Computers & Industrial Engineering 128 (2019) 514–525

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

Accelerating genetic algorithms with GPU computing: A selective overview

John Runwei Cheng[a,*], Mitsuo Gen[b]

[a] Broad Geophysical Technology, Inc., 10630 Haddington Dr., Houston, TX 77043, United States
[b] Research Institute of Science & Technology, Tokyo University of Science & Fuzzy Logic Systems Institute, Japan

ARTICLE INFO

Keywords:
Parallel genetic algorithms
GPU computing
Parallelism

ABSTRACT

The emergence of GPU-CPU heterogeneous architectures has led to a fundamental paradigm shift in parallel programming. Accelerating *Genetic Algorithms* (GAs) on these architectures has received significant attention from both practitioners and researchers ever since GPUs emerged. In the past decade we have witnessed many progresses on migrating parallel GAs from CPU to GPU (Graphical Processing Unit) architecture, which makes this research field 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 global search to explore large and complex search spaces for better solutions.

Designing a parallel algorithm on GPU is quite different from designing one on CPU. On CPU architecture, we typically consider how to distribute data across tens of CPU threads, while on GPU architecture, we have more than hundreds of thousands of GPU threads running simultaneously. Therefore, we should rethink the design approaches and implementation strategies of parallel algorithms to fully utilize the computing power of GPUs to accelerate the computation of GAs. The intention of this paper is to give an overview on selective works of parallel GAs designed for GPU architecture.

In this survey paper, we first reexamine the concept of granularity of parallelism for GAs on GPU architecture, discuss how the aspect of data layout affect the kernel design to maximize memory bandwidth, and explain how to organize threads in grid and blocks to expose sufficient parallelism to GPU. The comprehensive overview on selective works since 2010 then follows. The focus is mainly on the perspective of GPU architecture: how to accelerate GAs with GPU computing. Performance issues are not touched in this review, because most of these works are conducted on very early GPU cards, which are out of date already.

We finally discuss some future research suggestions in the last section, especially about how to build up an efficient implementation of parallel GAs for hyper-scale computing. Many industrial and academic disciplines will be benefited from the GPU accelerated parallel GAs, one of the promising area is to evolve better deep neural networks.

1. Introduction

As *Deep Learning* (DL) has led to important breakthroughs in both industrial world and academic community, much attention has been given overwhelmingly to accelerate various *Neural Network* (NN) using GPU (Graphical Processing Unit) computation, but less attention turns to accelerate *Genetic Algorithms* (GAs), one of the five tribes of *Machine Learning* (ML) (Domingos, 2015).

GAs are stochastic and global stochastic search methods which rely on a trade-off between exploiting good solutions and exploring global search space. GAs have been successfully applied to many optimization problems in different disciplines which are difficult to solve by conventional mathematical programming methods (Gen and Cheng, 1997, 2000).

As a population-based search method, GAs have a great potential for accelerations 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 important stages according to the progress of hardware architecture: the stage before the emergence of heterogeneous computing architecture and the stage after it. In the early stage, the basic idea to speed up GAs computation is to distribute computation among multiple concurrent computation, implemented either across multiple processes within one processor or across multiple processors connected within a local network. Each process essentially is a sequential GA. Typically, you run at most parallel GAs across tens or hundreds of computer nodes. Since the last decade, GPU has become a

* Corresponding author.
E-mail addresses: runweicheng@gmail.com (J.R. Cheng), gen@flsi.or.jp (M. Gen).

https://doi.org/10.1016/j.cie.2018.12.067

Available online 29 December 2018
0360-8352/

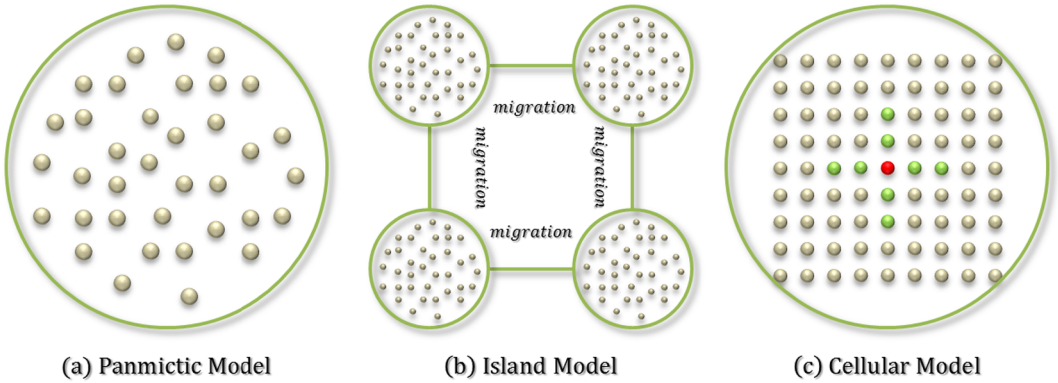

Fig. 2.1. Parallel GAs models (circle represents an isolated population).

mainstream computing platform that accelerates applications. You can have tens of thousands of GPU threads executing in parallel, which makes a huge difference in designing a parallel algorithm to accelerate your applications. The traditional parallel approaches for CPU architectures are not suitable for GPU architectures.

The intention of the paper is to give a comprehensive overview on how to accelerate GAs on GPU architecture. We will focus on literatures after 2010, because that since then, both GPU hardware architectures and CUDA software development platform have had significant progress. The Fermi architecture, released in 2010, is the world's first complete GPU computing architecture, which makes GPU computing truly applicable in industrial worlds. CUDA 4.2, released in 2012, is a stable version of parallel computing platform, which makes CUDA programming easy and productive.

Several early review literatures do not distinguish the difference of parallel design between CPU and GPU architecture. The authors may still view the parallel approach from the aspect of CPU architecture, not from GPU architecture. Actually, designing a parallel algorithm on CPU basically requires nothing about architecture knowledge, with a divide-and-conquer approach to run multiple sequential instance each with different sub dataset. Designing parallel algorithms is always motivated by performance and driven by profiling. GPU computing is unique in that the exposed architectural features enable us to extract every iota of performance from this powerful hardware platform, that is why we need to rethink and redesign algorithms suitable for this computing architecture.

This paper is organized as follows. A brief review of traditional parallel GAs on CPU architectures is given in the next section to highlight basic ideas behind the traditional approaches. Section 3 discusses major issues on how to accelerate GAs with GPU computing. Section 4 gives a detail overview of parallel GAs implemented on GPU architecture. The paper ends with a discussion of future research suggestions, especially about how to build up an efficient implementation of parallel GAs for hyper-scale computing.

2. Brief overview of traditional approaches of parallel GAs

GAs are stochastic search algorithms inspired by the mechanism of population genetics and the principles of natural selection, such as reproduction, recombination, mutation, and selection. Different from conventional search techniques, GAs work on a set of solutions, called a *population*. Each individual in the population, called a chromosome, represents a solution to the problem at hand. The set of solutions evolves through successive iterations, called *generations*. During each generation, the chromosomes are evaluated, using some measures of fitness. To create the next generation, new chromosomes, called *offspring*, are formed by either merging two chromosomes using a *crossover operator* or modifying a chromosome using a *mutation operator*. A new generation is formed by a *selection*, according to each *fitness*, some of the parents and offspring. Chromosomes with better fitness values have much higher possibilities of being selected. After several generations, the algorithm converges to the best one, which hopefully is close to the optimal solution.

As a generic metaheuristic, GAs possess many advantages over the conventional search methods. It requires neither much mathematical properties about the problem, nor domain knowledge, so it can handle much complex problems with any kind of objective functions and constraints, linear or nonlinear, defined on discrete, continuous or mixed search spaces. The ergodicity of genetic operators makes GAs very effective at performing global search. GAs also provide us a great flexibility to hybridize with domain-dependent heuristics to make an efficient implementation for a specific issue, such as *Job-shop scheduling problem* (JSP) (Cheng, Gen, and Tsujimura, 1996, 1999). Therefore, GAs often perform well on all types of combinatorial optimization problems because they do not make any assumption about the underlying fitness landscape (Gen, Cheng, & Lin, 2008).

GAs have a tremendous potential for parallelization. Since GAs work on a set of independent solutions, all genetic operators can be parallelized by data-parallel approach. Therefore, it is easy to distribute the computational load among multiple processors through a data parallel approach. The motivation for parallelizing 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 to parallelize GAs on CPU architecture (Gordon & Whitley, 1993, Chipperfield & Fleming, 1994): (see Fig. 2.1)

(1) Master-Slave model,
(2) Island model, and
(3) Cellular model.

