Survey of Parallel Distributed Algorithms on CUDA-based Platform

Kasidit Wijitsopon

Department of Computer Science, Faculty of Science, Khon Kaen University

Maung, Khon Kaen, Thailand, 40002

I. INTRODUCTION

II. LITERATURE REVIEW

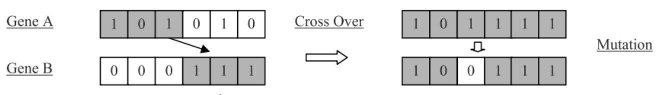
Any Colony Optimization (ACO) [1]

ACO มีลักษณะการหาอาหารของมด ซึ่งเป็นวิวัฒนาการของพฤติกรรมทางสังคม (Social Behavior) แทนที่จะเป็นทางด้านพันธุกรรม นโดยถูกคิดค ACO Dorigo และคณะ [1] ซึ่งเลียนแบบพฤติกรรมการหาอาหารของมดโดยค้นการหาเส้นทางที่สั้นที่สุดระหว่างรังกับแหล่งอาหาร โดยใช้ฟีโรโมน onPherom)) ที่มดวางไว้ระหว่างทางเพื่อใช้ในการสื่อสารทางอ้อมกับมดตัวอื่น ในฝูง ในระหว่างการเดินทางหากเจอสิ่งกีดขวางมดแต่ละตัวจะตัดสินใจเลือกเส้นทางเลี่ยงอย่างสุ่ม สมมติว่ามีสองเส้นทางที่เลี่ยงได้ในช่วงแรกปริมาณการ ระเหยของ Pheromone บนสองเส้นทางจะมีปริมาณที่เท่ากัน แต่เมื่อเวลาผ่านไป เส้นทางที่สั้นกว่าจะมีปริมาณ การระเหยของฟีโรโมนที่มากกว่า เพราะมดจะเลือกเส้นทางที่มีปริมาณ Pheromone มากกว่าหรือเส้นทางที่สั้นที่สุด

Genetic Algorithm [1]

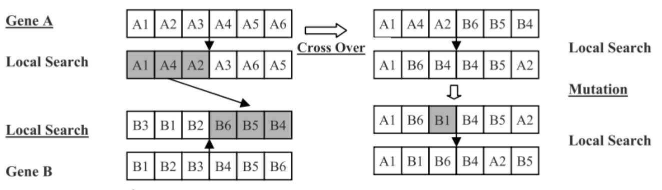
เป็นอัลกอริธึมด้านวิวัฒนาการแบบแรก และถูกคิดค้นโดย Holland [2] ซึ่งได้รับ แรงบันดาลใจมาจากทฤษฎีของ Darwin เกี่ยวกับการวิวัฒนาการอัลกอริธึม GA นี้ได้รับความนิยมอย่างมากในแวดวงปัญญาประดิษฐ์ (Artificial Intelligence) และได้ถูกประยุกต์ใช้กับปัญหาหลายแขนง เช่น ในวิทยาศาสตร์และวิศวกรรมศาสตร์

ขั้นตอนการทำงานของ GA เริ่มด้วย การสร้างประชากรของคำตอบหรือโครโมโซมขึ้นมา แล้วคำนวณหาค่าฟังก์ชัน ความเหมาะสม (Fitness Function) ของประชากรแต่ละตัวซึ่งเป็นขั้นตอนการถอดรหัสโครโมโซม เพื่อคำนวณ หาค่าความเหมาะสมตามฟังก์ชันเป้าหมายหลังจากนั้นจะเลียนแบบพฤตกรรมทางพันธุกรรมในธรรมชาติดังนั้นโครโมโซมที่มีค่าความเหมาะสมสูงจะแลกเปลี่ยนข้อมูลซึ่งกันและกันเพื่อสร้าง โครโมโซมใหม่ที่พัฒนาตัวเองผ่านกระบวนการสลับสายพันธุ์ (Crossover) และการกลายพันธุ์ (Mutation) ดังแสดง ในรูปที่ 1 โครโมโซมลูกหลาน (Offspring Chromosome) จะถูกตรวจสอบ ว่าให้คำตอบที่ดีกว่าโครโมโซมตัวที่แย่ที่สุดในประชากรหรือไม่ ถ้าดีกว่ามันจะแทนที่โครโมโซมตัว ที่แย่ที่สุด ขั้นตอนการค้นหาดังกล่าวจะถูกทำซ้ำไปซ้ำมา (Iterative Process) จนกระทั่งตรงกับ เงื่อนไขการหยุดการทำงานเพื่อให้ได้โครโมโซมตัวที่ให้ค่าความเหมาะสมที่ดีที่สุด หรือผลลัพธ์ใกล้เคียงกับค่าที่ดีที่สุด (Optimum Solution)



รูปที่ 1 ตัวอย่างการ Crossover และ Mutation

Fig. 1. Sample of Crossover and Mutation



รูปที่ 2 ตัวอย่างการ Crossover และ Mutation แบบมี Local Search

Fig. 2. Local Search of Crossover and Mutation

Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously,[5] operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level , data, and task parallelism.

Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling.[6] As power consumption (and consequently heat generation) by computers has become a concern in recent years,[7] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.[8]

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and gridsuse multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones,[9] because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common.

Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance. The maximum possible speed-up of a program as a result of parallelization is known as Amdahl's law

CUDA Parallel Computing Platform [10]

NVIDIA® CUDA™ technology leverages the massively parallel processing power of NVIDIA GPUs. The CUDA architecture is a revolutionary parallel computing architecture that delivers the performance of NVIDIA's world-renowned graphics processor technology to general purpose GPU Computing. Applications that run on the CUDA architecture can take advantage of an installed base of over one hundred million CUDA-enabled GPUs in desktop and notebook computers, professional workstations, and supercomputer clusters.

With the CUDA architecture and tools, developers are achieving dramatic speedups in fields such as medical imaging and natural resource exploration, and creating breakthrough applications in areas such as image recognition and real-time HD video playback and encoding.

CUDA enables this unprecedented performance via standard APIs such as the soon to be released OpenCL™ and DirectX® Compute, and high level programming languages such as C/C++, Fortran, Java, Python, and the Microsoft .NET Framework.

The CUDA Architecture consists of several components, in the green boxes

1. Parallel compute engines inside NVIDIA GPUs
2. OS kernel-level support for hardware initialization, configuration, etc.
3. User-mode driver, which provides a device-level API for developers
4. PTX instruction set architecture (ISA) for parallel computing kernels and functions

The CUDA Architecture consists of several components, in the green boxes below:

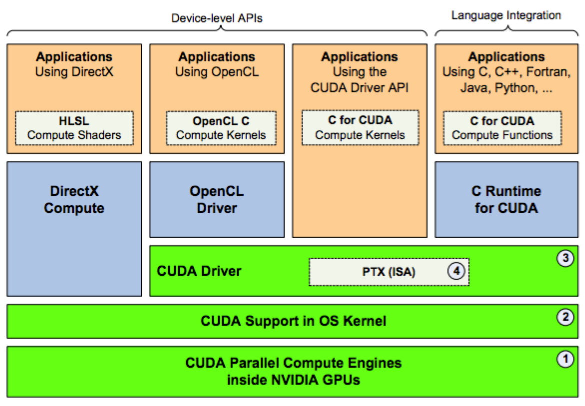

Device-level APIs
Language Integration
Applications Using DirectX
HLSL Compute Shaders
Applications Using OpenCL
OpenCL C Compute Kernels
Applications Using the CUDA Driver API
C for CUDA Compute Kernels
Applications Using C, C++, Fortran, Java, Python, ...
C for CUDA Compute Functions
DirectX Compute
OpenCL Driver
C Runtime for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines inside NVIDIA GPUs
1

Fig.3. CUDA Architecture

Travelling salesman problem

The travelling salesman problem (TSP) or travelling salesperson problem asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult,[11] a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder. TSP is a special case of the travelling purchaser problem.

In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether any tour is shorter than L) belongs to the class of NP-complete problems. Thus, it is likely that the worst-case running time for any algorithm for the TSP increases exponentially with the number of cities.

III. ALGORITHMS

a. Algorithm inspired by Ant

A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-In-Roulette Selection [12]

This paper presents and implements a parallel MAX-MIN Ant System (MMAS) based on a GPU+CPU hardware platform under the MATLAB environment with Jacket toolbox to solve Traveling Salesman Problem (TSP). The key idea is to let all ants share only one pseudorandom number matrix, one pheromone matrix, one taboo matrix, and one probability matrix. We also use a new selection approach based on those matrices, named AIR (All-In-Roulette). The main contribution of this paper is the description of how to design parallel MMAS based on those ideas and the comparison to the relevant sequential version. The computational results show that our parallel algorithm is much more efficient than the sequential version.

Fig.4 shows the experimental results when the matrix size is from 3×3 to 2500×2500

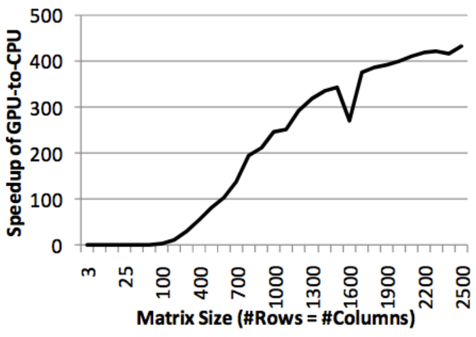


Fig.4. Speedup of GPU-to-CPU in multiplication between three matrices

Fig.5 gives the comparison of overall computation time between the GPU+CPU and CPU implementation of the MMAS algorithm for 9 TSP instances. Due to the tiny percentage of initialization, all the tests are limited to 50 iterations. The GPU can only work when the data is moved from CPU to it; the movement from GPU to CPU is also necessary. The absolute transfer time is linearly dependent on matrix size, and is with a small initialization penalty (less than a hundred microseconds).

