Chapter 6

Parallel Genetic Algorithms with GPU Computing

John Runwei Cheng and Mitsuo Gen

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.89152

Abstract

Genetic algorithms (GAs) are powerful solutions to optimization problems arising from manufacturing and logistic fields. It helps to find better solutions for complex and difficult cases, which are hard to be solved by using strict optimization methods. Accelerating parallel GAs with GPU computing have received significant attention from both practitioners and researchers, ever since the emergence of GPU-CPU heterogeneous architectures. Designing a parallel algorithm on GPU is different fundamentally from designing one on CPU. On CPU architecture, typically data or tasks are distributed across tens of threads or processes, while on GPU architecture, more than hundreds of thousands of threads run. In order to fully utilize the computing power of GPUs, the design approaches and implementation strategies of parallel GAs should be re-probed. In the chapter, a concise overview of parallel GAs on GPU is given from the perspective of GPU architecture. The concept of parallelism granularity is redefined, the aspect of data layout is discussed on how it will affect the kernel performance, and the hierarchy of threads is examined on how threads are organized in the grid and blocks to expose sufficient parallelism to GPU. Some future research is discussed. A hybrid parallel model, based on the feature of GPU architecture, is suggested to build up efficient parallel GAs for hyper-scale problems.

Keywords: GPU acceleration, parallel genetic algorithms, granularity of parallelism of parallel genetic algorithms on GPU, GPU-CPU heterogeneous systems

1. Introduction

GAs are stochastic and global stochastic search methods, which combine two major search strategies: exploiting better solutions and exploring the global search space. GAs have been successfully applied to many optimization problems in different disciplines that are difficult to solve by conventional mathematical programming methods [1]. GAs, as a population-based

search method, present great potential in performance improvement by parallel computation. The earliest attempt to parallelize GAs can be traced back to nearly 30 years ago. The research on parallel GAs can be divided into two major stages as the computing architectures evolve: the stage of parallel GAs on CPU architecture versus on GPU architecture.

On CPU architecture, you may run a parallel program over tens of threads or processes, while on GPU architecture, you can run it with more than hundreds of thousands of massive threads simultaneously, which leads to a fundamental paradigm shift in parallel programming. The basic idea to parallel GAs on CPU is to distribute computations among either multiple processors or multiple computers. Each instance of parallel processes on a processor is essentially a sequential GA, viewing from the perspective of algorithm implementation. The approach of parallel GAs on CPU is not suitable for massive GPU thread computing architecture.

Since the emergence of CPU + GPU heterogeneous architecture, how to accelerate GAs on the architectures has received significant attentions from both practitioners and researchers. GPU computing makes the research of parallel GAs truly enter into the world of high-performance computing (HPC) and demonstrates a great potential to many research disciplines and industrial worlds that can benefit from the power of GPU-accelerated stochastic and global search to explore large and complex search spaces for better solutions.

Most works on parallel GAs on GPU are published before 2013 on very earlier generation of GPU architectures, and most researches adopt a naive approach: let each GPU thread do a sequential GA, exactly the same way parallel GAs do on CPU. Even with the naive approach, you can still speed up GA computation greatly. Since massive threads are available on GPU, the way of designing a parallel program on GPU is quite different with the way of designing one on CPU. The distinguishing features of parallel programming on GPU over CPU are that CUDA platform exposes GPU memory model and execution model to us to enable us to have more control over massive threads in order to exert the computing power of GPUs.

The intention of the chapter is to give a concise review on accelerating GAs on GPU: how to organize the massive threads to access memories and how to harmonize massive threads to work efficiently. Fermi architecture, the world's first complete GPU computing architecture released in 2010, makes GPU computing truly applicable in industrial worlds. This is the reason why the review on the selected works after 2010. The performance issue is not touched, because since then, both GPU hardware architectures and CUDA platform have had significant progress.

This chapter is organized as follows. Section 2 briefly describes the basic concept of GA and parallel models on CPU. Section 3 discusses major issues on how to accelerate parallel GAs on GPU: the granularity of parallelism is redefined for parallel GAs on GPU, the aspect of data layout is examined to show how it will affect the kernel design to maximize memory bandwidth, and the hierarchy of thread execution is investigated to explain how to organize threads in grid and blocks to expose sufficient parallelism to GPU. A concise overview is given in Section 4 on selective works of parallel GAs on GPU. How to implement three parallel models of GAs on GPUs is explained through several famous combinatorial optimization problems to show that the efficient implementations of parallel GAs on GPUs share the common principles of kernel optimization. A case study of railway scheduling problem is

described in Section 5 to demonstrate how to accelerate multiple objective parallel GAs with GPU. A brief discussion of future research is given in Section 6, focusing on how to build up efficient parallel GAs on GPUs for hyper-scale computing. The chapter ends with a brief conclusion in Section 7.

2. Parallel GAs on CPU

This section introduces briefly the basic concepts of GAs and basic models of parallel GAs on CPU.

2.1. General structure of genetic algorithms

GAs are efficient and stochastic search methods based on principles of natural selection and genetics. Many optimization problems from the scientific and industrial research are very complex in nature and quite hard to solve by conventional optimization techniques. Since the 1960s, there has been an increasing interest in imitating living beings to solve such kinds of hard optimization problems. Simulating the natural evolutionary process of human beings results in stochastic optimization techniques called genetic algorithms or evolutionary algorithms, which can often outperform conventional optimization methods when applied to difficult real-world problems [2].

The canonical form of GAs is described by Goldberg [3]. The calculation of GAs begins with a set of individuals, called a *population*. Each individual represents a solution to the problem you want to solve, which is also called *chromosome.* GAs perform the stochastic search through a loop of iterations, and each iteration is called a *generation*. New solutions are created at each generation by either merging two solutions using a *crossover* operator or modifying a solution using a *mutation* operator. The new solution is called *offspring* and the solutions used to create them called *parents*. During each generation, each solution in the population is evaluated, using some measures of *fitness.* A set of new population for the next generation is then formed by a *selection* operator. The solutions with better fitness values have much higher possibilities of being selected to enter the next generation. The population evolves over several generations, converging to the better solutions which are hopefully close to the optimal solution. The pseudo code of the general structure of GA is described as follows (**Figure 1**).

GAs, as a metaheuristic, have many advantages over the conventional single-point search methods. Requiring not good mathematical properties about the problem as the conventional method does, GAs can handle much complex problems with any kind of objective functions and constraints, linear or nonlinear, on discrete, continuous, or mixed search spaces, and provide us a great flexibility to hybridize with domain-dependent heuristics to make the random search more powerful and effective. The ergodicity of genetic operators makes GAs very effective at performing global search. GAs often outperform over conventional methods on most types of combinatorial optimization problems because they do not make any assumption about the underlying fitness landscape [4].

```
Procedure: Genetic Algorithms
begin
  t ← 0;
  initialize P(t);
  evaluate P(t);
  while (not termination condition) do
    recombine P(t) to yield C(t);
    evaluate C(t);
    select P(t + 1) from P(t) and C(t);
    t ← t + 1;
  end
end
```

Figure 1. The general structure of GA.

2.2. Multi-objective genetic algorithms

Multi-objective optimization deals with problems of seeking solutions over a set of possible choices to optimize multiple criteria. Almost every important real-world decision problem involves multiple and conflicting objectives that need to be tackled while respecting various constraints, leading to overwhelming complexity for conventional optimization algorithms. Genetic algorithms have been receiving considerable attention as a novel approach to multi-objective optimizations since the early 1980s, resulting in a fresh body of research and applications, called multi-objective GAs.

The primary difference of multi-objective GAs from single-objective GAs is how to determine the fitness value of individuals according to multiple objectives. For multi-objective problems, we cannot find a point superior than all other points in the criterion space, as illustrated in **Figure 2**, such points called non-dominated, or non-inferior, or Pareto point, named after an Italian economist, who introduced the concept of Pareto efficiency in the field of microeconomics. A point is called Pareto optimal if none of the objectives can be improved in value without degrading some of the other objectives' values.

How to maintain a set of non-dominated individuals during evolutionary processes is a special issue for multi-objective GAs. Pareto points are identified at each generation and used to calculate fitness values of all points or rank all other points. No mechanism in canonical GAs is provided to guarantee Pareto points in current generation be selected into next generation. In other words, some Pareto points may get lost during the evolutionary process. To avoid such sampling errors, a special mechanism for preserving Pareto solutions is added to the basic structure of genetic algorithms. In each generation, a set of Pareto points, kept in a separate pool, is updated by deleting all dominated solutions and adding all newly generated Pareto solutions [2].

