GPU-Accelerated Genetic Algorithms

Rajvi Shah
International Institute of
Information Technology
Hyderabad, India
rajvi.shah@research.iiit.ac.in

P.J.Narayanan
International Institute of
Information Technology
Hyderabad, India
pjn@iiit.ac.in

Kishore Kothapalli
International Institute of
Information Technology
Hyderabad, India
kkishore@iiit.ac.in

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. We exploit the parallelism within a chromosome in addition to the parallelism across multiple chromosomes. The use of one thread per chromosome by previous efforts does not utilize the GPU resources effectively. Our approach uses multiple threads per chromosome, thereby exploiting the massively multithreaded GPU more effectively. This results in good utilization of GPU resources even at small population sizes while maintaining impressive speed up for large population sizes. Our approach is modeled after the GAlib library and is adaptable to a variety of problems. We obtain a speedup of over 1500 over the CPU on problems involving a million chromosomes. Problems of such magnitude are not ordinarily attempted due to the prohibitive computation times.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Parallel Programming*; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

General Terms

Algorithms,Performance

Keywords

GAs,GPU,Genetic Algorithm,CUDA,Parallel GAs

1. INTRODUCTION

Genetic Algorithms (GAs) are a set of evolutionary algorithms, powerful and effective in solving search and optimization tasks. This class of algorithms is inspired by the process of biological evolution. Similar to the process of 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 (Holland [4]).

A typical genetic algorithm starts with selecting random points in search space, representing them as chromosomes and building an initial population. This 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, which usually with a very low mutation probability gets mutated. Hence, main components of a genetic algorithm are chromosome representation, selection, crossover and mutation. A representation that encodes the solution of the problem in the best possible way is used. Crossover and Mutation operators are often limited by the representation being used. Many methods exist for the process of selection as well, a method is chosen based on the convergence and diversity needs.

The user needs to tune various parameters and experiment with genetic operations and selection methods to achieve desired results using a genetic algorithm. In such a scenario, a library-like utility provides users great flexibility and ease of experimentation, speeding up the process of actual problem solving. Many public libraries exist for genetic algorithms providing a unified and optimized approach to achieve desired results. GALib (Wall [16]) is one such widely accepted library which enables the users to represent and solve their problems using genetic algorithms in a simple and effective way with enough flexibility. The long execution times associated with Genetic Algorithms constraints its application in many domains, despite its popularity on many domains.

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 [7]), 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.

2. RELATED WORK

Genetic Algorithms have been well explored and used in many domains for a long time. Attempts have been made in recent times to accelerate their performance using GPUs. Yu et al. [18] implemented a fine-grained parallel genetic algorithm (Tomassini and Calcolo [14]) on the GPU using Cg shader mechanism. A hybrid genetic algorithm (HGA) was proposed and implemented on GPU using the graphics pipeline and shading languages by Wong and Wong [17].

These above approaches were implemented and tested on Nvidida's GeForce 6800GT GPUs using the graphics pipeline. As the GPUs became more powerful and popular, they have become fully programmable parallel processing units. With the availability of high-level programming languages such as CUDA (NVIDIA [7]) and OpenCL (Khronos OpenCL Working Group [6]), researchers now see GPUs as a high performance multi-core processors. This has established a trend for General-purpose computation on GPUs (GPGPU) (GPGPU [3]).

In a recent work, Pospíchal et al. [8] presented a mapping of the parallel-island model of GA (Cantu-Paz [1]) to the CUDA architecture. This approach was implemented and tested on Nvidia's high-end CUDA compatible GPUs, namely, the 8800 GTX and GTX 285. The mapping of population islands to blocks benefits tremendously by fast access to shared memory resources within a block accelerating the performance many times. The island model is further explored to solve 0-1 knapsack problem in Pospíchal et al. [9]. The islands model is especially well suited to the block structure of CUDA with limited shared memory. This approach doesn't extend well to the general GA framework, which is the focus of this work.