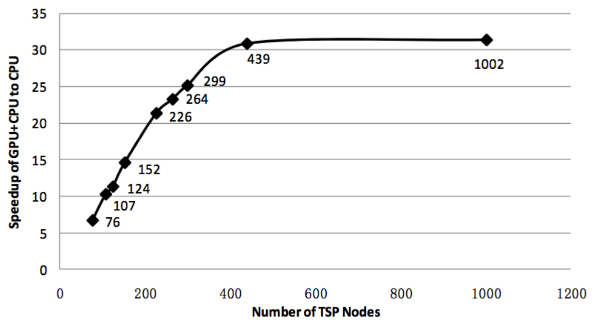


Fig.5. Overall Speedup of GPU+CPU to CPU. Running from pr76 to pr1002.

Fig.6 shows that within our parallel MMAS algorithm, the percentage of time-consuming in data transfer increases steadily with the growth of the number of TSP nodes. For matrix size in the range 0.25-1.25MB, the transfer rate reaches the peak, which is above 3.5GB/s

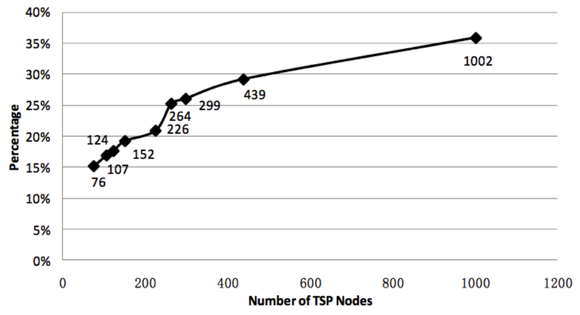


Fig.6. Percentage of time-consuming in data transfer within the parallel MMAS algorithm. Running from pr76 to pr1002.

The results indicate that, when the TSP nodes are approximately less than 450, the GPU-to-CPU speedup increases steadily with the increase of number of TSP nodes. After the number of nodes reaches 450, the growth of speedup slows down significantly.

Utilizing GPGPUs for Design Validation with a Modified Ant Colony Optimization [13]

This paper, them propose a novel parallel state justification tool, GACO, utilizing Ant Colony Optimization (ACO) on Graphical Processing Units (GPU). With the high degree of parallelism supported by the GPU, GACO is capable of launching a large number of artificial ants to search for the target state. A novel parallel simulation technique, utilizing partitioned navigation tracks as guides during the search, is proposed to achieve extremely high computation efficiency for state justification. Them present the results on a GPU platform from NVIDIA (a GeForce GTX 285 graphics card) that demonstrate a speedup of up to 228× compared to deterministic methods and a speedup of up to 40× over previous state-of-the-art heuristic based serial tools.

The technique is implemented on the NVIDIA GeForce GTX 285 with 30 cores and 8 SIMT execution pipelines per core. Them experimental results demonstrate that the proposed GACO can achieve between one and two orders of magnitude speedup in comparison with the state-of-the-art sequential ACO based state justification algorithm implemented on conventional processor architecture.

The pseudo-code for the algorithm is shown in Algorithm 1. Lines marked with * are executed on the CPU, others on the GPU.

```
Algorithm 1 Modified Ant Colony Optimization
 1: Initialize $Start\_state, Best\_fit$ *
 2: Initialize $trace$ *
 3: for all $N_{rounds}$ rounds * do
 4:   for all $N_{stride}$ strides do
 5:     for all $N_{ants}$ ants do
 6:       $W = Gen\_PI(Seed)$ // randomly generate PI
 7:       $V_{tmp} = Simulate(Start\_state, W)$
 8:       $Fit_{tmp} = Calc\_Fitness(V_{tmp})$
 9:       Add $V_{tmp}$ to $Trace_{tmp}$
10:       if $Fit_{tmp} > best\_fit$ then
11:         Update($Fit_{best}$, $V_{best}$)
12:       end if
13:     end for
14:   end for
15:   $Start\_State = V_{best}$ * // set guidepost for next round
16: end for
```

Fig.7. Modified Ant Colony Optimization

```
Algorithm 2 $eval\_kernel$ – evaluation kernel of OR2 gate
Input: $uint *t_{val}$, $uint *f_{val}$, $bool * flag$, $int\ id$,
$int\ in0$, $int\ in1$
 1: $tid = threadIdx.x$; // get thread index
 2: $val\_in0 = t_{val}[tid + in0 \times t]$;
 3: $val\_in1 = t_{val}[tid + in1 \times t]$;
 4: $f_{val}[tid + id \times t] = val\_in0 \mid val\_in1$;
```

Fig.8. Evaluation kernel of OR2 gate

They evaluated GACO on a set of ISCAS89 and ITC99 benchmark designs. The platform has them present consists of a workstation with Intel 8-core i7 3.33 GHz CPU, 2 GB memory and one NVIDIA GeForce GTX 285 graphic card which has 30 SMs and 240 processing core with the clock speed 1476 MHz. The on-chip memory contains 1 GB DDR3 global memory with a bandwidth of up-to 159.00 GB/s. The operating system on the host machine is a 64-bit Red Hat GNU/Linux distribution. The NVIDIA CUDA 2.3 SDK toolkit is used for all the program execution.

Implementation of Ant Colony Algorithm based on GPU [14]

In this paper, the ACS was applied to solve the typical combinatorial optimization problem TSP using modern GPU parallel calculation features.

