Parallel Neighbourhood Search on Many-Core Platforms

Yuet Ming Lam

Faculty of Information Technology,
Macau University of Science and Technology, Macau
E-mail: ymlam@must.edu.mo

Kuen Hung Tsoi

Department of Computing,
Imperial College London, UK
E-mail: khtsoi@doc.ic.ac.uk

Wayne Luk

Department of Computing,
Imperial College London, UK
E-mail: wl@doc.ic.ac.uk

Abstract: This paper presents a parallel search parallel move approach to parallelise neighbourhood search algorithms on many-core platforms. In this approach, a large number of searches are run concurrently and coordinated periodically. Iteratively, each search generates and evaluates multiple moves in parallel. The proposed approach can fully utilise the computing capability of many-core platforms under various platform specific constraints. A parallel simulated annealing algorithm for solving the Traveling Salesman Problem is developed using the parallel search parallel move scheme and implemented on an NVIDIA Tesla C2050 GPU platform. We evaluate the performance of our approach against a multi-threaded CPU implementation on a server containing two Intel Xeon X5650 CPUs (12 cores in total). The experimental results of 20 benchmark problems show that the GPU implementation achieves 99 times speedup on average in solution space exploration speed. In term of effectiveness, the GPU implementation is capable of finding good solutions 39.5 times faster or with 21.7% solution quality improvement given the same searching time.

Keywords: Parallel neighbourhood search; Many-core platforms; Graphics Processing Unit; Traveling Salesman Problem; Simulated annealing.

Biographical notes: Yuet Ming Lam received his Ph.D degree in Computer Science and Engineering at the Chinese University of Hong Kong in 2007. He worked as a Research Associate in Department of Computing at Imperial College London from 2007 to 2009. Currently, he is an Assistant Professor in Faculty of Information Technology at Macau University of Science and Technology. His research interests are parallel architectures, design automation and optimisation, and custom computing.

Kuen Hung Tsoi is a postdoctoral researcher in the Custom Computing Group in the Department of Computing, Imperial College London. He obtained his Ph.D from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, in 2007. His previous research work include cryptographic systems and computer arithmetic on reconfigurable platforms. His current research interests include many-core parallel computing, high performance clustering and heterogeneous architecture.

Wayne Luk is Professor of Computer Engineering with Imperial College London. He was a Visiting Professor with Stanford University, California, and with Queens University Belfast, UK. His research includes theory and practice of customizing hardware and software for specific application domains, such as multimedia, financial modeling, and medical computing. His current work involves high-level compilation techniques and tools for high-performance computers and embedded systems, particularly those containing accelerators such as FPGAs and GPUs. He received a Research Excellence Award from Imperial College, and 11 awards for his publications from various international conferences. He is a Fellow of the IEEE and the BCS.

1 Introduction

Neighbourhood Search algorithms have been widely used to solve many NP-complete problems where the time of finding solution increases faster than exponentially as the problem size increased. Examples of such problems are vehicle route planning, job scheduling, task assignment, cargo placement, and digital circuit optimisation. An neighbourhood search

algorithm starts with a feasible solution and attempts to improve it by searching its neighbours. These neighbouring solutions can be reached directly from the current solution by an operation called a *move*. This process is repeated until a local optimum or the termination condition is reached. Due to the NP-completeness nature of such problems, the search process is often tedious and time-consuming. Various parallel approaches using distributed systems or multi-threaded techniques have been proposed to enhance solution quality or to reduce search time.

One strategy is to explore data level parallelism where a problem instance is divided into multiple subsets for parallel processing in multiple processors. When the search terminates, each processor generates a sub-solution. In the final stage, all sub-solutions are combined and transformed into a feasible solution. For example, in a parallel placement algorithm for field programmable gate array technology (Ababei, 2009), a large netlist is partitioned into a number of small sub-netlists. Independent simulated annealing process is applied to each sub-netlist to produce a partial placement solution. Upon termination, all the partial solutions are combined to form a complete and feasible placement solution. A distributed implementation of simulated annealing for solving the Traveling Salesman Problem (TSP) (Allwright and Carpenter, 1989) is also proposed. After the entire path is divided into many sub-paths, multiple processors are used to work on the sub-paths in parallel where a number of trial city exchanges are performed on each processor. However, the drawback of this approach is that the result quality depends strongly on the solution space and the associated partition of it. Thus the final solution by combining sub-solutions is likely to be worse than that of the traditional equential search methods (Ababei, 2009).

A second approach is to employ parallel move strategies, in which multiple moves are generated in each search iteration and evaluated in parallel on a number of processors. Study in (Kravitz and Rutenbar, 1987) tackles placement problem by assigning the cost calculations of multiple placement solutions to multiple processors. A parallel 2-Opt local search approach is proposed for the Traveling Salesman Problem in (Mavroidis et al., 2007). Parallel moves, as swaps of cities, are evaluated on distinct processors. Similarly, (Gallego et al., 1997) proposed a parallel simulated annealing solution for the problem of power transmission network expansion.