3. GPU AND CUDA ARCHITECTURE

We present an overview of the CUDA programming and hardware models in this section. Please see (NVIDIA [7]) for more details about CUDA programming. Figure 1 depicts the CUDA programming model, mapping a software CUDA *block* to a hardware CUDA multiprocessor. A number of blocks can be assigned to a multiprocessor and they are time-shared internally by the CUDA programming environment. Each multiprocessors consists of a series of processors which run the *threads* present inside a block in a time-shared fashion based on the *warp* size of the CUDA device. Each multiprocessor further contains a small shared memory, a set of 32-bit registers, texture, and constant memory caches common to all processors inside it. Processors in the multiprocessor executes the same instruction on different data at any time. This makes CUDA an SIMD model. Communication between multiprocessors is through the device global memory which is accessible to all processors within a multiprocessor. Synchronization between threads of a block are possible. Synchronization across blocks is possible only at kernel boundaries.

The CUDA API provides a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The code executes as threads running in parallel in batches of warp size, time-shared on the CUDA processors. Each thread can use a number of private registers for its computation. A collection of threads (called a *block*) runs on a multiprocessor at a given time. Threads of each block have access to a small amount of common shared memory. Synchronization barriers are also available for all threads of a block. A group of blocks can be assigned to a single multiprocessor but their execution is time-shared. The available shared memory and registers are split equally amongst all blocks that timeshare a multiprocessor. An execution on a device generates a number of blocks, collectively known as a *grid* Figure 1.

Each thread executes a single instruction set called the *kernel.* Threads and blocks are given a unique ID that can be accessed within the thread during its execution. These can be used by a thread to perform the kernel task on its part of the data resulting in an SIMD execution. Algorithms may use multiple kernels, which share data through the global memory and synchronize their execution either at the end of each kernel or forcefully using barriers.

4. GPU ACCELERATED GA

Genetic algorithm execution is a parallel process. That is, there is no dependency across the chromosomes of a population for the process of fitness evaluation and genetic operations. Hence, the entire population can be operated in parallel within a generation. To exploit the parallelism at a greater level, we form groups of threads to handle a single chromosome, thus mapping the problem to a massively multithreaded model for which GPUs are best suited. Currently, we have implemented the generic genetic algorithm with uniform and roulette wheel selection strategies, one point crossover and flip mutation (Goldberg [2]). Figure 2 shows the overall flow of the genetic algorithm framework onto a GPU.

4.1 Data Organization

In past, efforts were made to effectively utilize the parallelism of chromosomes by employing one thread per chromosome to perform fitness evaluation as well as genetic operations (Pospíchal et al. [8], Robilliard et al. [10]). The key difference of our approach is that we use several threads to perform these operations on a single chromosome, resulting in a better utilization of GPU resources. This is realized in practice by organizing the data in GPU memory in such a way that genes of each chromosomes can be accessed efficiently in a coherent manner by multiple threads handling it. This section describes organization of thread and data, used by various CUDA kernels.

Population is laid out in main memory of GPU, as a two dimensional $N \times L$ matrix such that columns refer to chromosomes and rows corresponds to genes within chromosomes as shown in Figure 3, where N is population size and L is

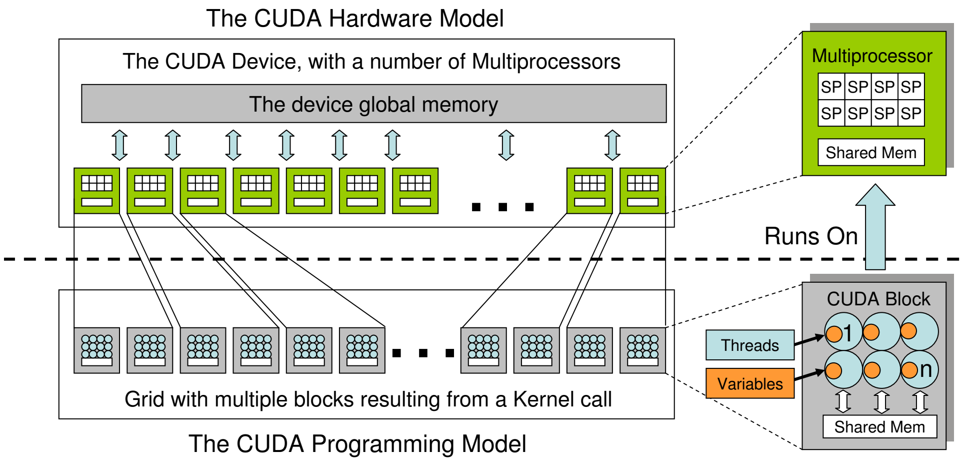

Figure 1: The CUDA hardware model (top) and programming model (bottom), showing the block to multiprocessor mapping.