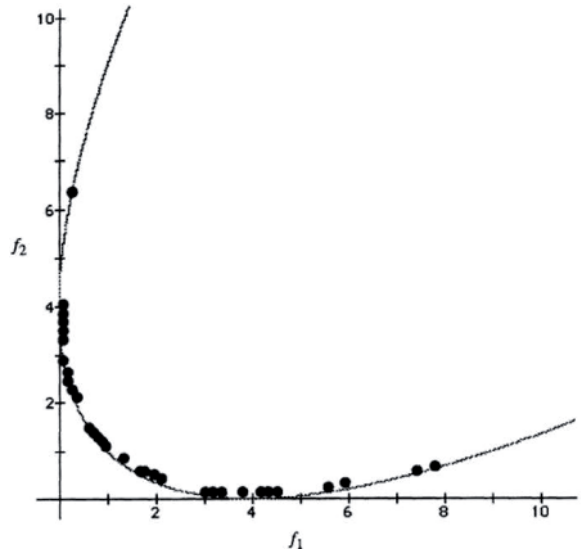

Figure 2. The Pareto frontier.

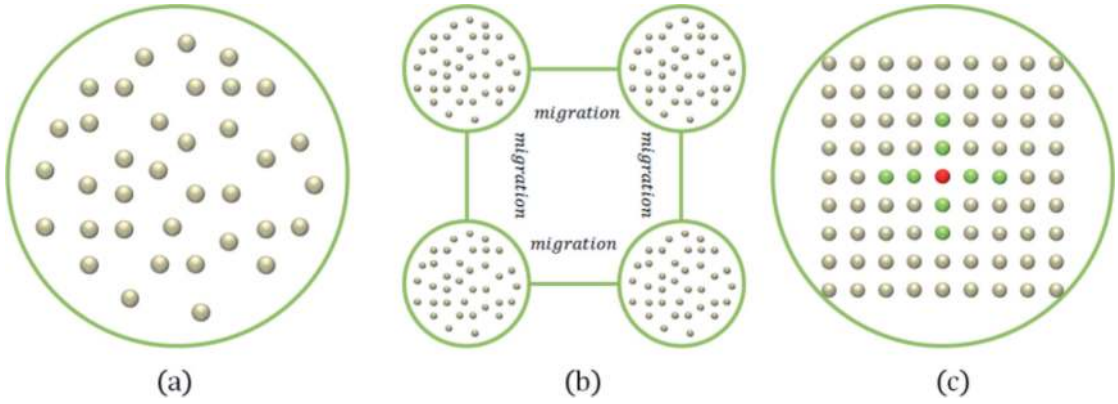

Figure 3. Models of parallel GAs. (a) Master-Slave model, (b) Island model, and (c) Cellular model.

2.3. Parallel genetic algorithms

Because GAs conduct the stochastic search over a set of solutions, there is a great potential for speeding up the evolutionary process of GAs: distributing the computational load among multiple processors through a data parallel approach. The motivation to parallelize GAs is twofold: to speed up the computation of GAs when solving large and complex problems and to improve the quality of solutions by exploiting distributed populations. There are three basic approaches of parallel GAs on CPU as shown in **Figure 3**:

1. Master-slave model
2. Island model
3. Cellular model

The basic idea behind all above algorithms is a divide-and-conquer approach: dividing the task into chunks and solving the chunks simultaneously using multiple processors [5]. In the *master-slave* model, there is a single population, and the evaluation of fitness is distributed among several processors. In the *cellular* model, there is a spatially structured population, and selection and mating are restricted to a small neighborhood, but neighborhoods overlap, permitting some interaction among all the individuals. This model is suited for massively parallel computers. In the *island* model, the population is divided into multiple sets, each called an *island*. In an island, individuals mate freely, while mating is prohibited across islands. A new operator is then introduced into the island model, the migration operator, exchanging a small portion of population among islands periodically in a predefined way in order to bring new genetic materials into each island.

From the perspective of mating mechanisms, these models can be classified into two major categories: mating globally and mating locally. In the master-slave model, individuals are permitted to mate freely within the entire population. In the cellular model, the population is imposed with geographical structure, and individuals are permitted to mate only with its close neighbors. The island model is in between the master-slave and cellular model: free within a given island but restricted among islands.

From the perspective of population, these models can be classified into two categories: the single population and the multiple populations. The master-slave and cellular models belong to the single population, and the island model belongs to the multiple populations.

Conceptually, all above models keep multiple instances of GAs, and each instance on a processor or a computer is a sequential program in nature. A communication mechanism among processors or computers is required in order to evolve a better solution from the overall populations, implemented either in a synchronizing way or in an asynchronizing way.

3. Major issues of parallel GAs on GPU

Many of early works of parallel GAs on GPUs take a naive approach: let each GPU thread run a sequential GA for a given portion of task, the same way used in parallel GAs on CPU to do a sequential GA over one processor. This approach, by ignoring the architectural feature of GPUs, cannot fully exploit the computing power of GPUs [6].

Because more than tens of thousands of threads run on GPUs, we cannot simply follow the old path of parallel GAs on CPU. Several issues need to be carefully examined according to the architectural feature of GPUs, such as how to arrange data access to maximize GPU memory bandwidth and how to organize threads to expose sufficient parallelism to GPUs in order to harness the power of GPUs [7].

This section first introduces briefly the basic concept of CUDA platform and then focuses on the issues of data layout, execution configuration, and memory access, which are the key enablers for parallel computing on GPU.

3.1. The CUDA platform

CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose parallel computing on GPUs [8]. With CUDA, compute-intensive applications can be dramatically speeded up by harnessing the power of GPUs. CUDA is also a scalable programming model that enables programs to transparently scale their parallelism to GPUs.

In CPU-GPU heterogeneous system, conceptually, the sequential part of the workload runs on the CPU, while the compute-intensive portion of the application runs on thousands of GPU threads in parallel, as shown in **Figure 4**.

CUDA C is an extension of standard ANSI C with a handful of language extensions to enable parallel programming with a shallow learning curve for ones familiar with the C programming language. The CUDA Toolkit, provided in CUDA platform, includes GPU-accelerated libraries, a compiler, development tools, and the CUDA runtime, everything you need to develop GPU-accelerated applications.

The distinguishing features of CUDA C programming, compared with C programming on CPU, are that the CUDA platform exposes the memory model and execution model of GPUs to programmers to enable programmers to have more control over massive threads in order to optimize GPU implementation. The top three principles of CUDA C program optimization, listed in order of importance, are the following:

1. Exposing sufficient parallelism
2. Optimizing memory access
3. Optimizing instruction execution

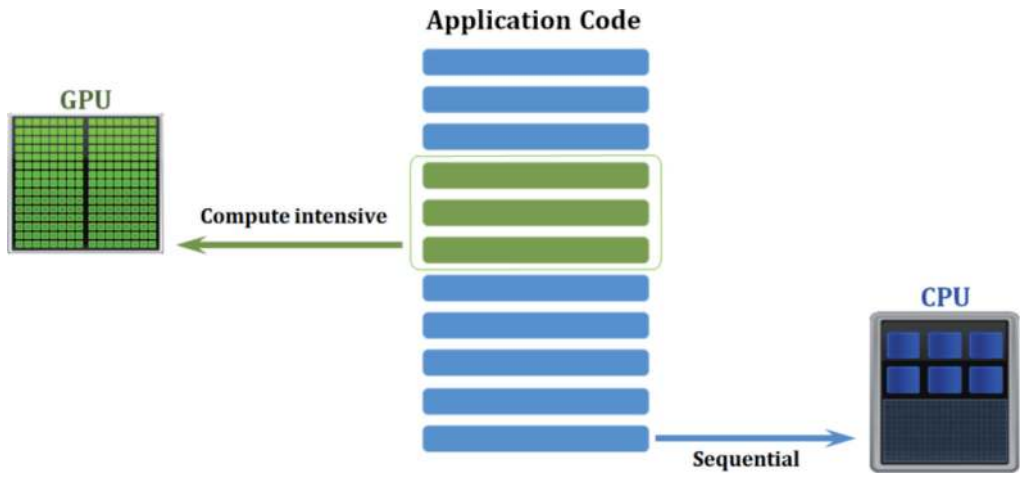

Figure 4. Acceleration on CPU-GPU heterogeneous system.

These principles are applicable almost to all kinds of computing architectures. The implementation of GPU kernels is architecture-dependent while applying these principles to harness the power of GPUs; therefore, it is necessary for a programmer to have some basic knowledge of the underlying architecture when parallelizing GAs on GPUs.

