Evolutionary Computing on Consumer Graphics Hardware

Ka-Ling Fok and Tien-Tsin Wong, *Chinese University of Hong Kong*

Man-Leung Wong, *Lingnan University*

Parallel algorithms run on consumer-grade graphics hardware achieve better execution times than ordinary evolutionary algorithms and offer greater accessibility than those run on high-performance computers.

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

An EA starts with a set of individuals in the search space. This set forms the algorithm's *population*. Usually, the initial population is generated randomly using a uniform distribution. For each iteration, the algorithm evaluates each individual using the fitness function and invokes the termination function. The algorithm terminates if it finds acceptable solutions or spends the computational resources. Otherwise, it selects several individuals and copies them to replace individuals in the population that weren't selected for reproduction so that the population size remains constant. Then, the algorithm manipulates individuals in the population by applying different evolutionary operators. Individuals from the previous population are called *parents*; those created by applying evolutionary operators to the parents are called *offspring*. The consecutive processes of selection, manipulation, and evaluation constitute a generation of the algorithm.

Although EAs can help solve many practical problems in science, engineering, and business domains, they can take a long time to find solutions for huge problems because they need to perform several fitness evaluations. One way to overcome this is to parallelize EAs for parallel, distributed, and networked computers. However, these high-performance computers are relatively more difficult to access, use, manage, and maintain. Consequently, we propose implementing a parallel EA on consumer graphics cards, which you can find in many PCs. This will let more people use our parallel algorithm to solve large-scale, real-world problems such as data mining.

Graphics processing unit

In the last decade, demand from the multimedia and games industries have pushed graphics hardware companies to develop high-performance parallel graphics accelerators. This resulted in the birth of the GPU (graphics processing unit), which handles rendering requests using a 3D graphics API. The pipeline consists of transformation, texturing, illumination, and rasterization. The games industry's need for cinematic rendering led to the introduction of programmability into the rendering process. Starting with the recent generation of GPUs launched in 2001 (including the nVidia GeforceFX series and the ATI Radeon 9800 series or above), developers could write their own C-like programs, called *shaders*, on a GPU.

Shaders control two major modules of the ren-

1541-1672/07/$25.00 © 2007 IEEE
Published by the IEEE Computer Society

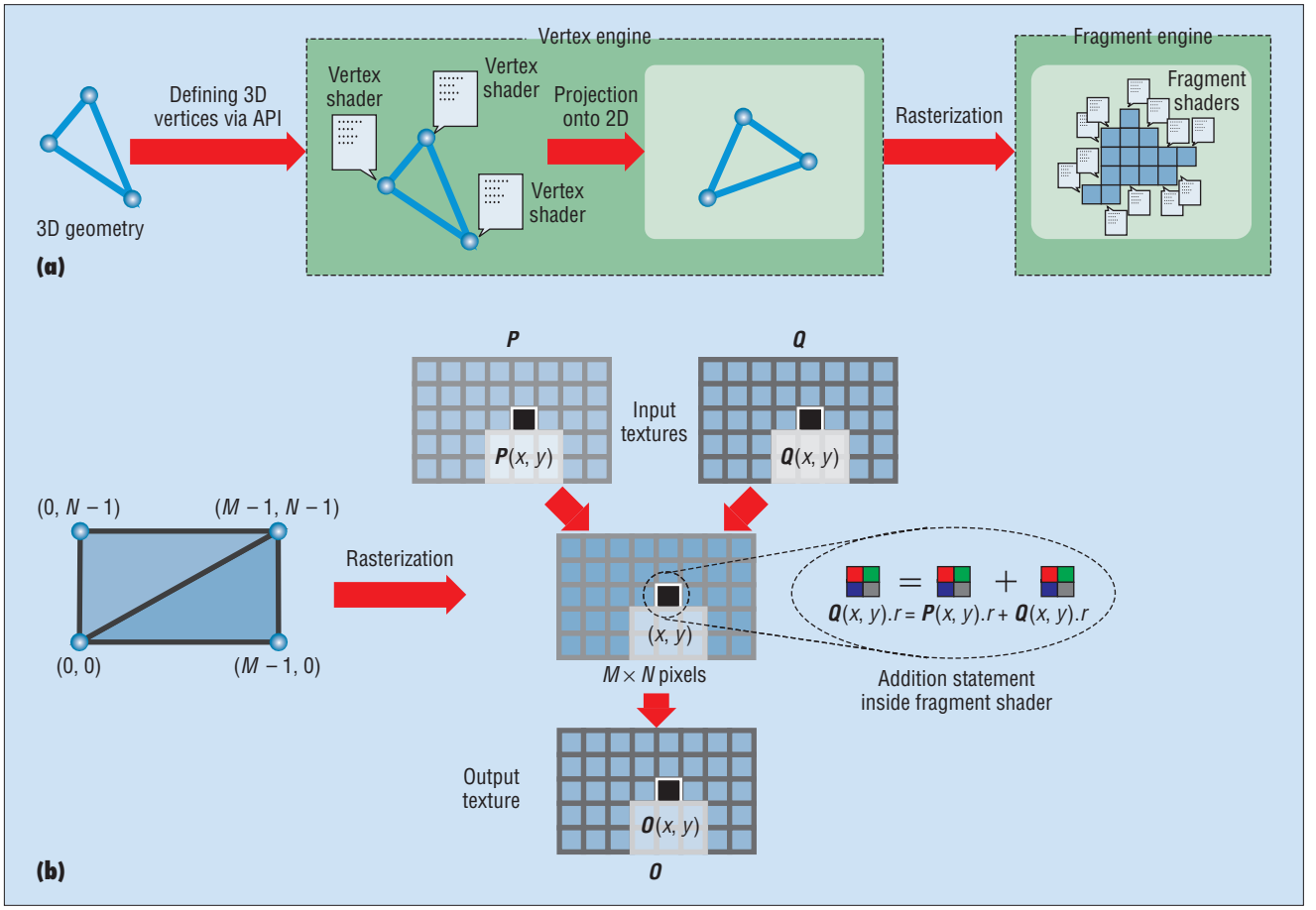

Figure 1. Illustration of a graphics processing unit: (a) the 3D rendering pipeline and (b) adding two matrices.

dering pipeline: the vertex and fragment engines. Take, for example, the rendering of a texture-mapped polygon. The user first defines each vertex's 3D position through the API in a graphics library (such as OpenGL or DirectX). It seems irrelevant to define 3D triangles for evolutionary computation. However, you need such a declaration to satisfy the graphics pipeline's input format. In our application, we define two triangles that cover the whole screen. At the same time, we also define the texture coordinate associated with each vertex. We need these texture coordinates to define the correspondence of elements in textures (I/O data) and the pixels on the screen (shaders are executed on a per-pixel basis). The defined vertices then pass to the vertex engine for transformation. In our case, the vertex engine performs only minimal operation.

