SCIENCE CHINA
Information Sciences

• **Supplementary File** •

Dynamic Strategy based Parallel Ant Colony Optimization on GPUs for TSPs

Yi ZHOU[1,2], FaZhi HE[1*] & YiMin QIU[3]

[1]*State Key Laboratory of Software Engineering, School of Computer Science, Wuhan University, Wuhan* 430072*, China;*
[2]*School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan* 430081*, China;*
[3]*School of Information Science and Engineering, Wuhan University of Science and Technology, Wuhan* 430081*, China*

Appendix A Background

Appendix A.1 Ant Colony Optimization for the TSP

In computational complexity theory, the TSP is an NP-hard problem. It plays a prominent role in research as well as in a number of application areas [1]. The objective of this problem is to find a minimum-weight Hamilton cycle in a complete weighted directed graph $G = (V, A, d)$, where $V = 1, 2, ..., n$ is a set of vertices (cities), $E = \{(i, j)|(i, j)\epsilon V \times V\}$ is a set of edges (paths), and $d : E \to \mathbb{N}$ is a function assigning a weight or distance (positive integer) d_{ij} to every edge (i, j).

Dorigo et al. [2] proposed a basic ACO algorithm named the *ant system* (AS) to solve the TSP. It involves using many artificial ants to perform parallel searches on a graph. Each ant moves independently on the graph until it has traveled to all of the vertices on the graph. Because each ant constructively builds a route, this process is referred to as the *tour construction*, which is the first stage. The second stage is the *pheromone update*. To obtain a better solution, each ant strengthens the pheromone on its path to guide other ants. The ants stochastically move to the next city based on the heuristic information obtained from the pheromone trail and inter-city distances. However, a pheromone-evaporation process is also applied to avoid falling into local optimum solutions.

In the tour-construction stage, each ant independently selects a route for traveling to all cities. Take ant k, for example; when this ant is placed at city i, the probability of visiting city j is calculated by (A1):

$$P_{i,j}^k = \frac{[\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{l\epsilon N_i^k} [\tau_{i,l}]^\alpha [\eta_{i,l}]^\beta}, j\epsilon N_i^k, \tag{A1}$$

where $\tau_{i,j}$ is the pheromone value on the path from city i to city j. $\eta_{i,j} = 1/d_{i,j}$ is a heuristic value. α and β are configurable parameters that represent the relative influences of the pheromone trail and the heuristic information, respectively. N_i^k is the number of cities that can be reached by ant k when at city i. The cities outside N_i^k are recorded in an array called the tabu list. Ant k stochastically moves to the next city using a roulette-wheel selection procedure [3], which is also known as fitness proportionate selection in genetic algorithms.

In the pheromone-update stage, each path holds a pheromone value that guides the ants' travel. The initial pheromone values are set equally. When the ants finish their travel, the pheromones on all the paths evaporate, as described in (A2):

$$\tau_{i,j} \leftarrow (1 - \rho)\tau_{i,j}, \forall(i, j)\epsilon L, \tag{A2}$$

where ρ is the evaporation rate, which ranges between 0 and 1. Then, the ants update the pheromone value of their travelled paths separately, as described in (A3):

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^k, \forall(i, j)\epsilon L, \tag{A3}$$

where $\Delta\tau_{i,j}^k$ is the amount of pheromone that ant k deposits on path $e(i, j)$, which is defined as follows:

$$\Delta\tau_{i,j}^k = \begin{cases} Q/L_k, & \text{if ant } k \text{ goes through } e(i, j), \\ 0, & \text{otherwise,} \end{cases} \tag{A4}$$

where Q is a constant and L_k is the tour length of ant k.

Algorithm A1 presents the general ACO framework in pseudo-code. In the tour-construction stage, m ants travel to n cities sequentially. In the pheromone-update stage, each ant deposits the pheromone on the n paths of its travel, one by one. These two stages are performed until the termination criterion is reached. In this work, we focus on the GPU acceleration

* Corresponding author (email: fzhe@whu.edu.cn)

Algorithm A1 ACO metaheuristic pseudo-code

```
1: InitializeData();
2: t:= 0;
3: repeat
4:     TourConstruction();
5:     PheromoneUpdate();
6:     t:= t+1;
7: until Termination_criterion()
8: end
```

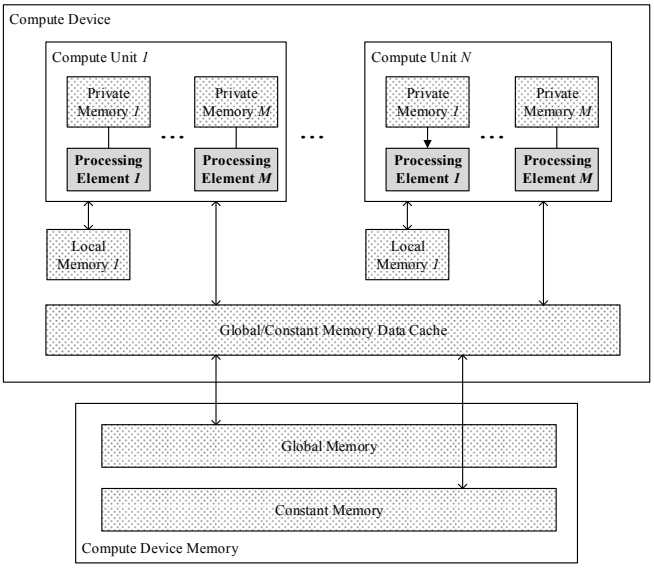

Figure A1 OpenCL compute device architecture [7].

of the ACO algorithm; further information on the original sequential algorithm can be found in [3].

