Parallel Genetic Algorithms on Programmable Graphics Hardware

Qizhi Yu[1], Chongcheng Chen[2], and Zhigeng Pan[1,2]

[1] College of Computer Science, Zhejiang University, Hangzhou 310027, P.R. China
qizhi.yu@gmail.com
[2] Spatial Information Research Center of Fujian Province,
Fuzhou University, Fuzhou 350002, P.R. China

Abstract. Parallel genetic algorithms are usually implemented on parallel machines or distributed systems. This paper describes how fine-grained parallel genetic algorithms can be mapped to programmable graphics hardware found in commodity PC. Our approach stores chromosomes and their fitness values in texture memory on graphics card. Both fitness evaluation and genetic operations are implemented entirely with fragment programs executed on graphics processing unit in parallel. We demonstrate the effectiveness of our approach by comparing it with compatible software implementation. The presented approach allows us benefit from the advantages of parallel genetic algorithms on low-cost platform.

1 Introduction

Genetic algorithms (GAs) are robust search algorithms inspired by the analogy of natural evolutionary processes [1]. They have demonstrated to be particularly successful in the optimization problems. As many GA solutions require a significant amount of computation time, a number of parallel genetic algorithms (PGAs) have been proposed in past decades [2][3]. These algorithms differ principally from the classical sequential genetic algorithm, but they seem to have even better optimization quality [4]. Previous proposed parallel implementations usually rely on parallel computers, distributed systems or specialized GA hardware which are not easily available to the common users. The goal of this paper is to implement PGA by utilizing graphics hardware found in PC.

The graphics processors (GPUs) on today's commodity video cards have evolved into an extremely powerful and flexible processor. Modern GPUs perform floating-point calculations much faster than today's CPUs [5]. Furthermore, instead of offering a fixed set of functions, current GPUs allow a large amount of programmability [6]. These desirable properties have attracted lots of research efforts to utilize GPUs for various non-graphics applications in recent years [7][8][9][10][11][12]. Previous research work has already shown that GPUs are especially adept at SIMD computation applied to grid data [9]. Therefore, we can envision that some type of parallel genetic algorithms should be a good fit for commodity programmable GPUs.

L. Wang, K. Chen, and Y.S. Ong (Eds.): ICNC 2005, LNCS 3612, pp. 1051–1059, 2005.

In this paper, we present a novel implementation of fine-grained parallel genetic algorithm on the GPU. Real-coded individuals of a population are represented as a set of 2D texture maps. We perform BLX-α crossover and non-uniform mutation by executing a fragment program on every pixel at each step in a SIMD-like fashion. Thus, when application related fitness evaluation is assumed to be implemented on GPU, the GA iterations can run entirely on GPU. We will demonstrate the effectiveness of GPU implementation by applying it to function optimization problem. Relative to software implementation, a speedup of about 15 times has been achieved with population size 512^2.

The rest of the paper is organized as follow: The subsequent section gives background of both genetic algorithms and graphics hardware to facilitate understanding of our implementation. In Section 3, we describe the proposed GPU-based implementation. Section 4 presents performance results, and the paper concludes with suggestions for future work in Section 5.

2 Background

2.1 Genetic Algorithms

A simple GA starts with a population of solutions encoded in one of many ways. The GA determines each string's strength based on an objective function and performs one or more of three genetic operators on certain strings in the population. As described in Golberg [13]: in general terms, a genetic algorithm consists of four parts.

1. Generate an initial population.
2. Select pair of individuals based on the fitness function.
3. Produce next generation from the selected pairs by applying pre-selected genetic operators.
4. If the termination condition is satisfied stop, else go to step 2.

The termination condition can be either:

1. No improvement in the solution after a certain number of generation.
2. The solution converges to a pre-determined threshold.

In real-code GA, a solution is directly represented as a vector of real-parameter decision variable [14]. This coding scheme is particularly natural when tackling optimization problems of parameters with variable in continuous domains.