The main idea of the novel implementation mapping MMAS to GUP is to make full use of parallel process feature and positive feedback mechanism of the MMAS algorithm. By applying the parallel processing techniques of fragment processor based on the SIMD, one can improve MMAS computing efficiency with GUP's accelerated features. Nowadays, GPU is still deficient on branch and iterative feedback processing, so the GPU computing implements were mainly focus on the calculation flow

They chose typical TSP example with 30 cities and 8 ants and realize MMAS to solve TSP with CPU and GPU respectively. The parameters of MMAS are listed below: City coordinates are shown in TABLE I

```
#define MAXCITY 30
int
xPos[MAXCITY]={41, 37, 54, 25, 7, 2, 68, 71, 54, 83, 64, 1
8, 22, 83, 91, 25, 24, 58, 71, 74, 87, 18, 13, 82, 62, 58, 45,
41, 44, 4};
int
yPos[MAXCITY]={ 94, 84, 67, 62, 64, 99, 58, 44, 62, 69, 6
0, 54, 60, 46, 38, 38, 42, 69, 71, 78, 76, 40, 40, 7, 32, 35, 2
1, 26, 35, 50};
```

TABLE I. City Coordinates

The results show that our implements can find the optimized path length that is 423.741. In order to test the GPU computing efficiency, we use different iteration counts in finding the shortest path respectively, see TABLE II.

Count of iteration	GPU costs (ms)	CPU costs (ms)
1000	2781	3390
2000	5218	6766
3000	7656	10141
5000	12500	16922
7000	17344	23656
10000	24531	33766

TABLE II. Comparision of time costs in solving MMAS

The comparison of running time cost between GPU and CPU is shown in Figure 3.

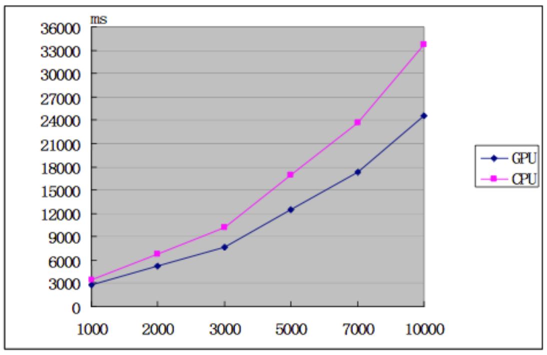


Fig.9. Comparision of computing cost.

The novel method realized fast solution of MMAS algorithm through constructing texture suitable to GUP processing. The experimental results show GPGPU has good accelerated performance on large scale computing. Because the GPU's SIMD features are designed for graphics pipeline, it is more difficult in realization than CPU. Our implement is suitable to GPGPU and can be used on solving other similar computing problem.

b. Algorithm inspired by Genetic Evolution

A Comparison of Many-threaded Differential Evolution and Genetic Algorithms on CUDA [15]

They have compared a many-threaded implementation of two nature inspired meta-heuristics, the GA and the DE. In contrast to previous GPU implementations of the GA and DE, the presented implementation processes each candidate solution with many threads and generates the random numbers on the GPU. This approach seeks to utilize the resources of the GPGPU as much as possible.

The implementations consists of a set of CUDA-C kernels for generation of initial population, generation of batches of pseudorandom numbers for decision making, merger of the old and new populations, the implementation of the operations specific for each meta-heuristic, and for evaluation of candidate solutions.

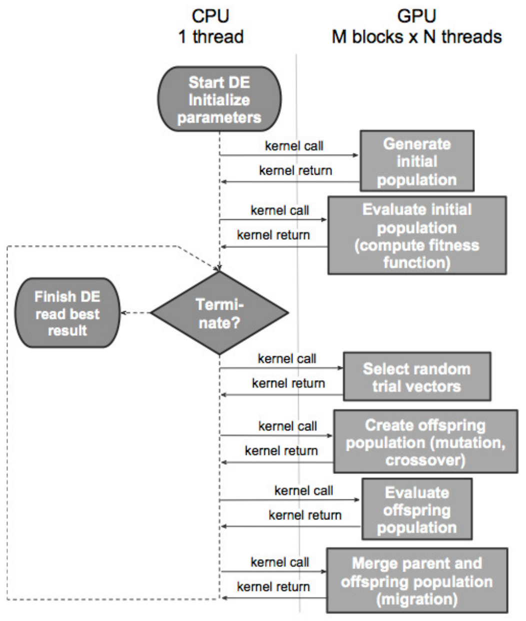

CPU
1 thread
GPU
M blocks x N threads
Start DE Initialize parameters
kernel call
kernel return
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish DE read best result
Termi-nate?
kernel call
kernel return
Select random trial vectors
kernel call
kernel return
Create offspring population (mutation, crossover)
kernel call
kernel return
Evaluate offspring population
kernel call
kernel return
Merge parent and offspring population (migration)

Fig.10. The flowchart of the DE implementation on CUDA.

