Parallel Evolutionary Algorithms on Graphics Processing Unit

Man-Leung Wong
Dept. of Computing & Decision Sci.
Lingnan University
Tuen Mun, Hong Kong
mlwong@ln.edu.hk

Tien-Tsin Wong
Dept. of Comp. Sci. & Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
ttwong@acm.org

Ka-Ling Fok
Dept. of Comp. Sci. & Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
klfok@cse.cuhk.edu.hk

Abstract- Evolutionary Algorithms (EAs) are effective and robust methods for solving many practical problems such as feature selection, electrical circuits synthesis, and data mining. However, they may execute for a long time for some difficult problems, because several fitness evaluations must be performed. A promising approach to overcome this limitation is to parallelize these algorithms. In this paper, we propose to implement a parallel EA on consumer-level *graphics cards*. We perform experiments to compare our parallel EA with an ordinary EA and demonstrate that the former is much more effective than the latter. Since consumer-level graphics cards are available in ubiquitous personal computers and these computers are easy to use and manage, more people will be able to use our parallel algorithm to solve their problems encountered in real-world applications.

1 Introduction

Evolutionary Algorithms (EAs) are weak search and optimization techniques inspired by natural evolution. They have been demonstrated to be effective and robust in searching very large and varied spaces in a wide range of applications such as feature selection [1], electrical circuits synthesis [2], and data mining [3, 4]. In general, EAs include all population-based algorithms that use selection and recombination operators to generate new search points in a search space. They include genetic algorithms, genetic programming, evolutionary programming, and evolution strategies [5].

Although EAs are effective in solving many practical problems in science, engineering, and business domains, they may execute for a long time to find solutions for some huge problems, because several fitness evaluations must be performed. A promising approach to overcome this limitation is to parallelize these algorithms for parallel, distributed, and networked computers. However, these computers are relatively more difficult to use, manage, and maintain. Moreover, some people may not have access to this kind of computers. Consequently, we propose to implement a parallel EA on consumer-level graphics cards which are available in ubiquitous personal computers. Given the ease of use, maintenance, and management of personal computers, more people will be able to use our parallel algorithm to solve huge problems encountered in real-world applications such as data mining.

In the following section, graphics processing unit will be discussed. We will present our parallel evolutionary algorithm in Sections 3 and 4. A number of experiments have been performed and the experimental results will be discussed in Section 5. We will give a conclusion and a description of our future work in the last section.

2 Graphics Processing Unit

In the last decade, the need from the multimedia and games industries for accelerating 3D rendering has driven several graphics hardware companies devoted to the development of high-performance parallel graphics accelerator. This results the birth of GPU (Graphics Processing Unit), which handles the rendering requests using 3D graphics application programming interface (API). The whole pipeline consists of the transformation, texturing, illumination, and rasterization to the framebuffer. The need for cinematic rendering from the games industry further raised the need for programmability of the rendering process. Starting from the recent generation of GPUs launched in 2001 (including nVidia GeforceFX series and ATI Radeon 9800 and above), developers can write their own C-like programs, which are called *shaders*, on GPU. Due to the wide availability, programmability, and high-performance of these consumer-level GPUs, they are cost-effective for, not just game playing, but also scientific computing.

These shaders control two major modules of the rendering pipeline, namely vertex and fragment engines. As an illustration to the mechanism in GPU, we describe the rendering of a texture-mapped polygon. The user first defines the 3D position of each vertex through the API in graphics library (OpenGL or DirectX). It seems irrelevant to define 3D triangles for evolutionary computation. However, such declaration is necessary for satisfying the input format of the graphics pipeline. In our application, we simply define 2 triangles that cover the whole screen. The texture coordinate associating with each vertex is also defined at the same time. These texture coordinates are needed to define the correspondence of elements in textures (input/output data) and the pixels on the screen (shaders are executed on per-pixel basis). The defined vertices are then passed to the vertex engine for transformation (dummy in our case).

For each vertex, a vertex shader (user-defined program) is executed (Fig. 1). The shader program must be *Single-Instruction-Multiple-Data* (SIMD) in nature, *i.e.* the same set of operations has to be executed on different vertices. The polygon is then projected onto the 2D screen and rasterized (discretized) into many fragments (pixels) in the framebuffer as shown in Fig. 1. From now on, the two terminologies, pixel and fragment, are interchangeable through-

out this paper. Next, the fragment engine takes place. For each pixel, a user-defined fragment shader is executed to process data associated with this pixel. Inside the shader, the input textures can be fetched for computation and results are output via the output textures. Again, the fragment shader must also be SIMD in nature.