Appendix A.2 GPU Computing

In this section, we briefly introduce the GPU architecture and programming model, for a better understanding of our work. GPUs, formerly designed as a fixed-function rendering devices, have evolved into highly parallel and many-core general-purpose computing devices, which work in a SIMD (single instruction, multiple data) manner. This architecture puts more emphasis on data processing than data caching and flow control; thus, it is very suitable for computation-intensive and highly data-parallel computation. More detailed material about the GPU architecture can be found in [4–6].

The Open Computing Language (OpenCL) and CUDA are two prevalent and successful programming models in the field of GPU computing; they share many similarities. We chose OpenCL to implement the parallel ACO algorithm. OpenCL is an open standard for general-purpose parallel programming across CPUs, GPUs, and other processors [7]. It provides software developers with low-level access to the hardware of these heterogeneous processing platforms and is portable in both code and performance [8, 9]. As a programming language, OpenCL supports application programming interfaces (APIs) for coordinating parallel computation across heterogeneous processors, and the C programming language for cross-platform ability with a well-specified computation environment.

In OpenCL, a GPU is regarded as a compute device. From a hardware point of view, a compute device is comprised of one or more compute units (CUs), which are further divided into one or more processing elements (PEs) (Figure A1). Computations on the GPU occur within the processing elements. The global memory, constant memory, and texture memory together represent the GPU's off-chip memory. Generally, a CPU in the OpenCL architecture is referred to as a host. The OpenCL application submits commands from the host to execute computations on the PEs within a GPU.

Execution of an OpenCL program on hardware is based on kernels. A kernel instance is called a work-item, which is the smallest execution unit and more generally called a thread. The host calls the GPU function using the kernel, which defines the computation to be executed by many work-items organized in work-groups. In a work-group, work-items are further grouped into wavefronts (warps) coordinated by a scheduler at runtime. They execute concurrently on the PEs of a single CU. The number of work-items in a work-group is also referred to as the work-group size or local size, while the total number of work-items is the global size. The number of work-items in a wavefront is regarded as the wavefront size, which is an implementation-defined constant. Each work-item has an explicit identifier in a work-group named *localid* and an implicit identifier named *laneid* (SIMD lane id) in a wavefront that is defined as: $localid\%wavefrontsize$. Based on the assumption that the work-items of a wavefront execute synchronously, we can exploit the wavefront-level parallelism by

omitting memory-synchronization instructions when using local memory.

The hierarchical memory model of the GPU consists of global memory, constant memory, local memory, and private memory. The global memory can be accessed by all work-items and is the largest memory region. The constant memory, which can only be allocated and initialized on a host, is a read-only region in global memory. The local memory is shared by work-items in a work-group; it is much faster than the global memory but has a limited size, typically within 100 KB. The private memory is a private memory region of a work-item that cannot be observed by other work-items. This memory contains the registers used by each PE.

Appendix A.3 Related Work

Appendix A.3.1 *Basic Implementation of Parallel ACO on GPUs*

Pioneer work on GPU-based parallel ACOs started with the emergence of the programmable shader. Catala et al. [10] and Wang et al. [11] implemented ACOs using vertex processors and fragment processors on GPUs. After CUDA was introduced by the Nvidia Corporation, researchers and developers could benefit from this GPU computing API, combining its ease of programming with their need for computing power. Since then, parallel ACO algorithms have mainly been implemented with CUDA [12–16]. These studies provide us with approaches focusing on accelerating ACO algorithms on GPUs.

Appendix A.3.2 *Parallelization Models of GPU-based ACO*

Parallel ACO algorithms can be implemented using GPU computing APIs, e.g., CUDA and OpenCL. Although a general computing API provides us with a more convenient programming interface than a graphics API, it is still very important to investigate parallel ACO models and strategies on GPU hardware to achieve the best performance. Since 2013, systematic studies on GPU-based ACO have been proposed. Notably, Delévacq et al. [17] and Cecilia et al. [18] offer data-parallel ACO models on GPUs with similar ideas that associated each ant with a work-group. This model is more suitable for a GPU architecture than the traditional task-parallel ACO model. The advantage of the data-parallel approach is that low-latency local memory becomes available for storing each ant's data. In essence, this approach exploits the spatial locality of a GPU memory system. Their experimental results confirmed that the data-parallel model largely outperforms the traditional task-parallel counterpart on GPUs.

In addition, Delévacq et al. integrated a three-opt local search into the MAX-MIN ant system (MMAS) algorithm to improve the solution quality. They also implemented and evaluated multiple ant colonies. They reported speedups as high as 19.47x with a solution quality similar to that of the original sequential implementation using ANT^{shared}_{block}. Their results also showed low speedups of up to only 5.84x using ANT_{thread}. Cecilia et al. address the data-parallelism scheme and proposed a new proportionate-selection method called I-Roulette that simulates the classic roulette-wheel selection process while improving the GPU parallelism.