For each vertex, the vertex engine executes a vertex shader (a user-defined program) (see figure 1a). The shader must be SIMD (single-instruction-multiple-data) in nature; that is, the shader must execute the same set of operations on different vertices. The engine then projects the polygon onto the 2D screen and rasterizes (discretizes) it into many fragments (pixels) in the frame buffer (see figure 1a). (From here on, we use the terms pixel and fragment interchangeably.) Next, for each pixel, the fragment engine executes a user-defined fragment shader to process data associated with this pixel. Inside the fragment shader, which must also be SIMD, the shader can fetch input textures for computation and output results via the output textures.

To demonstrate how you can use a GPU for scientific computing, we show the addition of two $M \times N$ matrices, $\boldsymbol{P}$ and $\boldsymbol{Q}$ (see figure 1b). First, we define two right triangles (one upper and one lower) covering the $M \times N$ pixels. The vertex shader basically does nothing but project the six vertices (of two triangles) onto the 2D screen. After rasterization, the engine breaks these two triangles into $M \times N$ fragments (or pixels). For each pixel, the fragment engine executes a fragment shader. We then feed matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ to this shader as two *input textures* (see figure 1b). A texture is basically an image, in which each pixel consists of four components, (r, g, b, α). We use the symbols r, g, b, and α following the convention in graphics. We can represent each component as 32-bit floating-point values. So, we could add the two matrices by storing $\boldsymbol{P}$'s elements in one input texture's r component and $\boldsymbol{Q}$'s elements in another texture's r component. Obviously, a more compact and practical representation is to store elements of $\boldsymbol{P}$ and $\boldsymbol{Q}$ in two components, say r and b, of the same texture. For clarity, we use two input textures. As the fragment shader executes at each pixel (x, y) independently and in parallel, it contains only a single addition statement and doesn't require looping (see figure 1b). The statement fetches and sums $\boldsymbol{P}(x, y).r$ and $\boldsymbol{Q}(x, y).r$, and stores the output in the third texture, $\boldsymbol{O}(x, y).r$. The notation $.r$ specifies the pixel's r component. This SIMD-type parallelism results in the algorithm's high performance.

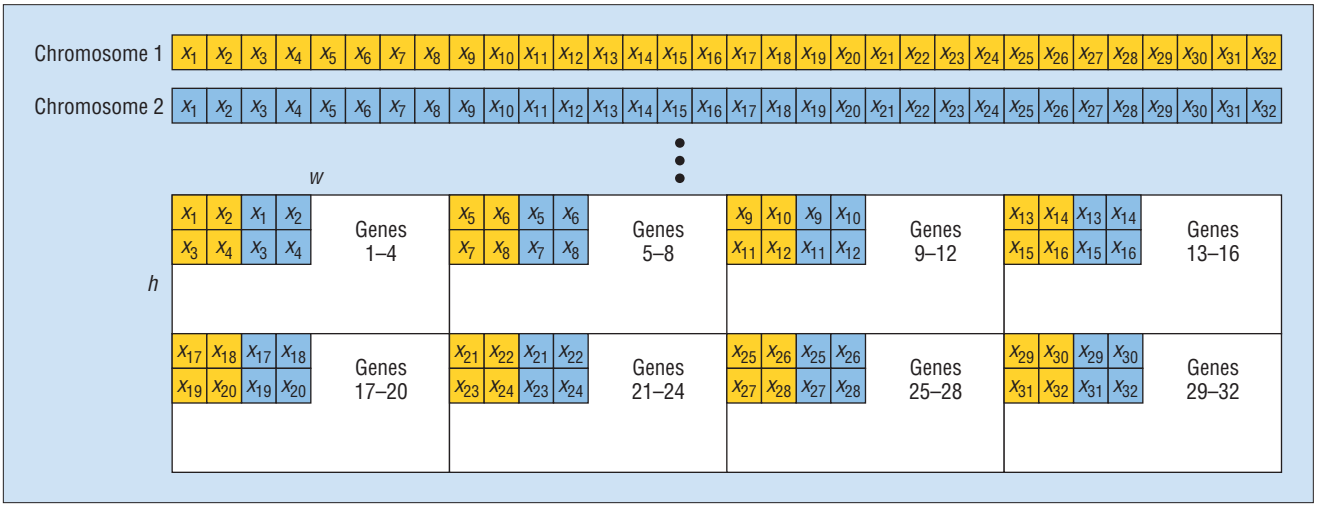

Figure 2. Representing individuals of 32 genes on textures.

Data organization

Suppose we have μ individuals, each containing k variables (genes). The most natural representation for an individual is an array. Because a GPU is tailored for parallel processing and for optimized, multichannel texture fetching, all input data to the GPU should be in the form of textures. Figure 2 shows how we represent μ individuals as textures. Without loss of generality, we use k = 32 as an example throughout this article. This number of variables reflects the typical size of real-world problems.