As an example of utilizing GPU for scientific computing, we illustrate the addition of two $M \times N$ matrices, P and Q. Firstly, we define two right triangles (one upper and one lower) covering the $M \times N$ pixels as shown on the left hand side of Fig. 2. The vertex shader basically does nothing but only projects the six vertices (of two triangles) onto the 2D screen. After rasterization, these two triangles are broken down into $M \times N$ fragments (or pixels). For each pixel, a fragment shader is executed. We then fed matrices P and Q to this shader as two *input textures* (Fig. 2). A texture is basically an image with each pixel composed of four components, (r, g, b, α). Each component can be represented as 32-bit floating point. Therefore, one way to add two matrices is to store P's elements in the r component of one input texture and Q's elements in the r component of another texture. Obviously, a more compact and practical representation is to store elements of P and Q in two components, say r and b, of the same texture. For presentation clarity, we use two input textures. As the fragment shader is executed at each pixel (x, y) *independently* and *in parallel*, it only contains one single addition statement and no looping is needed (Fig. 2). The statement fetches and sums $P(x, y).r$ and $Q(x, y).r$, and stores the output in the third texture, $O(x, y).r$. The notation $.r$ specifies the r component of the pixel. The high performance is mainly contributed by this SIMD-type parallelism. Most GPU nowadays impose certain limitations on the usage of textures. For example, the total number of textures being accessed simultaneously is usually limited (e.g. 16 textures on nVidia GeForceFX 6800). Furthermore, the input texture cannot be used for output.

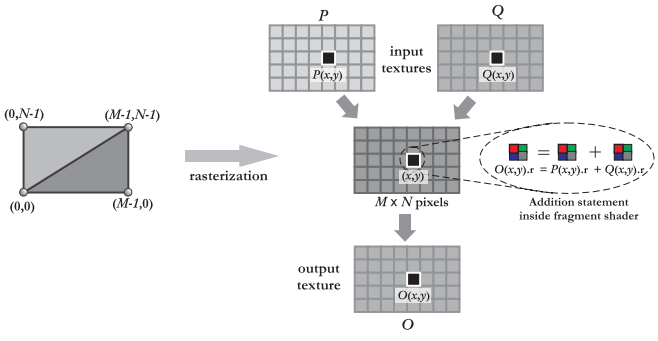

Figure 2: Addition of two matrices on GPU.

3 Data Organization

Suppose we have μ individuals and each contains k variables (genomes). The most natural representation for an individual is an array. As GPU is tailored for parallel processing and optimized multi-channel texture fetching, all input data to GPU should be loaded in the form of *textures*. Fig. 3 shows how we represent μ individuals in form of texture. Without loss of generality, we take k=32 as an example of illustration throughout this paper.

As each pixel in the texture contains *quadruple* of 32-bit floating point values (r, g, b, α), we can encode an individual of 32 genomes into 8 pixels. In other words, the memory is more efficiently utilized if k is multiple of 4. This is also why we take $k = 8 \times 4 = 32$ as a working example. Instead of mapping an individual to 8 consecutive pixels in the texture, we divide an individual into quadruple of 4 genomes. The same quadruples from all individuals are grouped and form a tile in the texture as shown in Fig. 3. Each tile is $w \times h = \mu$ in size. The reason we do not adopt the *consecutive-pixel* representation is that the implementation will be complicated when k varies. Imagine the complication of genomes' offsets within the texture when k increases from 32 to 48. On the other hand, the *fragmentation-and-tiling* representation is more scalable because increasing k can be easily achieved by adding more tiles. In our specific example of $k = 32$, 4×2 tiles are formed. It is up to user to decide the organization of these tiles in the texture. The first tile (upper-left tile) in Fig. 3 stores genomes 1 to 4, while the next tile stores genomes 5 to 8, and so on.

Texture on GPU is not as flexible as main memory. Current GPUs impose several limitations. One of them is the size of texture must not exceed certain limit, *e.g.* 4096×4096 on nVidia GeforceFX 6800. In other words, to fit the whole population in one texture on our GPU, we must satisfy $k\mu \leq 4 \times 4096^2$. For extremely large populations with a large number of variables, multiple textures have to be used. Note that there are also limitation on the total number of textures that can be accessed simultaneously. The actual number varies on different GPU models. Normally, at least 16 textures can be supported.

4 Evolutionary Programming on GPU

Evolutionary programming (EP) and genetic algorithm (GA) have been both successfully applied to several numerical and optimization problems. While classical GA requires the processes of crossover and mutation, EP requires the mutation process only. Hence, for each generation of evolution, EP is less computational intensive than GA. When implementing on GPU, the crossover process of GA induces more *rendering passes* than that of EP.

One complete execution of the fragment shader is referred as one rendering pass. On current GPU, there is a significant overhead for each rendering pass. The more rendering passes are needed, the slower the program is. Since fragment shaders are executed independently on each pixel, no information sharing is allowed among pixels. If the computation result of a pixel A has to be used for computing an equation at pixel B, the computation result of A must be written to an output texture first. This output texture has to be fed to the shader for computation in *next* rendering pass. Therefore, if the problem being tackled involves a chain of data dependency, more rendering passes are needed, and hence the speed-up is decreased.