A third strategy is parallel search, where each processor performs independent sequential searches on the complete problem and processors are coordinated periodically. In (Crainic et al., 1995), each processor runs a sequential Tabu search for a number of iterations independently and then broadcast the local best result to all the other processors. The global best solution is then chosen as the new starting point of all searches. Parallel simulated annealing algorithms using this approach are also proposed to solve routing problems (Ram et al., 1996; Czech and Czarnas, 2002). Instead of synchronising solutions globally, an alternative implementation is to exchange solutions with neighbouring processors (Cohoon et al., 1991; Whitley et al., 1991; Matsumura et al., 1997).

The main constraints of using distributed systems or multi-threading techniques for multi-core microprocessor systems are the communication overhead and thread switching cost. Recent many-core architectures, such as those employed in Graphics Processing Units (GPUs), have different architectures designed for massively parallel applications. The characteristics include a close to zero thread switching cost; a lower inter-core communication overhead; and a larger number of parallel processing cores. With the advances in many-core technologies, exploration of parallel neighborhood search on many-core platforms becomes more and more popular. Many of these approaches employ a parallel move strategy (Choi and Liu, 2010; Choong et al., 2010; Luong et al., 2010; Han et al., 2011; Fujimoto and Tsutsui, 2011). In particular, a parallel Tabu search (Luong et al., 2010) uses three different neighbourhood functions to generate the neighbours simultaneously. Each neighbourhood function generates a number of neighbours which are evaluated in different cores in an NVIDIA GTX 280 GPU. In these many-core based implementations, the host computer is often used to run a master search thread which iteratively distributes potential moves to the many-core platform for solution evaluation, collects the results and selects the next move (Choi and Liu, 2010; Han et al., 2011). Genetic algorithms are also explored on many-core platforms as shown in (Wong and Wong, 2006; Li et al., 2007). The principle is to utilise the large number of processing cores on the many-core platform to evaluate more individuals in parallel.

Although the above approaches have shown promise, there is scope for significant improvement in parallelism. In this work, a many-core based parallel search parallel move (PSPM) approach is proposed to improve the neighborhood search algorithms. The purposes are to increase the computational parallelism and to minimise the synchronisation overhead. The parallelism is achieved by employing both parallel search and parallel move strategies in a single design. In the PSPM scheme, a number of parallel searches are run and coordinated periodically. Each search generates and evaluates multiple moves in parallel in each iteration. The main contributions of this work are:

- A generic PSPM approach for parallelising neighbourhood search algorithms on many-core platforms is proposed. We focus on maximising the achievable parallelism when mapping the neighbourhood search algorithm on the target hardware platform. This approach can be used to improve the performance of generic neighbourhood search algorithms (Section 3).

- Using the PSPM approach, a simulated annealing process for solving the TSP is parallelised and implemented on a GPU platform. In this case study, we achieve significant improvement over a multi-core CPU design by applying application and platform specific optimisations (Section 4).

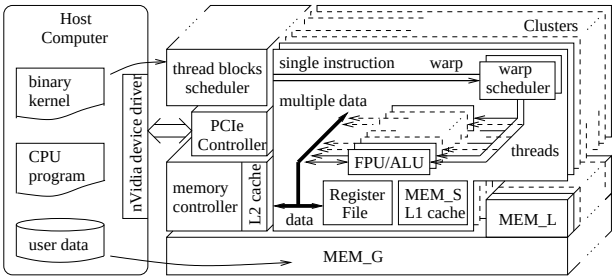

Figure 1 NVIDIA CUDA platform architecture.

- The results of a TSP benchmark suit are measured and evaluated to illustrate the effectiveness of the proposed techniques. The impacts of various runtime configurations and scalability of each platform are also analysed (Section 5).

The rest of this paper is organised as follows: Section 2 introduces the GPU architecture and the simulated annealing algorithm. Section 3 presents the proposed parallel search parallel move approach. Section 4 presents details of the application and platform specific optimisation techniques and the GPU-based implementation. The measured results and evaluations are given in Section 5. Finally, Section 6 draws conclusions and future work.

2 Background

2.1 GPU architecture

The GPU platform recently becomes one of the most popular many-core platforms for general purpose computing applications. In this work, we carry out the many-core experiments on an NVIDIA GPU card. Figure 1 illustrates an abstract structure of the NVIDIA graphics processors with the Fermi architecture (NVIDIA, 2009). A group of 8 to 32 floating point units (FPUs) and a shared control unit form a basic computing core called a streaming multiprocessor. Associated with each streaming processor is a multi-bank static memory partitioned into L1 cache and user-managed shared memory. Based on a single instruction in the control unit, multiple data can be read, processed or written in parallel. Depends on the die size, there may be 1 to 30 streaming multiprocessors on a single GPU chip. The size of on-chip shared memory on each streaming multiprocessor is in range of 16KB to 64KB and larger data structure must be stored in off-chip global memory which presents a unified address space to all threads in all streaming multiprocessors. The shared memory provides a much higher bandwidth than the global memory with less access conflicts. For simplicity, some components such as special function units and texture memory are not shown in the figure.