Because each pixel in the texture contains a quadruple of 32-bit floating-point values (r, g, b, α), we can encode an individual consisting of 32 genes into 8 pixels. We call this *consecutive-pixel representation*. In other words, the memory is used more efficiently if k is a 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 quadruples of 4 genes. We group the same quadruples from all individuals to form a tile in the texture (see figure 2). Each tile is $w \times h = \mu$ in size. We call this the *fragmentation-and-tiling representation*. We don't adopt the consecutive-pixel representation because such an implementation will be complicated when k varies. Imagine the complication of the genes' offsets within the texture when k increases from 32 to 48. However, the fragmentation-and-tiling representation is more scalable because we can easily increase k by adding more tiles. Another reason we don't adopt the consecutive-pixel representation is for the sake of visualization. Our example of k = 32 forms 4×2 tiles. The user must decide how to organize these tiles in the texture. The first (upper left) tile in figure 2 stores genes 1 to 4, the next tile stores genes 5 to 8, and so on.

Texture on a GPU isn't as flexible as main memory. Current GPUs impose several limitations. One is that the texture's size must not exceed a certain limit—for example, $4{,}096 \times 4{,}096$ on an nVidia GeforceFX 6800. In other words, to fit the whole population in one texture on our GPU, we must satisfy $k\mu \le 4 \times 4{,}096^2$. For extremely large populations with many variables, we need to use multiple textures. You can also access only a limited number of textures simultaneously. The number varies on different GPU models. Normally, they can support at least 16 textures. Moreover, you can't use the input texture for output.

GPU evolutionary programming

Evolutionary programming and genetic algorithms have been successfully applied to several numerical and optimization problems. Although a classic GA requires crossover and mutation, EP only requires mutation. So, for each generation of evolution, a GA is more computationally intensive than EP. More important, GA crossover induces higher data dependency than EP crossover. When implemented on a GPU, higher data-dependency leads to more *rendering passes*. A rendering pass is a complete execution of the fragment shader. On current GPUs, significant overhead exists for each rendering pass—the more rendering passes, the slower the program.

A simple scenario demonstrates why high data dependency induces more rendering passes. Consider pixels A and B. Because fragment shaders execute independently on each pixel, information sharing isn't allowed among pixels. If we need to use the result from pixel A's computation to compute an equation at pixel B, A's result must first be written to an output texture. We need to feed this output texture to the shader for computation in the next rendering pass. So, if the problem being tackled involves a chain of data dependency, it requires more rendering passes and lowers the speedup.

Because a GA's crossover process requires more passes and more data transfers, EP is more GPU friendly (more efficient to implement on a GPU) than a GA. In this article, we study a GPU implementation of EP instead of a classic GA. Without loss of generality, we assume the optimization is to minimize a cost function. So, we use EP to determine a $\vec{x}_{\min}$, such that

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

where

$$\vec{x} = \{x_i(1), x_i(2), \ldots, x_i(k)\}$$

is the individual containing k variables and $f : R^n \mapsto R$ is the function being optimized. David B. Fogel introduced EP using Gaussian distribution.[5] Xin Yao and Yong Liu proposed a mutation operation based on the Cauchy distribution to increase the speed of convergence.[6] We implement a fast evolutionary programming (FEP) based on the Cauchy mutation (see figure 3).[6]

1. Generate the initial population of μ individuals, each 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 of the population's individuals,

   (x_i, η_i), $i = 1, \ldots, \mu$ on the basis of the objective function $f(\vec{x})$.

3. For each parent

   $(\vec{x}_i, \vec{\eta}_i)$, $i = 1, \ldots, \mu$,

   create an offspring $(\vec{x}_i', \vec{\eta}_i')$; for $j = 1, \ldots, k$,

   $$x_i'(j) = x_i(j) + \eta_i C_j(0,1)$$

   $$\eta_i'(j) = \eta(j) \exp \frac{1}{\sqrt{2k}} R(0,1) + \frac{1}{\sqrt{2\sqrt{k}}} R_j(0,1)$$

   where $x_i(j)$, $\eta_i(j)$, $x_i'(j)$, and $\eta_i'(j)$ denote the jth component of $\vec{x}_i$, $\vec{\eta}_j$, $\vec{x}_i'$, and $\vec{\eta}_j'$. $R(0, 1)$ denotes a normally distributed 1D random number with zero mean and a standard deviation of one. $R_j(0,1)$ indicates a new random variable for each value of j. $C_j(0, 1)$ denotes a Cauchy random number.

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, randomly choose q (tournament size) opponents from all parents and offspring. For each comparison, if the individual's fitness is less than or equal to that of its 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 **wins** to become parents of the next generation.

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

Figure 3. Fast evolutionary programming based on the Cauchy mutation.

In figure 3's pseudocode, $\vec{x}_i$ is the individual evolving and $\vec{\eta}_i$ controls the vigorousness of mutation of $\vec{x}_i$. In general, we can roughly divide the computation of FEP into three stages: mutation and reproduction (step 3), evaluation of the fitness value (steps 2 and 4), and competition and selection (steps 5 and 6).

Mutation and reproduction

Following the pseudocode in figure 3, mutation is executed on each gene. We assume genes are independent. So, mutation is perfectly parallelizable. In a pure CPU implementation, you need to perform mutation on each gene sequentially. On a SIMD-based GPU, a fragment shader executes in parallel to perform mutation on each component (r, g, b, α) of each pixel. So, the GPU solution is ideal for this independent mutation and can achieve significant speedup.

To accomplish mutation on a GPU, we designed two fragment shaders, one for computing $\vec{x}'$ and the other for $\vec{\eta}'$. Figure 4 illustrates these two shaders. 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 $\vec{\eta}'$. One fragment shader computes $\vec{x}_i'$ while the other computes $\vec{\eta}'$. We also need to input two random number textures. Mutation requires normally distributed random variables. Unfortunately, our GPU isn't equipped with a random number generator. So, we divide random-number generation into two steps. First, we use the CPU to generate random variables with uniform distribution. We feed the generated random numbers to the GPU via input textures. Then, inside the two fragment shaders, the GPU converts the numbers from uniform distribution to Gaussian distribution in parallel.