3.2. Granularity of parallel GAs on GPU

In parallel computing, the granularity of computations of an algorithm is a measure of the amount of computation which is conducted by that task [9]. Typically, the parallelism of a program can be classified into two categories: fine-grained and coarse-grained. In a fine-grained parallel algorithm, a program is broken down to a large number of small tasks, while in a coarse-grained parallel algorithm, a program is split into several large tasks. Both coarse-grained parallel GAs and fine-grained parallel GAs have been proposed for CPU architecture, but these parallel algorithms are too "coarse" to GPU architecture. Therefore, a new definition of granularity is necessary to describe parallelism for programs on GPU architectures [10].

The granularity for parallel GAs on GPU can be defined by how many GPU threads are working together to handle one chromosome during genetic operations. Basically, the parallelism can be classified into two categories:

1. The granularity in the chromosome level
2. The granularity in the gene level

If a solution in GAs' population is handled by one GPU thread independently, the granularity of the parallel GAs is in the chromosome level. If a solution is handled by a group of GPU threads cooperatively, the granularity of the parallel GAs is in the gene level. Another type of granularity of parallel GAs is a mixed form: some genetic operators in the chromosome level and some in the gene level. The granularity in the gene level also can be divided into several types: cooperative threads in block size or in warp size. In the former, a block of threads handles one solution, while in the latter, a warp of threads handles one solution.

GPUs take a partitioning approach to allocate compute resources among blocks and threads. The more a kernel requires the resources, the less the parallelism is exposed to GPUs. Therefore, the compute resource used in a kernel is a vital factor for the performance of a GPU kernel function, since more resource requirement will limit a kernel to expose sufficient parallelism to GPU.

Generally speaking, if a kernel of genetic operator is designed in the granularity of chromosome level, it will require more shared memory and local registers than a kernel designed in the granularity of gene level. For a large-scale application, it will become a major issue that hinders the performance. GPUs execute groups of threads in a unit known as warps in *single instruction, multiple thread* (SIMT) fashion; the kernel implemented in a granularity in the gene level will have more efficient access to global memory.

In most of the early works of parallel GAs on GPUs, the granularity of kernels belongs to the category of the chromosome level, because it is a nature extension of CPU program to GPU.

It is also an easy way to migrate programs to GPU for the beginners of CUDA C programmers without knowing much in-depth knowledge of GPU architecture. This naive approach cannot fully utilize the computing power of GPU to accelerate GAs.

3.3. Data layout on global memory

Memory bandwidth and latency are key considerations in GPU applications. Bandwidth refers to the amount of data that can be moved to or from a given destination. Latency refers to the time the operation takes to complete. In designing a GPU kernel, we are primarily concerned about the global memory bandwidth. Most GPU kernels' performance is likely limited by memory accesses. To maximize global memory bandwidth, a vital step in kernel design is to make the kernel access global memory aligned and coalesced. How data are organized in the global memory will affect the way kernels access memory.

The population of GAs can be viewed logically as a 2D matrix with two dimensions: the index of chromosomes in the whole population and the index of gene within each chromosome. Physically GPUs store all data linearly in global memory. Therefore, there are two basic ways to layout the population in global memory:

1. The chromosome-based layout
2. The genotype-based layout

Let *pop_size* denote the population size and *nvar* the total number of genes in one chromosome, the chromosome-based layout is shown in **Figure 5**, and the genotype-based layout is shown in **Figure 6**.

In the chromosome-based layout, the fast dimension is the index of gene, that is, one chromosome is allocated in a contiguous memory space. A kernel designed with the granularity of gene level can take the benefit of the coalesced access of global memory. In the genotype-based layout, the fast dimension is the index of chromosome in the population, and all i^{th} genes of all chromosomes are allocated to a contiguous range. Therefore, the stride between two consecutive genes in one chromosome is the number of *pop_size*. A kernel designed with the granularity of chromosome level may take the benefit of coalesced memory access. But this layout will be not good for genetic operations, such as evaluation and migration operation.

The discussion above is for the real-coded or integer-coded GAs. Conceptually, the data layout for the binary GA belongs to the chromosome-based layout. There are two kinds of implementation for the binary GAs:

1. Byte-wise binary method
2. Bitwise binary method

In the byte-wise encoding, 8 bits is used to represent 0 or 1, while in the bitwise method, only 1 bit is used to represent 0 or 1. A nature way to implement the binary GA is to pack multiple

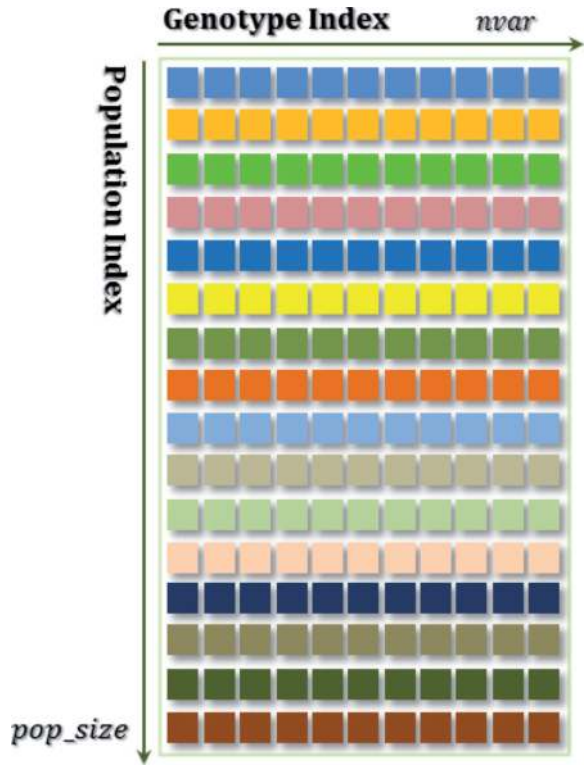

Figure 5. 2D view of the chromosome-based layout in global memory.

Figure 6. 2D view of the genotype-based layout in global memory.

bits into a non-Boolean data type, typically packing 32 bits into one unit for processing [11]. This bitwise approach can save global memory usage and also accelerate computation due to the less memory access.

3.4. Kernel execution configuration

The main difference between GPU and CPU programming is the level of programmer exposure to architectural features [7]. CUDA exposes the concepts of memory and thread hierarchy to grant the ability to control thread execution to a greater degree. GPU thread execution can be organized in a hierarchy structure: grid and block; therefore, programmers can configure the execution of a kernel function by specifying:

1. The number of threads per block and the dimension of the block
2. The number of thread blocks of a grid and the dimension of the grid

Grids and blocks represent a logical view of the thread hierarchy when executing a kernel function. Both block and grid can be arranged as 1D, 2D, and 3D layouts. How to configure a kernel execution heavily depends on the layout of data in the global memory in order to expose sufficient parallelism to GPU to saturate both instruction bandwidth and memory bandwidth. Exposing sufficient parallelism is the number one principle to optimize kernels, and grid and block heuristics play a vital role in kernel performance optimization.

3.5. Shared memory: the programmable cache

Shared memory can be viewed as a programmable cache and is a key enabler for kernel performance. By utilizing the shared memory, threads within the same block can easily cooperate with each other, facilitating the reuse of on-chip data and reducing greatly global memory traffic by caching data on-chip manually.

A programmer has the complete control over when data is moved into the shared memory and when data is evicted. CUDA makes it easier for you to optimize your application code by providing more fine control over data placement and on-chip data movement.

Sometimes, the non-coalesced accesses may not be easily avoided due to the nature of an algorithm. Shared memory provides a means to avoid non-coalesced memory accesses by staging data movement through the shared memory.

4. Parallel GAs on GPU

In this section, a concise review on selective works of parallel GAs on GPU is presented from the perspective of GPU computing. Several examples of combinatorial optimization problems are used to demonstrate on how to implement the master-slave model, island model, and cellular model on GPUs.

Although most published works are conducted on an early generation of GPU architecture, the basic idea on how to design parallel GAs on GPU is on the right track, and efficient implementations for different models share the common principles to optimize GPU performance, which is the focus of the chapter.

4.1. The master-slave model on GPU

The model of master-slave describes the relationship among multiple concurrent processes: one master process and many slave processes. The master process has the control over all other slave processes, distributing tasks among them and synchronizing them to work together. Each slave process does its own assigned task.

The earliest attempt at parallelizing GA on CPU is to let all slave processes calculate fitness values, the most time-consuming part of GA computation, while the master processes all other tasks. Lately, more tasks are moved into slave processes to accelerate the computation, as shown in **Figure 7**.

Because the relationship between GPU and CPU is the typical pattern of slave to master in nature, it is straightforward to implement the master-slave model on a GPU-CPU heterogeneous system. Earlier works take a very simple way to implement the model: GPU calculates the evaluation of the population only, while CPU does all other jobs, because evaluation takes major time comparing with other genetic operators. It is not an efficient implementation because it needs frequent data transfer between CPU and GPU at each generation of GAs, and the data movement between CPU and GPU dominates the execution time. Because most workloads are on CPU, it still is a bottleneck of performance, while GPU stays idle most time. Later works try to move more genetic operators onto GPU to speed up parallel implementations on GPU.