The CUDA (NVIDIA, 2011) driver groups and manages threads into *thread blocks*. All threads within a thread block view the same address space in the shared memory which is inaccessible by threads in any other thread blocks. A thread block is assigned and distributed to a single streaming multiprocessor during run time. This thread block, as well as any allocated per-block resources, resides statically inside the streaming multiprocessor until all threads inside the block are terminated. Thus the total number of active blocks are limited by the resource requirements of the blocks. Non-active blocks are kept waiting in a queue managed by the device.

Within a block, every 32 threads are grouped into an atomic structure call a *warp*. Threads in a block are scheduled for execution in warps. Different warps are scheduled in non-deterministic order and executed independently. Threads within a warp are designed to execute in parallel in instruction level for maximum performance. When threads within a warp branch into different execution paths, the warp scheduler will sequentially go through the divergent paths one by one until all paths merge at a single point. That means every time the diverged warp is scheduled, threads in the current paths are executed while the remaining threads are disabled. As a result, minimising intra-warp divergence is critical for performance optimisation.

Algorithm 1: Standard simulated annealing.

```
1  begin
2      T ⇐ T_initial;
3      prevSol ⇐ feasible_initial_solution;
4      prevCost ⇐ getCost(prevSol);
5      bestSol ⇐ prevSol;
6      bestCost ⇐ prevCost;
7      while T > T_termination do
8          newSol ⇐ generateNewSol(prevSol);
9          newCost ⇐ getCost(newSol);
10         p ⇐ exp((prevCost − newCost)/T);
11         rand ⇐ random(0, 1);
12         if rand < p then
13             prevSol ⇐ newSol;
14             prevCost ⇐ newCost;
15             if newCost < bestCost then
16                 bestSol ⇐ newSol;
17                 bestCost ⇐ newCost;
18             end if
19         end if
20         T ⇐ αT;
21     end while
22 end
```

2.2 Standard Simulated Annealing

Neighbourhood search starts with a feasible solution and attempts to improve it by searching its neighbours through the *move* operation. A move generates a new solution by modifying the previous solution. This process is repeated until a local optimum or the termination condition is reached. Simulated annealing is one of the popular neighbourhood search techniques, we employ it to verify the proposed techniques in this work. In simulated annealing, the process

accepts not only neighbours with improved solutions, but also those with increased cost based on certain design parameters. The acceptance probability p is determined using the Metropolis criterion:

$$p = \begin{cases} e^{(cost(b)-cost(c))/T_i} & \text{if } cost(c) > cost(b), \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

where b is the previous solution and c is the new solution. One can see that a better solution will always be accepted. For a worse solution, the acceptance probability will be higher if the cost is closer to the previous solution. It is given that T_i is the temperature in the i^{th} iteration and is updated as follows:

$$T_{i+1} = \alpha T_i \quad (2)$$

where α is the cooling constant with typical value in the range of 0.9 to 0.99. Algorithm 1 shows the steps of a standard simulated annealing process. A feasible solution is initially generated. Then a neighbourhood function ($generateNewSol$) is chosen to generate a new solution ($newSol$) which is accepted and used as a new starting point of the search if the Metropolis criterion is satisfied (Line 12). The probability of accepting worse solutions decreases with the temperature. Once a solution having lower cost than that of the best solution ($bestSol$) is found, this solution is recorded (Line 15 to 18). The search process is repeated iteratively until the termination condition is fulfilled such as the minimum temperature or maximum search iteration is reached.

3 Methodology

3.1 The challenges of algorithm mapping

Many-core platforms, such as GPU cards, are widely used as accelerators to enable fast implementations in many applications. It is expected that using many-core platforms can improve the run time of neighbourhood search algorithms. Thus, finding the solutions for bigger problems can be more practical. Furthermore, the many-core technologies should benefit the neighbourhood search algorithms in solution quality. Since massively parallel computing capability can be employed to explore the solution space more thoroughly, the probability of capturing good solutions is thus increased. However, there are some challenges to parallelise neighbourhood search algorithms on a many-core platform:

- Neighbourhood search algorithms are sequential searches in their original form. Operations in later iterations depend on the results of the previous one. So later iterations cannot start until the previous iteration is finished. This sequential nature makes it difficult to fully utilise the parallel many-core architecture. Although parallel search based neighbourhood search algorithms are proposed where a number of searches are performed in parallel, the achievable parallelism is limited by the memory hierarchy of the underlying many-core platform. For example, the standard simulated annealing approach needs to keep track of both the previous solution ($prevSol$) and the new solution ($newSol$). Since the memory resources on GPUs, in particular the fast shared memory, are limited, storing two solutions for each search constrains the number of parallel threads deployed on a GPU device.
- In each iteration of the neighbourhood search algorithm, solutions may be updated and the next iteration will start from the updated solutions. For example, in simulated annealing algorithm, the previous solution ($prevSol$) is updated based on the Metropolis criterion (Line 13 of Algorithm 1). Furthermore, the global best solution ($bestSol$) is updated when the new solution has a lower cost than that of the previous best solution (Line 16 of Algorithm 1). Frequent updates to both local and global solutions generate huge amount of memory access traffic. As a result, the memory bandwidth of many-core platforms becomes a bottleneck which again limits the parallelism of the neighbourhood search algorithm.