Since the crossover process of GA requires more passes

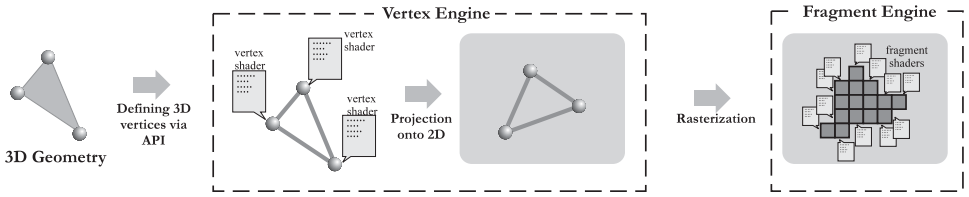

Figure 1: The 3D rendering pipeline.

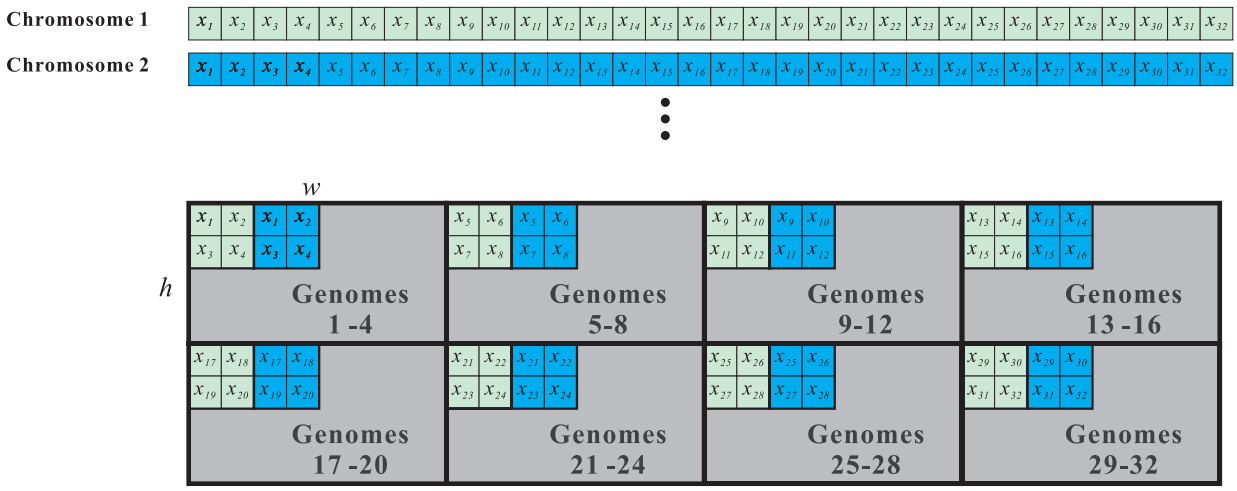

Figure 3: Representing individuals of 32 genomes on textures.

and more data transfer than that of EP, EP is more *GPU-friendly* (efficient to implement on GPU) than GA. Hence, in this paper, we study the GPU implementation of EP instead of the classical GA. Without loss of generality, we assume the optimization is to minimize a cost function. Hence, our EP is used to determine a $\vec{x}_{\min}$, such that

$$\forall \vec{x}, f(\vec{x}_{\min}) \leq f(\vec{x})$$

where $\vec{x} = \{x_i(1), x_i(2), \ldots, x_i(k)\}$ is the individual containing k variables; $f : R^n \mapsto R$ is the function being optimized. We implement a fast evolutionary programming (FEP) based on Cauchy mutation [6] as follows:

1. Generate the initial population of μ individuals, each of which can be represented as a set of real vectors, $(\vec{x_i}, \vec{\eta_i}), i = 1, \ldots, \mu$. Both $\vec{x_i}$ and $\vec{\eta_i}$ contain k independent variables,
   $\vec{x_i} = \{x_i(1), \ldots, x_i(k)\}$
   $\vec{\eta_i} = \{\eta_i(1), \ldots, \eta_i(k)\}$

2. Evaluate the fitness score for each individual (x_i, η_i), $i = 1, \ldots, \mu$, of the population based on the objective function, $f(\vec{x})$.

3. For each parent $(\vec{x_i}, \vec{\eta_i})$, $i = 1, \ldots, \mu$, create an offspring $(\vec{x_i}', \vec{\eta_i}')$ as follows:
   for $j = 1, \ldots, k$
   $x_i'(j) = x_i(j) + \eta_i(j)R(0,1)$,
   $\eta_i'(j) = \eta(j)\exp(\frac{1}{\sqrt{2k}}R(0,1) + \frac{1}{\sqrt{2\sqrt{k}}}R_j(0,1))$
   where $x_i(j), \eta_i(j), x_i'(j)$, and $\eta_i'(j)$ denote the j-th component of $\vec{x_i}, \vec{\eta_i}, \vec{x_i}'$, and $\vec{\eta_i}'$ respectively. $R(0,1)$ denotes a normally distributed 1D random number with zero mean and standard deviation of one. $R_j(0,1)$ indicates a new random variable for each value of j.

4. Calculate the fitness of each offspring $(\vec{x_i}', \vec{\eta_i}')$.

5. Conduct pairwise comparison over the union of parents $(\vec{x_i}, \vec{\eta_i})$ and offspring $(\vec{x_i}', \vec{\eta_i}')$, for $i = 1, \ldots, \mu$. For each individual, q (tournament size) opponents are chosen randomly from all the parents and offspring. For each comparison, if the individual's fitness is smaller than or equal to that of opponent, it receives a "`win`".

