p. 206.

[12] H. Rashid, B. C. Novoa, C. A. Qasem, and S. Marcos, “An Evaluation of Parallel Knapsack Algorithms on Multicore Architectures 2 . The Integral Knapsack Problem.”

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

[14] NVIDIA, “Cuda c best practices guide,” 2011.

[15] 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.

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

[17] O. Garnica, J. Risco-Martin, and J. Hidalgo, “Speeding-up resolution of deceptive problems on a parallel gpu-cpu architecture,” in *WPABA08 (PACT08)*, 2008, pp. 57-64.

[18] W. Banzhaf, S. Harding, W. B. Langdon, and G. Wilson, “Accelerating Genetic Programming through Graphics Processing Units,” in *Genetic Programming Theory and Practice VI*, R. L. Riolo, T. Soule, and B. Worzel, Eds. Springer, 2008, pp. 229-249.

[19] 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.

[20] R. Shah, P. Narayanan, and K. Kothapalli, “GPU-Accelerated Genetic Algorithms,” *cvit.iiit.ac.in*.

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

[22] A. Fog, “Optimizing software in C ++: An optimization guide for Windows, Linux and Mac platforms,” 2011.

[23] J. Jaros and P. Pospichal, “A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm under the Knapsack Benchmark,” in *Applications of Evolutionary Computation, EvoPar*, 2012.

[24] NVIDIA, “CUDA Toolkit 4 . 0 CURAND Guide,” 2011.

[25] 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.

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

[27] N. Corporation, “NVIDIA GPUDirect ™ Technology NVIDIA GPUDirect ™ : Eliminating CPU Overhead,” 2011.

[28] MiTAC International Corp. / TYAN Business Unit, “TYAN FT77B7015 Web page,” 2012. [Online]. Available: http://www.tyan.com/product_SKU_spec.aspx?ProductType=BB&pid=439&SKU=600000195.

[29] M. Wall, “GAlib : A C ++ Library of Genetic Algorithm Components,” *Statistics*, no. August, 1996.

[30] A. D. Malony et al., “Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs,” in *Performance Computing*.

[31] A. Danalis, G. Marin, C. Mccurdy, J. S. Meredith, and P. C. Roth, “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.

[32] J. Jaros, “Jiri Jaros’s software website,” 2012. [Online]. Available: http://www.fit.vutbr.cz/~jarosjir/prods.php.en.