4.1.1. Binary GA for one max problem

One-Max problem is a toy problem in GA community just as the "Hello World" for C programmer. The problem is to produce a string of all 1 s, starting from a set of random binary strings. It has been widely used to test GA programs since it can well illustrate the potential of genetic algorithms.

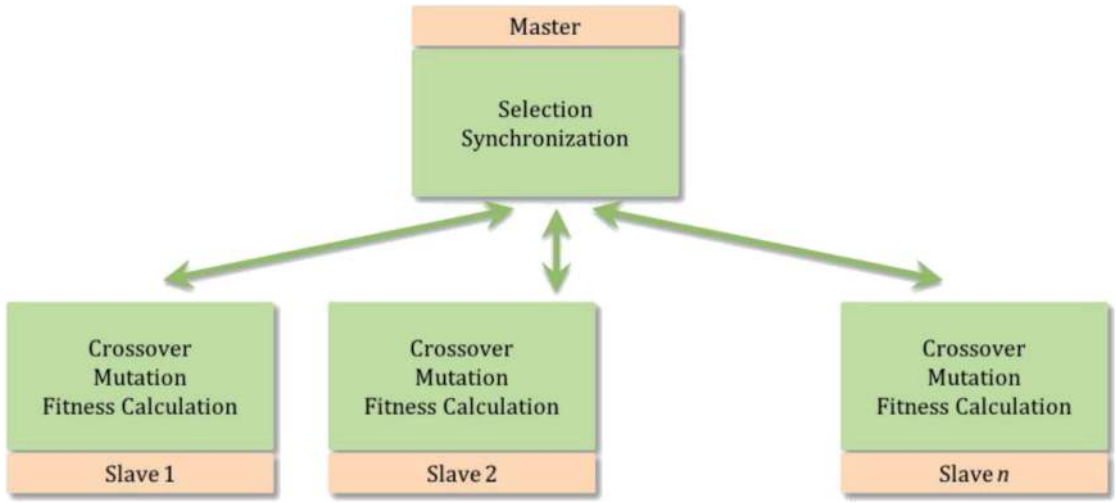

Figure 7. The master-slave model.

Debattisti et al. implemented a master-slave model of a fixed-length binary GA on GPU to solve the problem [12]. A byte-wise method is used to encode a solution: one byte is used to represent one bit of the genome, and a set of the unsigned char type data is used to encode a solution, that is, a string of binary value of 0 or 1. As mentioned in the last section, the bitwise encoding method is a better way to implement a binary GA on GPUs. Bitwise operations are available since CUDA 10.1 [13].

Except for initialization of the population, all other genetic operations, such as selection, crossover, mutation, and evaluation, are implemented on GPU. The crossover kernel implements the two-cut point method. Each thread block is responsible for dealing with two individuals to produce offspring, and each thread is responsible for handling 4 bytes each time. The mutation kernel implements the bitwise exclusive OR operations. Each thread block is responsible for dealing with one individual, and each thread is responsible for handling 4 bytes. All kernels are launched with 1D grid and 1D block configuration. Since multiple threads cooperatively deal with one individual, the granularity belongs to the gene level.

4.1.2. Real-coded GA for unconstrained optimization problems

Unconstrained optimization problems seek to maximize or minimize an objective function that depends on real variables without restrictions on these variables. For complex cases with many local optima, it highly depends on the initial point for conventional procedures to find the global optimum. Since GA is a population-based global search method, GAs can easily escape from local optimum during search process. That is why the unconstrained optimization problems are well tested in the GA community.

There are two typical ways to encode population of GAs: the binary-coded method and the real-coded method. For optimization problems, the real-coded approach outperforms the binary-coded approach [14].

Arora et al. implemented a master-slave model of the real-coded GAs on GPU to solve the unconstrained optimization problems [15]. The genotype-based layout is used to store the population in global memory, as shown in **Figure 6**.

The crossover kernel implements the simulated binary crossover (SBX) method, executing in the configuration of 1D block and 2D grid. Conceptually, each GPU thread works on a variable (or a gene) of two mates. Because thread blocks work independently with each other and one chromosome is operated by several different blocks, it will cause an issue that one chromosome may be selected to reproduce offspring many times.

The mutation kernel implements the one-variable change method using the execution configure of 1D block and 2D grid. Since one chromosome may be operated by multiple blocks independently, it has the same issue as what the crossover kernel has: one chromosome may be selected to reproduce offspring many times. The granularity of the implementation belongs to the gene level since multiple threads cooperatively deal with one or two individuals. Ellipsoidal function and Rosenbrock function are tested, which are well-used functions in GA literature.

4.2. Island model on GPU

In the island model, the population is divided into several subpopulations. By introducing geographical distribution of the whole population, the genetic diversity is hopefully preserved during the evolutionary process of different subsets. Each subset of population is an isolated breeding environment to evolve species locally. A new operator is introduced: the migration operator exchanges small portion of individuals among subsets to bring new genetic traits into each subset.

The island model is characterized by several factors: the number of islands, the topology of the migration, the interval of the migration, and the policy for selection and replacement during migration. The common migration topologies are given in **Figure 8**.

4.2.1. Binary GA for 0/1 knapsack problem

The 0-1 knapsack problem is a famous problem in combinatorial optimization: given sets of items, each with a weight and a value. The problem is which items should be put into a knapsack in order to maximize the total value without exceeding the capacity of the bag. Knapsack problems have been studied intensively, attracting both theorists and practitioners. The theoretical interest arises mainly from its simple structure, which allows exploitation of a number of combinatorial properties and permits more complex optimization problems to be solved through a series of knapsack-type subproblems. From a practical point of view, many industrial situations can be modeled as a knapsack problem. Since it is a well-known NP-hard problem, it has been well studied in GA community to test different implementations of GAs.

Jaros described implemented an island model of binary GA on multi-GPUs to solve the 0-1 knapsack problem [17]. The binary-coded method is used, since it is a nature representation for 0-1 binary knapsack problem. Each of the 32 items is packed into an integer data type; therefore, it is a kind of bitwise representation. The population is organized as a structure of arrays: the first array contains all chromosomes, and the second array contains the fitness values associated with each chromosome. The whole population is evenly distributed among multiple GPUs, and one GPU simulates one island to evolve a local population by using a steady-state method with elitism, the uniform crossover, the bit flip mutation, the tournament selection, and replacement. The unidirectional ring (1-way ring) is adopted as the migration topology where only adjacent nodes can exchange individuals. Migration among islands occurs after certain generations by exchanging the best individual and a number of randomly selected individuals. The tournament selection is used to pick emigrants. All the data exchanges among GPUs were implemented by means of MPI [18].

The chromosome-based layout is adopted to store the data in the global memory. The execution of all kernels is configured as 2D block and 2D grid. In the block layout, the fast dimension corresponds to the genes within a chromosome, while the slow dimension corresponds to different chromosomes, as shown in **Figure 9**.

The size of the fast dimension of 2D grid is fixed to 1, and the size of the slow dimension is set to the size of offspring divided by twice the slow dimension of 2D block, because two offspring were produced at once. Since each individual is processed by a warp, therefore, it belongs to the category of the granularity in the gene level.