We employ the direct inverse cumulative normal distribution function (ICDF) because it doesn't require looping. This makes it suitable for a SIMD-based GPU. Our experiment shows that our GPU implementation of ICDF is two times faster than a CPU ICDF implementation.

Our GPU transfers data slowly from the GPU texture to the main memory. So, we need to avoid this kind of data transfer as much as possible. Our strategy is to keep the parent and offspring residing in GPU memory. We only transfer the final result, after several generations of evolution, from the GPU textures to the main memory.

Fitness value evaluation

Fitness value evaluation, one of EP's core

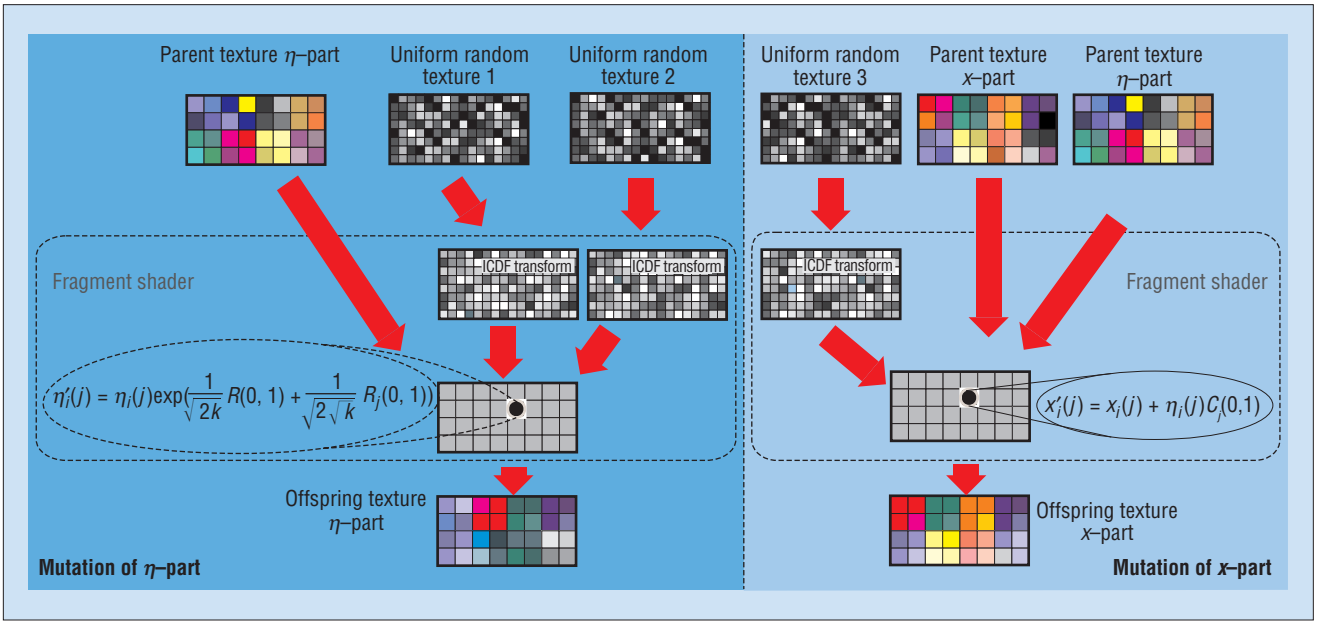

Figure 4. Two fragment shaders for mutation.

parts, determines an individual's "goodness." After each evolution, we calculate the fitness value of each individual in the current population. The result passes to a later stage. Each individual returns a fitness value by feeding the objective function f with the individual's genes. Fitness evaluation usually consumes most of the computational time.

Because we don't require interaction between individuals during evaluation, we can fully parallelize the evaluation. Figure 5 illustrates the evaluation shader. Recall that we break down individuals into quadruples and store them in the tiles within the textures. The evaluation shader 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 returns only a single value.

Competition and selection

The last stage of each evolution replaces the old generation. This involves two major processes: competition and selection.

Competition. We use competition to determine which searching direction is profitable on the basis of information gathered. EP employs a stochastic (soft) selection through the tournament schema. Individuals in the population compete with q randomly drawn opponents (from the union set of parents and offspring), where q is the predefined tournament parameter. The considered individual wins if its fitness value beats (in our case, is less than or equal to) its opponent's. We use the variable `win` to record the number of opponents defeated.

Competition can take place either on the GPU or CPU. For GPU implementation, the CPU must generate q textures of random values and load them to the GPU memory for picking q opponents in each evolution. The shader performs q fitness comparisons by looking up the opponent's fitness in the fitness textures obtained previously. The shader performs lookups by regarding the random values as indices to fitness textures. `win` is recorded and output to a texture.

For CPU implementation, we only have to transfer the fitness textures of both parent and offspring populations from GPU memory to the main memory. Then, we refer to these fitness textures to perform the tournament on the CPU.

You might think that the GPU implementation would be faster than the CPU implementation because it parallelizes the competition. However, our experiments show that it's slower. The GPU implementation hits a bottleneck in the transfer of q textures to the GPU memory. It demonstrates the limitation of the GPU's slow data transfer. Because the competition doesn't involve a time-consuming fitness evaluation (it only involves fitness comparison), the gain of parallelization doesn't compensate for the time lost in the data transfer. So, we suggest conducting the tournament in the CPU unless the GPU data transfer rate improves significantly.

Median searching and selection. After the tournament, selection chooses the best μ individuals, those with the highest `win` values, and assigns them as parents of the next generation. The most natural way is to sort the 2μ individuals in a descending order of win values. This approach then selects the first μ individuals. For large populations, the sorting time is unbearably slow even using the $O(N \log(N))$ sorting algorithm.

We aim to pick the best μ individuals, not sort 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 finding the median without sorting. We apply the partition-and-conquer algorithm with linear time complexity[7] to search for the median. Once we know the median, we can scan through the `win` values of all individuals and select those with values above or equal to the median. The process stops once μ individuals are selected.