The Island model is often named as the *coarse-grained* parallel GAs, and the Cellular model is often named as the *fine-grained* parallel GAs. There are various hybrid types which combines the coarse and fine-grained approaches (Cantú-Paz, 1998).

From the perspective of mating mechanisms, the parallel GAs can be classified into three major categories: mating globally, mating locally, and mating global within an isolated sub-population with migration among sub-populations. The *Master-slave model* belongs to the type of mating globally. Therefore, it is also called *Panmictic model* or *Global model*. In this model, individuals are permitted to mate freely within entire population. Controversial to the Panmictic model, the *Cellular model* belongs to the type of mating locally. In this 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 of the Panmictic model and Cellular model. In this model, whole population is divided into several isolated sub groups, called

Island, and within an island, individuals mate freely, while mating across islands are restricted. A new operator is introduced into the Island Model, the *migration operator,* which exchanges periodically a portion of sub-population among islands in a predefined way to bring new genetic materials into each island. In the Panmictic Model, since it mates globally, no need for the migration. In the Cellular model, though it mates locally, the migration feature is archived through its neighborhood overlapping.

From the perspective of parallelism, all of the traditional approaches belong to data parallelism. There are two fundamental approaches of parallelism: the *task-parallelism* and the *data-parallelism*. The data parallelism is a way to speed up computation by distributing the data across different processors in a parallel computing architecture. Since GAs work on a population of solutions, it is easy and straightforward to parallelize it with the data-parallel approach.

From the perspective of population, the traditional approaches can be classified into two basic types: *single population* and *multiple populations* (Knysh & Kureichik, 2010). The Master-Slave and Cellular models belong to the single population category, and the Island model belongs to the multiple population type. The multiple population approach evolves solutions in several isolated population pools, each pool running on one processor or computer, and then exchanges partial species among different pools at certain evolution stages, aiming at finding better solutions than the single solution approach.

From the perspective of implementation, the traditional approaches can be classified as follows (Umbarker & Joshi, 2013):

(1) parallel GAs run over multiple processors using OpenMP or POSIX threading.
(2) over cluster (tightly connected computers) using MPI (Message Passing Interface).
(3) over grids (loosely coupled computers) using MPI.
(4) over clouds (a network of remote computation resources hosted on the Internet) using Hadoop.

Conceptually, all of implementation of the traditional approaches are (Munawar, Wahib, Munetomo, & Akama, 2008)

(1) Keep multiple instances running across multiple processors or computers,
(2) Each instance on one processor/computer is a sequential algorithm in nature, and
(3) A communication mechanism is needed to keep overall populations to evolve to a better solution, which can be implemented either synchronizing way or asynchronizing way.

The approaches to parallelize GAs on CPU are not suitable for GPU architecture. The following section will discuss what special issue we should take into consideration when designing a parallel GAs with GPU computing.

3. Major issues of parallel GAs on GPU

This section will give a brief discussion of major issues on how to accelerate GAs with GPU computing. We assume that readers have basic understanding on GPU architecture and CUDA programming. If you need to know primary knowledge of GPU computing, refer to the recent book of professional CUDA C programming (Cheng, Grossman, & McKercher, 2014).

Many of early works on parallel GAs on GPU can be viewed as a kind of extension of CPU approaches onto GPU, therefore, which do not fully exploit the computation power of GPU architecture because of less understanding of the fundamental features of GPU computing architecture (Zheng et al., 2011).

The major difference between parallelizing GAs on CPU and on GPU is that CUDA architectural features, such as memory and execution models, are exposed directly to us, which enables us to have more control over the massively parallel threading environment. Top three principles on optimizing GPU implementation, listed in order of importance, are

(1) exposing sufficient parallelism,
(2) optimizing memory access, and
(3) optimizing instruction execution.

When designing parallel GAs on GPU architecture, we need to rethink some special issues related to the underlying architecture according to these principles. Therefore, it is necessary for us to have some basic knowledge of the underlying architecture for harnessing the power of GPUs.

From the aspect of algorithms design, the granularity of parallelism should be reexamined since we will have hundreds of thousands of threads running simultaneously on GPU. From the aspect of data layout in global memory, we need to consider how to design a kernel in order to make it in an aligned and coalesced memory access pattern to maximize bandwidth utilization. From the aspect of kernel execution, we should consider how organize threads in grid and blocks to expose sufficient parallelism to GPU so as to saturate both instruction bandwidth and memory bandwidth.

3.1. Granularity of parallel GAs on GPU

First, we introduce a new definition of granularity of parallel GAs on GPU with the special consideration of GPU architecture. The granularity is defined by how many GPU threads are working together to handle one chromosome during genetic operations. If a chromosome is handled by one GPU thread independently in genetic operations, we call its granularity is in a *grain size* of chromosome level. If a chromosome is handled by a group of GPU threads cooperatively, we call its granularity is in the grain size of gene level.

From the point of view of the parallel granularity, all parallel GAs proposed so far can be roughly classified into two basic categories, the granularity in chromosome level and the granularity in gene level. According to the group size of GPU threads, the granularity in gene level is 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 chromosome, while in the later, a warp of threads handles one chromosome.

Because that GPU partitions compute resources among blocks and threads, the more a kernel requires the shared memory and local register, the less the thread blocks can run simultaneously. Therefore, the compute resource usage in a kernel is a key factor that limits the sufficient parallelism exposed to GPU. Generally, a kernel designed with the granularity in chromosome level requires more shared memory and registers then a kernel designed with the granularity in gene level. For a large scale application, it will become a major issue that hinders the performance.

Most parallel GAs proposed for CPU architecture belongs to the category of the granularity in the chromosome level. Most of early works on parallel GAs proposed for GPU architecture also belong to the category of the granularity in chromosome level, since it is a nature extension of CPU approach to migrate CPU code to GPU, and an easy way for beginners of CUDA C programmers without the need of knowing much in-depth knowledge of GPU architecture. This naive approach cannot fully utilize the computational power of GPU to accelerate GA. Since more than hundreds of thousands of threads can run simultaneously on GPU, we should rethink and redesign parallel GAs in the granularity in gene level to fully exploit its computing power. A third type of granularity is the mixed level: some kernels is in the genotype level while others in the chromosome level.

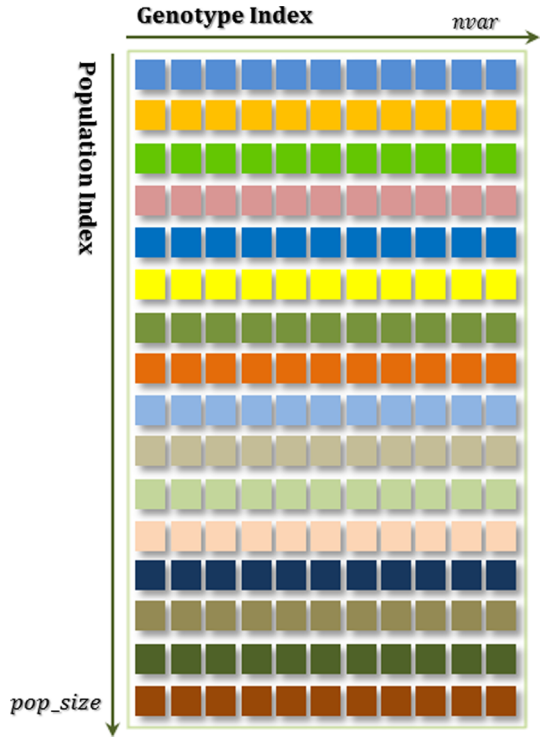

Fig. 3.1. 2D view of chromosome-based layout in global memory.

3.2. Data layout on global memory

Most GPU applications tend to be limited by memory bandwidth. To maximize global memory bandwidth utilization, it is a fundamental step in the kernel design to make global memory access to be both aligned and coalesced. How data are organized in global memory will affect the way kernels access memory.

Essentially, all data are stored linearly in global memory. The whole population can be viewed logically as 2D matrix with two dimensions: one dimension is the index of chromosomes in whole population, and the other is the index of genotype within one chromosome. Therefore, there are two basic ways to layout whole population in global memory: the chromosome-based layout and the genotype-based layout.

Given *pop_size* as the population size and *nvar* as the total number of gene in one chromosome, the chromosome-based layout is shown in Fig. 3.1, the whole population can be viewed logically as 2D matrix of $nvar \times pop\_size$. The genotype-based layout is shown in Fig. 3.2, the whole population can be view as 2D matrix of $pop\_size \times nvar$.