It has long been noted that genetic algorithms are well suited for parallel execution. There are three main type of parallel GAs: master-slave, fine-grained, and coarse-grained [2]. In a master-slave model, there is a single population just as in sequential GA, but the evaluation of fitness is distributed among several processors. In a coarse-grained model, the GA population is divided into multiple subpopulations. Each subpopulation evolves independently, with only occasional exchanges of individuals between subpopulations. In a fine-grained model, individuals are commonly mapped onto a 2D lattice, with one individual per node. Selection and crossover are restricted to a small and overlapping neighborhood.

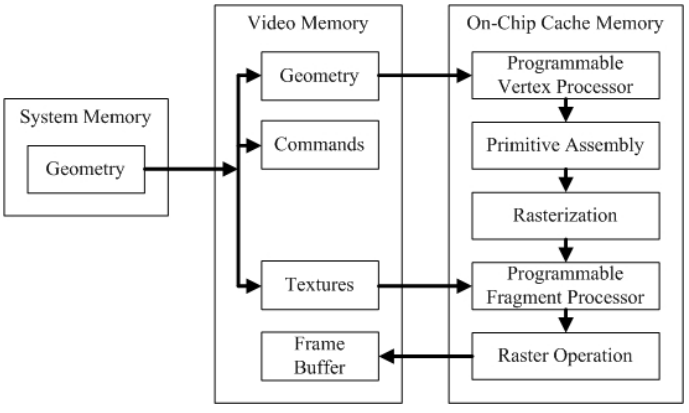

Fig. 1. The programmable graphics pipeline

2.2 Graphics Hardware

Graphics hardware is originally designed for accelerating rendering images. Figure 1 shows a simplified pipeline of modern programmable GPU. First, commands, textures, and vertex data are received from the host CPU through shared buffers in system memory or local frame-buffer memory. The vertex processor allow for a program to be applied to each vertex in the object, performing transformations and any other per-vertex operation the user specifies. Vertices are then grouped into primitives, which are point, lines, or triangles. Next, rasterization is the process of determining the set of pixels covered by a geometric primitive. After this, the results of rasterization stage, a set of fragments, are processed by a program which runs in the programmable fragment processor. Meanwhile, the programmable fragment processor also supports texturing operations which enable the processor to access a texture image using a set of texture coordinates. Finally, the raster operations stage performs a sequence of per-fragment operations immediately before updating the frame buffer.

Graphics cards hardware have features which help parallelism. A GPU contains a multiple number of pixel pipelines which process data in parallel (sixteen in our case). These pixel pipelines are each SIMD processing elements, carrying out operations typically on four color components in parallel [5].

3 A Real-Coded Parallel Genetic Algorithm on the GPU

3.1 Algorithm Overview

In this paper, we adopt the fine-grained parallel model suitable for SIMD implementation. A typical fine-grained parallel GA has been proposed and studied in [4]. We adopt a 2D toroidal grid as the spatial population structure where each grid point contains one individual. The neighborhood defined on the grid

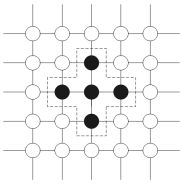

Fig. 2. Spatial population structure and neighborhood shape

always contains 5 individuals: the considered one plus the North, East, West, and South individuals (see Figure 2).

The crossover operator defines the procedure for generating a child from two parent genomes. For each individual, the best individual in its neighborhood will be selected as one of the parents, while the other one is itself.

Several crossover operators can be defined for real representation: averaging crossover, uniform crossover and blend crossover [14]. In this work, blend crossover is used. Let us assume that and are two chromosomes that have been selected to apply crossover to them. Blend crossover operator randomly selects a value for each offspring gene y_i, using a uniform distribution within the range:

$$[C_{\min} + \alpha \cdot I, C_{\max} - \alpha \cdot I]$$

where $C_{\min} = \min\{x_i^1, x_i^2\}$, $C_{\max} = \max\{x_i^1, x_i^2\}$, $I = C_{\max} - C_{\min}$, and α is the tunable parameter, the higher the value of α the more explorative the search.