6. Select μ individuals out of $(\vec{x_i}, \vec{\eta_i})$ and $(\vec{x_i}', \vec{\eta_i}')$, $i = 1, \ldots, \mu$, that receive more `win`'s to be parents of next generation.

7. Stop if the stopping criterion is satisfied; otherwise go to Step 3.

In the above pseudocode, $\vec{x_i}$ is the individual evolving and $\vec{\eta_i}$ controls the vigorousness of mutation of $\vec{x_i}$. In general, the computation of FEP can be roughly divided into three types: (a) mutation and reproduction (step 3), (b) fitness value evaluation (steps 2 and 4), and (c) competition and selection (steps 5 and 6). These types of operations will be discussed in the following sub-sections.

4.1 Mutation and Reproduction

Unlike GA, EP omits crossover and carries out mutation only. Fogel [7] introduced EP using Gaussian distribution. Yao and Liu [6] proposed a mutation operation based on Cauchy distribution to increase the speed of convergence. From the pseudocode above, mutation operation is executed on each genome. Genomes are assumed to be independent of each other. Thus mutation process is perfectly parallelizable. In pure software (CPU for short) implementation, a loop is needed to perform mutation on each genome. On the SIMD-based GPU, a fragment shader is executed *in parallel* to perform mutation on each component (r, g, b, α) of each pixel. GPU solution is thus ideal for this independent mutation and can achieve significant speed-up.

To accomplish the mutation process on GPU, we designed two fragment shaders, one for computing $\vec{x}'$ and the other for $\vec{\eta}'$. Fig. 4 illustrates these two shaders graphically. The parents $\vec{x_i}$ and $\vec{\eta_i}$ are stored in two input textures while the offspring are generated and written to two output textures $\vec{x_i}'$ and $\vec{\eta_i}'$. One fragment shader is responsible for computing $\vec{x_i}'$ while the other is responsible for $\vec{\eta_i}'$. Besides, we also need two input textures of random numbers.

Mutation requires normally distributed random variables. Unfortunately, current GPU is not equipped with random number generator. Hence the random numbers have to be generated by CPU and fed to GPU in the form of input textures. We divide the process of random number generation into two steps. Firstly, CPU is used to generate random variables with uniform distribution. This is a sequential process. The generated random numbers are fed to GPU via input textures. Then, inside the two fragment shaders, GPU converts them from uniform distribution to Gaussian distribution in parallel.

Traditionally, the well-known Box-Muller transformation [8] is used to transform random numbers of uniformly distribution to normal distribution. The polar form of Box-Muller transformation algorithm provides even faster and more robust solution [9]. However, the number of iterations in Box-Muller transformation depends on the data values. Such data-dependent looping is undesirable for SIMD-based GPU, as various processors execute different numbers of iterations.

Instead of using Box-Muller transformation, we employ the direct inverse cumulative normal distribution function (ICDF) as it does not require looping. It trades accuracy for speed. The algorithm uses minimax approximation and the error introduced is relatively little. Our experiment shows that GPU implementation of ICDF is 2 times faster than CPU implementation of ICDF. ICDF on GPU is even more than 4 times faster than Box-Muller transformation on CPU.

Current GPU has a slow performance in data transferal from GPU texture to main memory. Therefore, such data transfer should be avoided as much as possible. Hence, our strategy is to keep the parent and offspring resided in GPU memory. Only the final result, after several generations of evolution, is transferred from GPU textures to main memory.

4.2 Fitness Value Evaluation

Fitness value evaluation determines the "goodness" of individuals. It is one of the core parts of EP. After each evolution, the fitness value of each individual in the current population is calculated. The result is then passed to the later stage of EP process. Each individual returns a fitness value by feeding the objective function f with the genomes of the individual. This evaluation process usually consumes most of the computational time.

Since no interaction between individuals is required during evaluation, the evaluation is fully parallelizable. Fig. 5 illustrates the evaluation shader graphically. Recall that the individuals are broken down into quadruples and stored in the tiles within the textures. The evaluation shader hence looks up the corresponding quadruple in each tile during the evaluation. The fitness values are output to an output texture of size $w \times h$, instead of $4w \times 2h$, because each individual only returns a single value.