Minimizing data transfer. To minimize the data transfer between GPU memory and the main memory, we construct an index array storing each individual's offset in the textures before competition and selection. During selection, we record only the index of the

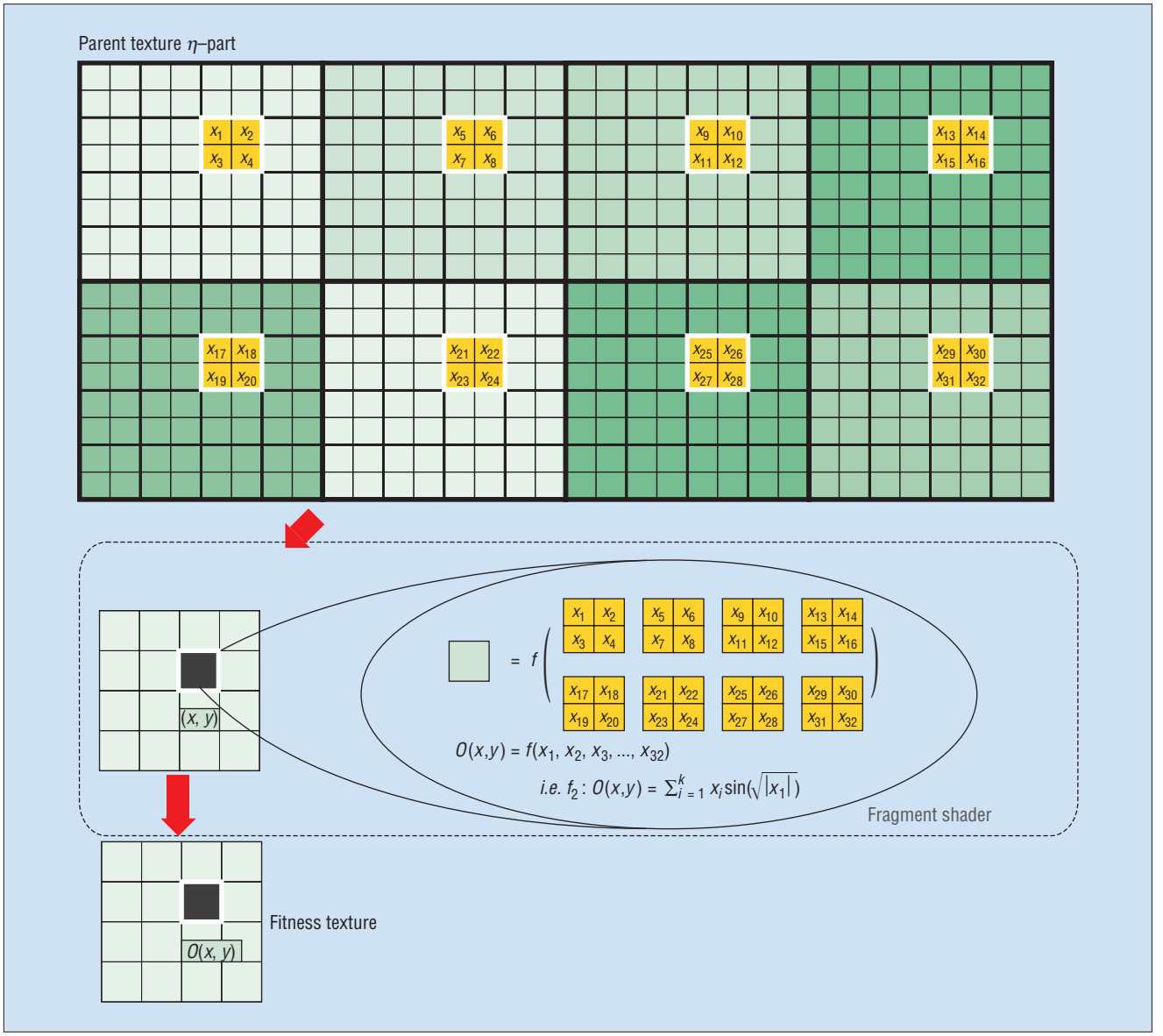

Figure 5. Shader for a fitness value evaluation.

selected individuals instead of all the individuals. We then load the index array to the GPU as a texture. Replacing individuals occurs on the GPU on the basis of the index texture. The process renders the final result to a texture that stores the new generation's individuals. With this approach, the GPU memory always retains the individuals' textures. We don't transfer these textures to the main memory during evolution until we've obtained the final generation. We transfer only fitness, random, and index textures between the GPU and CPU during evolution. Because these textures are smaller than the individuals' textures, this indexing approach minimizes data transfer and significantly improves speed.

Experimental results

We applied EP with Cauchy distribution to these benchmark optimization problems:

- $f_1: \sum_{i=1}^{32} x_i^2$
- $f_2: \sum_{i=1}^{32} \left(\sum_{j=1}^{i} x_j \right)^2$
- $f_3: \sum_{i=1}^{31} \left\{ 100\left(x_{i+1} - x_i^2\right)^2 + \left(x_i - 1\right)^2 \right\}$
- $f_4: -\sum_{i=1}^{32} x_i \sin\left(\sqrt{|x_i|}\right)$
- $f_5: \sum_{i=1}^{32} \left\{ x_i^2 - 10\cos\left(2\pi x_i\right) + 10 \right\}$

We performed experiments for 20 trials on both the CPU and GPU implementations and report on the average performance. The experiment testbed was a 2.4 GHz Pentium IV with a AGP 4X-enabled (consumer grade) GeForce 6800 Ultra display card, with 512 Mbytes of main memory and 256 Mbytes of GPU memory. The experiments used these parameters:

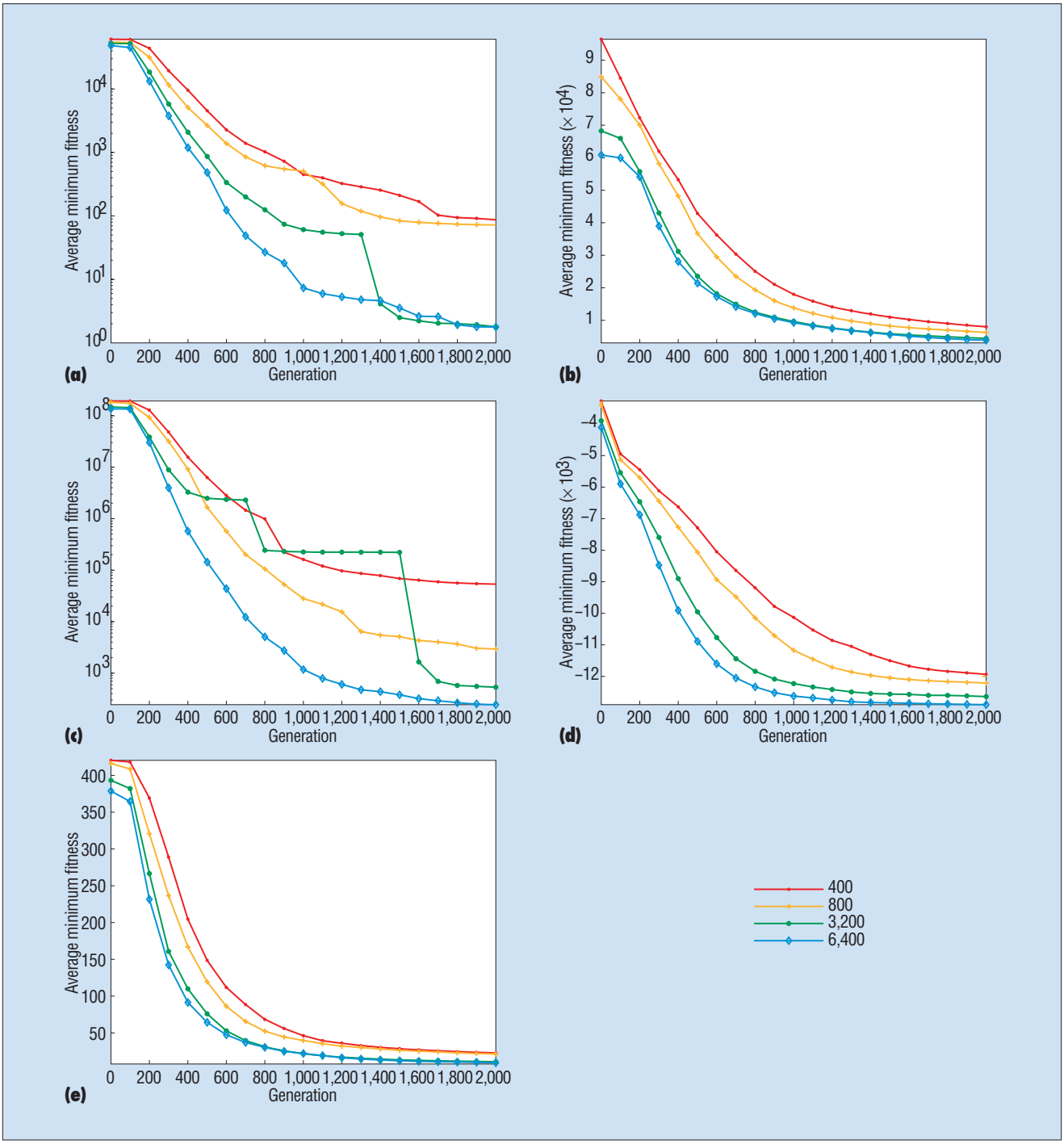

Figure 6. The fitness value of the best solution found by the GPU approach for five benchmark optimization problems: (a) f_1, (b) f_2, (c) f_3, (d) f_4, and (e) f_5. The results were averaged over 20 independent trials.

- population size: $\mu = 400$, 800, 3,200, and 6,400;
- tournament size: $q = 10$;
- standard deviation: $\sigma = 1.0$; and
- maximum number of generation: $G = 2{,}000$.

Figure 6 shows, by generation, the average fitness value of the best solutions our GPU approach found in 20 trials with various population sizes. It obtained better solutions for all functions when we used a larger population size. This occurs because a larger population has more search points, and EP is less likely to be trapped in a local optimum.

However, EP with a larger population size will take longer to execute. Figure 7 displays, by generation, the average execution time of the GPU and CPU approaches for different