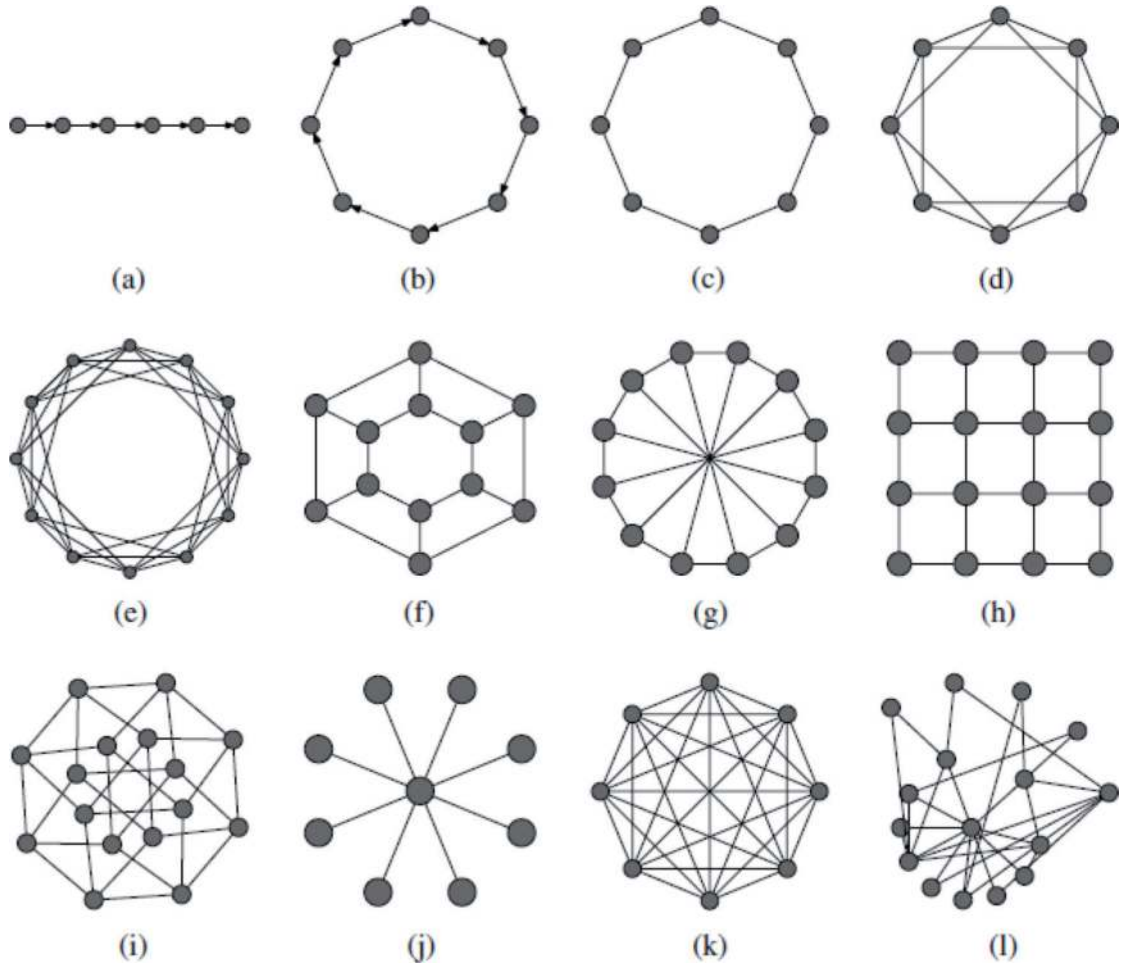

Figure 8. Common migration topologies (from [16]). (a) Chain, (b) I-way ring, (c) Ring, (d) +1+2 ring, (e) +1+2+3 ring, (f) Tours, (g) Cartwheel, (h) Lattice, (i) Hypercube, (j) Broadcast, (k) Fully-connected, and (l) Barabasi-Albert (3.2).

A knapsack instance with 10,000 items is used as a test case to simulate the real-world problem with a reasonable large data set. The tests are conducted on a two-node cluster, each equipped with seven GPUs, connected with InfiniBand.

4.2.2. Real-coded GA for unconstrained optimization problems

Luong et al. implemented an island model of real-coded GA for the unconstrained optimization problems [19]. The basic idea of their implementation is that each thread block is treated as an island, and each chromosome is handled by one thread for all operations of selection, crossover, mutation, and evaluation. Therefore, its granularity belongs to the chromosome level.

How data is organized in global memory was not described in the paper. If data is organized in the genotype-based layout, all kernels may take the merits of the coalesced memory access; if data is organized in the chromosome-based layout, all kernels will violate the principle of the coalesced memory access.

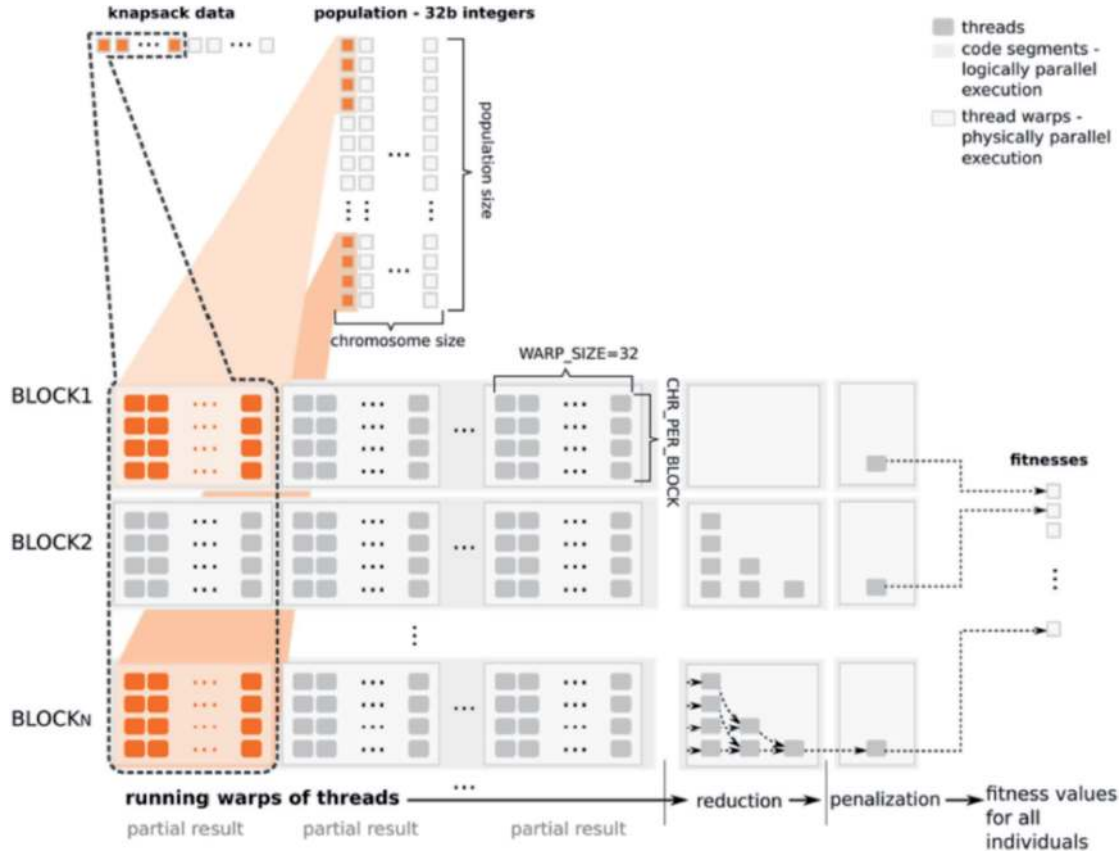

Figure 9. The configuration of kernel execution (from [17]).

The shared memory is used to staging data during the computation: all local subpopulations of each island stay in the shared memory. One major limitation of this way is the size of the memory, which makes only the cases with a very small population possible.

4.3. Cellular model on GPU

The cellular model imposes a geographic restriction on genetic search: individuals can only mate with its neighbors. The whole population is constricted with a spatial structure, defined as a connected graph. The common topology of the structure is a 2D toroidal mesh as shown in **Figure 10**, which limits the interactions between individuals. The use of cellular structure in populations helps for a better exploration over the search space with respect to the equivalent one with panmictic and decentralized populations.

The neighborhood of a given point in the mesh is defined in terms of Manhattan distance from it to its neighbor points. The two commonly used neighborhoods are L5, also called Von Neumann, and C9, also known as Moore neighborhood (Here, L stands for linear, while C stands for compact), as shown in **Figure 10**.