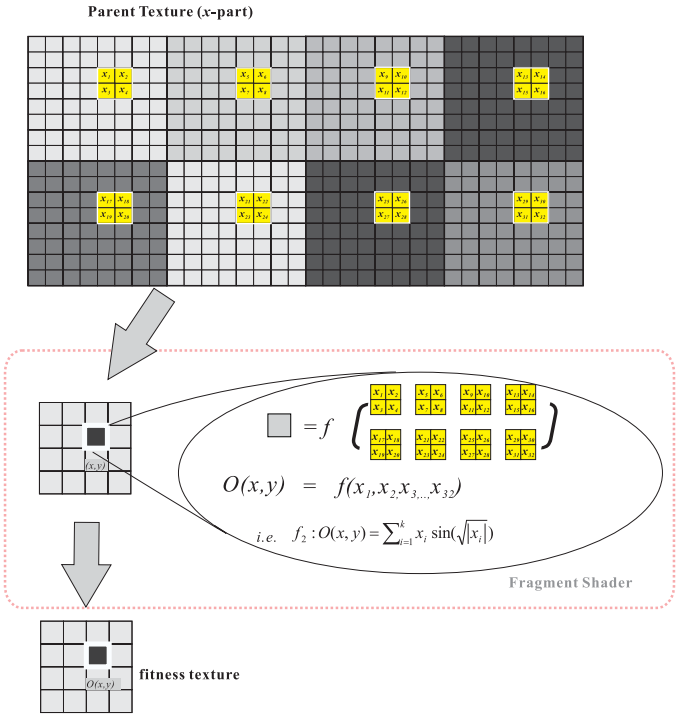

Figure 5: The shader for fitness evaluation.

4.3 Competition and Selection

Replacing the old generation is the last stage of each evolution. There are two major processes involved, competition and selection. EP employs a stochastic selection (soft selection) through the tournament schema. Each individual in the union set of parent and offspring population takes part in a q-round tournament. In each round, an opponent is randomly drawn from the union set of parents and offsprings. The number of opponents defeated is recorded by the variable `win`. After the tournament, a selection process takes place and chooses the best μ individuals having the highest `win` values as parents for next generation.

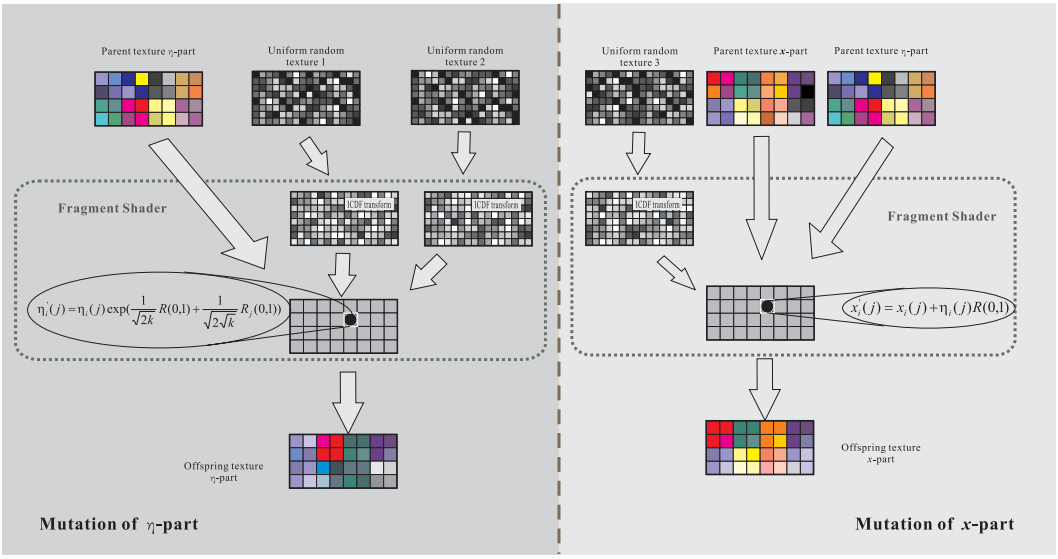

Figure 4: The two fragment shaders for mutation process.

4.3.1 Competition

Exploitation is the process of using information gathered to determine which searching direction is profitable. In EP, exploitation is realized by competition. Individuals in the population compete with q randomly drawn opponents, where q is the predefined tournament parameter. The considered individual wins if its fitness value is better than (in our case smaller than or equal to) that of the opponent. The times of winning is recorded in `win`. It tells us how good this individual is.

Competition can be done either on GPU or CPU. For GPU implementation, q textures of random values have to be generated by CPU and loaded to GPU memory for each evolution. On the other hand, for CPU implementation, only the fitness textures of both parent and offspring population have to be transferred from GPU memory to main memory. The final result of selected individuals is then transferred back to GPU memory.

It seems that GPU implementation should be faster than the CPU one, as it parallelizes the competition. However, our experiments show that using GPU to implement the competition is slower than that of using CPU. The major bottleneck of GPU implementation is the transfer of q textures towards GPU memory. It evidences the limitation of slow data transfer of current GPU. As the competition does not involve any time-consuming fitness evaluation (it only involves fitness comparison), the gain of parallelization does not compensate the loss due to data transfer. The data transfer rate is expected to be significantly improved in the future GPU.

4.3.2 Median Searching and Selection

After the competition process, selection is performed based on the `win` values. It selects the best μ individuals having highest `win` values and assigns them as the parents for the next generation. The most natural way is to sort the 2μ individuals in a descending order of `win` values. The first μ individuals are then selected. For large population size, the sorting time is unbearably slow even using $O(N \log(N))$ sorting algorithm.

Note that our goal is to pick the best μ individuals, instead of sorting the individuals. These two goals are different. We can pick the best μ individuals without sorting them if we know the median `win` value. The trick is to find the median without any sorting.