Appendix A.3.3 *Optimization of GPU-based ACOs*

Focusing on implementation strategies for higher efficiency, Uchida et al. [19] implemented ACOs for the TSP using CUDA, while considering many GPU programming issues, e.g., the coalesced access of global memory and the shared memory bank conflict. In addition, these authors proposed a strategy named Stochastic Trial to avoid the prefix-sum calculation as much as possible. Dawson et al. [20] extended the data-parallel ACO approach. In the tour-construction stage, they proposed a new parallel implementation of the roulette-wheel selection called Double-Spin Roulette to fully exploit the warp-level parallelism, which significantly reduced the running time.

Appendix A.3.4 *GPU-based ACOs on Open Platforms*

CUDA is a powerful platform with mature toolkits, but it is a closed platform that only works on Nvidia GPUs. To implement and test parallel ACOs on various devices, OpenCL is a good candidate. Recently, Angelo et al. [21] explored the recent developments for parallel ACO algorithms on GPUs. They introduced parallelism strategies for each step of an ACO algorithm in OpenCL. Guerrero et at. [22] implemented the ACO algorithm using OpenCL, and extensively experimented with it on a wide variety of GPU platforms. They benchmarked the algorithm against existing implementations, and offered extensive analyses.

Appendix A.3.5 *Summary of Related Work*

We summarize the previous work in Table A1. These studies intensively investigate ACO parallel-implementation strategies on GPUs, including parallel models and new algorithms. They demonstrate the promising computational efficiency of GPU platforms, while preserving the solution quality. They provide us with insights on the following major issues. First, several parallelization models in the tour-construction stage are studied and compared, especially the roulette-wheel selection process. Second, approaches for efficient management of the pheromone trails matrix are proposed. Third, GPU memory strategies for efficient management of ACO data structures are supplied.

However, due to GPU resource limitations [17, 18, 20], these authors did not test any TSP instances larger than 2,392 cities. In this paper, we greatly extend previous work on large TSPs.

Table A1 Summary of GPU-based parallel ACO proposals.

Author	Year	Algorithm	Problem	CPU	GPU	Speedup	Largest problem
Catala et al. [10]	2007	ACO	Orienteering problem	Intel Pentium IV 2.4 GHz	Nvidia GeForce 6600GT	80	Graph with 3,000 nodes
Wang et al. [11]	2009	MMAS	TSP	AMD 2.79 GHz	Nvidia Quadro Fx 4500	1.38	TSP with 30 cities
Zhu and Curry [12]	2009	ACO	Bound constrained optimization	Intel Xeon E5420	Nvidia GeForce GTX 280	403.91	-
Li et al. [13]	2009	MMAS	TSP	Intel Pentium 2.6GHz	Nvidia GeForce 8600GT	15.27	tsp225[1]
Bai et al. [14]	2009	MMAS	TSP	AMD Athlon 3600+	Nvidia GeForce 8800GTX	2.3	rd400
Fu et al. [16]	2010	MMAS	TSP	Intel Core i7 3.3 GHz	Nvidia Tesla C1060	31	pr1002
Bai et al. [15]	2011	MMAS ACS	TSP	AMD Athlon 3600+	Nvidia GeForce 8800GTX	45	u1060
Uchida et al. [19]	2012	AS	TSP	Intel Core i7 860	Nvidia GTX 580	43.47	pr2392
Delévacq et al. [17]	2013	MMAS	TSP	Intel Xeon E5640	Nvidia Tesla C2050	19.47	d2103
Cecilia et al. [18]	2013	AS	TSP	Intel Xeon E5620	Nvidia Tesla C2050	21	pr2392
Dawson et al. [20]	2013	AS	TSP	Intel Core i7 950	Nvidia GTX 580	82	pr2392
Angelo et al. [21]	2013	AS	TSP	-	-	-	-
Guerrero et al. [22]	2014	AS	TSP	Intel Xeon E5620, AMD Llano E350 and AMD Llano A6-3420	Nvidia Tesla C2050, AMD FirePro V8800 and three AMD HD6000-series GPUs	21.5	pr2392

[1)] This is a TSPLIB instance, and the number represents its size.

Appendix B Proposed GPU Acceleration Strategies

Appendix B.1 Previous GPU-based ACO Algorithm Strategies

Appendix B.1.1 *Data Parallel Model*

From a hardware point-of-view, to fully exploit the computing power of a GPU, the thread strategy should be configured properly using two parameters: the work-group size and the number of work-items. The work-group size should be a multiple of the wavefront size to achieve higher hardware utilization. An easy strategy is to consider a work-item as an ant. However, the memory bandwidth is limited by the hardware constraints and the complexity of the algorithm.

Delévacq et al. [17] suggested an ANT_{block} strategy to harness the benefits of a GPUs parallel data-processing capability by associating each ant with a CUDA block. Cecilia et al. [18] proposed a similar idea that associated a work-group, composed of w worker ants, with a queen ant. All w worker ants obtain a solution cooperatively, thus enhancing the data parallelism by a factor of w. In this paper, we call these strategies ST-GROUP. The experimental results in [17] and [18] both demonstrate the significantly larger acceleration ratio of the ST-GROUP strategy. Therefore, we selected the ST-GROUP in the tour-construction step, which we will discuss in the next section. Our new approach is based on this strategy.

Appendix B.1.2 *GPU Memory Management*

ACO has two types of data structures. The first type is city data, which contains the path-length data. These data could be considered as an $n \times n$ matrix, where n is the number of cities. The data size increases dramatically with the number of cities. It is suitable for writing to global memory because of the limitations of the local memory size. The second type of data is the ant data. Each ant has its own private data to record the path and visits. These data are $m \times n$, where m is the number of ants.