Previous study (Cope et al., 2010) has shown that mapping the originally sequential algorithm to a parallel architecture requires attention to both the characteristics of the platforms and the operations in the algorithms. Due to the low arithmetic operation to memory access ratio, we believe that the optimisation of memory access is essential for achieving high performance on the many-core platform. This will be discussed in Section 4.1.

3.2 Maximising parallelism with PSPM

A parallel search parallel move (PSPM) approach is proposed to fully utilise the computing capability of many-core platforms. It has three features, as shown in Figure 2.

- Multiple searches, each iteratively generates and evaluate moves, run in parallel. The processing units in a many-core platform can be grouped into multiple blocks, such that each block of processing units supports one search. During the search process, each search must keep a set of intermediate solutions. If a many-core platform provides user-managed on-chip memory, this set of solutions can be stored in the on-chip memory to improve performance, by avoiding overhead in time and power consumption when going off-chip. As the required memory size is proportional to the number of searches, running a large number of searches requires a significant amount of on-chip memory. On a platform with limited on-chip memory, the number of parallel searches will be restricted. On the other hand, storing solutions on the main memory can increase the number of parallel searches albeit at the cost of reduced performance.
- In each iteration of a search, multiple moves are generated and evaluated in parallel; each processing unit in a block covers one move and its evaluation.

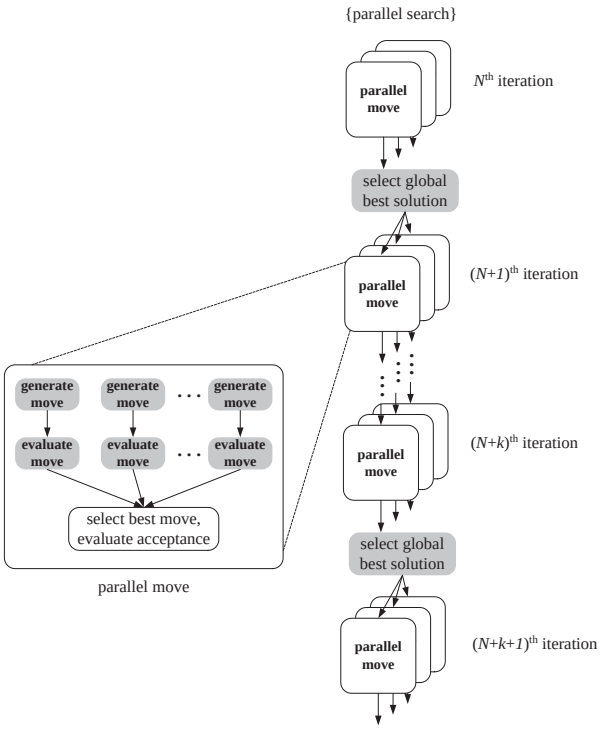

Figure 2 The proposed parallel search parallel move approach. Global best solution is chosen periodically for every k iterations.

Although multiple moves are generated in one iteration, only the move with the lowest cost can survive in each iteration of a search. The acceptance of this move depends on the parameters of the employed neighbourhood search algorithm, such as Metropolis criterion for simulated annealing. In theory, one copy of solution is required for each move. Using the application and platform specific optimisation techniques, shown in Section 4, we can reduce the memory requirement of parallel move to the size of a single solution. As a result, increasing number of moves does not increase the memory requirement for storing solutions. The number of moves is thus limited by the number of registers, which are normally used to store local variables, intermediate values, etc.

- The concurrent searches are coordinated periodically. For every k iterations where k is a fixed number (Figure 2), the best solution among all concurrent searches is chosen and used as a new starting point of all searches in the next iteration, and this process is repeated until the algorithm terminates, e.g. the termination temperature is reached in simulated annealing. The frequency of global synchronisation depends on a number of factors such as run time performance and solution quality. The impact of synchronisation frequency is highly problem specific.

In the PSPM approach, the parallel move process has a higher probability for capturing a better solution due to the increased sampling rate in the neighbouring solution space. This approach makes the search algorithm more greedy and results in a faster converging speed. On the other hand, the parallel search process can compensate the lost of diversity and avoid trapping in local optimums. Performing inter-search global synchronisations refocuses more computing power on a better solution from time to time. Reducing the synchronisation frequency to a single move will effectively transform the process to a single search parallel move approach. The advantage of the PSPM approach is that the achievable parallelism for the search process and for the move process can be adapted for various platform constraints. Two cases are as follows.

- If the on-chip memory size of a many-core platform constrains the achievable parallelism for the parallel search process, our approach can still increase parallelism by increasing the amount of parallel moves.
- If the achievable parallelism is bounded by register count, the amount of parallel moves can be limited. Our approach can then increase the amount of parallel searches which is bounded by the solution storage requirement.

As a result, the proposed PSPM approach allows more solutions to be explored by utilising the parallel computing resources of many-core architectures.

3.3 Application to solving TSP

In this work, the PSPM approach is applied to parallelise a simulated annealing algorithm for solving the Traveling Salesman Problem (TSP). It is a typical model for vehicle routing, circuit design and networking applications.