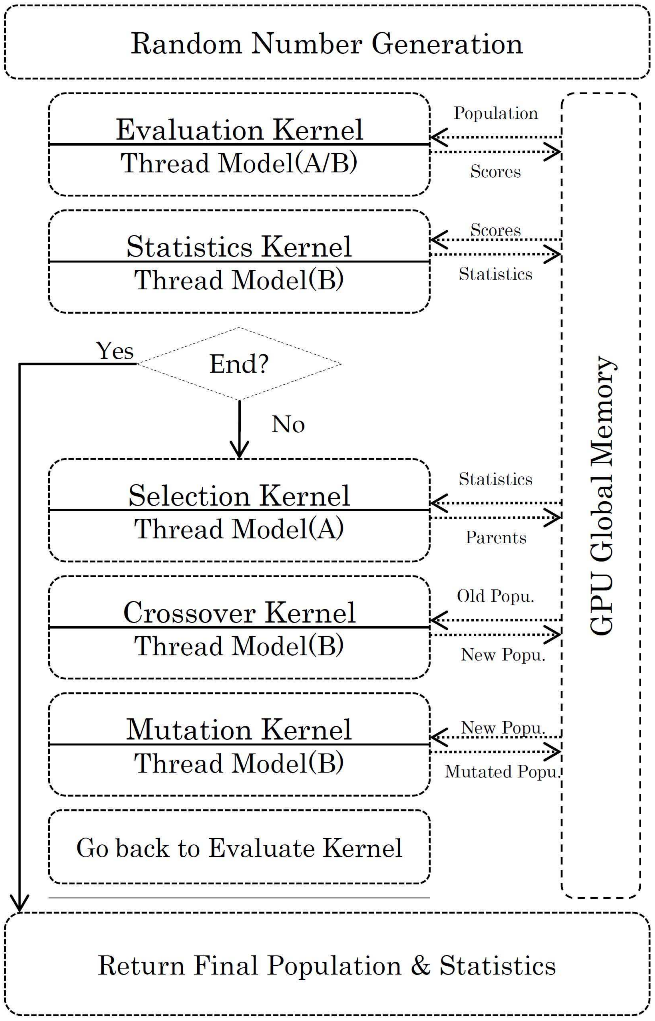

Figure 2: Program Execution and Memory Transfers

chromosome length. For a thread per chromosome model, threads in a block are arranged as a one dimensional array as shown in Figure 4 with one thread per chromosome. For a fully parallel approach, threads in a block are also arranged as a two dimensional matrix as shown in Figure 5. Kernel parameter *blockDim.x* is controlled by the number of threads per block (TPB), which is a CUDA block parameter. This layout leads to a one-to-one mapping between thread indices and genes. So, all genes of a chromosome can be accessed simultaneously.

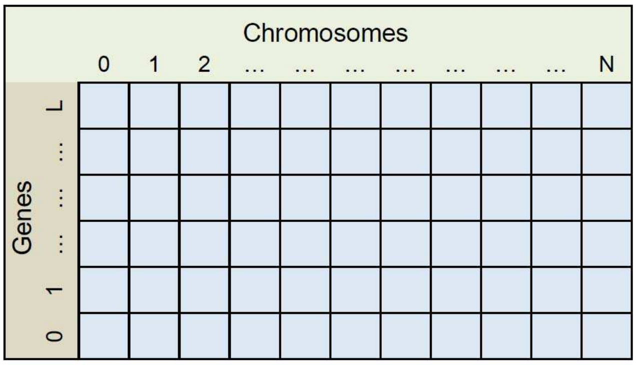

Figure 3: Population Matrix in memory

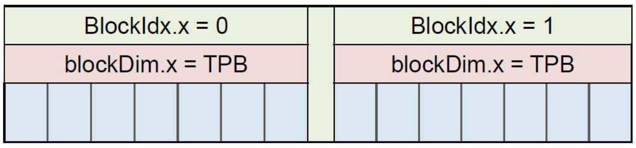

Figure 4: Thread Layout A

A detailed description of the execution flow depicted in Figure 2 and the mapping of the data layout shown in Figure 3 to a massively multithreaded model in each of the kernels is given in the subsequent subsections.