As noted above, the ST-GROUP is a strategy based on data parallelism. Therefore, an ant data structure that is used frequently in the tour-construction period could be saved to local memory to reduce the overhead of data accesses to global memory. Typically, the ant data structure is a one-dimensional array with size n, e.g., the tabu list, city indices, city visits, and transfer probabilities. Table B1 shows the data distribution of the GPU memory.

Table B1 Data Distribution of GPU Memory

ACO structure	Memory type	Data size
City distance	Global memory	n * n
Pheromone information	Global memory	n * n
Parameters	Constant memory	-
Tabu list	Global memory	m * n
City visit	Global memory	m * n
Tabu list (per ant)	Local memory	n
City visit (per ant)	Local memory	n
Probability array (per ant)	Local memory	n
Temporary variable	Private memory	-

Appendix B.2 Our Approaches on Kernel Control

The GPU has some hardware-related limitations, e.g., the local memory of each work-group and registers for each PE. As stated in [17, 20], the GPU memory feature reaches its limits on larger TSPs. If the kernel algorithm only considers small-scale problems, the GPU would be out of resources when solving large-scale problems. On the other hand, a kernel suited for large-scale problems would face efficiency degradation for small-scale problems. Therefore, we propose two kernels in the tour-construction stage. One kernel is called KE-ALL, and the other is called KE-ONE. KE-ALL creates tours for all of the ants when the tour-construction kernel is invoked by the host. It contains a loop with count $n-1$ for the full tour construction of each ant, as shown in Algorithm B1. The main concept is that m ants create their paths in parallel,

Algorithm B1 KE-ALL

```
i:=GetWorkGroupId();
InitializeData(i);
k:=0;
repeat
    for each i → m − 1 parallel do
        Calculate transfer probabilities
        Barrier();
        Randomly select next city id
        Barrier();
        Move ant i to next city
    end for
    Barrier();
    k:=k+1;
until k==n-1
end
```

using m work-groups. We use the GPU memory-synchronization function, e.g., barrier(), to guarantee that all work-items in a work-group finish reading or writing data before the next calculation step.

Because this kernel contains a loop associated with the number of cities, and the register size of a work-group is limited, it is not a scalable solution. KE-ONE moves all of the ants forward one step per call from the host. The loop control is performed by the host to relieve the pressure on GPU registers, as shown in Algorithm B2.

Algorithm B2 KE-ONE

```
i:=GetWorkGroupId();
InitializeData(i);
for each i → m − 1 parallel do
    Calculate transfer probabilities
    Barrier();
    Randomly select next city id
    Barrier();
    Move ant i to next city
end for
end
```

This method could be adopted to solve a large-scale TSP. However, in the small-scale case, the CPU calls the GPU kernel frequently, which has a considerable time cost.

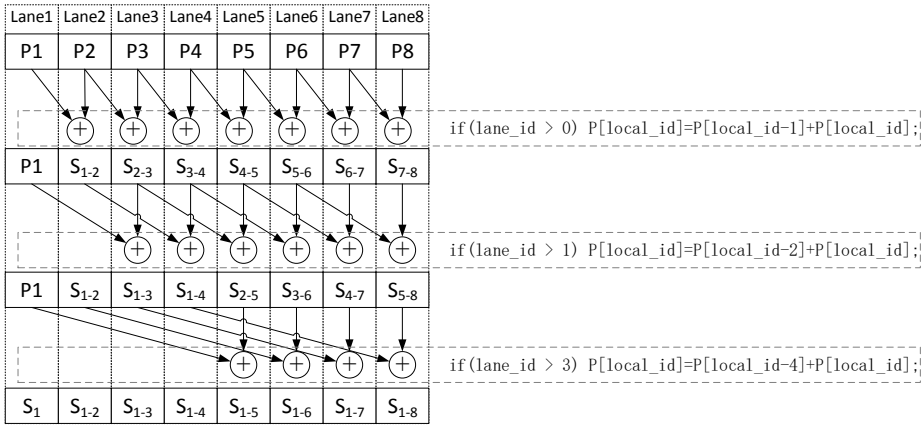

Figure B1 Optimization of parallel prefix sum on GPUs by exploiting wavefront-level parallelism, in which $n = 8$ and S_{i-j} is the sum of the items from index i to index j in array P.

Appendix B.3 Pheromone Update

The pheromone-update stage contains two steps. In the first step, each path evaporates independently, as configured by the evaporation rate. This process could be fully parallelized by creating a thread for each path. Each thread reads the pheromone value of its corresponding path to private memory, calculates the new pheromone value, and then writes it back to global memory. The next process is initiated by the ants, which deposit pheromones along their paths. Unlike the prior step, two or more ants can walk along the same path. Global-memory writing conflicts occur in this situation. However, modern GPUs support atomic instructions, e.g., the atomic add function, to guarantee that the pheromone value could be updated by several ants in parallel. Algorithm B3 presents the pheromone-update kernel. The global synchronization method is needed to avoid data-writing conflicts, because there is no synchronization mechanism between work-groups [24]. In practice, this kernel is divided into two kernels that are called sequentially by the host.