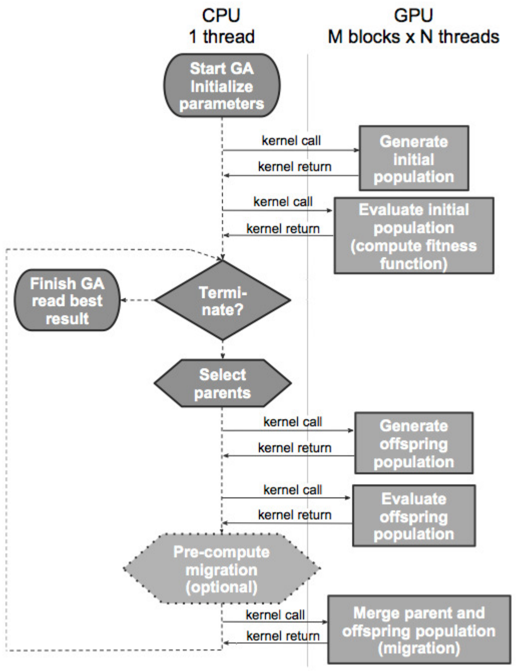

CPU
1 thread
GPU
M blocks x N threads
Start GA Initialize parameters
kernel call
kernel return
Generate initial population
kernel call
kernel return
Evaluate initial population (compute fitness function)
Finish GA read best result
Termi-nate?
Select parents
kernel call
kernel return
Generate offspring population
kernel call
kernel return
Evaluate offspring population
Pre-compute migration (optional)
kernel call
kernel return
Merge parent and offspring population (migration)

Fig.11. The flowchart of the GA implementation on CUDA.

To evaluate the performance of the GA and DE for minimizing the makespan and flowtime, we have used the benchmark proposed in [18]. The simulation model is based on the ETC matrix for 512 jobs and 16 machines. The instances of the benchmark are classified into 12 different types of ETC matrices.

GA		DE	
paremeter	value	parameter	value
population size	64	population size	64
mut. probability	0.01	F	0.9
cros. probability	0.8	C	0.9
selection	semi elitary		

TABLE III. GA and DE settings

ETC matrix	GA	DE
ThMhCc	2.34994e+07	9.55294e+06
ThMhCi	1.38284e+07	3.17031e+06
ThMhCs	1.45571e+07	4.28296e+06
ThMlCc	207687	189457
ThMlCi	182313	78844.7
ThMlCs	171689	104447
TlMhCc	755855	331510
TlMhCi	468615	104894
TlMhCs	493757	142368
TlMlCc	6834.15	6158.24
TlMlCi	5731.92	2550.79
TlMlCs	5873.45	3391.38

TABLE IV. Optimization results

The average final fitness obtained for each ETC matrix by the GA and DE is shown in Table II. We can clearly see that the DE was able to find significantly better schedules within the given minute. The differencies between final fitness for the DE and GA are also illustrated in Fig. 11. Indeed this is an interesting results that can be attributed to several reasons

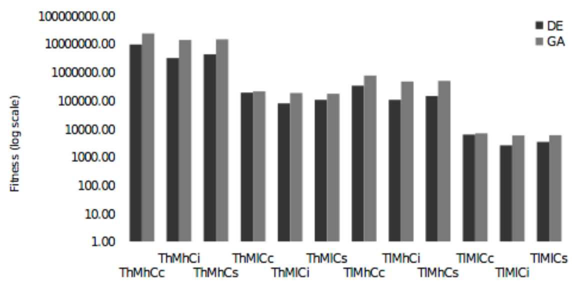


Fig.11. The flowchart of the GA implementation on CUDA.

The CUDA implementation of the DE was easier because the variant of the algorithm that was implemented can be almost entirely expressed using matrix vector operations. The parallelization of the GA was not so straightforward because some parts of the algorithm (parent selection, transition from one population to another) are not suitable for parallelization in a SIMD environment such as the CUDA.

A Parallel Genetic Algorithm with GPU Accelerated for Large-scale MDVRP in Emergency Logistics [16]

Routing the vehicles delivering relief supplies effectively and efficiently right after the disaster is one of the important issues in emergency logistics. Mostly, these kinds of problems are the MDVRP, which are NP-hard problems. When the problem scale goes large, it is hard to find a reasonable solution rapidly enough to meet the requirement of the emergency situation.

In this paper, The proposed a parallel Genetic Algorithm (GA) with Graphics Processing Unit (GPU) accelerated. By assigning the computing tasks for each chromosome to independent threads, the algorithm can process all the operations in GA in parallel. Experimental results show that our parallel algorithm can reduce the computing time of MDVRP to a large degree, which can improve the efficiency and effectiveness of the decision-making process.

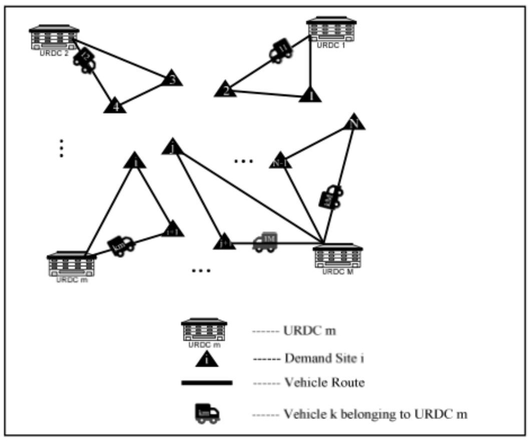


Fig.12. The emergency logistic system.