4.2 Fitness Evaluation Kernel

The process of fitness evaluation determines how fit each chromosome is to be the solution. Unlike other genetic op-

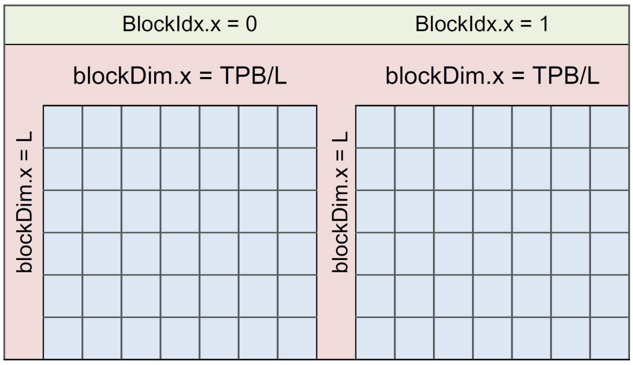

Figure 5: Thread Layout B

erators, fitness evaluation is a problem specific process and has to be provided by the user. In our framework, we provide a partially parallel and a fully parallel methods for the process of fitness evaluation.

The partially parallel method uses thread layout A as shown in Figure 4 with one thread per chromosome. In this method, user can access the chromosome as a 1D array and write an expression for fitness evaluation by accessing this array. User's C code fragment is used in fitness evaluation kernel by each of the threads to evaluate fitness of each individual. This is possible as CUDA is compatible to C. The calculated fitness scores are written back to the GPU global memory. As only chromosome level parallelism is exploited, this method may prove less efficient. But, it doesn't require a user to be familiar with CUDA architecture or programming. Hence, it makes the utility useful to a larger community at a small loss in performance.

The fully parallel method is provided for CUDA proficient users wherein the user can supply an evaluation function including a fitness evaluation kernel which may utilize the GPU resources in a more effective manner. This provides the user a way to achieve maximum performance.

Consider an example of 0-1 Knapsack problem. We are given a set of items with associated weights and costs. The aim is to pick items such that the total cost is maximum and total weight does not exceed knapsack capacity. A binary string is a convenient representation for chromosomes in this problem, where 1 indicates presence of an item and otherwise. Length of the chromosome is set to total number of items. In such a problem the fitness evaluation will involve finding cost sum and weight sum for all the chromosomes.

In partially parallel method, every thread will read one chromosome, its weight and cost, calculate total sum and total cost and write the score, providing parallelism across the chromosomes. Whereas in a fully parallel approach, we copy a block of cromosomes to shared memory. According to thread layout B (Figure 5), threads in each column read genes of corresponding chromosome, multiply it with cost and weight arrays and perform a log-sum as shown in Figure 6.

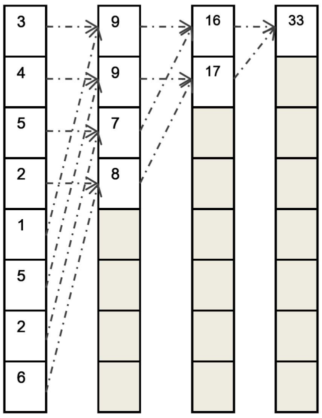

Figure 6: Parallel Sum

Fully parallel approach with careful utilization of shared resources can make the evaluation process much faster, especially for problems involving intensive fitness calculation.

4.3 Statistics Kernel

After the fitness scores are calculated, population statistics need to be updated. Population statistics are used for the process of selection and to decide termination. The maximum, minimum and average and total fitness scores are calculated using standard parallel reduce algorithms (Jaja [5]). Best and worst chromosomes are recorded to ensure elitism, if selected by user. Also the selection probability for each of the chromosome is calculated.