Median searching has been well studied. Floyd and Rivest [10] proposed a complex partition-and-conquer linear time algorithm. When the valid range of values is known and countable, median searching can be done easily. To illustrate the idea, we go through a running example in Fig. 6. Suppose we have $\mu = 9$ values (top of Fig. 6) and the valid range of value is an integer within $[0, q]$ where $q = 5$ in our example. Now, we construct $q + 1 = 6$ bins as shown in the middle of Fig. 6. Therefore, in a linear time, we scan through all 9 values and form a histogram showing the count of each valid value (bin). Next we compute the cumulative distribution function (bottom part of Fig. 6) based on the histogram. The median is the value whose cumulated sum exceeds half the number of elements, i.e. $\mu/2 = 4.5$. In our example, value 3 is the median. Both the construction of histogram and cumulative distributed function are linear in time complexity. Therefore, we can find the median in linear time without sorting. Once the median is known, we can scan through the fitness values of all individuals and select those with fitness values below or equal to the median. The process stops once μ individuals are selected.

4.3.3 Minimizing Data Transfer

To minimize the data transfer between the memory on GPU and the main memory, an index array storing the offset of each individual in the textures is constructed before the competition and selection process. During the selection, we only record the index of the selected individuals, instead of

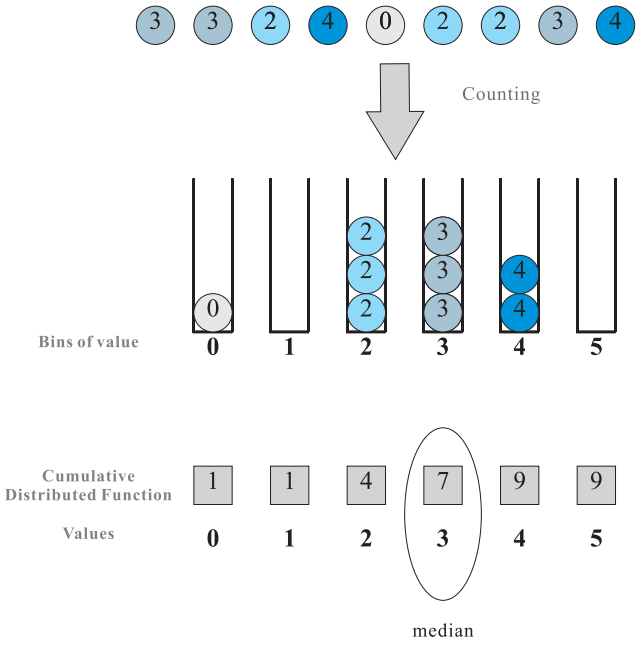

Figure 6: Running example of median picking algorithm.

Test Functions	N	S
$f_1 : \sum_{i=1}^{N} x_i^2$	32	$(-100, 100)^N$
$f_2 : \sum_{i=1}^{N} (\sum_{j=1}^{i} x_j)^2$	32	$(-100, 100)^N$
$f_3 : \sum_{i=1}^{N-1} \{100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\}$	32	$(-30, 30)^N$
$f_4 : -\sum_{i=1}^{N} x_i \sin(\sqrt{\lvert x_i \rvert})$	32	$(-500, 500)^N$
$f_5 : \sum_{i=1}^{N} \{x_i^2 - 10\cos(2\pi x_i) + 10\}$	32	$(-5.12, 5.12)^N$

Table 1: The set of test functions. N is the number of variables and S indicates the ranges of the variables.

the whole individuals (all genomes). The index array is then loaded to GPU in form of a texture. The actual individual replacement is performed on GPU based on the index texture. The final result is rendered to a texture which stores the individuals of the new generation. With this approach, the textures of individuals are always retained in the GPU memory. These textures of individuals are never transferred to the main memory during the evolution, until the final generation is obtained. Only fitness textures, random textures and index textures are transferred between GPU and CPU during the evolution. Since the fitness, random and index textures are smaller than the textures of individuals, this indexing approach minimizes the data transfer and improves the speed significantly.

5 Experimental Results

We applied EP with Cauchy distribution to a set of benchmark optimization problems. Table 1 summarizes the benchmark functions, number of variables and the search ranges. We conducted the experiments for 20 trials on both CPU and GPU. The average performance is reported in this paper. The experiment test bed was an Pentium IV 2.4 GHz with AGP 4X enabled consumer-level GeForce 6800 Ultra display card, with 512 MB main memory and 256 MB GPU memory. The following parameters were used in the experiments:

- population size: μ = 400, 800, 3200, 6400
- tournament size: $q = 10$
- standard deviation: $\sigma = 1.0$
- maximum number of generation: $G = 2000$

In these experiments, we find that better solutions can be obtained for all functions if a larger population size is used. However, EP with a larger population size will take longer execution time. Fig. 7 displays, by generation, the average execution time of the GPU and CPU approaches with different population sizes. From the curves in this figure, the execution time increases if a larger population is applied. However, our GPU approach is much more efficient than the CPU implementation because the execution time of the former is much less than that of the latter if the population size reaches 800. Moreover, the efficiency leap becomes larger when the population size increases.