The routing problem in emergency logistics is a typical MDVRP. The emergency logistics system considered in this study is depicted in Figure 12

As with the normal genetic algorithm, our parallel genetic algorithm also contains four steps on the whole: evolution, selection, crossover and mutation. But some of them are modified in order to fit the parallel architecture of GPU. The algorithm is illustrated as follow:

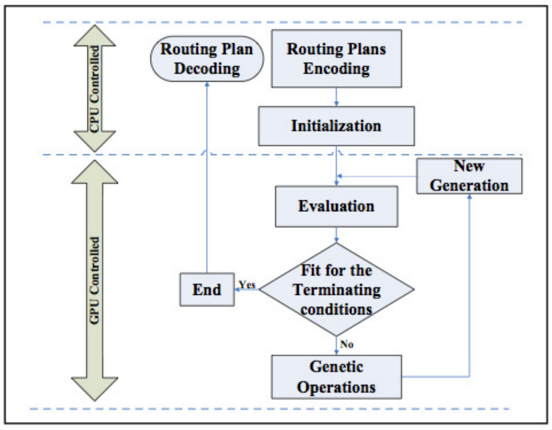


Fig.13. GPU accelerated parallel genetic algorithm.

In the GPU kernel, each chromosome is dispatched to one thread, and all the genetic operations are executed in parallel. The assignment of chromosome in GPU is as follows:

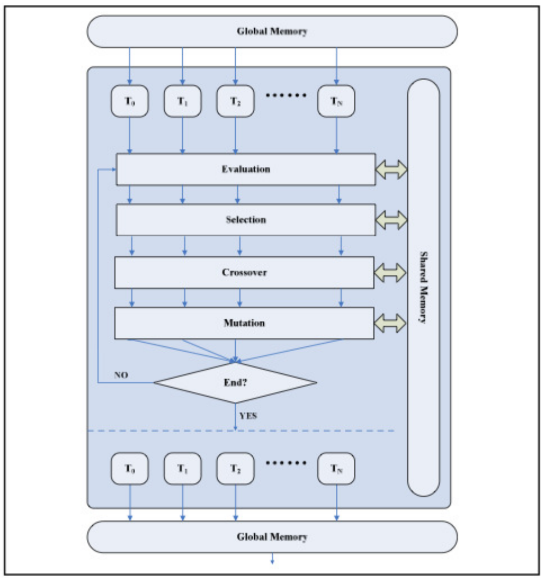


Fig.14. Chromosomes assignment and computing process in kernel

Table V. shows the computing time of both CPU and GPU algorithms for the datasets with the same chromosome population size and generations. And the speedup for different problem scale is also given. Figure 4 shows the difference of algorithm execution time between CPU and GPU with different problem scale.

Dataset	N	CPU time (ms)	GPU time (ms)	Speed-up
CGW1	50	53674	155	346.28
CGW3	75	74265	163	455.61
CGW7	100	105075	172	610.90
CGW16	200	182499	191	964.92

TABLE V. The speed-ups of them algorithm with different datasets.

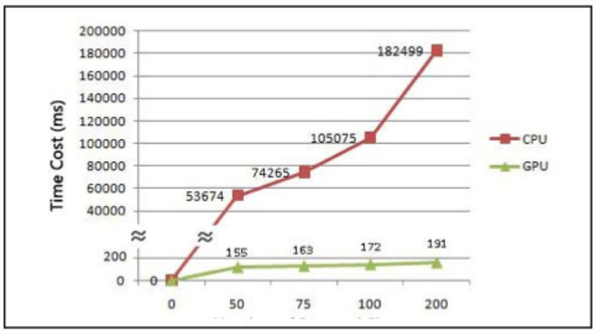


Fig.15. The comparison of time cost between CPU and GPU

The results indicate that our parallel Genetic Algorithm has an ideal speedup ratio from 346.28 up to 964.92 with different number of demand sites from 50 to 200. Compared to CPU, it can dramatic reduce the time of vehicle routes planning in emergency logistic. The accelerate effect is more obvious as the problem scale becomes larger.

A Multi-GPU Implementation of a Cellular Genetic Algorithm [17]

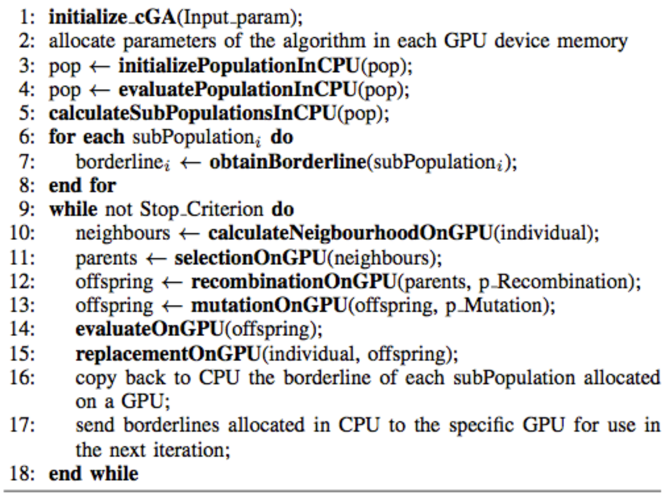
In this work, they implement a multi-GPU cGA algo- rithm that runs entirely on GPUs. We demonstrate that the proposed optimization technique (called cGA MultiGPU) is quite amenable for massive parallelism to obtain larger performances. With regarding to the CPU cGA version, the multi-GPU cGA obtains a speedup of up to 771 and respect the version on a single GPU the time consuming is only a 17% more in the worst case. This approach offers the possibility to solve large problem instances. All this will be shown on a benchmark of discrete and continuous problems to claim not only for time reductions but also for numerical advantages of this swarm intelligence algorithm.