Mutation operation is the final step of genetic operation. The role of mutation in GA is to restore lost or unexpected genetic material into a population to prevent the premature convergence of GA to a local result. Some of the commonly used mutation operators for real-coded GA are reviewed in [14]. In our approach, a non-uniform mutation [15] is adopted. If the operator is applied at generation step t and tmax is the maximum number of generations then the new value of the i-th gene in an individual will be:

$$y_i = \begin{cases} x_i + \delta \cdot (U_i - x_i) & \tau = 0 \\ x_i - \delta \cdot (x_i - L_i) & \tau = 1 \end{cases}$$

where τ is a random number taking value 0 or 1 with equal probability, L_i and U_i are the lower bound and upper bound of x_i, and

$$\delta = 1 - r^{(1-t/t_{\max})^b}$$

where r is a random number within the range $[0, 1]$ and b is a user defined parameter.

3.2 Representation of Population

In this section we describe the internal representation of population. If the GPU is to perform GA operators for us, the first thing we need to do is to represent

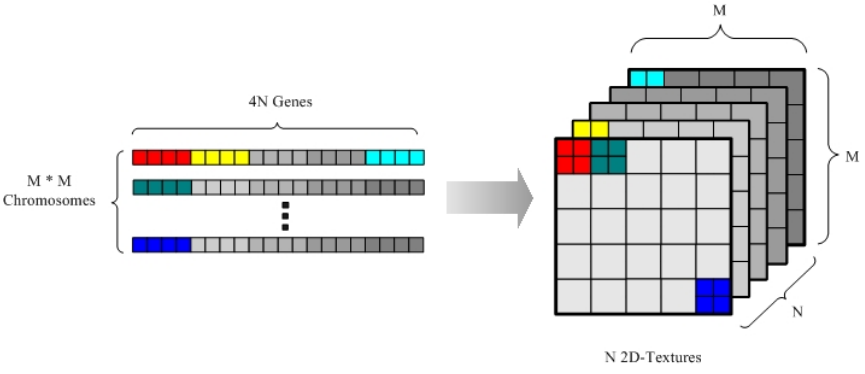

Fig. 3. The representation of chromosomes in a population as a set of 2D textures is shown

population data in a format that is accessible by the GPU. The general idea is to store population in a set of texture maps and to exploit fragment programs to implement genetic operators.

In our representation, the chromosome of each individual is sequentially divided into several segments which are distributed in a number of textures with the same position (see Figure 3). Every segment contains four genes packed into a single pixel's RGBA channels. We call those textures *population textures*. Another 2D texture map, called *fitness texture*, is used to hold the fitness scores of each individual in the population. The position of the fitness of a particular individual maps to the position of the individual in the population.

The proposed representation enables the efficient computation of genetic operators. It has several advantages: First, it naturally keeps 2D grid topology of the population described in Section 3.1. Second, for each individual, fragment programs of genetic operators only need lookup considered pixel or near neighborhood pixels in each texture map. Thus it keeps the memory footprint of each fragment as low as possible to efficiently utilize texture cache. Third, packing of four concessive genes in one texel makes use of the wide SIMD computations carried out by the GPU. Up to four times as many genes can be processed simultaneously.

3.3 Fitness Evaluation

It is important to emphasize that our framework is designed for solving problems whose fitness function can be implemented entirely in GPU. Only in this case can we avoid the bottleneck of reading population data from graphics hardware to system memory in each iteration of GA. On the other hand, executing fitness evaluation on GPU can take advantage of the GPU's parallel processing capabilities,

Implementation of fitness evaluation on GPU is application related. In some cases, such as solving function optimizer problems, fitness evaluation can be

easily implemented in a single fragment program. For more complicated applications, we refer readers to a homepage of research on general purpose use of GPU (http://www.gpgpu.org). After the fragment program responsible for fitness evaluation has been executed, the fitness values are stored in fitness texture. This texture is then redisplayed in following rendering pass, and other fragment program is run to perform genetic operators.

