Parallel Compact Genetic Algorithm on CUDA-C Platform

Vuppuluri Sumati
Dayalbagh Educational Institute
Dayalbagh
Agra

ABSTRACT
This paper deals about the parallel implementation of the compact Genetic Algorithm on the Compute Unified Device Architecture (CUDA) platform of GPU. We elaborate implementation details on the parallel platform.

General Terms
Parallel Implementation, High Performance Computing, Evolutionary Algorithm.

1. INTRODUCTION
The paper focuses on the parallel implementation of compact Genetic Algorithm on the platform of CUDA-C. This is the age of technology where we want things to be done at a fast pace. Here comes the power of modern day computers. The technology has enabled modern day computers to store and manipulate large amount of data. Thus the generation is shifting towards utilizing power of computers from the age old manual data handling. The power of computers has provided acceleration at the research frontier too. The field of biology is generating enormous amount of biological data which needs to be interpreted. As the data in the gene banks is increasing at very fast rates, the challenges of biology have actually become the challenges of computing. Such an approach is ideal because of the ease with which computers can handle large quantities of data and probe the complex dynamics observed in nature. This new field of study called as bioinformatics. According to definition in the oxford dictionary bioinformatics is conceptualizing biology in terms of molecules (in the sense of physical chemistry) and applying informatics techniques (derived from disciplines such as applied mathematics, computer science and statistics) to understand and organize the information associated with these molecules, on a large scale [1]. Bioinformatics is an emerging area of research where the current trend is towards solving the computationally intensive problems like cancer classification with important gene identification, protein sequence analysis, DNA and RNA structure prediction and various other such problems.

The use of bioinfomatics implies handling with huge data sets. This requires clever data management, process and inference techniques. Many a times we harness the power of computational intelligence techniques to infer the data available. Computational intelligent techniques comprise of neural networks [2], fuzzy logic [3] and evolutionary algorithms [4]. Recently, the focus on the evolutionary algorithms (EA's) has increased. EAs concentrate on a broad class of structural and parametric optimization tasks omnipresent in system design. The EA is basically a randomized search and optimization technique. An EA is able to cope with ill-behaved problem domains, exhibiting attributes such as multimodality, discontinuity, time-variance, randomness, and noise. The problem at hand is represented as a population of strings (each string being a candidate solution) that are made to evolve by applying a set of stochastic operators (mutation, crossover and selection) [5]. Mutation is random permutation of a candidate solution. Crossover or recombination is the formation of new solutions by mixing parts of two decomposed and distinct solutions. The last operator is the selection operator which is replication of the most successful solution in the population. Genetic algorithms (GAs), evolutionary strategies (ES), and genetic programming (GP) are the major components of evolutionary algorithms. Of these, GAs are the most widely used because they are efficient, robust, and adaptive search processes producing near optimal solutions and having a large amount of parallelism. GAs can deal with large amounts of data very efficiently and robustly. Evolutionary algorithms have been integrated into different neural networks for evolutionary learning [6].

An algorithm which is of interest to us is the compact evolutionary algorithm (cEA). The Compact Genetic Algorithm is an Estimation of Distribution Algorithm (EDA), also referred to as Population Model-Building Genetic Algorithms (PMBGA), an extension to the field of Evolutionary Computation [7]. The algorithms belonging to this class do not store and process an entire population and all its individuals therein but on the contrary make use of a statistical representation of the population in order to perform the optimization process. In this way, a much smaller number of parameters must be stored in the memory. Thus, a run of these algorithms requires much less memory requirements compared to standard EAs. The first cEA was the compact genetic algorithm (cGA) [8]. This cGA evolves a probability vector (PV) that describes the hypothetical distribution of a population of solutions in the search space. A cGA iteratively processes the PV with updating mechanisms that mimic the typical selection and recombination operations performed in a standard GA (sGA) until a stopping criterion is met. The cGA performs equivalent to the sGA with significant reduction of memory requirements, as it needs to store only the PV instead of entire population of solutions. Thus, cGA mimics the "order-one" behaviour of the sGA. This feature makes cGA particularly useful for memory constrained applications [9]. cGA seems suitable in optimizing a nonlinear approximator such as a neural network. Many variants of the cGA have also been proposed in the literature [10, 11] to solve real world problems such as in the fields of computational biology, evolvable hardware, and embedded systems.

We have divided this paper with first section containing the introduction. The second section deals with the basic concept of parallel computing, third containing GPU and CUDA-C. The fourth section deals with steps for implementing cGA on CUDA platform. The fifth section is conclusion.