Algorithm B3 Pheromone Update Kernel

```
i:=GetGlobalId();
for each i → n * n − 1 parallel do
    Pheromone evaporate
end for
Global synchronization
for each i → m − 1 parallel do
    for each k → n − 1 parallel do
        Ant j deposit pheromone on its path k using atomic add operation
    end for
end for
end
```

Appendix B.4 Parallelism Optimization Strategies

In the tour-construction process, the selection operates on the probability array to select which city to move to next. Because only one city index value must be saved for the next movement, selecting a city in parallel on the GPU is not straightforward. However, parallel reduction [25] and scan techniques [27] for each work-group have been proposed to address this issue. The sequential roulette-wheel method could be parallelized using several steps.

First, the prefix sum of the probabilities array is calculated in parallel. This process is also called a parallel scan operation. Our implementation is shown in Figure B1. The host preliminarily generates a random array and sends it to GPU memory. Each ant receives a random number r, which is uniformly between 0 and S_{1-j}. The parallel-search method is used to determine the next city index by finding the first element in P that is larger than r. We exploit parallelism in each wavefront by having the lanes within a wavefront execute synchronously. Thus, the thread communication overhead is reduced.

After the ants have travelled to all of the cities, we use the parallel-reduction method to sum up the travel length of each ant and to find the shortest travel length for all of the ants. Since parallel reduction is a very commonly used technique, we suggest that readers refer to [25] for detailed information.

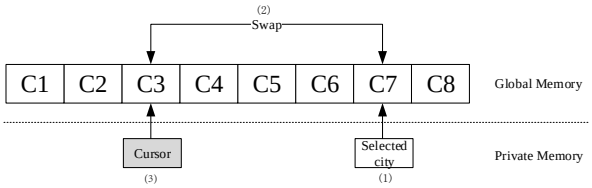

Figure B2 Node-swap process for GPUs. (1) The roulette-wheel selection process produces the result *selected_city*; (2) the current city that the cursor points to is swapped with the selected city in global memory; (3) the cursor moves forward one step.

Appendix B.4.1 *Advantages of the TR Approach*

The advantages of the TR approach are as follows:

1) **Fewer random numbers generated**. In the previous GPU-based parallel roulette-wheel selection approach [18], each ant requires n (the number of cities) random numbers to construct a path of the entire tour. In our TR approach, each ant requires only one random number, which saves considerable computation time.

2) **Improved memory locality**. To efficiently utilize the GPU registers, the tile size should match the register's size within a work-group. For example, the variables that are repeatedly accessed are saved in registers (see Algorithm 1, line 3).

3) **Reduced scan computation**. A full scan of the entire probability array often wastes time. For example, if the selected city lies in the first tile, then computation for the other tiles is useless. In our TR approach, if a tile is selected, the scan of the remaining tiles is omitted (see Algorithm 1, lines 11 to 13).

Appendix B.4.2 *GPU-based Implementation of the STC*

We use the tour list and cursor to replace the tabu list and city visit list. A node-swap operation is defined to operate on the tour list. In step 1 of our algorithm, the scale of the tour list decreases as C^k increases, thus reducing the amount of computations. In Algorithm B4, we present a new GPU strategy called ST-DYNAMIC, which dynamically changes the work-group size for each movement of ants. We call this type of strategy a dynamic work-group. The main idea of this approach is to tune the work-group size in each step. We set the work-group size to a power of 2 because this size is suitable for the parallel-reduction method.

Each ant reads the tour-list data from global memory to local memory using the $readTourListOfAnt()$ method with a parameter cursor to only copy the cities have not been visited. Then, the ant randomly selects a city to visit from *localCityList* in local memory and saves it to *nextCityId*. By the performing node-swap operation, the visited city is placed at the position to which the cursor points. Figure B2 illustrates the node-swap process on GPUs. In this case,

Algorithm B4 Dynamic Strategy

```
 1: i:=GetGlobalId();
 2: repeat
 3:    workGroupSize := nearestPowerOf2 (n - cursor);
 4:    totalThreadNum := workGroupSize * m;
 5:    SetGPUKernel (workGroupSize, totalThreadNum);
 6:    for each i → m − 1 parallel do
 7:       localTourList = readTourListOfAnt(i, cursor, n);
 8:       Parallel calculate transfer probability of cities in localTourList
 9:       Barrier();
10:       Parallel calculate next city id of ant i and set value in nextCityId
11:       Barrier();
12:       RightShift(i, cursor, nextCityId);
13:    end for
14:    cursor[i]++;
15: until cursor[i] == n - 1
16: end
```

there are eight cities and the ant has traveled through $C1$ and $C2$. The cursor is pointing to $C3$. After the roulette-wheel selection process, we obtain the *selected_city*, which is $C7$. Then, $C3$ and $C7$ are swapped in global memory. Finally, the cursor moves forward and points to $C4$ before the next selection process.

Appendix B.4.3 *Time Complexity Analysis*

The differences between ST-DYNAMIC and ST-GROUP are lines 7 to 12 in Algorithm B4. In this section, we analyze both algorithms and predict the improvement of our algorithm. We can assume that a GPU work-group is a parallel

Table C1 Average execution times (ms) of the CPU-based sequential version and the GPU-based parallel version.

Instances	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392	pcb3038	fnl4461
CPU-based	15	54	198	1519	2909	10157	47987	115058	397991
GPU-based	1.09	2.06	5.72	37.64	61.62	270.73	1001.83	3315.79	8903.6

Table C2 Execution times (ms.) on GPU with different work-group size configuration.