3.4 Random Numbers Generator

As described above in Section 3.1, we can find random numbers are involved in both crossover and mutation operator. However, current graphics hardware does not provide the function for generating random numbers. We use the Linear Congruential Generator (LCG) to generate pseudo-random numbers [16]:

$$I_{j+1} = a \cdot I_j + c \pmod{m}$$

where m is called the modulus, and a and c are multiplier and the increment respectively. LCG can be implemented in a simple fragment program. We store a matrix of random numbers in a texture called *random texture*. It is updated once by the fragment program in each iteration of GA loop.

3.5 Genetic Operators

Selection, crossover and mutation operators described in Section 3.1 can be easily mapped to a single fragment program. The fragment program needs lookup *population textures*, *fitness texture* and *random texture* described in above sections. System parameters such as mutation probabilities and crossover probabilities etc. are passed to the fragment program by uniform parameters. We invoke the fragment program by rendering a screen-parallel quadrilateral. The result is written into a new population texture.

In our implementation, for a population represented by n population textures, n rendering passes have to be performed in every generation of GA. In each rendering pass, four genes of each chromosome are processed parallelly. This is possible because the crossover operator and mutation operator we used all can be performed on independent gene.

4 Experimental Results

Our performance results were measured using an AMD Athlon 2500+ CPU with 512M RAM and an NVidia GeForce 6800GT GPU. The GPU-based implementation was developed with Cg code [6]. We used the Colville minimization problem as benchmark. It is defined as:

$$\begin{aligned} f(\overline{x}) = {} & 100(x_1^2 - x_2)^2 + (1 - x_1)^2 + 90(x_3^2 - x4)^2 + (1 - x_3)^2 \\ & + 10.1((1 - x2)^2 + (1 - x_4)^2) + 19.8(x2 - 1)(x4 - 1) \end{aligned}$$

Table 1. GA Parameters

Parameters	Value
Crossover Rate	1.0
Mutation Rate	0.05
Blend Crossover Parameter α	0.5
Non-uniform Mutation Parameter b	3

Table 2. Time cost and speed up for different GA module (500 generations)

Population Size	Genetic Operators			Fitness Evaluation		
	GPU(s)	CPU(s)	Speedup	GPU(s)	CPU(s)	Speedup
32^2	0.211	0.296	1.4x	0.044	0.013	0.3x
64^2	0.262	1.201	5.8x	0.046	0.062	1.4x
128^2	0.444	5.230	11.8x	0.074	0.587	7.9x
256^2	1.187	21.209	17.9x	0.176	2.725	15.4x
512^2	4.075	81.882	20.1x	0.602	10.299	17.1x

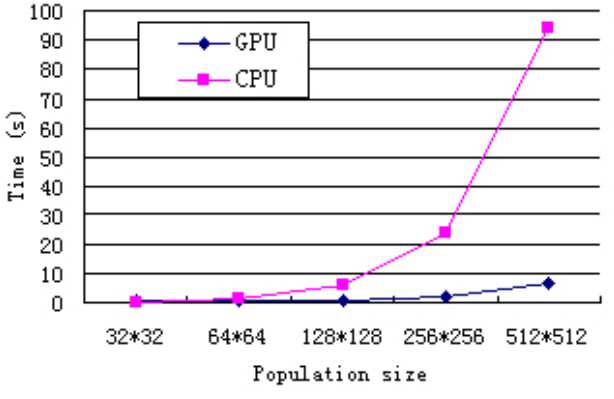

Fig. 4. The effects of population size on the run time (500 generations)

where $-18 \leq x_i \leq 10, i = 1, 2, 3, 4$; with the global solution $(1, 1, 1, 1)$ and $f(1, 1, 1, 1) = 0$.

GPU-based implementation was compared with software implementation running on single CPU with different population size. GA parameters are shown in Table 1. Figure 4 shows GPU-based implementation is much faster than the software implementation. We see that speedup increases as we increase the population size. Table 2 shows performance improvement of the GPU-based implementation stems from both genetic operators and fitness evaluation. The results also show that when the population size is 32^2, fitness evaluation of GPU-based implementation is slower than that of software version. This happens because

when the objective function is simple and meanwhile the population is small, the evaluation time is mainly consumed by the overhead of graphics pipeline.