2. PARALLEL COMPUTING
Parallel computing is the simultaneous use of multiple CPU's to solve a computational problem. A problem is broken into

discrete parts that can be solved concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs. Parallel computing attempts to emulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet with a sequence.

Parallel computing is considered to be the high end of computing, and is used to model difficult scientific and engineering problems found in the real world. Its innumerable applications are found in the field of atmospheric sciences, earth and environmental science and many areas of physics.

Commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. In order to achieve fast computation in the beginning, software developers relied on the advancements in hardware to increase the speed of their applications. This drive has slowed down since 2003 because heat-dissipation and energy consumption issues have limited the clock frequency in a single CPU. Nowadays, the advanced versions of CPU's containing multiple processor cores are used to increase the processing power.

Traditionally, the vast majority of software applications are written as sequential programs. However, to enjoy the actual benefit of speed with each new generation of processors one needs to switch to parallel programming and processing in which multiple threads of execution cooperate to complete the work faster. This is high performance computing.

The multicores began as two-core processors, with the number of cores roughly doubling with each semiconductor process generation [12]. A current example is Intel Core i7 microprocessor, which has four processors cores. The microprocessor supports hyperthreading with two hardware threads and is designed to maximize the execution speed of sequential programs.

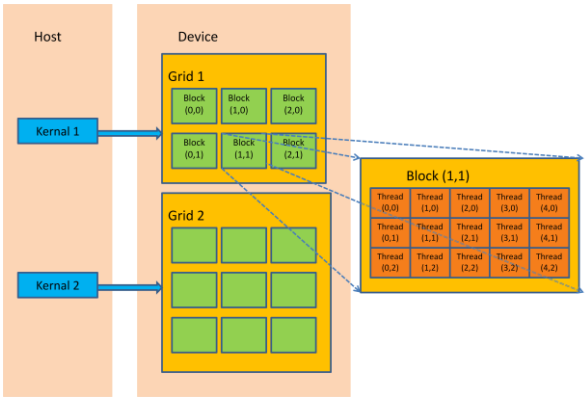

Fig 1a: Grid level hierarchy in GPU (courtesy NVIDIA Programming Guide 2.0)

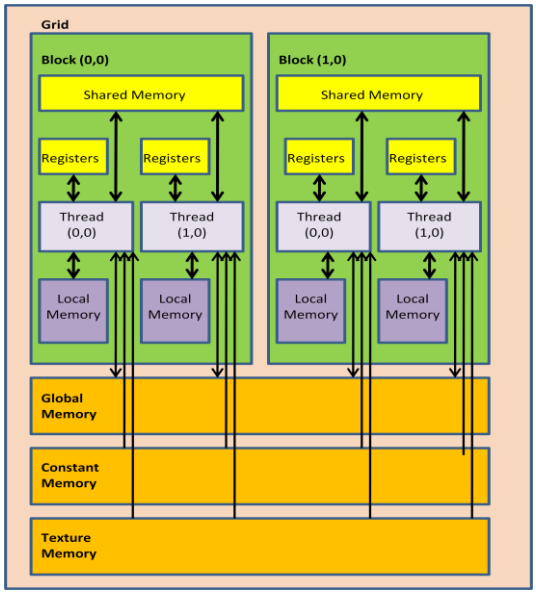

Fig 1b: Grid level hierarchy in GPU (courtesy NVIDIA Programming Guide 2.0)

There is a possibility that we can research cGA and its variants for solving computationally intensive problems in bioinformatics. We can parallelize this algorithm to make it suitable for handling huge data set of bioinformatics. In this paper we study the parallel implementation of compact genetic algorithm.

3. GPU and CUDA-C

The main idea of general-purpose parallel computing on a graphic processing unit (GPU) is to devote more transistors to process data and process in parallel. This is the main difference between CPU and GPU [13]. GPUs are primarily oriented on parallel data manipulations rather than data caching and flow control. CUDA (Compute Unified Device Architecture) is high level language for GPU programming. During the last twenty years GPUs came through massive development. They evolved from simple 2D graphic processing units, into multi-purpose, programmable, highly parallel, multi-core processor units with extraordinary computational power and very high memory bandwidth. GPU computing with CUDA is a new approach to computing where hundreds of on-chip processors simultaneously communicate and cooperate to solve complex computing problems up to 100 times faster than traditional approaches. A CUDA-enabled GPU operates either as a flexible thread processor, where thousands of computing programs called threads work together to solve complex problems, or as a streaming processor in specific applications such as imaging where threads do not communicate. CUDA-enabled applications use the GPU for fine grained data-intensive processing, and the multi-core CPUs for complicated coarse grained tasks such as control and data management.