Fitness scores may need to be sorted depending upon the selection method to be used. Sorting is not required if stochastic selection method is used. For probabilistic selection, like roulette wheel or rank selection, scores need to be sorted. A fast GPU based radix-sort, provided by CUDPP (CUDA Data Parallel Primitives) library is used for the same (Satish et al. [12]). Some method-specific statistics are also calculated, which is described later.

4.4 Selection Kernel

The execution of a Genetic Algorithm begins with the process of selection. In the process of selection, parent chromosomes are selected to go through the process of crossover to produce offspring. Selection Kernel will vary according to the selection method being used. Here, Uniform and Roulette Wheel selection kernels are described in detail. A uniform selection kernel is described in Pseudo-code 1.

Pseudo-code 1 Uniform Kernel

```
$N \leftarrow popSize$
$numThreads \leftarrow \frac{N}{2}$

{For all threads in parallel}

$i \leftarrow threadIdx$
$parent1(i) \leftarrow random(0, N-1)$
$parent2(i) \leftarrow random(0, N-1)$
$parent1(i+1) \leftarrow parent1(i)$
$parent2(i+1) \leftarrow parent2(i)$
```

Roulette wheel selection is more expensive than uniform selection. To simulate the roulette wheel, the population is sorted based on the fitness score values (Satish et al. [12]). These score values are normalized to calculate selection probabilities. A sum-scan is performed on the normalized array (Sengupta et al. [13]). This new array is stored in global memory and used as a roulette wheel array (*rouletteArray*). These calculations are done in statistcs update stage, prior to execution of selection kernel. This selection kernel is described in Pseudo-code 2.

Pseudo-code 2 Roulette Wheel Kernel

```
$GLOBAL : rouletteArray$
$N \leftarrow popSize$
$numThreads \leftarrow \frac{N}{2}$

{For all threads in parallel}

$i \leftarrow threadIdx$
$p1 \leftarrow random(0-1)$
$p2 \leftarrow random(0-1)$

$parent1(i) \leftarrow rotateWheel(p1, N)$
$parent2(i) \leftarrow rotateWheel(p1, N)$
$parent1(i+1) \leftarrow parent1(i)$
$parent2(i+1) \leftarrow parent2(i)$
```

The rotateWheel function used in selection, performs a binary search on prefix-summed *rouletteArray* for the nearest smaller real number and returns parent index. This subroutine is described in Pseudo-code 3.

Pseudo-code 3 rotateWheel(n,N)

```
$GLOBAL : rouletteArray$
$flag = 0$
$start = 0, end = N, middle = \lceil \frac{end}{2} \rceil$
while !flag do
  $left \leftarrow rouletteArray[middle]$
  $right \leftarrow rouletteArray[middle + 1]$
  if n >= left then
    if n < right then
      $index \leftarrow middle$
      $flag = 1$
    else
      $start = middle$
      $middle = start + \lceil \frac{end-start}{2} \rceil$
    end if
  else
    $end = middle$
    $middle = \lceil \frac{end-start}{2} \rceil$
  end if
end while
$return(index)$
```

Both, Uniform and Roulette Wheel Selection use thread layout A as shown in Figure 4.

4.5 Crossover Kernel

A pair of chromosomes selected in selection process undergoes the process of crossover to produce offsprings. The process of crossover is controlled by the crossover probability. Our implementation performs one-point crossover, but the same approach can be adapted to other crossover methods as well.

4.5.1 Crossover Preprocess

In our implementation, crossover points are calculated and stored prior to invoking actual crossover kernel. As the approach used for crossover uses multiple threads per chromosome, all the threads performing crossover between two chromosomes should know a common crossover point value. This prohibits generation of crossover points in crossover kernel itself due to a restrictive memory model and synchronization issues across the blocks in CUDA (NVIDIA [7]).

A kernel for selecting crossover points for one-point crossover is described by Pseudo-code 4. This kernel uses thread model A as shown in Figure 4.

Pseudo-code 4 Crossover Points Kernel

```
$N \leftarrow popSize$
$L \leftarrow chromoLength$
$numThreads \leftarrow \frac{N}{2}$

{For all threads in parallel}

$i \leftarrow threadIdx$
$r1 \leftarrow random(0, 1)$
if $r1 \geq probCross$ then
  $crossPoint(i) \leftarrow random(0, L-1)$
  $crossPoint(i+1) \leftarrow crossPoint(i+1)$
else
  $crossPoint(i) \leftarrow 0$
  $crossPoint(i+1) \leftarrow 0$
end if
```

In practice, crossover points are also selected along with parents in selection kernel as it uses the same thread layout.