In the chromosome-based layout, the fast dimension is the index of genotype within one chromosome, and the slow dimension is the index of chromosome in the population. Since one chromosome is allocated to a contiguous memory space, a kernel with the granularity in gene level will take the benefit of the coalesced memory access in this data layout. On the contrary, in the genotype-based layout, the fast dimension is the index of chromosome in the population and the slow dimension is the index of genotype within one chromosome. In this layout, genes in one chromosome are interleaved and the stride between two consecutive genes in one chromosome is the number of *pop_size*. Since all i^{th} gene of different chromosome are allocated to a contiguous memory space, a kernel with the granularity in the chromosome level may take the benefit of coalesced memory access in this data layout. But this layout will be not friendly for evaluation operation and migration operation.

Fig. 3.2. 2D view of genotype-based layout in global memory.

During the evolutionary computation, two types of data are accessed repeatedly: the chromosome and its fitness values associated. How to arrange them in memory will also affect the memory utilization. There are two basic ways of arranging the data: *Structure of Arrays* (SoA) and *Array of Structures* (AoS). The SoA is a layout which separates different type of data elements into different array, while the AoS is the opposite layout, in which data of different types are interleaved. The AoS is often more intuitive. C programmers has a habit to organize data into structures, then create arrays of these structures to hold the whole data set. The SoA arrangement allows more efficient use of the parallelism of the SIMD (Single Instruction Multiple Data) technologies, because the homogeneous data is ready for computation How to organize data elements in memory depends on how they are used by different kernels. Since different genetic operator kernel accesses different type of data elements, the recommended way is to use separate array layouts, but not organize them into structure.

All discussion above is about the real coded GA or integer coded GA on GPU. It will be a different story for the binary GA. Two implementations have been reported for the binary GA: (1) byte-wise binary encoding and (2) bit-wise binary encoding. In the byte-wise method, 8 bits is used to represent 0 or 1, while in the bit-wise method, only 1 bit is used to represent 0 or 1, therefore saving one eighth memory space. A nature way to implement the binary GA is to pack multiple bits into a non-Boolean data type, typically, using an unsigned integer data type to pack 32 bits into one unit for processing (Pedemonte, Alba, & Luna, 2011). This bitwise approach can not only save memory usage, but also accelerate execution time due to the less memory access. It is very easy to implement bitwise operations by using CUDA toolkit.

3.3. Execution configuration

GPU organizes thread execution using a grid of thread blocks that gives us the freedom to choose the kernel launch configuration by specifying:

- The number of threads per block and the dimension of the block
- The number of thread blocks of a grid and the dimension of the grid

Through the execution configuration, we can control how threads are arranged into thread blocks so as to expose sufficient parallelism on the GPU to saturate both instruction bandwidth and memory bandwidth. Exposing sufficient parallelism is the number one principle to optimize kernels. GPU devices with different compute capabilities have different hardware limits; therefore, grid and block heuristics play a very important role in optimizing kernel performance for different platforms.

Both block and grid can be organized as 1D, 2D and 3D layouts. How to decide the execution configuration for a kernel is heavily depends on how the data is laid out in the global memory and how the kernel access the data. We will give detailed explanation when reviewing each selected works in the subsequent sections.

3.4. Shared memory

Shared Memory is a key enabler for GPU kernel performance, which can be viewed a program-managed cache. You have full control over when data is moved into shared memory, and when data is evicted. By allowing you to manually manage shared memory, CUDA makes it easier for you to optimize your application code by providing more fine-grained control over data placement and improving on-chip data movement.

There are two main reasons to use shared memory in a kernel. One is to cache data on-chip and reduce the amount of global memory traffic. The other is to transform how data is arranged in shared memory to avoid non-coalesced global memory accesses.

Depending on the nature of your algorithms and the corresponding access patterns, non-coalesced accesses may be unavoidable. However, it is possible to improve global memory coalesced access by using shared memory in many cases. Shared memory enables threads within the same thread block to cooperate, facilitates reuse of on-chip data, and can greatly reduce the global memory bandwidth needed by kernels.

4. Lecture review of parallel GAs on GPU

This section will give a brief overview on some selected works since 2010. The focus will be mainly on the perspective of GPU architecture: how accelerate GAs on such kind of many-core computing architecture. The survey is divided three parts for easy access: works related to Master-Slave model, Island model, and Cellular model, respectively.

4.1. The Master-Slave model on GPU

It is straightforward to implement the Master-Slave scheme on a heterogeneous computing system, as the relationship between CPU and GPU is master and slave in nature. Earlier studies take a very simple way to implement the scheme on the heterogeneous system: the GPU only calculates the fitness evaluation, while the CPU will do all other jobs (Zheng et al., 2011). It is a very inefficient implementation because it needs frequently data transfer between CPU and GPU at each iteration. Because that the computation at each generation is very little on GPU, so the communication between CPU and GPU dominates the execution time. Since most of workloads are on the CPU, so that the CPU becomes the bottleneck while the GPU stays idle most time.

4.1.1. Binary GA for one max problem

Debattisti, Marlat, Mussi, & Cagnoni (2009) implemented a fixed length binary GA on GPU to solve the One-Max problem, a well-studied problem in GA community (Mühlenbein & Chakraborty, 1997). Except for the initialization of individuals, all other operations of selection, crossover, mutation, and evaluation were implemented on GPU, so as to avoid data transfer between GPU and CPU memories during the evolutionary process.

One-Max problem is a toy problem in the evolutionary computation world, just as the “Hello World” for C programmer (Du, Ma, Sakamoto, Furutani, & Zhang, 2014). The problem is to produce a string of all 1s, starting from a population of random binary strings. This problem is widely used since it is very simple and illustrates well the potential of evolutionary algorithms.

It is a nature way to take a binary representation for the problem. Authors adopted a byte-wise method to encode population: each individual is encoded as a string of the unsigned char type data, that is, using one byte to represent one bit of the genome. As discussed in the last section, it is better to use bitwise method for the binary GA.

The crossover kernel was implemented with two-cut point method. Each thread block was responsible for dealing with two individuals to produce offspring, and each thread was responsible for handling 4 bits each time. The mutation kernel was implemented by means of bitwise exclusive OR operations, each block was responsible for dealing with one individual, and each thread was responsible for handling 4 bits. The evaluation kernel was implemented by means of parallel reduction, each block was responsible for dealing with one individual and each thread was responsible for calculating 4 bits, save the sum onto a vector of the shared memory, and the final fitness value was then calculated by means of a parallel reduction. Since multiple threads cooperatively deal with one individual, its granularity belongs to the genotype level.

All kernels were launched with the execution configuration of 1D grid and 1D blocks. Grids and blocks represent a logical view of the thread hierarchy of a kernel function, which affect the performance since the key is to expose sufficient parallelism. Not much details about this issue was discussed this paper.

Their experiments were conducted on a GeForce 8800 GT, an early generation of GPU card produced around 2007 by NVIDIA, with the compute capability 1.1. The CUDA Toolkit, the software development environment, in that time is around version 2.0. The compute capability of current GPU card is about 7.x and CUDA Toolkit version is about 9.x (CUDA, 2018). Although the hardware and software environment of their experiment is out-of-date, the basic idea on how to design genetic operators on GPU is on the right track: exposing parallelism as much as possible and keeping the coalesced memory access as possible as we can.

4.1.2. Real-coded GA for unconstrained optimization problems

Arora et al. reported their works on the parallelization of binary and real-coded GA on GPU to solve the unconstrained optimization problems (Arora, Tulshyan, & Deb, 2010).

The genotype-based layout was used to shore whole population in global memory, that is, the index of chromosome in the population is in the fast dimension, and the index of variables in a chromosome is in the slow dimension, as shown in Fig. 3.2.

In the initialization kernel, each thread was responsible for generating one variable at one time independently. The kernel execution configuration was 1D block and 2D grid: each block contains *nthreads* and the grid was organized as (*pop_size*/*nthreads*, *nvar*) to keep a coalesced global memory writing.

The Simulated Binary Crossover (SBX) was implemented as the crossover kernel. Its execution was organized in 1D block and 2D grid. The grid execution configure was ($pop\_size/(2 \times nthreads)$, *nvar*). Conceptually, each thread was responsible for doing crossover operation on a variable x_i of two mates. Because each block works independently with each other and each chromosome is operated by *nvar* different blocks, it will cause an issue that a chromosome may be selected to reproduce offspring many times. No detail was given about how offspring is created (Arora et al., 2010), so we cannot make more comment on the implementation.