In the TSP model, a list of cities and their positions are given. The task is to find the shortest route to visit all cities and each city is visited once and once only. Our design runs a number of simulated annealing based parallel searches on the GPU device. Each search generates multiple TSP solutions in parallel in each iteration, where each solution is produced by randomly relocating a city to another position in the route. The path lengths of all generated TSP solutions are then calculated in parallel and the solution with the shortest length is chosen to test against the Metropolis criterion.

In our algorithm, each search records a local best solution which is the best solution this search can find. The local best solution may be updated according to the path length of the newly chosen solution. As a result, there are a number of local best solutions in this parallel algorithm. Periodically, the algorithm checks all the local best solutions to find the one with the shortest length and use it as the new starting point of the subsequent iterations of all the concurrent searches. Upon termination, the best TSP solution among all searches is reported as the final solution of the algorithm.

4 Application and platform specific optimisations

To organise the parallelism of the parallel search parallel move approach for the GPU platform, a parallel search is mapped to a block of threads and each move in the search

Algorithm 2: The proposed light weight simulated annealing (LWSA).

```
1  begin
2      T ⇐ T_initial;
3      curSol ⇐ feasible_initial_solution;
4      curCost ⇐ getCost(curSol);
5      bestSol ⇐ curSol;
6      bestCost ⇐ curCost;
7      while T > T_termination do
8          (relocate_info, newCost) ⇐
           tryNewSol(curSol);
9          p ⇐ exp((curCost − newCost)/T);
10         rand ⇐ random(0, 1);
11         if (rand < p) then
12             if (curCost < bestCost) and
               (newCost > curCost) then
13                 bestSol ⇐ curSol;
14                 bestCost ⇐ curCost;
15             end if
16             commitSol(curSol, relocate_info);
17             curCost ⇐ newCost;
18         end if
19         T ⇐ αT;
20     end while
21 end
```

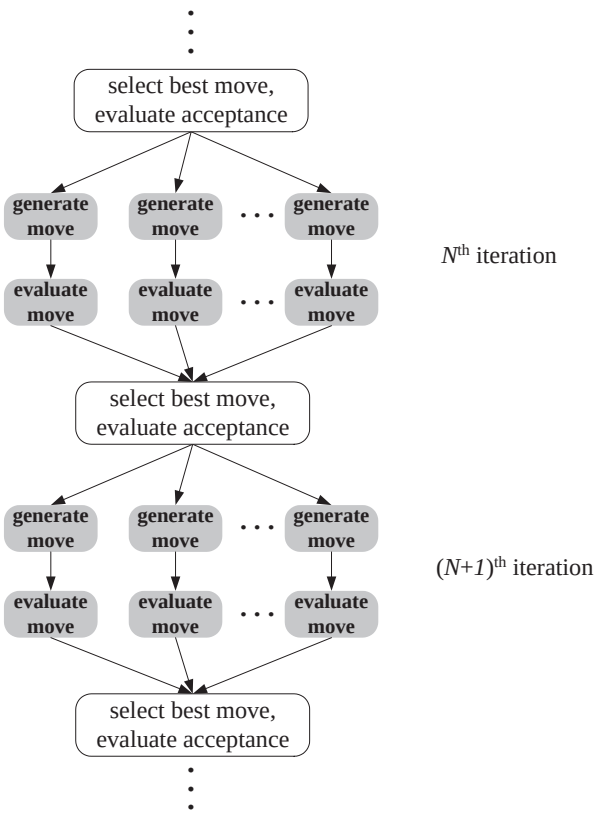

Figure 3 A single search parallel move approach.

is mapped to an individual thread within the block. Since a distinct copy of the solution is required for each search, the parallelism of parallel search is affected by memory storage size and memory access efficiency. The parallelism of parallel move is limited by the complexity of move process which affects the performance of running simultaneous threads. As a result, the achievable parallelism of parallel search parallel move is determined by the combination of the number of blocks and the number of threads per block. Since the global best solution is used as the starting point of all searches periodically, thread synchronisation is required to update and fetch the global best solution. It is modelled as a configurable parameter in the proposed PSPM approach and adjusted based on empirical feedbacks.

4.1 Memory optimisations

To address the size limitation of the shared memory and reduce memory access traffic, a light weight simulated annealing (LWSA) is proposed (Algorithm 2). Instead of storing two solutions in the standard simulated annealing approach, the new LWSA algorithm only stores one solution ($curSol$). Compared with the standard simulated annealing approach where a new solution is generated at the start of each iteration (Line 8 of Algorithm 1), the function $tryNewSol$ in LWSA does not generate a complete solution (Line 8 of Algorithm 2). It only records the city relocation information ($relocate\_info$) of the new solution and the corresponding cost ($newCost$). This $newCost$ is checked against the Metropolis criterion in Line 11. Based on the recorded $relocate\_info$, a new solution can be generated and stored into the $curSol$ by executing the function $commitSol$ if the Metropolis criterion is satisfied (Line 16 of Algorithm 2).