Work-group Size	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392	pcb3038	fnl4461
16	20.9	39.0	78.0	387.5	838.5	N/A[1]	N/A	N/A	N/A
32	19.03	35.8	62.5	252.2	593.5	3036.5	15399.1	31278.5	192960.3
64	18.57	34.3	55.2	195.0	381.5	1757.5	8321.5	32691.6	100197.1
128	18.72	34.4	52.3	162.3	280.5	1122.5	4869.5	18110.6	53760.8
256	20.12	37.4	55.6	156.1	239.5	835.5	3216.9	11141.5	31532.3
512	19.5	43.7	70.5	174.7	276.5	981.5	2621.5	8093.3	21691.8
1024	19.5	43.7	71.2	N/A	N/A	N/A	4281	7377.5	18564.7

[2)] N/A indicates not available due to register constraints.

Table C3 Optimized work-group size configuration.

Instances	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392	pcb3038	fnl4461
Work-group size	64	64	128	128	256	256	512	1024	1024

random-access machine (PRAM) with n processors. In line 7, ST-GROUP reads the tabu list and the city-visit list from global memory. In line 12, ST-GROUP writes the next city ID to the tabu list. We obtain the result of $O(1)$ according to Brent's theorem in [26]. Therefore, we can focus on lines 8 and 10, in which the parallel reduction and parallel prefix-sum method are used. The time complexity of these lines has been analyzed in [25] and [27]. In our implementation, with the same steps of ST-GROUP, the number of operations decreases with increasing i; thus, reducing the threads launched in each step.

Our algorithms time complexity is $O(\sum_{i=1}^{n} \log n)$, which is equal to $O(\log n!)$. In theory, the time complexities of the two algorithms are both linearithmic. However, in practice, the work-items in a work-group cannot execute at the same time. These work-items are organized into sub-groups called wavefronts or warps, which have additional execution overhead. In ST-DYNAMIC, the number of warps decreases as the ant moves forward a warp-size step and is thus more efficient. The performances of the algorithms are compared in the next section.

Appendix C Performance Evaluation

Appendix C.1 Experimental Configuration

Our experiments were carried out on an Intel Core i7-4770 (@3.4 GHz) machine with 8 GB DDR3 memory. The GPU platform was an Nvidia Kepler card (GTX780) with 3 GB GDDR5 memory and 2,304 CUDA cores. The OS was Windows 7 64-bit Ultimate Edition. Our development environment was Microsoft Visual Studio 2012 with the AMD APP SDK version 2.9. Our experiment used a standard set of benchmark instances from the TSPLIB library [28]. We use single-precision floating-point numbers for the pheromone information and city distances.

On the algorithm side, our key parameters were configured according to the experimental principles adopted by Dorigo et al. [2]. We set $m = n$ (n being the number of cities), $\alpha = 1$, $\beta = 2$, and $\rho = 0.5$. On the hardware side, we chose the optimal work-group configuration using the methodology proposed in [18].

We tested each TSP instance for a fixed number of iterations (1000 times in 10 independent runs), and used the average, max, and min values for comparison. The sequential CPU version provided by Stüzle in [3] was the baseline algorithm. The speedup was calculated by dividing the GPU algorithm time with the baseline time. The computation time of the CPU-based version executed on our platform is provided in Table C1. Although we focused on the efficiency of the parallel implementation using a GPU, the solution quality comparison between the sequential CPU and parallel GPU algorithm is also provided to ensure the correctness of our algorithms.

Appendix C.2 Determination of the Optimized Work-group Configuration

The work-group size configuration affects on the parallelism efficiency. Table C2 demonstrates that the optimal group-size configuration varies based on the size of the TSP problem. Therefore, the work-group size should be determined by the problem size. When the work-group size is considerably larger than the problem size, thread execution time is wasted. In contrast, an overly small work-group would cause serialization on a GPU. We obtained the optimized work-group sizes listed in Table C3 based on experimental results.

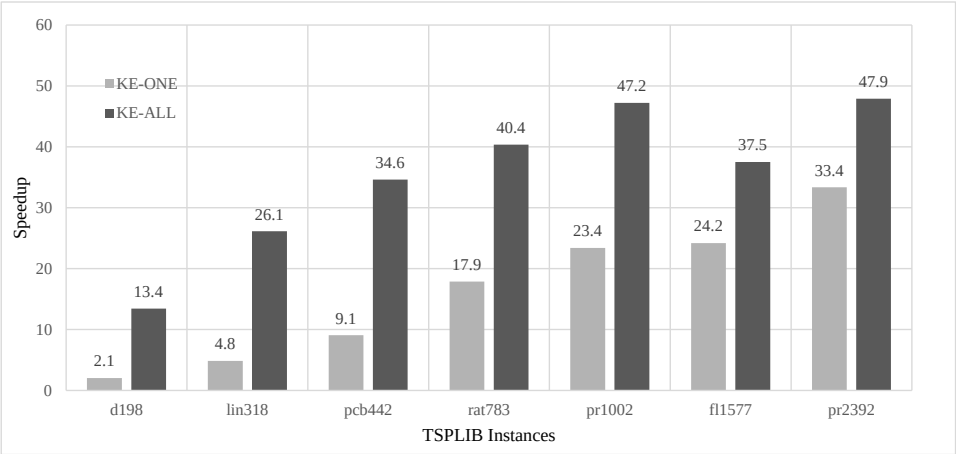

Figure C1 Speedup comparison of KE-ALL and our KE-ONE.