The one-variable mutation (corresponding to one-bit mutation of the binary coded GA) was implemented with a kernel, using the same execution configure as the one of the initialization kernels, (*pop_size*/*nthreads*, *nvar*). Since one chromosome may be operated by multiple blocks independently, it has the same issue as what the crossover kernel will have: one chromosome may be selected to reproduce offspring many times. No details about the evaluation kernel was given in the paper (Arora et al., 2010).

The granularity of their implementation belongs to the one in the gene level since multiple threads cooperatively deal with one or two individuals.

Their experiments were conducted on Tesla C1060, an early generation of GPU card produced around 2009 by NVIDIA, with the compute capability 1.3. Three unconstrained optimization problems were tested: One-Max function, Ellipsoidal function, and Rosenbrock function, which are well used in used in the GA literature. Although it is deliberative about how we organize GA data in GPU global memory since it will affect the way of kernels implementation, it is on the right track that authors intensively designed their GA implementation in

light of GPU architecture.

4.1.3. Binary GA for unconstrained optimization problems

Oiso et al. implemented a binary GA on GPU to solve unconstrained optimization problems (Oiso, Matsumura, Yasuda, & Ohkura, 2011). Unconstrained optimization problems consider the problem of minimizing or maximizing a nonlinear function of real variables over a given domain. Often it is practical to replace the constraints of an optimization problem with penalized terms in the objective function and to solve the problem as an unconstrained problem. It is a well-tested problem in GA community (Gen & Cheng, 1997).

The binary code was used to represent the population, and each float value was coded with 16 bits. Data layout was not mentioned in the paper, from the description of kernel implementations, the chromosome-based layout should be adopted to store whole population in global memory.

The *one-cut point crossover* method was used in the crossover kernel. It is quit confusing how it works from the description of the kernel implementation, mainly because of grammar mistakes. Each thread block was responsible for dealing with two individuals to produce offspring, and each thread was responsible for handling one gene. The point mutation was implemented with a kernel, each tread block responsible for dealing with one individual and each thread responsible for handling one gene. Since the binary encode will generate illegal offspring after the crossover and mutation operations: the offspring representing a point outside the give domain. No any information was given in the paper about how to handle the illegal offspring in their implementation. The parallel reduction was used in the evaluation kernel. Each block was responsible for dealing with one individual in such a way that, each thread in the block loading the corresponding variables from global memory to the shared memory, calculating the partial fitness value, and then saving back to the shared memory. The fitness value of the individual was then calculated by a parallel reduction on the shared memory. Finally, the thread zero writes the fitness value back to the global memory to its corresponding location. The granularity of their implementation belongs to the gene level since multiple threads cooperatively deal with one or two individuals.

The *tournament selection* was implemented with two kernels: one kernels performing tournament selection, and the other kernel copying winner from the current generation into next generation. Two kernels were executed in sequential. No much details were given to describe how the grid and blocks were organized to execute kernels. For the copying kernel, each block handles one individual copy, it should be 1D grid and 1D block arrangement. For the tournament kernel, each thread was responsible for handling a tournament, that is, reading the fitness values of two selected individuals, and writing the winner index onto the global memory.

Bitonic sort method, as the sorting kernels, was implemented for the selection process. Conceptually, the *tournament method* chooses a set of chromosomes and picks out the best one as the winner for the next generation. Since the method usually picks a very small set of individuals for competing, it is disputable if it is necessary to implement the sorting kernels for the selection process.

By the way, we now can easily implement various sorting method by utilizing *CUB*, a library of high-performance parallel primitives, implemented by NVIDIA as a C++ template library with reflective type structure (CUB, 2018). CUB primitives are not bound to any particular width of parallelism or any particular data type, that allows them to be flexible and tunable to fit an application need. Three types of cooperative primitives are available in CUB: warp-wide, block-wide, and device-wide primitives.

Their experiments were conducted on Geforce GTX280, an early version of GPU card produced around 2009 with the compute capability 1.1. The CUDA Toolkit is 2.3 in their test. Eight functions were tested on two small cases: one with 32 variables and the other with 128 variables.

4.1.4. Others

Cavuoti et al. reported their works on accelerating GAs with GPU computation for an astrophysical data mining problem (Cavuoti et al., 2013). GA was used to build up a polynomial function based on massive astrophysical data, which was used to classify astronomical objects. The coefficients of the polynomial function were coded as the chromosomes. Only the population initialization and the fitness calculation were executed on GPU, because these two phases took major execution time. Their experiments were conducted on GeForce GT540M, a very early version of GPU card produced around 2011 with the compute capability 2.1.

No details about kernel implementations were given in the paper (Cavuoti et al., 2013), neither for kernel execution configuration. Therefore, we cannot say much about the efficiency of their implementation. As we mentioned before, all phases of GA can be parallelized with GPU computation, so as to eliminate the need of the data transfer between CPU memory and GPU memory. For a large scale application, the frequent data movement between CPU and GPU memories may cost a lot.

Zhang, Qiu, Li, and Liu (2014) and Chen, Chen, Liu, and Zhang (2015) reported their works on accelerating GA for graph coloring problem. They took a naive approach to parallelizing GA on GPU: each thread did a sequential job independently, from the initialization to crossover, mutation and evaluation. Several small cases were tested on NVIDIA GeForce GTX 480 with capability 2.0, an early version of GPU produced in 2010.

4.2. Island model on GPU

4.2.1. Migration topologies

The basic idea of the island model introduces geographic population distribution by dividing the global population into several sub-populations in order to keep the genetic diversity of population. Each sub-population is an isolated breeding unit to evolve species locally. A new genetic operator is introduced in island model: the migration operator: individuals from one sub-population migrate into another sub-population. Several new factors are introduced to control the island model (Izzo, Rucinski & Biscani, 2012): 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 Fig. 4.1.

4.2.2. Real-coded GA for unconstrained optimization problems

Luong et al. reported their works of the real-coded GA on GPU, implemented as Island model, to solve the unconstrained optimization problems (Luong, Melab, & Talbi, 2010). The basic idea in their scheme is that each thread block was treated as an island, and each chromosome was handled by one thread during genetic operations of selection, crossover, mutation, and evaluation. Therefore, it belongs to the category of the granularity in the chromosome level.

The same evolutionary parameters were used for all islands (all blocks) in their implementation. It would be better to use different parameters in order to set up different environment for each island during the evolutionary process.

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

Authors also described a second implementation staging through the shared memory: all local sub-population of each island stay in the shared memory, and all genetic operations were performed on this kind of user-managed cache to avoid frequently access the global memory. Actually, by using the shared memory, we can make kernels accessing the global memory in a coalesced way, which will significantly improve the bandwidth. No details on how kernels work on the shared memory were described. One major limitation of the way staging through the