In contrast to the standard simulated annealing approach that generates and updates a new solution in every iteration (Line 8 of Algorithm 1), the LWSA algorithm only stores the relocation information which can be used to reconstruct the new solution (Line 8 of Algorithm 2). The solution is not updated unless the Metropolis criterion is satisfied. As a result, the number of solution updates in a search is reduced. In addition, the standard simulated annealing approach updates the local best solution immediately once a new solution with a lower cost is found. However, the local best solution is not updated immediately in the LWSA approach. The $curSol$, instead of the local best solution, is updated if a better solution is found. The local best solution is updated only when the next move will accept a worse solution and the $curSol$ is better than the stored local best solution (Line 12 of Algorithm 2). The idea is to skip the write operations to the local best solution in a sequence of better moves until a worse move is being accepted. By doing so, the frequency of memory updates, and thus the memory bandwidth requirement, is significantly reduced.

Table 1 A summary of notations.

symbol	description	adopted value
N_d	number of nodes in the TSP instance	100 to 493
P_m	number of parallel moves in the CPU SSPM implementation	12
P_s	number of parallel searches in the CPU PSSM implementation	12
S_e	number of temperature steps between best solution exchanges	100
S_s	total number of temperature steps performed in a complete search	
S_p	number of sweeps to reduce temperature	10 (Schneider and s. Kirkpatrick, 2006)
T	simulated annealing temperature	
T_s	starting temperature	10000 (Schneider and s. Kirkpatrick, 2006)
T_e	termination temperature	0.1 (Schneider and s. Kirkpatrick, 2006)
α	cooling constant	0.99
N_{sm}	number of streaming processors on GPU	14
M_s	shared memory size of a GPU streaming processor	48KBs
TB	number of thread blocks in GPU implementation	98
TPB	number of threads per block in GPU implementation	12

4.2 GPU implementation details

The following pseudo code segment shows the GPU implementation of the parallel simulated annealing algorithm for solving the TSP:

```
main() {
   read node coordinates;
   while (!terminate) {
     launch Tb*Tpb threads;
     find the global best solution among all threads blocks;
     copy the global best solution to all threads blocks;
   }
}

kernel_function {

  while (step < Se) {
    generate relocation information of a new solution
    from the thread block's shared solution;

    if (thread_id == 0) {
      find the thread in the block contains
      minimum cost solution;
    }

    if (thread_id == minimum_cost_thread) {
      generate a random number;
      calculate the acceptance probability;
      if (new solution is accepted) {
        if (block's shared solution is better than the
            best solution of the thread block && new
            solution is worse than block's shared solution) {

            store block's shared solution to the best solution
            of the thread block;
        }
        commit the thread block's shared solution
        using the relocation information;
      }

      if (iteration % (Nd * Sp) == 0) {
        step++;
        T = T * ALPHA;
      }
      iteration++;
    }
  }
}
```

The *main* function is responsible for parameter setting, launching the simulated annealing kernel and collecting the global best solutions. It iteratively launches $TB \times TPB$ threads which are divided into TB thread blocks and each block contains TPB threads. Each thread block implements a single search parallel move (Figure 3) simulated annealing process and each thread implements a LWSA. As a result, the $TB \times TPB$ threads implement a TB searches TPB moves (Figure 2) simulated annealing. All the launched threads run for S_e temperature steps where a temperature step is the temperature being decreased once. After the kernel call is finished, the global best solution among all thread blocks is selected. In the next iteration, all thread blocks continue the search from this selected solution. Iteratively, each thread in a block generates the relocation information of a new solution and calculates the path length. The thread that generates the best solution within each block is selected. This thread is then used to evaluate the acceptance of the solution and update the local best solution accordingly. Note that the temperature is not decreased in each iteration. It is decreased once every $N_d \times S_p$ iterations where N_d is the number of nodes in the TSP instance and S_p is the number of sweeps to reduce temperature (Schneider and s. Kirkpatrick, 2006). In this implementation, each thread block keeps a copy of the node coordinates and the current solution in the shared memory since they are accessed frequently. All threads in the block use these shared information to generate and evaluate new solutions. Each thread block updates the current solution only

when it is accepted. Putting the current solution and the coordinates in the shared memory can reduce memory access latency by utilising the high bandwidth of the shared memory.

In this work, the TSP solution is stored in the form of a double link list implemented in 1-D array structure. A node is represented as a pair of array indices, `prev` and `next`, of the previous and the next node in the path. Using the `short` type for the array index, each node occupies 4 bytes of memory. A complete solution thus requires $4N_d$ bytes where N_d is the number of nodes. Since each node requires two 32-bit floating point numbers to store the coordinate, $8N_d$ bytes are needed to store the coordinates for all nodes. As a result, the maximum number of thread blocks TB can be mapped on a GPU is determined as follows:

$$TB = \lceil \frac{M_s}{4N_d + 8N_d + r} \rceil \times N_{sm} \tag{3}$$

where M_s is the size of shared memory in a streaming multiprocessor and N_{sm} is the number of streaming multiprocessors in a single GPU. Parameter r is the number of bytes in shared memory for variables such as the path lengths of all moves in one iteration. A summary of all these notations is given in Table 1.

4.3 Multi-core CPU-based multi-threaded designs

To evaluate the performance of our PSPM implementation on GPU, two reference multi-threaded designs for multi-core CPU are implemented. For a fair comparison, these CPU-based implementations also use the proposed LWSA approach.