The neighborhood of all points is defined with the identical shapes. The neighborhood of each point of the mesh overlaps the neighborhoods of nearby points. The overlapping provides an

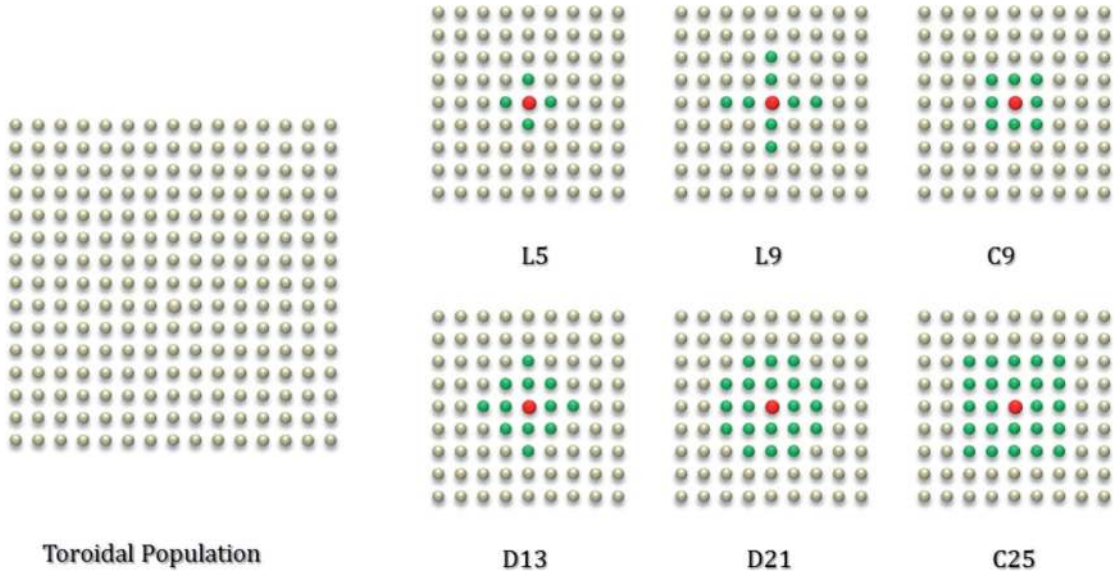

Figure 10. The toroidal mesh and neighborhoods.

implicit mechanism of migration to the cellular model: the genetic traits can spread smoothly through the overlapping neighborhoods. The mechanism helps the preservation of genetic diversity in the population longer than in a non-structured population and, therefore, provides a good trade-off between exploration and exploitation during the evolutionary process. The trade-off could be tuned by modifying the size of the neighborhood.

4.3.1. Integer-coded GA for quadratic assignment problem

The quadratic assignment problem is one of the fundamental combinatorial optimization problems in operations research. Conceptually, the problem is trying to find the optimal assignment of n facilities to n locations, knowing the distances between facilities and the flow between locations. Many combinatorial optimization problems can be written in this form. It has been a subject of extensive research in combinatorial optimization because of its importance in theoretical and practical domain. The problem is NP-hard, so there is no known algorithm for solving this problem in the polynomial time. Therefore, it becomes a target for GA community to show its stochastic and global search power for the complex optimization problems.

Cárdenas et al. implemented a cellular model of the integer-coded GAs for solving this problem [20]. Four types of neighborhoods, L5, C9, D17, and C25 (where L is linear, D is diamond, and C is compact), are mentioned, as shown in **Figure 11**.

All kernels are executed in a way that each GPU block corresponds to a chromosome and each GPU thread corresponds to a gene of the chromosome. Therefore, its granularity belongs to the gene level.

The crossover kernel implements the modified order crossover (MOX) method [21]. The first parent to MOX operator is decided by the block index, and the second parent is selected according to C9 neighborhood. The mutation kernel implements the random exchange method: selects two genes (two GPU threads) and exchanges their values.

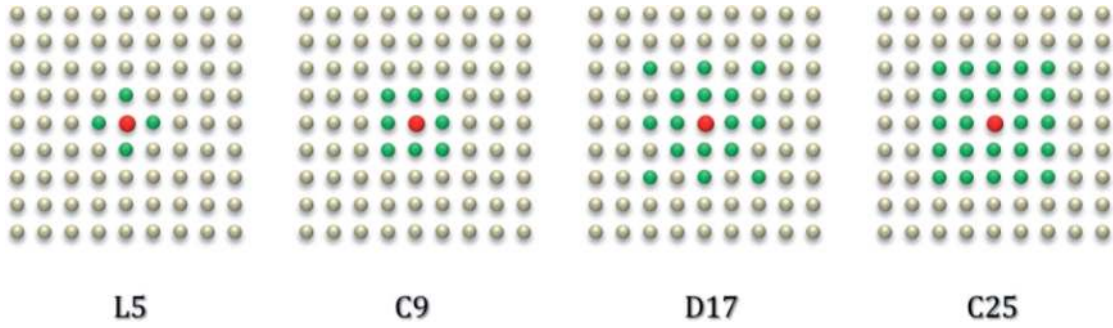

Figure 11. Neighborhood topologies.

Additional two nonstandard genetic operators are proposed: the transposition kernel and 2-opt kernel. In the transposition kernel, a portion of genes between two points is randomly generated in a chromosome and then reversed to obtain a new offspring. The 2-opt kernel mimics the 2-opt heuristic, a simple local search method to produce one offspring, both operators much like the mutation operator. It is a common practice to incorporate a kind of local search into genetic search to empower GAs when solving combinatorial optimization problems.

4.3.2. Integer-coded GA for independent task scheduling problem

Independent task scheduling problem is a kind of machine scheduling problem: assigning a set of independent computational tasks onto the different processors in a heterogeneous cluster. Finding a schedule that minimizes makespan to the problem is known to be NP-complete. The background of the problem is how to assign independent tasks onto the different processors in a cluster; therefore, time constraint is vital to this problem; usually, a limited amount of time is available for calculating the task schedule.

Pinel et al. implemented a cellular model of the integer-coded GA to solve the problem [22]. The population is arranged into a 2D toroidal mesh. The solution to GAs is encoded as a string of integer: the index of the string corresponds to a task, and the integer for a given index in the string corresponds to the machine to which this task is assigned. Uniform proportional recombination (UPR) is implemented as the crossover kernel, as shown in **Figure 12**.

The circled task in the middle string is the task being updated. Its neighborhood is defined by L5, the tasks in above, below, right, and left strings. The offspring is generated by the winner among the possible solutions according to its neighborhood. Two different criteria are defined for choosing the winner. The crossover operator is implemented with two kernels: one kernel is used to compute the probability for each solution in the neighborhood, and the other kernel is used to update tasks to generate offspring.

The crossover operator runs in a way that one thread handles with one task of a solution. Therefore, its granularity belongs to gene level. The other genetic operators of mutation, fitness, and replacement kernels, are launched in a way of one thread per solution, which belongs to the chromosome level.

Several instances are tested ranging from 512 tasks over 16 machines to 65,536 tasks over 2048 machines.

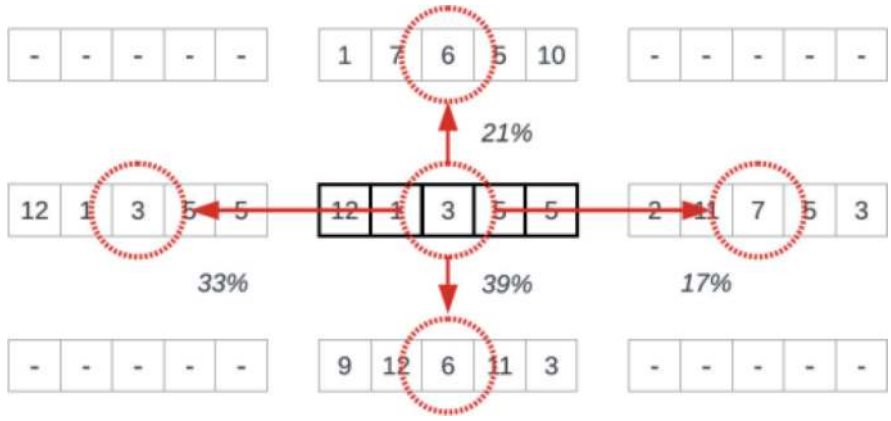

Figure 12. Illustration of crossover operator (from [22]).

5. A case study: multi-objective GAs for railway scheduling

The railway scheduling problem consists of building the timetable of trains, moving on a railway network under certain constraints. Nitisiri et al. reported their works of parallel and multi-objective GA implementation on GPU to generate train schedule for the Bangkok mass transit system (BTS) [23].

Bangkok BTS is a double-track railway system, and each track is operated in a single direction, as shown in **Figure 13**.

The BTS Silom Line consists of 13 stations, starting at National Stadium (W1) in central Bangkok and traveling southward to the last station S12 at Bang Wa station with the total length of 13 kilometers. The average number of passengers during the peak time in 2015 is above 2000 peoples. The BTS system operates daily from 05:00 am to 24:00 pm. The regular timetable is shown in **Figure 14**; the headways during normal operation are 15 mins. The total number of operating cycle is 91 cycles.

The train scheduling problem can be mathematically formulated as multi-objective integer programming problem with complex constraints. Two objectives are involved in this scheduling: (1) minimize average waiting time for the passenger, and (2) minimize a total number of the operating cycles during the operating period. These objectives are in conflict with each other: the more trains operating during the day, the less waiting time for the passenger but increasing the operational cost for the company.

A schedule is encoded as a $C \times 2S$-length matrix G, where C is the number of train operating cycles and S is the number of the stations, as shown in **Figure 15**.