Appendix C.3 Comparison of KE-ALL and KE-ONE

As noted in the last section, the optimum work-group size varies based on the problem size; thus, our tests were based on the optimized configurations listed in Table C3. In addition, in [22], TSP problems were divided into small, medium, and large datasets. Figure C1 shows the effect of kernel-launch overhead on small- and medium-scale problems. Because KE-ALL has only $1/n$ launch counts of KE-ONE, it is significantly faster for a small number of cities. However, as the problem scale increases, the kernel-launch overhead is hidden by the kernel-execution time. Therefore, in TSP problems with a city number equal to or less than 1,577, we recommend using the KE-ALL strategy because of its lower latency, which obtains a solution in a few seconds. This could make the strategy more user friendly in certain real-time applications. We attempted to run KE-ALL on a large dataset; however, it was not applicable due to register constraints. Thus, we gained a speedup up to 47.2x in pr1002 with KE-ALL.

Appendix C.4 Evaluation of the Dynamic Work-group Kernel

To evaluate the contributions of STC and TR to acceleration, we firstly added STC to ST-GROUP (ST-GROUP-STC). As shown in Figure 1, ST-GROUP-STC starts by accelerating ST-GROUP from pcb442 and improves with increases in the problem size. Then, we added the TR method to ST-GROUP-STC (ST-GROUP-STC-TR), which results in nearly the same contribution to the acceleration with the same trend.

Both the ST-GROUP-STC and ST-GROUP-STC-TR methods outperform ST-GROUP as the number of cities increases. In theory, these methods have the same level of time complexity; however, in this case, the processor number in a work-group is not always equal to the number of cities. Two situations are possible. In one situation, the work-group size is larger than the city number. In the other situation, the work-group size is smaller than the city number. In the former situation, the reduction of candidate cities was only limited help because both kernels are fully parallel. However, in the latter situation, both kernels are forced to run serially to prepare data for reduction or prefix-sum calculation. Therefore, STC and TR could further reduce the calculation time in larger problems. The work-group size of ST-GROUP is fixed, which means that the GPU launches all of the work-items in a work-group, even if there are fewer cities to visit, which causes additional overhead.

Furthermore, our ST-DYNAMIC method avoids this overhead by changing the work-group size for each step of the ants. As the remaining city numbers decrease at runtime, the serialization level decreases, which reduces the calculation time.

We can also learn from Figure 1 that ST-DYNAMIC is more suitable for large datasets, and can solve problem sizes as large as 4,461. The speedups are 41.8x and 44.7x in pr2392 and fnl4461 respectively with ST-DYNAMIC.

The speedup of all algorithms decreases at the scale of the pcb3038 problem for two reasons. First, the city number is considerably larger than the work-group size; thus, some calculations are forced to serialize. Second, the global-memory access counts increase, which have a higher latency than local memory accesses. Even in this situation, our ST-DYNAMIC method is still the best one.

Appendix C.5 Comparison with Existing GPU ACO Algorithms

To ensure the efficiency of our algorithm, we compared it against the first data-parallel GPU implementation of AS (DP-GPU-AS) provided by Cecilia et al. [18]. We rebuilt the original DP-GPU-AS on our hardware platform to ensure a fair comparison. In this experiment, we chose the KE-ALL kernel, which performs better with smaller TSPs (198 to 2,392 cities inclusive). The results show our algorithm is up to 3.6x faster than DP-GPU-AS (Table C4).

We also compared our algorithm with two improved GPU ACO algorithms [19,20]. Because no source code was provided for either of them, we could not reproduce their results for a direct comparison. Therefore, we ran our program on a

Table C4 Average execution times (ms) and speedups of our algorithm.

Instances	d198	lin318	pcb442	rat783	pr1002	fl1577	pr2392
DP-GPU-AS	2.22	7.51	18.09	86.53	149.85	586.40	2026.47
Our KE-ALL	1.09	2.06	5.72	37.64	61.62	270.73	1001.83
Speedups	2.04x	3.65x	3.16x	2.30x	2.43x	2.17x	2.02x

Table C5 Average execution times (ms) on GTX580. Smaller is better.

Instances	d198	a280	lin318	pcb442	rat783	pr1002	nrw1379	pr2392
Uchida et al. [19].	2.64	5.06	8.97	11.54	56.73	87.06	x	2084.78
Dawson and Stewart [20]	1.16	2.68	3.39	7.79	42.7	85.11	323	1979.31
Our KE-ALL	**1.12**	**2.41**	**2.92**	**7.32**	**26.69**	**53.58**	**143.43**	**904.10**

Table C6 Solution-quality comparison results. We give the percentage deviations from the average CPU results in parentheses for the GPU algorithm.

Instance	CPU avg.	CPU min.	CPU max.	GPU avg.	GPU min.	GPU max.
d198	17302	17018	17483	17371 (0.40%)	17200	17498
lin318	47406	47029	47878	47517 (0.23%)	47007	47928
pcb442	61752	60653	62378	61790 (0.06%)	60748	62823
rat783	10987	10762	11101	10994 (0.06%)	10786	11140
pr1002	328936	325376	333244	330234 (0.39%)	325100	332859
fl1577	26183	25689	26488	26159 (-0.09%)	25948	26361
pr2392	507012	501376	510957	506913 (-0.02%)	502677	511122
pcb3038	186249	185835	187900	186871 (0.33%)	186132	187853
fnl4461	250037	249630	250790	249887 (-0.06%)	249352	250036

GTX580 GPU with 512 CUDA cores, which is the same as [19, 20], and used the average execution time as an evaluation criterion. The results demonstrate that the performance of our proposed GPU ACO algorithm is the best one (Table C5).