4.5.2 One-point Cross-over

Instead of making one thread read two chromosomes, perform crossover and write the offspring chromosomes back, we make use of multiple threads to read a single chromosome. This approach results in a coalesced read and write of data speeding up the execution greatly.

As shown in the Figure 3, a chromosome occupies a column in the population matrix. Hence, the column index becomes the chromosome index. For the process of crossover we make use of total NL threads where N is the population size and L is the chromosome length. These threads are also laid out as a 2D matrix with N columns and L rows across the bloacks as shown in Figure 5. Now, instead of using one thread per crossover operation we use $2L$ threads, utilizing the massively multithreaded GPU model. Pseudo-code 5 describes a one-point crossover kernel using NL number of threads.

4.6 Mutation Kernel

In the process of genetic evolution, some chromosomes of the population mutate with a small mutation probability. Mutation is very crucial to bring genetic algorithm out of a local maxima or minima. The process of mutation is controlled by the mutation probability. We consider mutation probability as a probability for a gene to get mutated. For mutation kernel we again make use of matrix layout of Figure 3 for population and thread layout of Figure 5. Each

Pseudo-code 5 Crossover Kernel

```
$GLOBAL : Parent1, Parent2, crossPoint$
$N \leftarrow popSize$
$L \leftarrow chromoLength$
$numThreads \leftarrow N \times L$

{For all threads in parallel}

$C_{idx} \leftarrow threadIdx.x$
$R_{idx} \leftarrow threadIdx.y$

$p1 \leftarrow Parent1(C_{idx})$
$p2 \leftarrow Parent2(C_{idx})$
$xPoint \leftarrow crossPoint(C_{idx})$

if $R_{idx} \leq xPoint$ then
  $newPopulation(C_{idx}, R_{idx}) = oldPopulation(p1, R_{idx})$
else
  $newPopulation(C_{idx}, R_{idx}) = oldPopulation(p2, R_{idx})$
end if
```

thread now corresponds to a gene and decides whether or not to mutate the gene.

4.7 Random Numbers

Random numbers are extensively used throughout a genetic algorithm. CUDA does not provide any support for on the fly generation of a random number by a thread because of many synchronization issues associated. To solve this issue, an estimate of required random numbers is made. For example, a GA set up for a uniform selection with one-point crossover and flip mutation requires nearly $T = 2N+N+NL$ random numbers in one iteration, where N is the population size and L is length of the chromosome. Based on this estimate and memory limits imposed by hardware, a large pool of random numbers are generated and stored on GPU global memory before initiating the genetic algorithm. To speed up the process of generation and avoid transfer, we make use of rand routine provided by CUDPP, which uses MD5 algorithm for pseudo random number generation (Tzeng and Wei [15]). If high quality random numbers are needed, this is replaced by a CPU based random number generation followed by a copy to global memory.

5. PROGRAMMING INTERFACE

GALib (Wall [16]) is built around a few base classes, the main two being a Genome class and a Genetic Algorithm class. A user is allowed to tune a Genetic Algorithm according to the problem by setting various parameters through these classes.

Our framework is built around three main structures: GA Context, Genome Context and GAStatistics Context.

Out of these three, GA Context and Genome Context are mainly filled by user and contains various parameters for execution of genetic algorithm like population size, chromosome size, crossover and mutation probabilities, selection method, termination method etc. GAStatistics is mainly filled by the program along with execution of genetic algorithm. It holds fitness scores and other population related statistics. Support functions are provided to fill these structures, print parameters and destroy the structures.