- *CPU SSPM*: Since it is inefficient to run a massive number of threads in the multi-core CPU, this multi-threaded CPU design is a single search parallel move approach (SSPM) as shown in Figure 3. Within a single search, P_m moves are generated and then evaluated in parallel by individual CPU threads.
- *CPU PSSM*: This implementation is a traditional parallel search single move (PSSM) scheme where each processing core performs an independent search and only one move is generated in each step. All processing cores are coordinated periodically to fetch the global best solution and use it as a new starting point of the search.

5 Experimental results

5.1 Test environment

The proposed PSPM simulated annealing design for many-core platform is tested on an NVIDIA Tesla C2050 GPU system. This GPU has 448 FPUs which are grouped into 14 streaming multiprocessors. There are 48K bytes of shared memory and 32K registers in each streaming multiprocessor (NVIDIA, 2009). The reference CPU-based implementation is tested on a server with two Intel Xeon X5650 CPUs at 2.67GHz. With 6 processing cores per CPU chip, the system has 12 physical CPU cores. Thus 12 parallel threads are created in the CPU-based reference implementation for generating and evaluating the solutions. The Intel C compiler is used to compile the multi-threaded programs with all optimisation options enabled. With private state memory, the random number generator provided in the Intel Maths Kernel Library is more suitable for parallel design. It is used in our implementation instead of the standard random number generator from the GNU C library. Experiments are performed on 20 problem instances from the TSPLIB benchmark suit (Reinelt, 1991) with 100 to 493 cities.

From the CUDA compiler message, the kernel function consumes 40 registers and 6206 bytes of shared memory. The number of concurrent thread blocks is bounded by the shared memory size but not the register count. Determined by Equation 3, the number of thread blocks created in the target GPU is 98 ($TB = 98$). The application and platform specific parameters used in the experiments are shown in Table 1.

5.2 Solution space exploration speed comparison

To compare the efficiency of the multi-threaded CPU implementation and the GPU-based PSPM implementation in solving the TSP, a solution space exploration speed (SSES) is introduced and defined as:

$$SSES = \frac{\frac{\log T_e - \log T_s}{\log \alpha} \times N_d \times S_p \times TB \times TPB}{T_r} \tag{4}$$

where T_r is the run time of a complete search. The $SSES$ merit measures the number of moves a particular approach can explore per second. A solution error coefficient (SEC) is used to measure the quality of the achieved TSP solution by a particular approach. It is defined as:

$$SEC = \frac{PL - PL_{opt}}{PL_{opt}} \times 100\% \tag{5}$$

where PL is the path length of the final TSP solution obtained by a particular approach and PL_{opt} is the path length of the optimal solution. A smaller SEC value means that the path length of the achieved solution is closer to the optimal solution and the approach performs better in term of solution quality.

Table 2 shows the $SSES$ and SEC values of the GPU-based and the CPU-based implementations where both are run from temperature 10000 to 0.1 (Schneider and s. Kirkpatrick, 2006). To reduce the effect of system fluctuations, each result in Table 2 is obtained by averaging the measured values for 10 independent runs of the same experiment. The optimal length for each TSP is provided by the TSPLIB benchmark suit. The $SSES$ speedup of the GPU implementation is calculated as:

$$GPU\ SSES\ speedup = \frac{SSES_{GPU}}{SSES_{CPU}} \tag{6}$$

As shown in the results, the GPU-based implementation outperforms both CPU-based implementations for all TSP

Table 2 Performance comparisons between CPU-based and GPU-based implementations. The SSES results are in unit of 10^6 solutions/second and the SEC results are in form of $\%$.

TSP instance	CPU SSPM		CPU PSSM		GPU PSPM		SSES speedup of GPU over CPU SSPM	SSES speedup of GPU over CPU PSSM
	SSES	SEC	SSES	SEC	SSES	SEC		
rd100	4.6	2.4	19.4	31.2	456.8	0.14	99.7	23.6
eil101	4.3	3.8	19.6	19.4	447.8	0.49	105.1	22.9
bier127	3.6	4.0	17.6	23.4	466.4	0.92	130.0	26.4
ch150	5.5	2.9	16.2	35.9	459.4	1.19	83.3	28.3
kroA150	4.2	5.1	16.1	39.5	461.5	1.08	109.2	28.6
pr152	4.6	1.9	16.1	75.8	466.6	0.17	102.2	29.0
u159	4.6	4.3	15.4	40.7	463.8	1.00	99.9	30.1
rat195	4.3	6.2	13.9	37.0	446.2	2.50	104.5	32.1
ts225	5.6	5.5	11.9	28.4	468.7	1.06	83.5	39.4
tsp225	5.4	4.6	12.1	35.2	447.9	2.37	82.2	36.9
gil262	4.0	6.1	11.3	50.7	447.0	2.19	110.4	39.7
pr264	4.8	11.2	11.3	116.0	462.1	4.20	96.7	40.7
a280	4.6	5.9	11.1	53.2	443.9	2.75	95.6	40.0
pr299	5.6	5.7	10.7	48.7	463.2	1.81	82.2	43.5
lin318	4.5	9.1	10.4	63.6	465.8	3.63	103.3	44.7
rd400	4.8	7.5	9.1	40.5	453.0	3.62	93.9	49.9
fl417	4.4	6.6	8.8	116.2	449.6	1.41	102.7	51.3
pr439	4.7	10.5	8.6	72.1	462.2	5.16	98.8	53.9
pcb442	4.5	7.5	8.4	43.5	458.2	3.81	101.0	54.3
d493	4.7	6.7	7.7	47.0	452.0	4.30	96.7	58.9
average	4.7	5.9	12.8	50.9	457.1	2.2	99.0	38.7