In this study, They present a cGA algorithm running over a Description of CUDA architecture □NVIDIA GeForce GTX 285. This GPU consists of 16 stream multi-processors, each of wich has 8 processors, in total we have 240 processors capable of concurrent floating point operation which can accelerate computation many times over a CPU.

```
 1: initialize_cGA(Input_param);
 2: allocate parameters of the algorithm on GPU device memory
 3: for each individual in parallel do
 4:     individual ← initializeIndividual(individual);
 5:     individual ← evaluateIndividual(individual);
 6: end for
 7: while not Stop_Criterion do
 8:     for each individual in parallel do
 9:         neighbours ← calculateNeigbourhood(individual);
10:         parents ← selection(neighbours);
11:         offspring ← recombination(parents, p_Recombination);
12:         offspring ← mutation(offspring, p_Mutation);
13:         evaluateIndividual(offspring);
14:         replacement(individual, offspring);
15:     end for
16: end while
```

Fig.16. Pseudocode of Cellular GA on a GPU

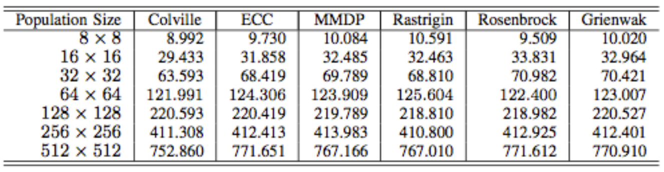
They implementation of a cGA algorithm for a single GPU (called cGA GPU) exploits the inherent parallelism of a GPU using a direct mapping between the population structure and the threads of the GPU. The algorithm have show on Figure 16 and Multi-GPU algorithm show on Figure 17.



```
initialize_cGA(Input_param);
allocate parameters of the algorithm in each GPU device memory
pop ← initializePopulationInCPU(pop);
pop ← evaluatePopulationInCPU(pop);
calculateSubPopulationsInCPU(pop);
for each subPopulation$_i$ do
    borderline$_i$ ← obtainBorderline(subPopulation$_i$);
end for
while not Stop_Criterion do
    neighbours ← calculateNeigbourhoodOnGPU(individual);
    parents ← selectionOnGPU(neighbours);
    offspring ← recombinationOnGPU(parents, p_Recombination);
    offspring ← mutationOnGPU(offspring, p_Mutation);
    evaluateOnGPU(offspring);
    replacementOnGPU(individual, offspring);
    copy back to CPU the borderline of each subPopulation allocated
    on a GPU;
    send borderlines allocated in CPU to the specific GPU for use in
    the next iteration;
end while
```

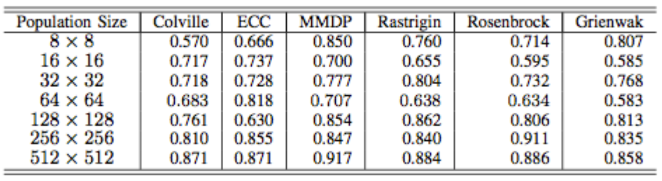
Fig.17. Pseudocode of Cellular GA on a multi-GPU

They selected for tests the following problems, 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 Rastri- gin function and Shifted Rosenbrock function. For the last three problems we decided to use instances of size 100 Il.



Population Size	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8 × 8	8.992	9.730	10.084	10.591	9.509	10.020
16 × 16	29.433	31.858	32.485	32.463	33.831	32.964
32 × 32	63.593	68.419	69.789	68.810	70.982	70.421
64 × 64	121.991	124.306	123.909	125.604	122.400	123.007
128 × 128	220.593	220.419	219.789	218.810	218.982	220.527
256 × 256	411.308	412.413	413.983	410.800	412.925	412.401
512 × 512	752.860	771.651	767.166	767.010	771.612	770.910

TABLE VI. AVERAGE SPEEDUP OBTAINED WITH DIFFERENT POPULATION SIZE FOR BETWEEN CGA MULTIGPU AND CPU CGA



Population Size	Colville	ECC	MMDP	Rastrigin	Rosenbrock	Grienwak
8 × 8	0.570	0.666	0.850	0.760	0.714	0.807
16 × 16	0.717	0.737	0.700	0.655	0.595	0.585
32 × 32	0.718	0.728	0.777	0.804	0.732	0.768
64 × 64	0.683	0.818	0.707	0.638	0.634	0.583
128 × 128	0.761	0.630	0.854	0.862	0.806	0.813
256 × 256	0.810	0.855	0.847	0.840	0.911	0.835
512 × 512	0.871	0.871	0.917	0.884	0.886	0.858