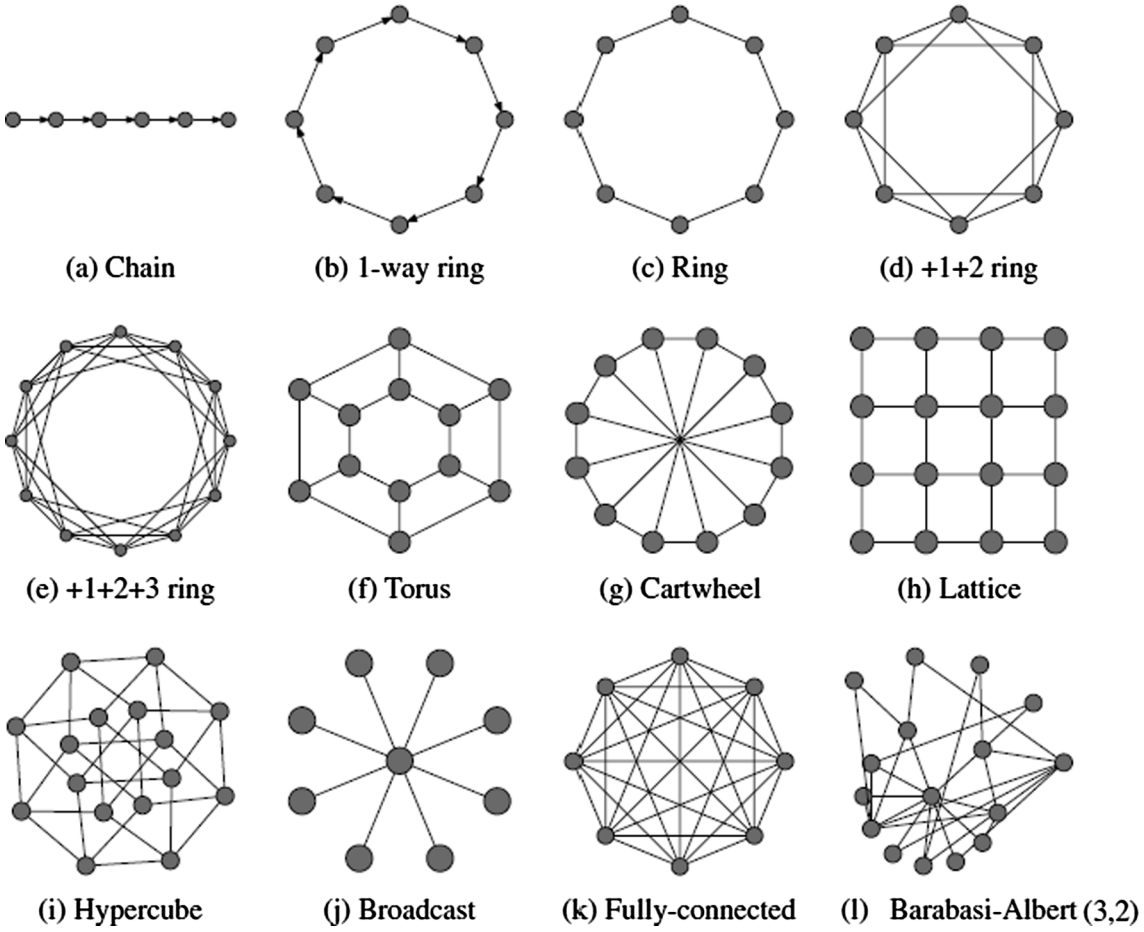

Fig. 4.1. Common migration topologies (from Izzo et al., 2012).

shared memory is the size of the memory which make only the case with a very small population possible to run this scheme.

The migration topology can be either ring or 2D toroidal grid, depending on how the execution configuration of kernels were organized: in 1D grid or 2D grid. No much details were given in this short paper.

Their experiments were conducted on GeForce GTX 280, a very early generation of GPU card produced around 2009 by NVIDIA, with the compute capability 1.3. Weierstrass-Mandelbrot functions was used to test their implementation (Lutton & Vehel, 1998).

4.2.3. Binary GA for 0/1 knapsack problem

Jaros described an implementation of binary GA on a multi-GPU cluster to solve single-objective 0/1 knapsack problem (Jaros, 2012). The island model was employed, where each GPU evolves a single island. The individuals were processed by a warp, therefore, it belongs to category of the granularity in the gene level. The population of a single island on one GPU evolved using a steady-state approach with elitism, uniform crossover, bit flip mutation, tournament selection and replacement. Migration of individuals occurred after a predefined number of generations exchanging the best local solution and an optional number of randomly selected individuals. The tournament selection was used to pick emigrants and incorporate immigrants. The unidirectional ring was adopted as the migration topology where only adjacent nodes can exchange individuals. All the data exchanges among GPUs were implemented by means of MPI (OpenMPI, 2018).

Since it was designed for 0/1 Knapsack problem (Rashid, Novoa, & Qasem, 2010), the binary encoding was used, and 32 items were packed into a single integer. Given the chromosomes size L, $L/32$ integers were used to represent one chromosome, this will help to reduce memory usage. The population was implemented as the struct of arrays: the first array containing all chromosome, and the second array containing fitness value associated with each chromosome.

The thread block was organized in 2D layout: the fast dimension x corresponding to the genes within a chromosome while the slow dimension y corresponding to different chromosomes as outlined in Fig. 4.2. The grid was also organized in 2D layout: the size of the fast dimension set to 1 and the size of the slow dimension set to the offspring population size divided by the double of the y of the block size, because two offspring were produced at once. Since the x dimension of the grid was set to 1, the warps may process the individuals in multiple rounds.

The binary tournament over the parents and offspring was adopted to create the new parent population. The kernel execution configuration was nearly the same layout as other kernels.

Their experiments run on a two-node cluster, each equipped with 7 GeForce GTX 580 cards, an early generation of GPU card produced around 2010 by NVIDIA, with the compute capability 2.1. A knapsack instance with 10,000 items was chosen as a test case to simulate the real-world problem with a reasonable large data set.

4.3. Cellular model on GPU

4.3.1. Cellular model

A cellular model is a kind of algorithm that the genetic search works on a structured population, and individuals cannot mate arbitrarily, can only compete and mate with its neighbors (Alba & Dorronsoro, 2008). The model has only one population imposed with a spatial structure, defined as a connected graph. A common topology is a 2D toroidal mesh as shown in Fig. 4.3, that limits the interactions between individuals. The neighborhood of a particular point of the mesh where an

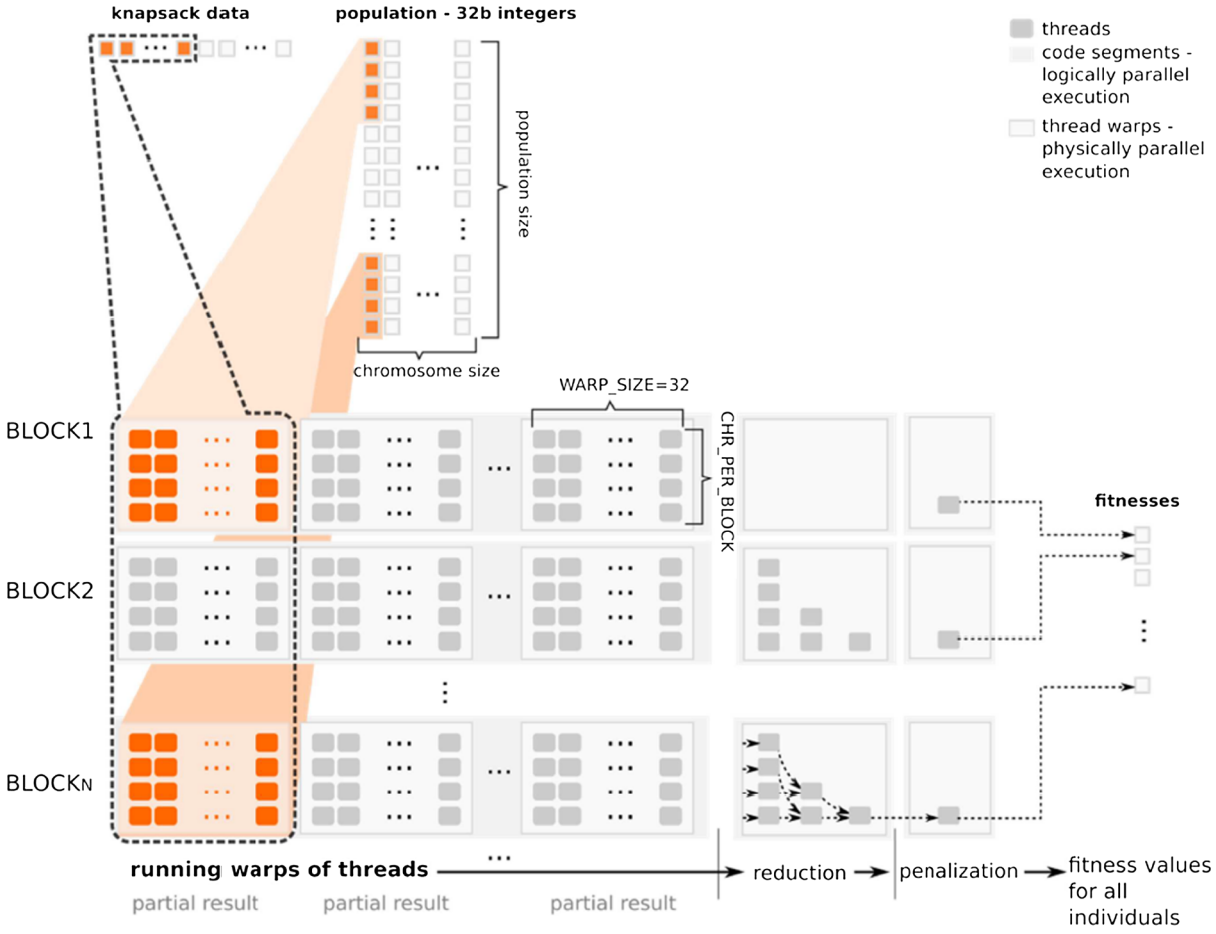

Fig. 4.2. Kernel Execution Configuration (from Jaros, 2012).

individual is placed is defined in terms of the Manhattan distance from it to others in the population. Each point of the mesh has a neighborhood that overlaps the neighborhoods of nearby individuals. In the basic algorithm, all the neighborhoods have the same size and identical shapes. The two most 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 Fig. 4.3.