Other than these three structures, user in his program needs to declare a void pointer to the population, and define and declare a user data structure which user might want to use for fitness evaluation. User also needs to supply fitness evaluation related code fragment as explained previously. A typical example user program 1 is listed below.

```
int main()
{
  void *population;
  UDATA udata;

  GAContext ga;
  GNMContext genome;
  GAStats stats;

  GASetParameters(&ga,&genome,&stats);
  GAPrintParameters(&ga,&genome,&stats);

  gaEvolvePopulation(&population,&ga,&genome
     ,&stats,&udata);
  PrintSolution(population,&genome,&stats);

  GAdestroyContexts(&ga,&genome,&stats);
  return 0;
}

__device__ float FitnessFunc(BIN1D *g,
   GNMContext genome,UDATA *udata)
{
        // Code to find fitness of a genome
        return score;
}
```

Program 1: An Example User Program

6. RESULTS AND DISCUSSION

We use a quarter of Nvidia's Tesla S1070 GPU to test our implementation. Tesla is a massively parallel platform with 30 multiprocessors each having 8 cores. We compare the performance of our GPU implementation with the serial implementation provided by GAlib (Wall [16]), running on an Intel Core2 Duo E7500(2.93 GHz) CPU. The direct comparison of running time (or speedup) of the CPU and the GPU is not entirely meaningful to evaluate their merits and demerits. The timing statistics is provided to provide the readers a sense of what they can expect with respect to a standard package.

We chose a standard 0-1 knapsack problem (Sahni [11]) to measure the performance speedup. We performed various experiments by changing chromosome length, population size, number of generations and selection methods. In all the experiments, the GPU accelerated approach showed significant speed up against serial implementation with comparable results. The quality of results is not degraded, as the basic algorithm was not modified but only paralleled.

A side-by-side performance comparison is given in Table 1 and Table 2, showing the GPU and CPU execution times for various population sizes for uniform selection and roulette wheel selection methods respectively. Chromosome length and number of generations were fixed at 50 and 100 respectively. All the timings are averaged over 5 trials. The numbers clearly indicate that problems of huge magnitude can be solved in seconds with a GPU accelerated approach.

N	GPU	Std.Dev	CPU	Std.Dev	Speed Up
100	0.025	0.000006	0.127	0.006148	5.04
1000	0.031	0.000472	1.364	0.002490	43.61
10000	0.153	0.000485	19.799	0.270023	129.40
100000	1.561	0.001172	342.814	4.714645	219.06
1000000	3.662	0.001435	4803.381	214.557712	1311.36

Table 1: Timing Comparison for Uniform Selection, Time for 100 generations is given in seconds

N	GPU	Std.Dev	CPU	Std.Dev	Speed Up
100	0.046	0.000051	0.141	0.003834	3.01
1000	0.053	0.000041	1.629	0.017473	30.38
10000	0.209	0.000859	21.609	0.624903	103.19
100000	1.724	0.001149	492.927	9.326317	286.04
1000000	4.727	0.000327	7233.716	176.666921	1530.14

Table 2: Timing Comparison for Roulette Wheel Selection, Time for 100 generations is given in seconds

Table 3 shows the average execution time of the GPU-based approach for various chromosome lengths and population sizes for 100 generations, with a plot of the same in Figure 7. It is apparant from Figure 7 that the run-time growth is sublinear as the product NL increases.

	Population Size (N)				
L	100	1000	10000	100000	1000000
16	0.022	0.026	0.073	0.625	6.472
32	0.024	0.030	0.111	1.164	2.671
64	0.026	0.035	0.202	2.105	4.124
128	0.031	0.052	0.443	4.523	11.592
256	0.041	0.109	1.137	11.092	39.792
512	0.062	0.265	2.443	10.492	93.301

Table 3: Run-time in seconds for varying parameters for 100 generations

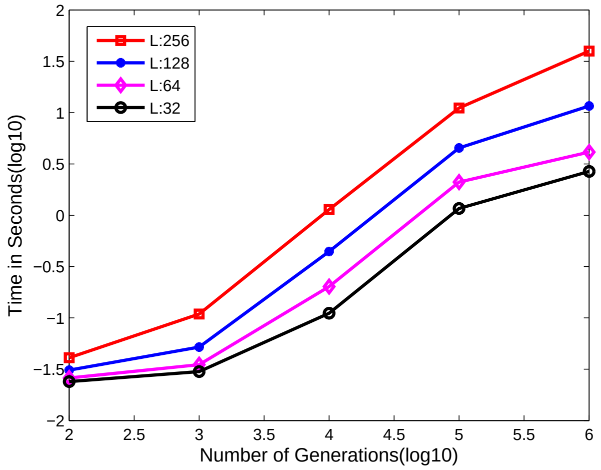

Figure 7: Run-time growth with N and L

Run-time growth of our approach with increasing number of iterations is linear as can be seen from Figure 8.