A master-slave model is implemented on GPU for multi-objective GA with hybrid sampling strategy and learning-based mutation to solve the railway train scheduling problem. Data layout in the global memory belongs to the chromosome-based layout. Two-point cut crossover and learning-based mutation are used to perform genetic search. For evaluating each individual with respect to two objectives, a method called Pareto dominating and dominated

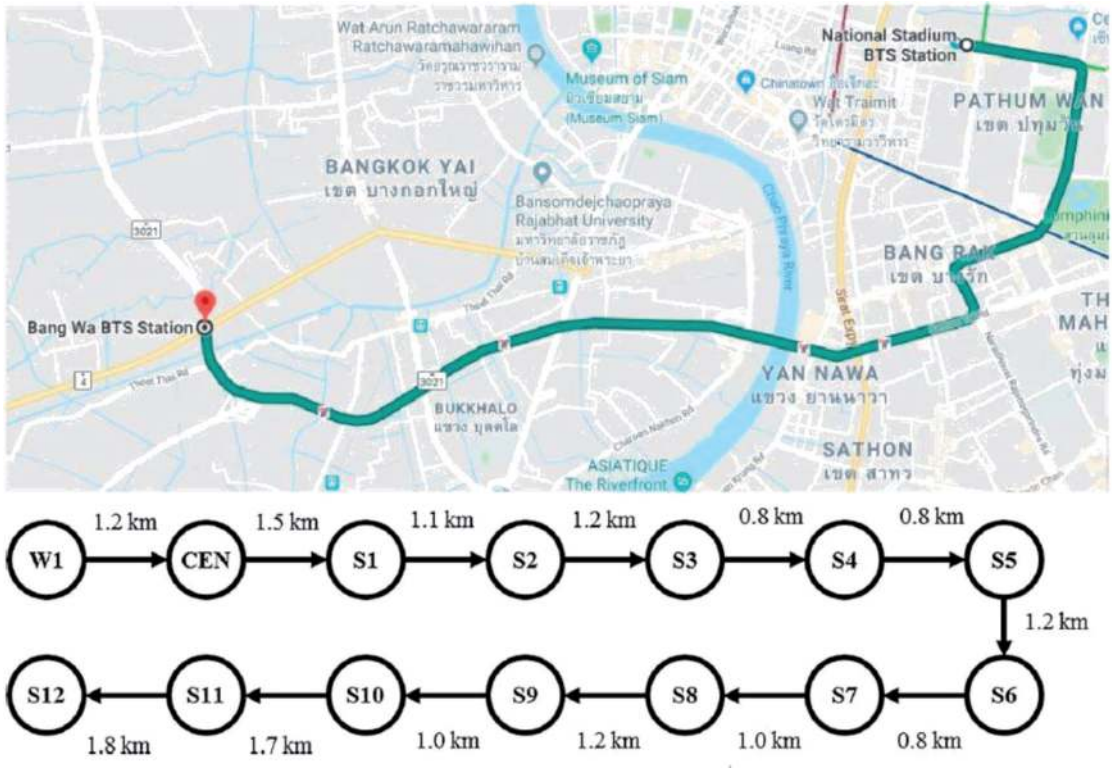

Figure 13. The Bangkok BTS transit line.

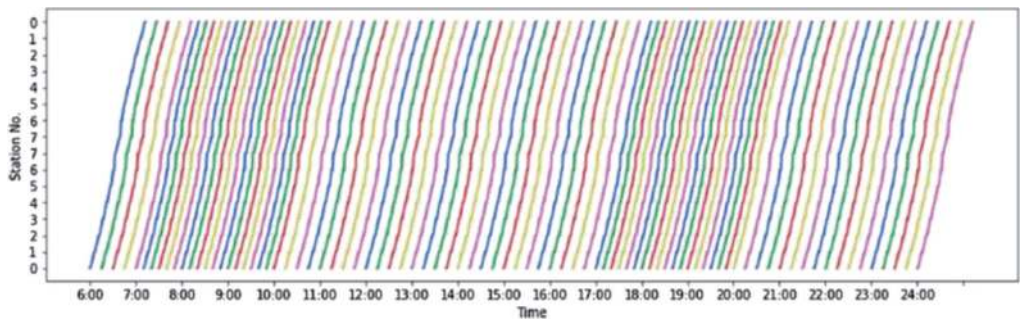

Figure 14. Regular timetabling scheme for Bangkok BTS transit line.

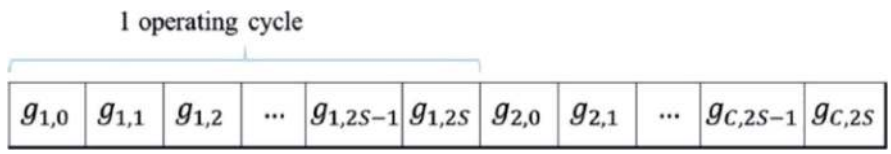

Figure 15. Genetic representation of a schedule.

relationship-based fitness function (PDDR-FF) is implemented as the evaluation kernel, which prefers the solutions in the edge area of the Pareto front [24]. A hybrid sampling strategy is adopted to select next generation, which has a tendency to converge toward the central area of the Pareto front [25].

Kernels are launched with 2D grid and 3D blocks and one GPU thread healing with one chromosome; therefore, its granularity belongs to the type of the chromosome level.

6. Discussion

Most published works on parallel GAs on GPU are conducted before 2013 on the earlier generations of GPU architecture. Since 2008, six generations of Tesla architectures for high-performance computation are released: Tesla, Fermi, Kepler, Maxwell, Pascal, and Volta. The average life span of one generation is about 2 years. The latest versions of GPU are Tesla V100 and Tesla P100, delivering a unified platform for accelerating both HPC and AI works.

Most published works tackle relatively small-scale problems. In handling large-scale problems, an efficient implementation of parallel GAs on multiple GPUs becomes imperative. A promising approach to do this is the hybrid model of island and cellular model on modern GPUs.

The earliest effort to combine these two models is given by Gruau in 1194, implemented in CPU to evolve and train a neural network [26]. Dorronsoro et al. reported their research works to use the hybrid model to solve a very large scale of vehicle routing problem in 2007, executed on a grid composed of 125 heterogeneous CPU nodes [27].

It is worthy to make further investigation on this hybrid model over multiple GPUs: implement a cellular mode in each GPU and treat each GPU as an island, as shown in **Figure 16**, where a 2D torus migration topology is defined among GPUs.

The hybrid model possesses both merits of island and cellular model: keeping the genetic diversity of population among islands and empowering local search ability on a structured population within each island. Implementing parallel GAs on multiple GPUs enables us to tackle larger-scale problems.

One of the promising areas to apply this hybrid model is to evolve better deep neural network. The notorious issue with deep net is what is called hyper-parameters, the very values of the deep net that cannot be learned and must be predetermined subjectively. Ever since the late 1990s, researchers have tried to evolve the best structure for network with GAs to solve the hyper-parameter issue [28]. There is a special term for this research that combines two powerful AI algorithms of genetic algorithms and neural networks: neuroevolution. Since 2016, the idea of architecture search through neuroevolution is attracting a number of researchers. Gaier and Ha reported their interesting results on using evolutionary algorithms to find minimal neural network architectures that can perform several reinforcement learning tasks without weight training [29]. For a large-scale application, weight training will be a painful and time-consuming process for most deep learning network methods.

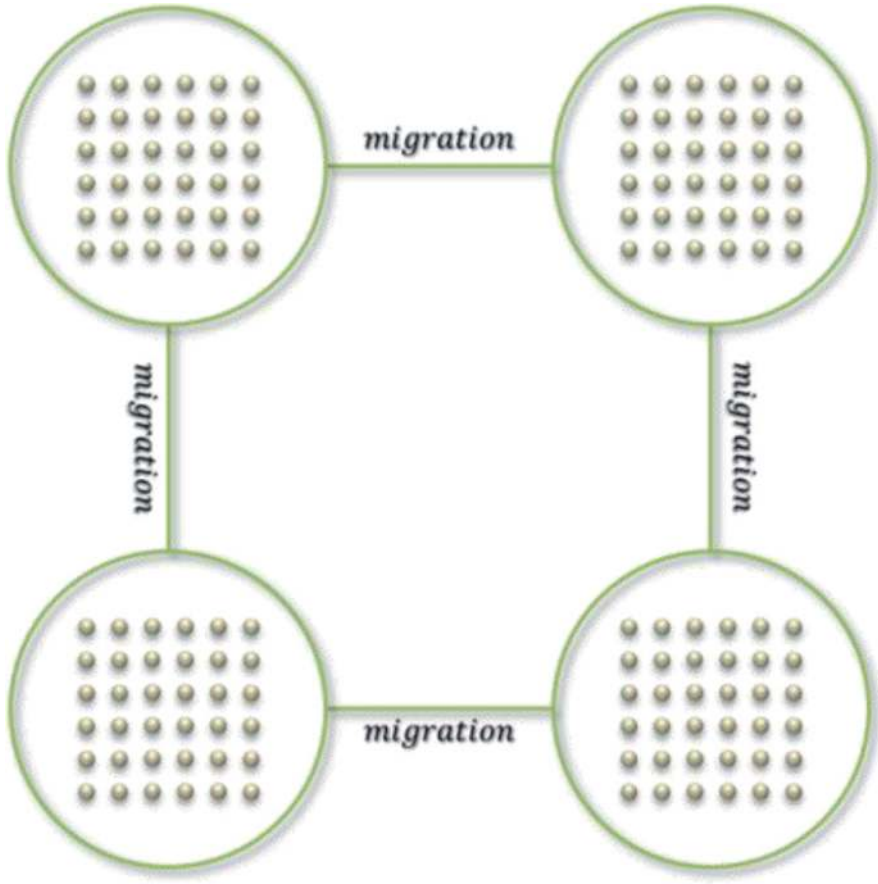

Figure 16. An example of a hybrid model over multiple GPUs: each GPU is an island, a cellular model is implemented within each GPU, and a 2D torus migration topology is defined among GPUs.