The overlap of the neighborhoods provides an implicit mechanism of solution migration to the cellular model. Since the best solutions spread smoothly through the entire population, genetic diversity in the population is preserved longer than in a non-structured population, which provides a good tradeoff between exploration and exploitation

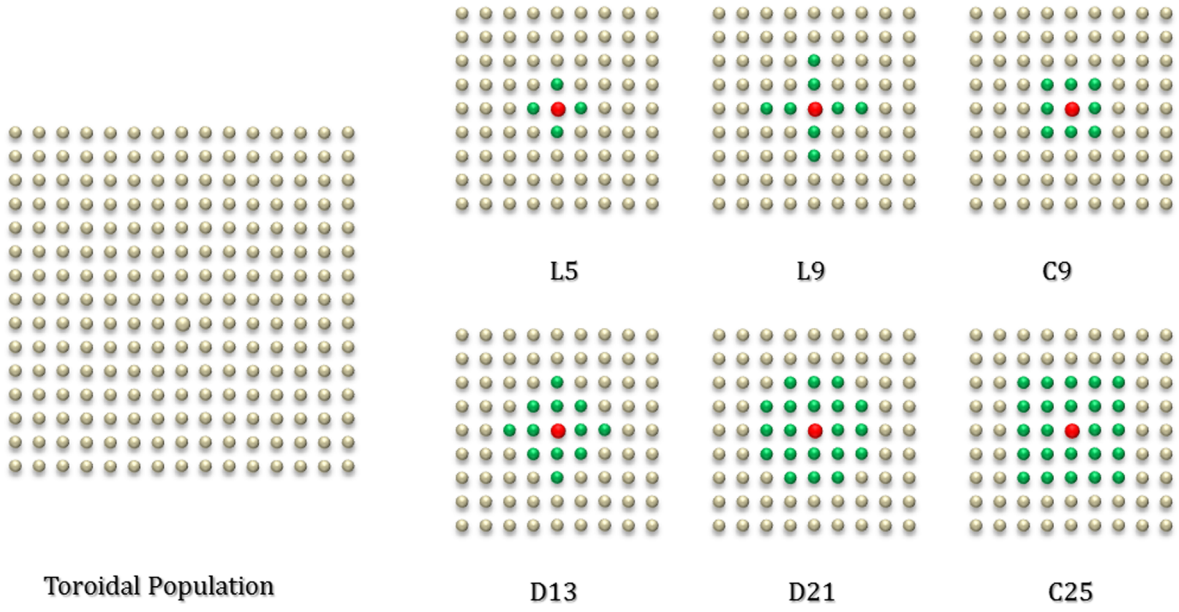

Fig. 4.3. The toroidal population (left) and most typically used neighborhoods. L: linear C: compact D: diamond.

during the evolutionary process. The tradeoff could be tuned by modifying the size of the neighborhood used, as the overlap degree between the neighborhoods grows according to the size of the neighborhood.

4.3.2. Real-coded GA on multi-GPUs for discrete optimization problems

Vidal and Alba described an implementation of real-coded GA on a multi-GPU cluster to solve continuous and discrete optimization problems (Vidal & Alba, 2010). The basic idea of their implementation was that each GPU thread was responsible for each individual of a 2D grid, by using a direct mapping between the population structure and the grid of thread blocks. Each GPU thread execute all the genetic operations over an individual. Therefore, the granularity belongs to the category in chromosome level. A potential issue is that the way cannot take the benefit of the coalesced memory access, if the data were stored in a chromosome-based layout in the global memory, which will harm the performance.

When implementing the Cellular Model on multiple GPUs, the whole population was divided into sub-populations, each on one GPU. At each step of evolutionary computation, partial data were exchanged among GPUs, as they were needed by adjacent GPUs as the corresponding neighbor individuals.

Their experiments were conducted on NVIDIA GTX 285, an early generation of GPU card produced around 2009 by NVIDIA, with the compute capability 1.3. Authors tested three discrete optimization problems: Colville Minimization, Error Correcting Codes Design Problem (ECC) and Massively Multimodal Deceptive Problem (MMDP), and three continuous ones, Shifted Griewank function, Shifted Rastrigin function and Shifted Rosenbrock function, on two GTX 285 cards.

4.3.3. Integer-coded GA for quadratic assignment problem

Cárdenas et al. implemented the integer-coded GA based on the Cellular Model to solve Quadratic Assignment Problem (QAP) (Cárdenas, Poveda, & García, 2017). The problem consists of finding the optimal assignment of n facilities to n locations, knowing the distances between facilities and the flow between locations. The problem is one of the fundamental combinatorial optimization problems in the operations research. Many other combinatorial optimization problems may be written in this form (Burkard, 2013). The problem is NP-hard, so there is no known algorithm for solving this problem in the polynomial time, GAs are the robust and flexible alternative to solve the complex optimization problems.

The population was imposed with a toroidal mesh structure, and four different definitions of neighborhoods, L5, C9, D17, and C25 (where, L-linear, D-diamond, C-compact) were mentioned, as shown in Fig. 4.4.

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

The crossover kernel implemented the method of *Modified Order Crossover* (MOX). The first parent was decided by the block index, and the second parent was selected according to C9 neighborhood topology. MOX works as follows: a common crossing point was selected for both parents, the genes of the first parent were stored to the left side of the offspring, and remaining genes were copied in the order as they were in the second parent to obtain one offspring (Wroblewski, 1996). The mutation kernel implemented the method of exchanging random selected genes to produce an offspring.

Additional two non-standard genetic operators were also applied: the transposition kernel and 2-opt kernel. In the transposition kernel, a portion of genes two points randomly generated in a chromosome were reversed to obtain a new offspring, much like a kind of mutation operator. The 2-opt kernel mimicked the 2-opt heuristic, a simple local search method to produce one offspring, also much like a kind of mutation operator.

Their experiments were conducted on Geforce GTX 760M, a GPU card produced around 2013 with the compute capability 3.0. Eight small cases were tested, size of instance was varied from 19 to 32.

4.3.4. Integer-coded GA for independent task scheduling problem

Pinel et al. implemented an integer-coded GA based on Cellular Model to solve a kind of independent tasks schedule problem (Pinel, Dorronsor, & Bouvry, 2013). It 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 this problem is known to be NP-complete. In scheduling computational tasks problems, we usually have a limited amount of time to find the best possible schedule of the tasks. Therefore, there is a need for algorithms that are able to find highly accurate solutions within this important time constraint.

The population is arranged into a 2D toroidal grid. The use of cellular populations in metaheuristics allows for a better exploration of the search space with respect to the equivalent one with other panmictic and decentralized populations. An array of integer numbers was used to present the solution: the index of the array corresponds to a task, and the integer of the array for a given index corresponds to the machine to which this task (denoted by the index of the array) is assigned.

Authors proposed a new crossover operator for their cellular GA implementation, called *Uniform Proportional Recombination* (UPR). Fig. 4.5 shows how this operator update each task of a solution.

The circled task of the center solution is the task being updated. Its neighborhood is defined by L5, the task above, below, right, and left. The offspring solution is generated by assigning to each task, the machine of one of the neighboring solutions, according to a proportionate selection mechanism. Authors defined two different criteria for choose the winner. The crossover operator was implemented with two kernels: one kernel was used to computes the probability for each solution in the neighborhood to be chosen under fitness proportionality. The other kernel was used to update task to generate offspring. The crossover operator was run in a way that one thread per task of a solution. Therefore, its granularity belongs to the category of the grain size in the gene level. The other kernels, mutate, fitness, and replace are launched

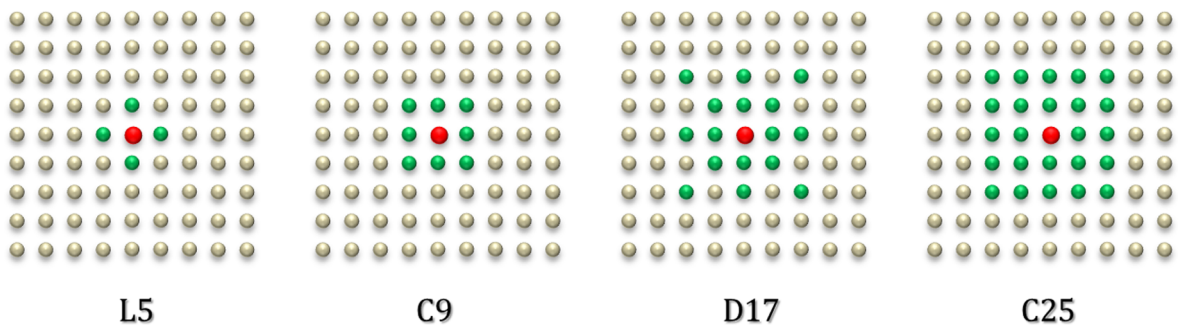

Fig. 4.4. Neighborhood Topologies.