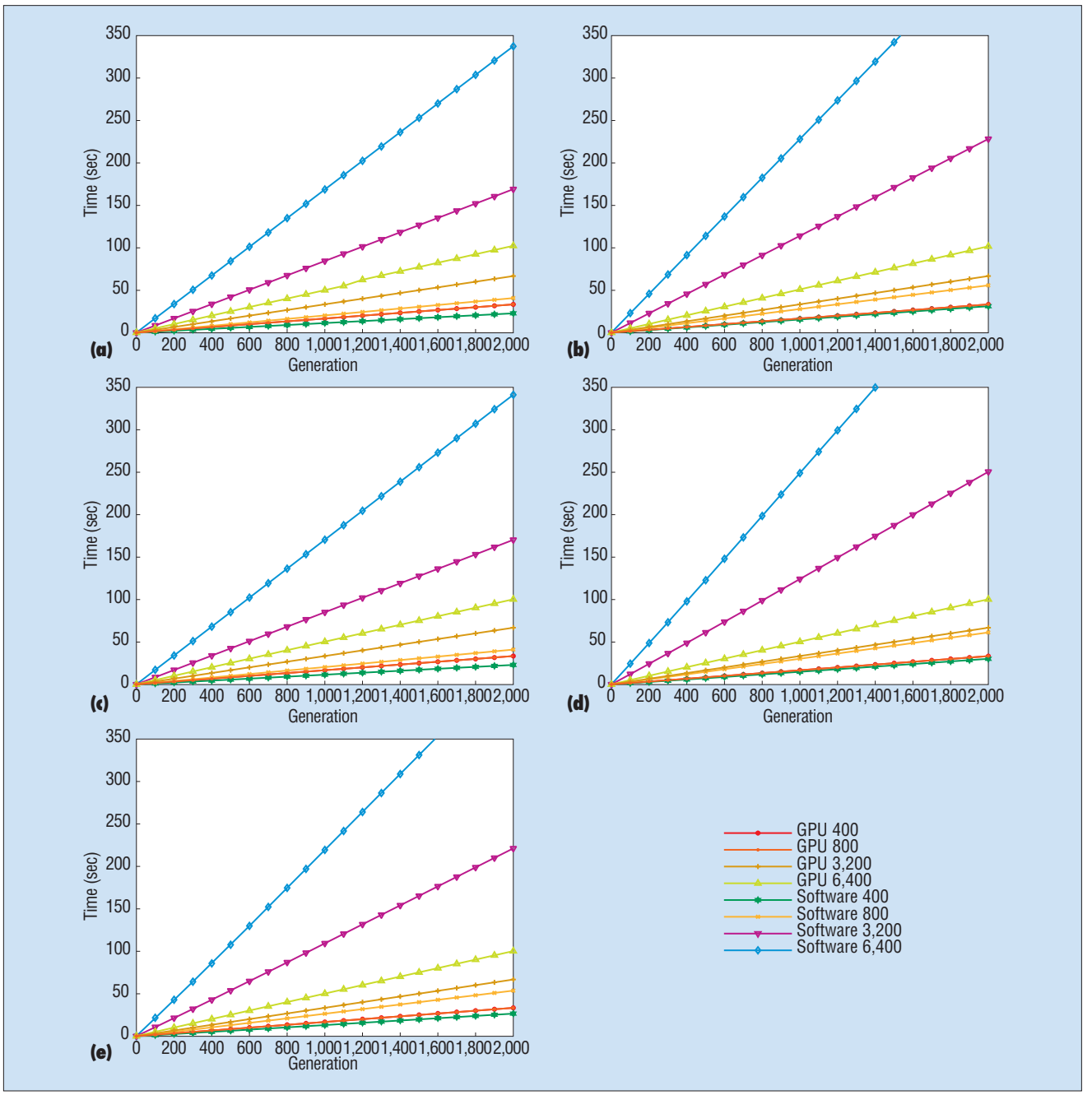

Figure 7. The execution time of the GPU and CPU approaches for five benchmark optimization problems: (a) f_1, (b) f_2, (c) f_3, (d) f_4, and (e) f_5. The results were averaged over 20 independent trials.

Table 1. The ratios of the average execution time of the GPU and of the CPU approaches with different population sizes to that with a population size of 400.

	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
3,200	2.02	2.02	2.02	2.02	2.02	8.30	8.24	8.37	8.12	8.29
6,400	3.11	3.09	3.04	3.05	3.05	16.57	16.45	16.75	16.40	16.53

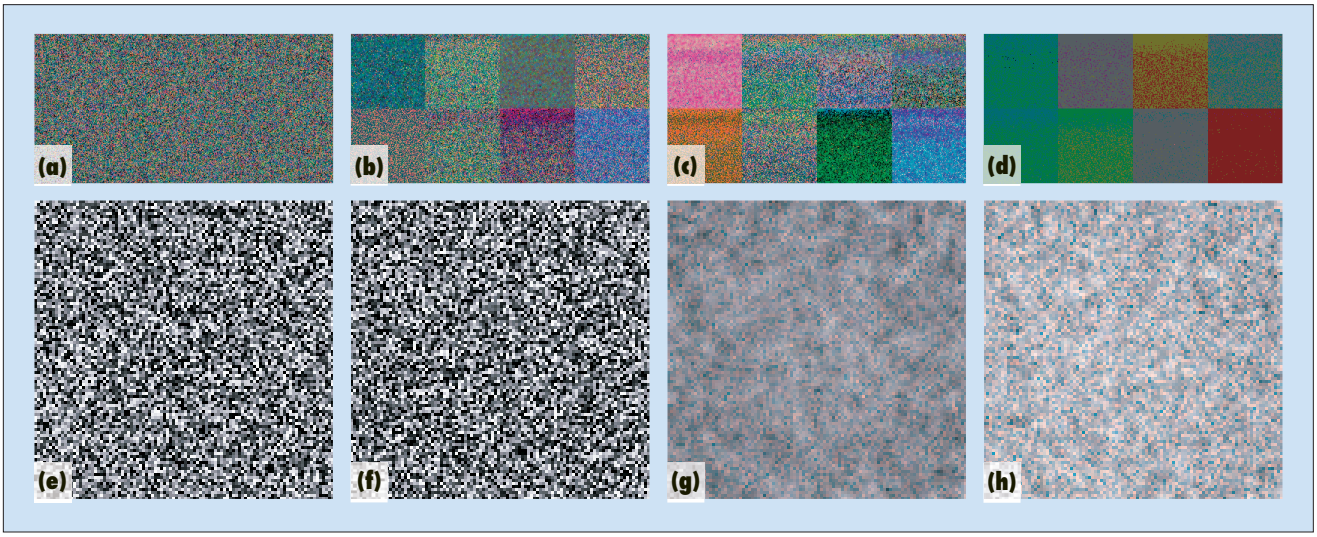

Figure 8. A visualization of evolution: (a)–(d) are four snapshots of genome-convergence; (e)–(h) are four snapshots of fitness convergence.

Table 2. An experimental results summary for f_5.

μ	Type	Competition & selection time (sec)	Speedup	Fitness evaluation time (sec)	Speedup	Mutation time (sec)	Speedup	Total time (sec)	Speedup
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	
3,200	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	
6,400	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	

population sizes. We see that execution time increases for larger populations. However, our GPU approach is more efficient than the CPU implementation because once the population size reaches 800, the GPU approach executes more quickly. Moreover, the leap in efficiency increases as the population size increases.