5 Conclusion

In this work, we have presented a novel implementation of parallel genetic algorithms on commodity graphics hardware. Our approach gives a representation of population suitable for GPU processing. All genetic operators have been implemented on GPU. Tests on a function optimization problem show that the larger the population size is, the better speedup over the software implementation can be achieved. Our work has provided a promising platform for implementation of PGAs. Looking toward future, programmable GPUs are on a much faster performance growth than CPUs. They also have many other advantages: inexpensive, readily available, easily upgradeable, and compatible with various operating systems and hardware architectures.

There are still several constrains in our approach. For problems whose fitness function is not suitable for GPU implementation, the performance of our method will be seriously limited because of the bottleneck of transferring data between system memory and video memory in each GA loop. Another limitation is that commonly used binary encoding scheme of GAs seems hard to be implemented on the GPU because there is no bit-operator supported in current GPUs.

In the future, we will apply the presented approach in real-world problems such as GA-based image processing [17]. Another future work is further implementations of other variants of genetic algorithms. Using GPU cluster [18] to perform parallel genetic algorithms is also of interest.

Acknowlegement

This project is co-supported by 973 Program (No.2002CB312100) and Excellent Youth Teacher Program of MOE in China.

References

1. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press Cambridge, MA, USA (1992)
2. Tomassini, M.: A survey of parallel genetic algorithms. World Scientific **III** (1995) 87–118
3. Konfrst, Z.: Parallel genetic algorithms: advances, computing rends, applications and perspectives. In: Parallel and Distributed Processing Symposium. (2004) 162
4. Spiessens, P., Manderick, B.: A massively parallel genetic algorithms: Implementation and first analysis. In: Int. Conf. Genetic Algorithms, San Diego, Morgan Kaufmann Publishers (1991) 279–285
5. Fernando, R.: GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Addison Wesley (2004)
6. Fermando, R., Kilgard, M.J.: The Cg Tutorial. Addision-Wesley (2003)

7. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general-purpose computing: a framework and analysiy. In: Internaltional Symposium on Microarchitecture, Istanbul, Turkey, IEEE Computer Society Press (2002) 306–317
8. Krger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. ACM Transactions on Graphics **22** (2003) 908–916
9. Harris, M.J.: Real-Time Cloud Simulation and Rendering. Dissertaion, University of North Carolina at Chapel Hill (2003)
10. Bolz, J., Farmer, I., Grinspun, E., Schroder, P., Schrder, P.: Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. ACM Transactions on Graphics **22** (2003) 917–924
11. Hillesland, K.E., Molinov, S., Grzeszczuk, R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. ACM Transactions on Graphics **22** (2003) 925–934
12. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: International Conference on Management of Data. (2004) 215–226
13. Goldberg, D.: Genetic Algorithms in Search Optimization and Machine Learning. Addison Wesley, New York (1989)
14. Raghuwanshi, M., Kakde, O.: Survey on multiobjective evolutionary and real coded genetic algorithms. In: The 8th Asia Pacific Symposium on Intelligent and Evolutionary Systems, Cairns, Australia (2004)
15. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. 3rd edn. Springer (1996)
16. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C++: The Art of Scientific Computing. Cambridge University Press (2002)
17. Lukac, R., Plataniotis, K.N., Smolka, B., Venetsanopoulos, A.N.: Color image filtering and enhancement based on genetic algorithms. In: The 2004 IEEE International Symposium on Circuits and Systems. (2004)
18. Houston, M., Fatahalian, K., Sugerman, J., Buck, I., Hanrahan, P.: Parallel computation on a cluster of gpus. In Lastra, A., Lin, M., Manocha, D., eds.: ACM Workshop on General-Purpose Computing on Graphics Processors, Los Angeles, California (2004) 50