The difference in speed between CPU and GPU is due to the architecture of GPU. While the CPU is conceptualized for general purposes carrying out arbitrary operations like input/output access, processing, etc; the GPU is conceptualized for performance optimization for defined tasks [12]. However, not all algorithms can be effectively implemented on the GPU. Only numerical problems that are inherently parallel may have profit of this technology. Since programming the GPU is a hard task, Nvidia [14] developed a software platform named Compute Unified Device

Architecture, for short, CUDA, which allows almost the direct translation of C code onto the GPU. The CUDA C extends the syntax of C language. Thus it has become easy to convert the existing code from C language to CUDA.

```
counter t = 0

for i =1:n
  initialize PV[i]= 0.5
end for

generate elite by means ofPV

while stopping criterion not met do
  generate1 individual cv by means of PV
  winner, loser = compete(a,elite)
  if cv == winner then
    elite = cv
  end if
  for i=1:n do
    if winner[i]!=loser[i]then
      if winner[i] = 1 then          PV Update
        PV[i] = PV[i]+1/Np
      else
        PV[i] = PV[i]-1/Np
      end if
    end if
  end for
  counter =counter+1
end while
```

Fig 2: compact GA with elitist strategy

Threads are the basic building blocks of GPUs. The concept of thread embodies the GPU with the technique of SIMD (Single Instruction, Multiple Data) from vector computers. A thread is a sequence of instructions that can be executed on different data units in parallel. There are three types of basic functions in CUDA-C: The host functions, which are called and executed only by the CPU. This kind of functions is the same as those functions implemented in C. The phases that exhibit little or no data parallelism are implemented in the host function. The phases that exhibits rich amount of data parallelism are implemented in the device code. The device code is written using ANSI C extended with keywords. These are called kernels. The second class of functions is kernel functions, which are only executed on the device and callable only by the CPU. For this kind of function, the qualifier, __global__, must be inserted before the type of return of the function that is always void. Finally, the device functions, which are called and executed only by the device, whereas the qualifier, __device__, must be declared before the type of return of the function. In this case, it is allowed to return any type of value [13].

There exists a level of hierarchy as grid, block and threads. The grid can have up to two dimensions of blocks, and the blocks, in turn, three dimensions of threads. These concepts allow perform all the parallelism of the program that will be executed into the device. When a kernel is invoked or launched, it is executed as grid of parallel threads. Each CUDA thread is comprised of thousands to millions of lightweight GPU threads per kernel invocation. Threads in a grid are organized in a two level hierarchy as illustrated in figure 1a and 1b. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. Each block has a unique two dimensional coordinate given by CUDA specific keywords blockIdx.x and blockIdx.y. All thread blocks must have the same number of threads organized in the same manner.

Each thread has an address that is represented by two vectors. The vector threadIdx of three position x, y and z, shows the address of the thread into the block. Two other additional vectors, which are very important are blockDim, and gridDim, of three and two positions, respectively. The first one contains the existing number of threads in each dimension of the block. The last one contains the number of blocks in each dimension of the grid.

The threads in a same block share a memory which is called shared memory and is located close to the GPU core (streaming multiprocessor). The use of this kind of memory provides advantages due to its low latency [12]. All the sequential operations are executed onto the host computer, e.g., for memory allocation and freeing within the device CUDA function cudaMalloc and cudaFree, respectively. To copy data to and from the device as well as to and from the host is used the CUDA function cudaMemcpy.

```
__global__ void cga(float *finalPV, curandState *globalState)
int tid = threadIdx.x;
      PV[tid] = generateinitPV();
                  while(flag==0)
      {
      curandState localState = globalState[tid];
      elite[tid] =
GenerateABinaryVector(tid,curand_uniform(&localState),PV[tid]);
      c_1_v[tid] =
GenerateABinaryVector(tid,curand_uniform(&localState),PV[tid]);
      __syncthreads();

      //Fitness evaluation
      fitness_elite = fitnessEval(elite);
      fitness_1_v = fitnessEval(c_1_v);
      __syncthreads();

      //Competition
      WhoIsTheWinner = Competition(fitness_0_v, fitness_1_v);
                  theta = theta +1;

      // Updating PV
      PV[tid] = updatePV(WhoIsTheWinner,tid, PV[tid], elite[tid], c_1_v[tid]);

     //elite vs existing
      if(WhoIsTheWinner == 1 || theta>=eta)
                  {
            elite[tid] = c_1_v[tid];
                                    theta = 0;
                  }
      gen = gen+1;
                  flag = checkifpvconverged(gen,PV);

      __syncthreads();
                  globalState[tid] = localState;
                  }
                  finalPV[tid]= PV[tid];
  }
```