The ratios of the average execution time of the GPU (CPU) approach with population sizes of 800, 3200, and 6400 to that of the corresponding approach with population size of 400 are summarized in Table 2. It is interesting to notice that, the CPU approach shows a linear relation between the number of individuals and the execution time, while our GPU approach has a sub-linear relation. For example, our GPU approach with population sizes of 400 and 800 take about the same execution time. Moreover, the execution time of our approach with population size of 6400 is about 3 times of that with population size of 400. Definitely, this is an advantage when huge population sizes are required in some real-life applications.

To study why our approach can achieve this phenomenon, the average execution time of different types of operations of the GPU (CPU) approach for the test function f_5 are presented in Table 3. It can be observed that the fitness evaluation time of our GPU approach with different population sizes are about the same, because all individuals are evaluated in parallel. Moreover, the mutation time does not increase proportionally with the number of individuals, because the mutation operations are also executed in parallel [1]. Similar results are also obtained for other test functions. Table 4 displays the speed-ups of our GPU approach with the CPU approach. The speed-ups depend on the population size and the problem complexity. Generally, GPU outperforms CPU when the population size is larger than or equal to 800. The speed-up ranges from about 1.25 to about 5.02. For complicated problems that require huge population sizes, we expect that GPU can achieve even better performance gain.

[1]The mutation time increases with the number of individuals, because our GPU approach requires a number of random numbers generated by CPU.

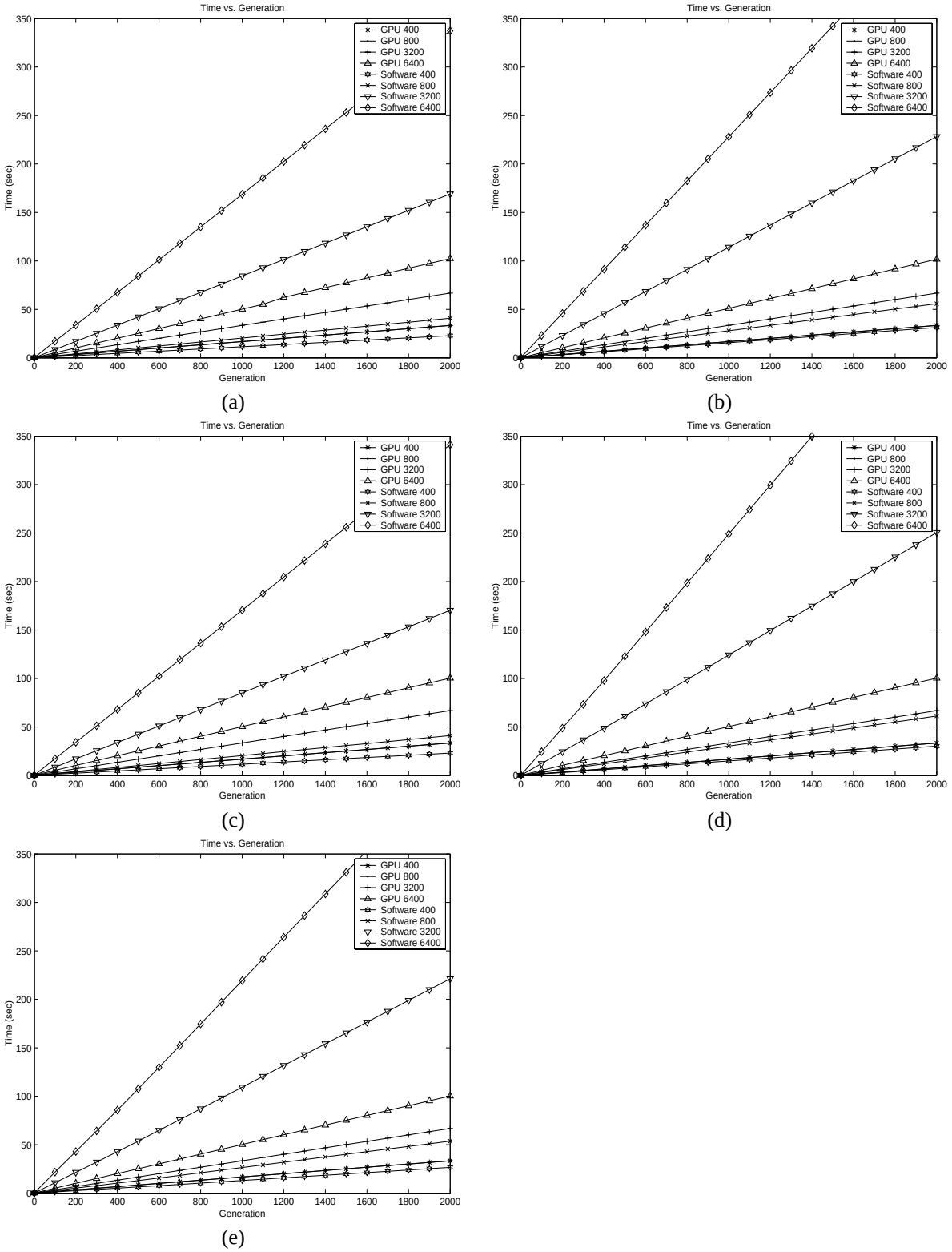

Figure 7: Execution time of the GPU and CPU approaches for functions f_1 - f_5. The results were averaged over 20 independent trials. (a)-(e) correspond to functions f_1 - f_5 respectively.