TABLE VII. AVERAGE SPEEDUP OBTAINED WITH DIFFERENT POPULATION SIZE BETWEEN CGA GPU AND CGA MULTIGPU

In Table VI, they present the resulting speedups for each problem. This value is the average time of the cGA algorithm in CPU divided by the average time of the cGA multiGPU. Thus, a value over 1.0 means a more efficient performance of the multi-GPU over the CPU. The results of them tests show that the speedup ranges from 8 to 771. They can observe that the speedup increases when the population size is larger. Another interesting observation is that there is not significant difference between the speedup of the discrete and continuous domains. This indicates that the multi-GPU approach is effective to evaluate problem instances of both domains.

Now, They analyze the speedup resulting from comparing the execution time between the cGA GPU and cGA multiGPU (showed in Table VII).

IV. COMPARISION

Ant

Ref Problems	Modified	Languages	Case Study
[12] Travelling Salesman Problem	No	MATLAB	MMAS
[13] ISCAS89 and ITC99	Yes	C++	N/A
[14] Travelling Salesman Problem	No	C++	MMAS

GA

Ref Problems	Modified GA	GPU
[15] ETC Matrix	Yes	Single-GPU
[16] MDVRP	Yes	Single-GPU
[17] Colville, ECC, MMDP, Rastrigin, Rosenbrock, Grienwak	Yes	Multi-GPU

V. REFERENCES

[1] W. Wongthatsanekorn, “Reviewing and Comparisons of Five Evolutionary-based Algorithms” The Journal of KMUTNB., Vol. 19, No. 2, May - Aug. 2009

[2] J. Holland, Adaptation in natural and artificial systems. Ann Arbor, MI: University of Michigan Press, 1975. accessed on May 20, 2008.

[3] Dijkstra, Edsger; Thomas J. Misa, Editor (2010-08). "An Interview with Edsger W. Dijkstra". Communications of the ACM 53 (8): 41–47. doi:10.1145/1787234.1787249. "What is the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path which I designed in about 20 minutes. One morning I was shopping with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."

[4] Dijkstra 1959

[5] Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[6] S.V. Adve et al. (November 2008). "Parallel Computing Research at Illinois: The UPCRC Agenda" (PDF). Parallel@Illinois, University of Illinois at Urbana-Champaign. "The main techniques for these performance benefits – increased clock frequency and smarter but increasingly complex architectures – are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster."

[7] Asanovic et al. Old [conventional wisdom]: Power is free, but transistors are expensive. New [conventional wisdom] is [that] power is expensive, but transistors are "free".

[8] Asanovic, Krste et al. (December 18, 2006). "The Landscape of Parallel Computing Research: A View from Berkeley" (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183. "Old [conventional wisdom]: Increasing clock frequency is the primary method of improving processor performance. New [conventional wisdom]: Increasing parallelism is the primary method of improving processor performance … Even representatives from Intel, a company generally associated with the 'higher clock-speed is better' position, warned that traditional approaches to maximizing performance through maximizing clock speed have been pushed to their limit."

[9] Hennessy, John L.; Patterson, David A., Larus, James R. (1999). Computer organization and design : the hardware/software interface (2. ed., 3rd print. ed.). San Francisco: Kaufmann. ISBN 1-55860-428-6.

[10] NVIDIA. CUDA Parallel Computing Platform. 2013. Available: http://www.nvidia.com/object/cuda_home_new.html

[11] http://www.mjc2.com/logistics-planning-complexity.htm Why is vehicle routing hard - a simple explanation

[12] Jie Fu, Lin Lei, Guohua Zhou, “A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-In-Roulette Selection” Third International Workshop on Advanced Computational Intelligence , August 25-27, 2010 - Suzhou, Jiangsu, China

[13] Min Li, Kelson Gent, and Michael S. Hsiao. “Utilizing gpgpus for design validation with a modified ant colony optimization.” High-Level Design, Validation, and Test Workshop, IEEE International, pp. 128–135, 2011.

[14] Wang Jiening, Dong Jiankang, Zhang Chunfeng, " Implementation of Ant Colony Algorithm based on GPU" 2009 Sixth International Conference on

Computer Graphics, Imaging and Visualization , pp. 50-53 , 11-14 August 2009 , Tianjin, China

[15] Kromer, P., Platos, J., Snasel, V., Abraham, A., "A comparison of many-threaded differential evolution and genetic algorithms on CUDA" Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 1595-1602, 2011, New York, NY, USA

[16] Jianming Li, Ximeng Lv, Linlin Liu, "A Parallel Genetic Algorithm with GPU Accelerated for Large-scale MDVRP in Emergency Logistics," cse, pp.602-605, 2011 14th IEEE International Conference on Computational Science and Engineering, 2011

[17] Pablo Vidal, Enrique Alba, “A multi-GPU implementation of a Cellular Genetic Algorithm.” IEEE Congress on Evolutionary Computation, pp.1-7, 2010, Barcelona, Spain, 18-23 July