Table 1 summarizes the ratios of the average execution time of the GPU and CPU approaches with population sizes of 800, 3,200, and 6,400 to that of the corresponding approach with a population size of 400. Notably, the CPU approach shows a linear relation between the number of individuals and the execution time, while our GPU approach has a sublinear relation. For example, our GPU approach with populations of 400 and 800 take about the same time to execute. Moreover, our approach's execution time with a population of 6,400 is about three times that of a population of 400. This offers a real advantage to real-world applications with huge populations.

Table 3. The speedup of the GPU approach.

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
3,200	2.55	3.45	2.57	3.74	3.35
6,400	3.31	4.50	3.42	5.02	4.42

To study why our approach can achieve this phenomenon, table 2 shows the average execution time for different types of operations with the GPU and CPU approaches for the test function f_5. The fitness evaluation times of our GPU approach with different population sizes are about the same because our approach evaluates all individuals in parallel. Moreover, the mutation time doesn't increase proportionally with the number of individuals because the mutation operations also execute in parallel. The mutation time increases with the number of individuals because our GPU approach requires a number of random numbers generated by the CPU. We obtained similar results for other test functions.

Table 3 displays the speedups of our GPU approach compared with the CPU approach. The speedups depend on the population size and problem complexity. Generally, the GPU approach outperforms the CPU approach when the population size is greater than or equal to 800. The speedup ranges from approximately 1.25 to 5.02. For complicated problems that require extremely large populations, we expect that the GPU approach can perform even better.

The Authors

Ka-Ling Fok is a MPhil student at the Chinese University of Hong Kong. His research interests focus on performing scientific computing on graphics processing units. He received his BSc in computer science from the Chinese University of Hong Kong. Contact him at the Dept. of Computer Science & Eng., the Chinese Univ. of Hong Kong, Shatin, Hong Kong; klfok@cse.cuhk.edu.hk.

Tien-Tsin Wong is a professor in the Chinese University of Hong Kong's Department of Computer Science and Engineering. His research interests include computer graphics involving general-purpose computing on graphics processing units, image-based modeling and rendering, medical visualization, natural-phenomena modeling, and photorealistic and nonphotorealistic rendering. He received his PhD in computer science from the Chinese University of Hong Kong. He's a member of the IEEE and ACM. Contact him at the Dept. of Computer Science & Eng., the Chinese Univ. of Hong Kong, Shatin, Hong Kong; ttwong@cse.cuhk.edu.hk.

Man-Leung Wong is an associate professor at Lingnan University's Department of Computing and Decision Sciences. His research interests include evolutionary computation, data mining, machine learning, knowledge acquisition, and approximate reasoning. He received his PhD in computer science from the Chinese University of Hong Kong. He's a member of the IEEE and ACM. Contact him at the Dept. of Computing & Decision Sciences, Lingnan Univ., Tuen Mun, Hong Kong; mlwong@ln.edu.hk.

Visualization

Because the GPU is designed for display, we can easily visualize all onboard textures in real time without much additional cost. Such visualization lets users instantly observe the current generation's individuals. We designed two visualization schemes: *genome convergence* and *fitness convergence*.

For genome convergence visualization, by mapping the minimum and maximum gene values to [0, 255] (an 8-bit integer), we can regard the gene values in each quadruple as a color and output to the screen. At the beginning of evolution, the gene values are basically random and hence appear as a noise image (see figure 8a). As the population converges, each tile's color becomes less noisy and converges to a single color. In fact, each tile's actual color doesn't matter. The most important observation is how apparent the boundaries between two consecutive tiles are (a useful indicator of convergence), because different genes can converge into different values. Figures 8a–d show the genome-convergence map at iterations 100, 500, 1,000, and 2,000.

The fitness convergence visualization scheme is more traditional. During our fitness evaluation, we obtain a texture of fitness values. We can output this texture onto the screen for inspection. Again, we can map the fitness values to a range of color values for better visualization using a simple shader program. Because our test functions are all minimization problems, we map the minimum and maximum fitness values to [0, 255] gray levels for visualization. Figures 5e–h are four snapshots of fitness values.

The GPU implementation of a parallel EP algorithm is a hybrid of master-slave and fine-grained models.[8] The CPU (that is, the master) performs competition and selection while the GPU—essentially a massively parallel machine with shared memory—performs fitness evaluation, mutation, and reproduction. Unlike other fine-grained parallel computers such as MasPar's MP-2, our GPU lets processors communicate with any other processor directly, enabling us to implement more flexible, fine-grained EAs.

We plan to implement a parallel GA on our GPU and compare it with the approach we reported here. You can access a demo of our program at www.cse.cuhk.edu.hk/~ttwong/demo/ecgpu/ecgpu.html. ■

Acknowledgments

The Chinese University of Hong Kong Young Researcher Award (4411110) and the Earmarked Grant LU 3009/02E from the Research Grant Council of the Hong Kong Special Administrative Region supported this work.

References

1. I.S. Oh, J.S. Lee, and B.R. Moon, "Hybrid Genetic Algorithms for Feature Selection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, 2004, pp. 1424–1437.
2. J.R. Koza et al., *Genetic Programming IV: Routine Human–Competitive Machine Intelligence*, Kluwer Academic Publishers, 2003.
3. M.L. Wong and K.S. Leung, "An Efficient Data Mining Method for Learning Bayesian Networks Using an Evolutionary Algorithm-Based Hybrid Approach," *IEEE Trans. Evolutionary Computation*, vol. 8, no. 4, 2004, pp. 378–404.
4. D.B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, 2000.
5. D.B. Fogel, "An Introduction to Simulated Evolutionary Optimization," *IEEE Trans. Neural Networks*, vol. 5, no. 1, 1994, pp. 3–14.
6. X. Yao and Y. Liu, "Fast Evolutionary Programming," *Evolutionary Programming V: Proc. 5th Ann. Conf. Evolutionary Programming*, MIT Press, 1996.
7. R.W. Floyd and R.L. Rivest, "Expected Time Bounds for Selection," *Comm. ACM*, vol. 18, no. 3, 1975, pp. 165–172.
8. E. Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms*, Kluwer Academic Publishers, 2000.