We also tried a numerical optimization problem using our implementation for its effectiveness. Our system found the minima of Rosenbrock's function effectively and fast.

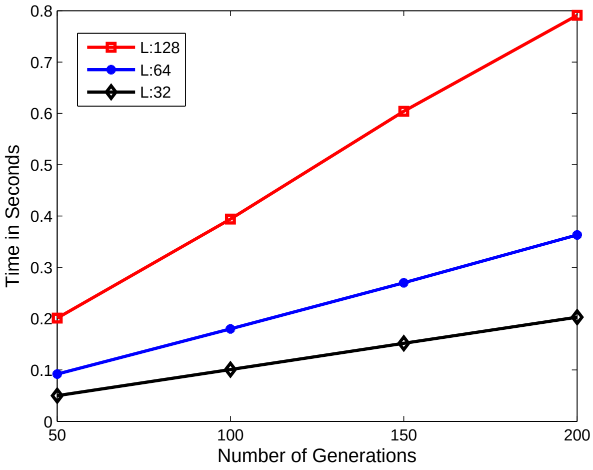

Figure 8: Run-time growth with number of generations

Direct performance comparison of our approach with other GPU based approaches is not meaningful. We achieve much higher speed up than that achieved by (Wong and Wong [17], Yu et al. [18]), but this comparison is not justified as they use a relatively old and less powerful hardware. Pospíchal et al. [8] use comparable hardware and demonstrates great speedup but using a parallel-island model of GA which can benifit greatly by shared resources.

7. CONCLUSION

In this paper, we demonstrate an approach to accelerate a simple genetic algorithm using the GPUs by exploiting gene level parallelism. We provide a mapping of various GA kernels to massively multithreaded model of GPUs using the CUDA programming model. Our GA framework is built around three basic structures to make the implementation extensible and flexible. Current implementation discusses a simple genetic algorithm with 1D chromosome and two different selection methods. The proposed framework can be extended to a GPU accelerated genetic algorithms library by incorporating more and more features. With speedup achieved over a factor of 1000 and a programmable library-like interface, GPU accelerated GA can find applications in many new domains.

8. ACKNOWLEDGEMENT

We would like to thank Nvidia for providing equipment support.

References

[1] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[2] D. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[3] GPGPU. General purpose computation on Graphics Processing Units. URL `http://www.gpgpu.org`.

[4] 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.

[5] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[6] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[7] NVIDIA. *NVIDIA CUDA Programming Guide Version 3.0*. NVIDIA Corporation, 2010.

[8] P. Pospíchal, J. JaroŽ, and J. Schwarz. Parallel Genetic Algorithm on the CUDA Architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[9] P. Pospíchal, J. Schwarz, and J. JaroŽ. Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. In *16th International Conference on Soft Computing MENDEL 2010*, pages 64–70. Brno University of Technology, 2010.

[10] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel gp on the g80 gpu. In *EuroGP'08: Proceedings of the 11th European conference on Genetic programming*, pages 98–109, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] S. Sahni. Approximate Algorithms for the 0/1 Knapsack Problem. *J. ACM*, 22(1):115–124, 1975.

[12] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[13] S. Sengupta, M. Harris, and M. Garland. M.: Efficient parallel scan algorithms for GPUs. NVIDIA. Nvidia technical report, NVIDIA Corporation, 2008.

[14] M. Tomassini and C. S. D. Calcolo. A Survey of Genetic Algorithms, 1995.

[15] S. Tzeng and L.-Y. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 79–87, New York, NY, USA, 2008. ACM.

[16] M. Wall. GAlib, A C++ Library of Genetic Algorithm Components. http://lancet.mit.edu/ga/, 2008.

[17] M. Wong and T. Wong. Parallel Hybrid Genetic Algorithms on Consumer-Level Graphics Hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980, 2006.

[18] Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III*, volume 3612, pages 1051–1059. Springer, August 27-29 2005.