μ	type	competition & selection time (sec)	speed-up	fitness evaluation time (sec)	speed-up	mutation time (sec)	speed-up	total time (sec)	speed-up
400	CPU	3.32	0.80	7.28	0.28	16.19	7.39	26.79	0,82
	GPU	4.14		26.46		2.19		32.79	
800	CPU	6.70	0.91	14.86	0.68	32.49	9.00	54.05	1.65
	GPU	7.33		21.84		3.61		32.78	
3200	CPU	28.26	1.06	60.25	2.31	133.52	9.92	222.03	3.35
	GPU	26.72		26.12		13.46		66.30	
6400	CPU	56.69	1.08	118.91	5.41	267.30	10.44	442.95	4.42
	GPU	52.57		21.96		25.61		100.25	

Table 3: Experimental result summary of f_5.

	GPU					CPU				
μ	f_1	f_2	f_3	f_4	f_5	f_1	f_2	f_3	f_4	f_5
800	1.00	1.00	1.00	1.00	1.00	2.01	2.02	2.02	2.02	2.02
3200	2.02	2.02	2.02	2.02	2.02	8.30	8.24	8.37	8.12	8.29
6400	3.11	3.09	3.04	3.05	3.05	16.57	16.45	16.75	16.40	16.53

Table 2: The ratios of the average execution time of the GPU (CPU) approach with different population sizes to that with population size of 400.

μ	f_1	f_2	f_3	f_4	f_5
400	0.62	0.85	0.62	0.93	0.82
800	1.25	1.71	1.25	1.88	1.65
3200	2.55	3.45	2.57	3.74	3.35
6400	3.31	4.50	3.42	5.02	4.42

Table 4: The speed-up of the GPU approach.

6 Conclusion

In this research, we have implemented a parallel EP on consumer-level graphics cards and proposed indirect indexing and many optimization skills to achieve maximal efficiency. The parallel EP is a hybrid of master-slave and fine-grained models [11]. Competition and selection are performed by CPU (i.e. the master) while fitness evaluation, mutation, and reproduction are performed by GPU which is essentially a massively parallel machine with shared memory. Unlike other fine-grained parallel computers such as Maspar, GPU allows processors to communicate with any other processors directly, thus more flexible fine-grained EAs can be implemented on GPU. We have done experiments to compare our parallel EP on GPU and an ordinary EP on CPU. It is found that the speed-up factor of our parallel EP ranges from 1.25 to 5.02, when the population size is large enough. Moreover, there is a sub-linear relation between the population size and the execution time. Thus, our parallel EP will be very useful for solving difficult problems that require huge population sizes.

For future work, we plan to implement a parallel genetic algorithm on GPU and compare it with the approach reported in this paper.

Acknowledgment

This work is supported by The Chinese University of Hong Kong Young Researcher Award (Project No. 4411110) and the Earmarked Grant LU 3009/02E from the Research Grant Council of the Hong Kong Special Administrative Region.

Bibliography

[1] Il-Seok Oh, Jin-Seon Lee, and Byung-Ro Moon, "Hybrid genetic algorithms for feature selection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, pp. 1424–1437, 2004.

[2] John R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, 2003.

[3] M. L. Wong, W. Lam, K. S. Leung, P. S. Ngan, and J. C. Y. Cheng, "Discovering knowledge from medical databases using evolutionary algorithms," *IEEE Engineering in Medicine and Biology Magazine*, vol. 19, no. 4, pp. 45–55, 2000.

[4] Man Leung Wong and Kwong Sak Leung, "An efficient data mining method for learning Bayesian networks using an evolutionary algorithm based hybrid approach," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 4, pp. 378–404, 2004.

[5] David B. Fogel, *Evolutionary Computation: Toward a New Philosohpy of Machine Intelligence*, IEEE Press, 2000.

[6] X.Yao and Y.Liu, "Fast evolutionary programming," in *Evolutionary Programming V: Processdings of the 5th Annual Conference on Evolutionary Programming*. 1996, Cambridge, MA:MIT Press.

[7] David B. Fogel, "An introduction to simulated evolutionary optimization," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 3–14, 1994.

[8] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Annals of Mathematical Statistics*, vol. 29, pp. 610–611, 1958.

[9] D. E. Knuth, "The art of computer programming. volume 2: Seminumerial algorithms (second edition)," *Addison-Wesley, Menlo Park*, 1981.

[10] Robert W. Floyd and Ronald L. Rivest, "Expected time bounds for selection," *Communications of the ACM*, vol. 18(3), pp. 165–172, 1975.

[11] Erick Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms*, Kluwer Academic Publishers, 2000.