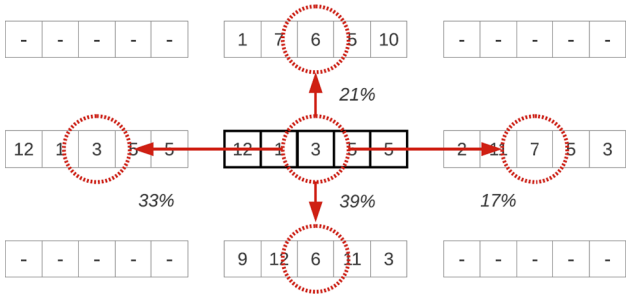

Fig. 4.5. Illustration of crossover operator (from Pinel, Dorronsoro, and Bouvry (2012)).

with one thread per solution, that belongs to the granularity in the chromosome level. These kernels were implemented as standard operators in GAs.

Their experiments were conducted on Tesla C2050, a GPU card produced around 2011 with the compute capability 2.0. Several instances were tested ranging from 512 tasks over 16 machines, to 65,536 tasks over 2048 machines.

4.3.5. *Integer-coded GA for dependent task scheduling problem*

Zhao et al. implemented an integer-coded GA based on the Cellular model to solve a kind of dependent tasks schedule problem (Zhao, Chen, Xie, Zhao & Ding, 2018). It is a kind of *Machine Scheduling Problem* (MSP): assigning a set of computational tasks under precedence constraints among tasks onto the different processors in a heterogeneous cluster. Due to its NP-hard nature, scheduling is always a challenging problem, and has been extensively studied in the past decades.

The population was imposed with 2D toroidal mesh structure, and L5 were used as neighborhoods. The chromosome consists of two substrings: one substring represents the machine allocation, one substring represents a task queue, introduced by Gupta, Agarwal and Kumar (2010). The crossover kernel was implemented with two separated parts: a classical two-cut points crossover method was applied to the machine allocation part of the chromosome, and a modified two-points crossover method was applied to the task queue part, avoiding illegal offspring due to the precedence restriction among tasks. The mutation kernel was also implemented with two separated parts: a classical single-point mutation method was applied to the machine allocation part of the chromosome, a kind of single-point re-ordering method, given by (Xu, Li, Hu, & Li, 2014), was applied to the task queue part. An evaluation kernel was implemented to calculate makespan.

All kernels were executed in the configuration of 1D grid and 1D block, the size of the block corresponds to the number of tasks, and the size of the grid corresponds to the number of the population. Since each GPU block corresponds to a chromosome, essentially, its granularity belongs to the category of the grain size in the gene level.

Their experiments were conducted on Geforce GTX 560Ti, a GPU card produced around 2011 with the compute capability 3.0. Several cases, task number from 128 to 1024, were tested.

5. Discussion and conclusion

In this paper, a comprehensive survey has been presented on how to accelerate GAs with GPU computing. Selective literatures after 2010 have been discussed from the perspective of GPU architecture. A concise overview of traditional approaches on parallelizing GAs on CPU has been briefly introduced, and the major difference has been discussed between parallelizing GAs on CPU and on GPU. Conceptually, parallel GAs on CPU run multiple instances concurrently across processors or computers, and each instance is a sequential algorithm in nature, therefore, CPU approach on parallel GAs is not suitable for the implementation on GPUs.

When designing a parallel algorithm for CPU architectures, we typically consider how to distribute data across tens of CPU threads. When designing a parallel algorithm for GPU architectures, we have more than hundreds of thousands of GPU threads run simultaneously, therefore, we should rethink and redesign the parallel mechanism to fully exploit the computing power of GPU architectures.

The major difference between parallelizing GAs on CPU and on GPU is that CUDA architectural features are exposed directly to us for programming. This enables us to have more control over the massively parallel threads computing. Top three principles on optimizing GPU implementation, listed in order of importance, are (1) exposing sufficient parallelism, (2) optimizing memory access, and (3) optimizing instruction execution.

Most published works on parallel GAs on GPU architectures are conducted before 2013 on the earlier generations of NVIDIA GeForce GPU cards. NVIDIA GeForce products target the gaming market while NVIDIA Tesla products target the high-performance computing market. Tesla GPUs outperform GeForce GPUs in both peak throughput and peak memory bandwidth. There are six generations of NVIDIA Tesla architectures since 2008: Tesla, Fermi, Kepler, Maxwell, Pascal, and Volta. The average lifespan of a generation of GPU architectures is about two years. Latest version GPU, powered by ground-breaking technologies and multiple revolutionary features, brings extraordinary speed and scalability for hyper-scale computing. The current mainstream GPUs used in industry and academia are Tesla V100 and Tesla P100, which deliver a unified platform for accelerating both HPC and AI works, dramatically increasing throughput while also reducing costs. Therefore, it is a better choice to conduct the research works on accelerating GAs at the latest Tesla architecture.

Most published works on parallel GAs on GPU tackled small scale problems. To handling large scale problems, an efficient implementation of parallel GAs on multiple GPUs becomes imperative. A promising approach to parallelizing GAs on multiple GPUs is to hybrid Island and Cellular Model. The earliest effort to combine these two models was given in (Gruau, 1994), which was implemented in CPU architecture to evolve and train a novel neural network. Dorronsoro et al. their works of applying this hybrid model to solve a very large scale of the capacitated vehicle routing problem, executed on a grid composed of up to 125 heterogeneous CPU nodes (Dorronsoro, Arias, Luna, Nebro, & Alba, 2007).

It is worthy of further consideration to implement this hybrid model over multiple GPUs: each GPU is an island, a cellular model is implemented within each GPU, a certain migration topology can be defined among GPUs, and migration among GPUs conducts at regular intervals, as shown in Fig. 5.1.

This hybrid model combines both merits of Island and Cellular Model: keeping the genetic diversity of population among multiple islands and improving local search ability on a structured sub-population within each island. Comparing with a pure island model over multiple GPUs, we have more chance to exploit neighborhood of each individual; while comparing with a pure cellular model over multiple GPUs, we have more chance to reserve genetic diversity of population and less data exchanges among GPUs. Deploying parallel GAs over multiple GPUs enables us to tackle real world lager scale problems.

Many industries and research fields can benefit from the efficient implementation of parallel GAs for hyper-scale computing. One of the promising area is to evolve better deep neural network with GAs. *Deep Neural Networks* (DNNs) are currently widely used for many artificial intelligence (AI) applications including computer vision, speech recognition, and robotics, which deliver state-of-the-art accuracy on many AI tasks. DNNs are typically trained via gradient-based learning algorithms. The Problem with DNNs is what is called *hyper-parameters*, the very values required by the DNN to perform properly, and the only values that cannot be learned and must be pre-determined subjectively. Genetic algorithms, with its family of evolutionary algorithms, might be

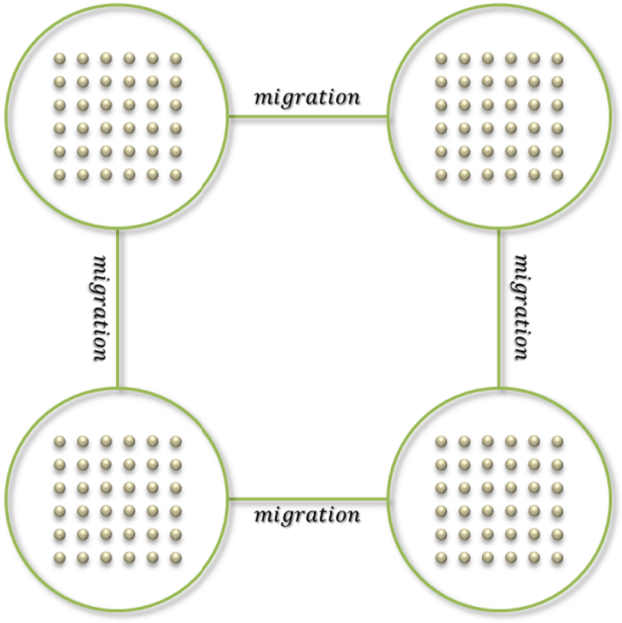

Fig. 5.1. A Hybrid model over Multiple GPUs: each GPU is an island, a Cellular model is implemented within each GPU, and a 2D torus migration topology is defined among GPUs.

able to evolve the best structure for a network intended for training with stochastic gradient descent. There is a special term for this study for years, *neuroevolution*, a good survey paper on this topic given by Yao (1999).