As aforementioned, we used a new SIMD-oriented algorithm for ACO and a new implementation of roulette-wheel selection in the tour-construction stage. We believe our ideas could accelerate other population-based metaheuristic algorithms.

Appendix C.6 Solution Quality

We followed the guidelines of Cecilia et al. [18] to evaluate the solution quality of our GPU-based ACO algorithm. Each algorithm was tested by a fixed number of 1,000 iterations in 10 tries. To ensure a fair comparison, the algorithmic parameter settings of both the CPU and GPU versions were set as described in Section 4.1.

The results indicate that the solution quality of our GPU algorithm is similar to the solution quality of the sequential CPU algorithm (Table C6). The percentage deviations of the GPU average results from the CPU average results are within 0.4%. However, because of the limitations of the original Ant System algorithm as explained by Stützle and Hoos [29], the solutions we obtained are not optimal. This could be improved by attaching a local search process to each ant. In this paper, we focused on the computational performance of the GPU-based ACO algorithm. We will leave improving the solution quality as our future work.

References

1 Hoos H H and Stützle T. Stochastic Local Search: Foundations and Applications. San Francisco: Morgan Kaufmann Publishers, 2005. 357–367
2 Dorigo M, Maniezzo V, and Colorni A. Ant system: optimization by a colony of cooperating agents. IEEE T Syst Man Cy B, 1996, 26: 29–41
3 Dorigo M and Stützle T. Ant Colony Optimization. Cambridge: MIT Press, 2004. 65–90
4 Junkins S. Compute Architecture of Intel Processor Graphics Gen8. Intel Technical Report IDF-2014. 2014
5 AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide Version 2.7, 2013
6 Nvidia. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110. Nvidia Whitepaper. 2012
7 Khronos. The OpenCL Specification Version 1.2, 2011
8 Du P, Weber R, Luszczek P, et al. From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. Parallel Comput, 2012, 38: 391–407
9 Brodtkorb A R, Hagen T R, and Sætra M L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. J Parallel Distr Com, 2013, 73: 4–13

10 Catala A, Jaen J, and Mocholi J A. Strategies for accelerating ant colony optimization algorithms on graphical processing units. In: Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC), Singapore, 2007. 492–500

11 Wang J N, Dong J K, and Zhang C F. Implementation of ant colony algorithm based on GPU. In: Proceedings of the Sixth International Conference on Computer Graphics, Imaging and Visualization (CGIV), Tianjin, 2009. 50–53

12 Zhu W H and Curry J. Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In: Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC), San Antonio, 2009. 1803–1808

13 Li J M, Hu X P, Pang Z L, and Qian K M. A parallel ant colony optimization algorithm based on fine-grained model with GPU-acceleration. Int J Innov Comput I, 2009, 5: 3707–3716

14 Bai H T, Ouyang D T, Li X M, et al. MAX-MIN ant system on GPU with CUDA. In: Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), Kaohsiun, 2009. 801–804

15 Bai H T, Ouyang D T, Li X M, et al. Multiple ant colonies sharing common pheromone matrix based on CPU. J Jilin U: Techno Ed, 2011, 41: 1678-1683. In Chinese

16 Fu J, Lei L, and Zhou G H. A parallel ant colony optimization algorithm with GPU-acceleration based on all-In-roulette selection. In: Proceedings of the 2010 Third International Workshop on Advanced Computational Intelligence (IWACI), Suzhou, 2010. 260–264

17 Delévacq A, Delisle P, Gravel M, et al. Parallel ant colony optimization on graphics processing units. J Parallel Distr Com, 2013, 73: 52–61

18 Cecilia J M, García J M, Nisbet A, et al. Enhancing data parallelism for ant colony optimization on GPUs. J Parallel Distr Com, 2013, 73: 42–51

19 Uchida A, Ito Y, and Nakano K. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In: Proceedings of the 2012 Third International Conference on Networking and Computing (ICNC), Okinawa, 2012: 94–102

20 Dawson L and Stewart I. Improving ant colony optimization performance on the GPU using CUDA. In: Proceedings of the 2013 IEEE Congress on Evolutionary Computation (CEC), Cancun, 2013. 1901–1908

21 Angelo J S, Augusto D A, and Barbosa H J C. Strategies for parallel ant colony optimization on graphics processing units. In: Barbosa H J C, ed. Ant Colony Optimization - Techniques and Applications. InTech, 2013. 63–83

22 Guerrero G D, Cecilia J M, Llanes A, et al. Comparative evaluation of platforms for parallel ant colony optimization. J Supercomput, 2014, 69: 318-329

23 Nvidia. Nvidia CUDA C Programming Guide Version 4.0, 2011

24 Khronos. The OpenCL Specification Version Version 1.1, 2011

25 Harris M. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology. 2007

26 Gustafson J L. Brent's theorem. In: David Padua, ed. Encyclopedia of Parallel Computing. Springer US, 2011. 182–185

27 Harris M, Sengupta S, and Owens J D. Parallel prefix sum (scan) with CUDA. In: Nguyen H, ed. GPU Gems 3. Addison-Wesley, 2007. 851–876

28 Reinelt G. TSPLIB - A traveling salesman problem library. Informs J Comput, 1991, 3: 376–384

29 Stützle T and Hoos H H. MAX-MIN ant system. Future Gener Comp Sy, 2000, 16: 889–914