7. Conclusion

This chapter provides a concise yet comprehensive overview of selective works of parallel GAs on GPU from the perspective of GPU computing architecture.

Nowadays, the CPU-GPU heterogeneous architecture has become the mainstream computing platform that accelerates applications, which has led to a fundamental paradigm shift in parallel programming. The difference between the parallel GAs on CPU and the parallel GAs on GPU is clearly distinguished in the chapter. Conceptually, a parallel GA on CPU runs multiple instances of a sequential GA over several processors, while a parallel GA on GPU organizes massive threads of hundreds of thousands to work truly in parallel to exert the power of GPU computing.

Early tries to accelerate GAs with GPU computing takes a naive approach: let each GPU thread run a sequential GA, which is not the way we should take to design a parallel program over such massive threads of GPU computing architecture. In the chapter, the parallel model,

the design approaches, and implementation strategies of parallel GAs are examined from the perspective of GPU computing architecture. Although designing a parallel GAs on GPU is both problem-dependent and architecture-dependent, the efficient implementations of parallel GAs on GPUs share the common principles of kernel optimization: (1) exposing sufficient parallelism, (2) optimizing memory access, and (3) optimizing instruction execution. CUDA platform exposes GPU architectural features directly to programmers that enables us having more control over the massively GPU threads in order to fully exploit the computing power of GPUs.

Most research works on parallel GAs on GPU are published around 2013 on the earlier generations of GPU architecture. Since then, several generations of Tesla architectures for high-performance computation are released. The latest versions are Tesla V100 and P100. It would be a very promising research area to probe all these parallel models on the latest generations of GPU architectures to find out how can we accelerate parallel GAs on them.

The research on hybridizing island and cellular model on GPU clusters would be a very promising area that enables us to tackle hyper-scale computing applications. The hybrid model have both merits of island and cellular models: keeping the diversity of solutions among islands and empowering the ability of local search onto the global and stochastic search within each island. It is worthy to try this model for neuroevolution in deep learning applications. Now neuroevolution is making a comeback, and GPU-accelerated parallel GAs will make this happen certainly.

Author details

John Runwei Cheng[1]* and Mitsuo Gen[2]

*Address all correspondence to: runweicheng@gmail.com

1 Broad Geophysical Technology, Inc., Houston, TX, United States

2 Research Institute of Science and Technology, Tokyo University of Science and Fuzzy Logic Systems Institute, Japan

References

[1] Gen M, Cheng R. Genetic Algorithms and Engineering Design. New York: John Wiley & Sons; 1997. p. 427

[2] Gen M, Cheng R. Genetic Algorithms and Engineering Optimization. New York: John Wiley & Sons; 2000. p. 511

[3] Goldberg DE. Genetic Algorithms in Search, Optimization and Machine Learning. Reading: Addison-Wesley; 1989. p. 432

[4] Gen M, Cheng R, Lin L. Network Models and Optimization: Multi-Objective Genetic Algorithms Approach. New York: Springer; 2008. p. 692

[5] Luque G, Alba E. Parallel Genetic Algorithms: Theory and Real-World Applications. Berlin: Springer; 2011. p. 95

[6] Zheng L, Lu Y, Ding M, Shen Y, Guo M, Guo S. Architecture-based performance evaluation of genetic algorithms on multi/many-core systems. In: Proceedings of 14th IEEE International Conference on Computational Science and Engineering (CSE 2011). 24-26 August 2011. Dalian, China: IEEE Computer Society; 2011. pp. 321-334

[7] Cheng J, Grossman M, McKercher T. Professional CUDA C Programming. New Jersey: John Wiley & Sons; 2014. p. 497

[8] CUDA Zone. Available from: http://developer.nvidia.com/cuda-zone [Accessed on: 18-06-2019]

[9] Hwang K. Advanced Computer Architecture: Parallelism, Scalability, Programmability. New York: McGraw-Hill Higher Education; 1992. p. 771

[10] Cheng JR, Gen M. Accelerating genetic algorithms with GPU computing: A selective overview. Computers and Industrial Engineering. 2019;**128**:514-525. DOI: 10.1016/j.cie.2018.12.067

[11] Pedemonte M, Alba E, Luna F. Bitwise operations for GPU implementation of genetic algorithms. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2011). Ireland: Dublin; 2011. pp. 439-446

[12] Debattisti S, Marlat N, Mussi L, Cagnoni S. Implementation of a simple genetic algorithm within the CUDA architecture. In: Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2009). Montreal, Canada; 2009

[13] CUDA Toolkit Documentation v10.1.168. Available from: https://docs.nvidia.com/cuda/index.html [Accessed on: 20-06-2019]

[14] Michalewicz Z. Genetic Algorithm + Data Structure=Evolution Programs. 3rd rev ed. New York: Springer; 1996. p. 387

[15] Arora R, Tulshyan R, Deb K. Parallelization of binary and real-coded genetic algorithms on GPU using CUDA. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC 2010). Spain: Barcelona; 2010. pp. 1-8

[16] Izzo D, Ruciński M, Biscani F. The generalized Island model. In: de Vega FF et al., editors. Parallel Architectures & Bioinspired Algorithms. Berlin: Springer-Verlag; 2012. pp. 151-169

[17] Jaros J. Multi-GPU Island-based genetic algorithm for solving the knapsack problem. In: IEEE Congress on Computational Computation (CEC 2012). Brisbane, Australia; 2012. pp. 1-8

[18] OpenMPI. Open source high performance computing. Available from: http://www.open-mpi.org/ [Accessed on: 18-06-2019]

[19] Luong TV, Melab N, Talbi E. GPU-Based Island model for evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2010). Portland, Oregon; 2010. pp. 7-11

[20] Cárdenas GE, Poveda Ch R, García HO. A solution for the quadratic assignment problem (QAP) through a parallel genetic algorithm based grid on GPU. Applied Mathematical Sciences. 2017;**11**(57):2843-2854

[21] Wroblewski J. Theoretical foundations of order-based genetic algorithms. Fundamenta Informaticae. 1996;**28**(3-4):423-430

[22] Pinel F, Dorronsor B, Bouvry P. Solving very large instances of the scheduling of independent tasks problem on the GPU. Journal of Parallel and Distributed Computing. 2013;**73**(1):101-110

[23] Nitisiri K, Gen M, Ohwada H. A parallel multi-objective genetic algorithm with learning based mutation for railway scheduling. Computers and Industrial Engineering. 2019;**130**:381-394. DOI: 10.1016/j.cie.2019.02.035

[24] Gen M, Cheng JR, Nitisiri K. Advances in hybrid genetic algorithms with learning and GPU for scheduling problems. In: The International Conference for High Performance Computing (SC19). Denver, Colorado; 2019. pp. 17-22

[25] Gen M, Lin L. Genetic algorithms. In: Wah B, editor. Wiley Encyclopedia of Computer Science and Engineering. Hoboken, N.J.: John Wiley & Sons; 2009. pp. 1367-1381

[26] Gruau F. Neural network synthesis using cellular encoding and the genetic algorithm [Unpublished doctoral dissertation]. L'Universite Claude Bernard-Lyon I. 1994

[27] Dorronsoro B, Arias D, Luna F, Nebro A, Alba E. A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP. In: High Performance Computing & Simulation Conference (HPCS 2007). 2007. pp. 759-765

[28] Yao X. Evolving artificial neural networks. Proceedings of the IEEE. 1999;**87**(9):1423-1447

[29] Gaier A, Ha D. Weight agnostic neural networks. Available from: https://arxiv.org/abs/1906.04358