Evolutionary algorithms can be introduced into DNN at many different levels.

The evolution of connection weights provides a global approach to connection weight training, especially when gradient information of the error function is difficult to obtain. Evolution can be used to find a better network architecture automatically, which has several advantages over heuristic methods of architecture design. Recently the idea of architecture search through neuroevolution is attracting a number of researchers since 2016. Now neuroevolution is making a comeback. GPU-accelerated computing power will make this happen certainly.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 61572100 and in part by the Grant-in-Aid for Scientific Research (C) of the Japan Society of Promotion of Science under Grant 15K00357.

References

Arora, R., Tulshyan, R., & Deb, K. (2010). Parallelization of binary and real-coded genetic algorithms on GPU using CUDA. *Proceedings of IEEE Congress on Evolutionary Computation* (pp. 8). .

Alba, E., & Dorronsoro, B. (2008). *Cellular genetic algorithms.* Heidelberg: Springer.

Burkard, Rainer E. (2013). Quadratic assignment problems. In P. M. Pardalos (Ed.). *Handbook of combinatorial optimization* (pp. 2741–2814). (2). New York: Springer.

Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs Parallèles, Réseaux et Systèmes Répartis, 10*(2), 141–171.

Cárdenas, G. E. R., Poveda, Ch, & García, O. H. (2017). A solution for the quadratic assignment problem (QAP) through a parallel genetic algorithm based grid on GPU. *Applied Mathematical Sciences, 11*(57), 2843–2854.

Cavuoti, S., Garofalo, M., Brescia, M., Pescape', A., Longo, G., Ventre, G. (2013). Genetic algorithm modeling with GPU parallel computing technology. In Apolloni, B. et al. editors. Neural Nets and Surroundings, SIST 19, pp. 29–39.

Chen, B., Chen, B., Liu, H., & Zhang, X. (2015). *A fast parallel genetic algorithm for graph coloring problem based on CUDA. International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*.

Cheng, R., Gen, M., & Tsujimura, Y. (1996). A tutorial survey of job-shop scheduling problems using genetic algorithms, Part I. Representation. *Computers & Industrial Engineering, 30*(4), 983–997.

Cheng, R., Gen, M., & Tsujimura, Y. (1999). A tutorial survey of job-shop scheduling problems using genetic algorithms, Part II: Hybrid genetic search strategies. *Computers & Industrial Engineering, 36*(2), 343–364.

Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional CUDA C programming.* NJ: John Wiley & Sons.

Chipperfield, A. J., & Fleming, P. J. (1994). Parallel Genetic Algorithms: A Survey. Research Report. ACSE Research Report 518. Department of Automatic Control and Systems Engineering.

CUB (2018). CUB v1.8.0. (Accessed July 25, 2018) https://nvlabs.github.io/cub/.

CUDA (2018). (Accessed July 25, 2018) https://en.wikipedia.org/wiki/CUDA#Version_features_and_ specifications.

Debattisti, S., Marlat, N., Mussi, L., & Cagnoni, S. (2009). *Implementation of a simple genetic algorithm within the CUDA architecture. Proceedings of the GECCO 2009 Workshop on Computational Intelligence on Consumer Games and Graphic Hardware*.

Domingos, Pedro (2015). *The master algorithm: How the quest for the ultimate learning machine will remake our world.* New York: Basic Books.

Dorronsoro, B., Arias, D., Luna, F., Nebro, A., Alba, E. (2007). A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP. In: Smari, W. W. editor, High performance computing & simulation conference (pp. 759–765).

Du, Y., Ma, Q., Sakamoto, M., Furutani, H., & Zhang, Y. (2014). Runtime analysis of onemax problem in genetic algorithm. *Journal of Robotics, Networking and Artificial Life, 1*(3), 225–230.

Gen, M., & Cheng, R. (1997). *Genetic algorithms and engineering design.* New York: John Wiley & Sons.

Gen, M., & Cheng, R. (2000). *Genetic algorithms and engineering optimization.* New York: John Wiley & Sons.

Gen, M., Cheng, R., & Lin, Lin (2008). *Network models and optimization: multi-objective genetic algorithms approach.* New York: Springer.

Gruau, F. (1994). Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. Unpublished doctoral dissertation, L'Universite Claude Bernard-Lyon I.

Gupta, S., Agarwal, G., & Kumar, V. (2010). Task scheduling in multiprocessor system using genetic algorithm. *Proceedings of second international conference on machine learning and computing* (pp. 267–271). .

Luong, T. V., Melab, N., & Talbi, E. (2010). *GPU-based island model for evolutionary algorithms. Proceedings of the Genetic and Evolutionary Computation Conference*.

Lutton, E., & Vehel, J. L. (1998). Holder functions and deception of genetic algorithms. *IEEE Transactions on Evolutionary Computation, 2*(2), 56–71.

Izzo, D., Rucínski, M., & Biscani, F. (2012). The generalized island model. In F. F. de Vega (Ed.). *Parallel architectures & bioinspired algorithms* (pp. 151–169). SCI.

Jaros, J. (2012). Multi-GPU island-based genetic algorithm for solving the knapsack problem. *WCCI 2012 IEEE world congress on computational intelligence* (pp. 1–8). .

Gordon, V. S., & Whitley, D. (1993). Serial and parallel genetic algorithms as function optimizers. *Proceedings of the fifth international conference on genetic algorithms* (pp. 177–183). .

Knysh, D. S., & Kureichik, V. M. (2010). Parallel genetic algorithms: a survey and problem state-of-the-art. *Journal of Computer and Systems Sciences International, 49*(4), 579–589.

Mühlenbein, H., & Chakraborty, U. K. (1997). Gene pool recombination, genetic algorithm, and the Onemax function. *Journal of Computing and Information Technology. 5*, 167–182.

Munawar, A., Wahib, M., Munetomo, M., & Akama, K. (2008). A survey: Genetic algorithms and the fast evolving world of parallel computing. *Proceedings of the 10th IEEE international conference on high performance computing and communications* (pp. 897–902). .

Oiso, M., Matsumura, Y., Yasuda, T., & Ohkura, K. (2011). Implementation genetic algorithms to CUDA environment using data parallelization. *The Journal Tehnički vjesnik, 18*(4), 511–517.

Open MPI, Indiana University (2018). OpenMPI: Open Source High Performance Computing. (Accessed July 25, 2018) http://www.open-mpi.org/.

Pedemonte, M., Alba, E., & Luna, F. (2011). Bitwise operations for GPU implementation of genetic algorithms. *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation* (pp. 439–446). .

Pinel, F., Dorronsoro, B., & Bouvry, P. (2012). Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel and Distributed Computing, 16*, 1–3.

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

Rashid, H., Novoa, C., & Qasem, A. (2010). An Evaluation of Parallel Knapsack Algorithms on Multicore Architectures, CSC.

Umbarkar, A. J., & Joshi, M. S. (2013). Review of parallel genetic algorithm based on computing paradigm and diversity in search space. *ICTACT Journal on Soft Computing, 3*(4), 615–622.

Vidal, P., & Alba, E. (2010). A multi-GPU implementation of a cellular genetic algorithm. *IEEE Congress on Evolutionary Computation,* 1–7.

Wroblewski, J. (1996). Theoretical foundations of order-based genetic algorithms. *Fundamenta Informaticae, 28*(3–4), 423–430.

Xu, Y., Li, K., Hu, J., & Li, K. (2014). A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences, 270*, 255–287.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE, 87*(9), 1423–1447.

Zhang, K., Qiu, M., Li, L., & Liu, X. (2014). Accelerating genetic algorithm for solving graph coloring problem based on CUDA architecture. In L. Pan, G. Păun, M. J. Pérez-

Jiménez, & T. Song (Eds.). *Bio-inspired computing - theories and applications. Communications in computer and information science* (pp. 578–584). Berlin, Heidelberg: Springer.

Zhao, Y., Chen, L., Xie, G., Zhao, J., & Ding, J. (2018). GPU implementation of a cellular genetic algorithm for scheduling dependent tasks of physical system simulation programs. *Journal of Combinatorial Optimization, 35*(1), 293–317.

Zheng, L., Lu, Y., Ding, M., Shen, Y., Guo, M., & Guo, S. (2011). Architecture-based performance evaluation of genetic algorithms on Multi/Many-core Systems. *Proceedings of 14th IEEE international conference on computational science and engineering* (pp. 321–334). .