Fig 3: Global function implemented on the device

4. COMPACT GA IMPLEMENTATION ON CUDA-C

In this section we show the main parts of the cGA code developed in CUDA-C.As mentioned earlier, compact genetic algorithm simulates the behaviour of a standard binary genetic

algorithm(GA). The performance comparison shows that the cGA is almost as good as GA. Lesser memory requirement is the advantage with cGA than GA [8]. The cGA is described in figure 2. A binary vector of length n is randomly generated by assigning a 0.5 probability to each gene to take either the value 0 or the value 1. This describing the probabilities, initialized with n values all equal to 0.5, is named as probability vector (PV). Two individual candidate solutions are generated from this PV. The winner solution with higher optimum value biases the PV on the basis of a virtual population parameter Np as shown in the figure 2. We take the elitist strategy cGA which is proved to have better performance over simple cGA [15].

The cGA is implemented on the CUDA-C platform. For this, we assign the number of threads in a block to be equal to the number of variables in PV vector. The number of variables in PV is determined by the number of binary bits and dimension of the optimizing function. When the cga kernel is invoked, the threads start to evaluate the PV values as shown in the figure 3. Each of the binary bit is manipulated by one GPU thread.

5. CONCLUSION

We have implemented the compact Genetic algorithm on the CUDA-C platform. We are proposing different strategies of parallelism to be applied on the compact Genetic algorithm. The implementation is being checked on different functions and is being compared to the compact Genetic Algorithm implemented on the standard C platform.

6. ACKNOWLEDGMENTS

My sincere thanks to the Department of Science and Technology, Government of India, which has contributed funding under the INSPIRE Fellowship Scheme. I also extend my thanks to my supervisor, my colleagues and my friends.

7. REFERENCES

[1] Fogel, G. B. and Corne, D. W. 2003. Evolutionary Computing in Bioinformatics. San Francisco, CA: Morgan Kaufmann, Elsevier Science.

[2] Kumar, S. 2004. Neural Networks: A Classroom Approach. New Delhi, India: Tata McGraw-Hill Education Private Limited.

[3] L. A. Zadeh, "Outline of a new approach to the analysis of complex systems and decision processes," IEEE Transactions on Systems, Man, and Cybernetics, vol. smc-3, no. 1, pp. 28-44.

[4] Pawlak, Z. 1991. Theoretical Aspects of Reasoning About Data. Dordrecht: Kluwer Academic.

[5] Tettamanzi, A. and Tomassini, M. 2001, Soft Computing: Integrating Evolutionary, Neural, and Fuzzy Systems. Berlin, Heidelberg: Springer-Verlag.

[6] Back, T. 1996. Evolutionary Algorithms in Theory and Practice. New York: Oxford University Press.

[7] Larranaga, P. and Lozano, J. A. 2002. Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation. Massachusetts, USA: Kluwer Academic.

[8] G. R. Harik, F. G. Lobo, and D.E. Goldberg, , "The compact Genetic Algorithm," IEEE Transactions on Evolutionary Computation, vol. 3, no. 4, pp. 287-297, 1999.

[9] J. C. Gallagher, S. Vigraham, and G. Kramer, "A family of compact genetic algorithms for intrinsic evolvable hardware," IEEE Transactions on Evolutionary Computation, vol. 8, no. 2, pp. 111-126, 2004.

[10] E. Mininno, F. Neri, F. Cupertino, and David Naso, "Compact Differential Evolution," IEEE Transactions on Evolutionary Computation, vol. 15, no. 1, pp. 32-54, 2011.

[11] E. Mininno, F. Cupertino, and D. Naso, "Real-Valued Compact Genetic Algorithms for Embedded Microcontroller Optimization," IEEE Transactions on Evolutionary Computation, vol. 12, no. 2, pp. 203-219, 2008.

[12] Kirk, D. B. and Hwu W. W. 2010. Programming Massively Parallel Processors: A Hands-on Approach. Burlington, MA, USA: Morgan Kaufman, Elsevier.

[13] Sanders, J. and Kandrot, E. 2010. CUDA by Example: An Introduction to General Purpose GPU Programming. Upper Saddle River, NJ: Addison-Wesley.

[14] NVIDIA Website, www.nvidia.com.

[15] C. W. Ahn and R. S. Ramakrishna, "Elitism based compact genetic algorithms," IEEE Transactions on Evolutionary Computation, vol. 7, no. 4, pp. 367-385.