instances. When compared to the CPU-based SSPM, the GPU-based PSPM achieves speedups from 82.2 times to 130 times without quality degradation. The speedup variation is due to the differences in the problem size, the nature of the solution space, and solution update frequency. On average, a speedup of 99 times is obtained over the CPU-based SSPM. When compared to the CPU-based PSSM, the achieved speedups are ranged from 23.6 times to 58.9 times with an average of 38.7 times. This is lower than that in the SSPM case since the PSSM approach has a lower communication overhead. Despite having a better $SSES$ performance, the CPU-based PSSM produces worse solutions. The average error coefficient of the CPU-based PSSM is 50.9%, which is almost 9 times higher than the CPU-based SSPM.

5.3 Exploration effectiveness comparison

Exploring more solutions per unit time in the GPU-based PSPM design is only half of the story. A superior approach should find a solution not only quicker but also better. To compare the effectiveness of the GPU-based PSPM design and the CPU-based implementations, two sets of experiments are conducted.

- *throughput experiments:* This experiment measures the effectiveness of a particular implementation in finding a good enough solution. Shorter search time enables an approach to solve more TSP instances. For each TSP instance, we locate the shortest path length achieved by the CPU-based SSPM implementation in the experiments in Section 5.2. This shortest path length is used as the threshold length in this experiment. We then re-run the experiments for different implementations. The elapsed search time of each approach is recorded once it can find a solution with a path length shorter than the threshold. The average run time of 10 independent trials, RT_{avg}, is used to calculate the throughput coefficient (TC) as:

$$TC = \frac{1}{RT_{avg}} \tag{7}$$

Table 3 Effectiveness comparisons between CPU-based and GPU-based implementations. The TC results are in unit of 10^{-2} solutions/second and the SEC results are in form of $\%$. The convergence improvements are shown as the SEC differences of two particular implementations. (The TC results of CPU-based PSSM are not available due to excess search time.)

TSP instance	Throughput (TC)			Convergence (SEC)				
	CPU SSPM	GPU PSPM	GPU speedup	CPU SSPM	CPU PSSM	GPU PSPM	GPU improvement over CPU SSPM	GPU improvement over CPU PSSM
rd100	0.99	48.31	48.8	3.5	23.3	0.3	3.2	23.0
eil101	4.57	35.34	7.7	3.8	17.9	0.5	3.3	17.4
bier127	8.85	66.67	7.5	7.4	22.2	1.5	5.9	20.7
ch150	0.45	32.36	72.3	13.9	35.3	0.8	13.1	34.5
kroA150	1.97	43.67	22.1	7.5	35.6	0.9	6.6	34.7
pr152	0.26	29.85	115.4	1.9	64.5	0.6	1.3	63.9
u159	0.50	37.31	75.1	4.4	35.3	1.2	3.2	34.0
rat195	1.69	21.98	13.0	59.7	36.8	1.6	58.1	35.2
ts225	2.27	44.84	19.7	57.3	30.4	1.1	56.2	29.3
tsp225	0.09	19.80	223.2	42.0	33.9	2.8	39.2	31.1
gil262	1.58	16.13	10.2	54.5	45.6	2.8	51.7	42.8
pr264	0.93	24.04	25.9	41.4	104.6	6.1	35.3	98.5
a280	1.01	15.13	15.0	86.4	56.9	2.5	83.9	54.4
pr299	0.31	16.26	53.0	7.1	49.2	2.7	4.3	46.5
lin318	0.69	18.52	26.7	15.1	64.0	4.1	11.0	59.8
rd400	1.21	12.14	10.0	22.9	54.9	4.2	18.7	50.7
fl417	0.93	9.43	10.1	6.9	112.0	1.8	5.0	110.2
pr439	2.66	16.42	6.2	30.8	77.6	5.0	25.8	72.6
pcb442	1.39	13.02	9.4	8.9	41.8	4.2	4.7	37.5
d493	0.49	9.09	18.4	7.2	46.7	4.4	2.8	42.3
average	1.64	26.52	39.5	24.1	54.9	2.5	21.7	47.0

- *convergence experiments:* This experiment analyses the effectiveness of a particular implementation to improve the solution quality. For each TSP instance, the averaged GPU search time for reaching the threshold length in the throughput experiments is used as the maximum allowed search time. All implementations are run for this allowed time before they are stopped. The path lengths of the resulting solutions are measured. The SEC values of 10 independent runs are averaged to measure the convergence of solution error.

Table 3 shows the exploration effectiveness of the CPU and GPU implementations. Throughput results of the CPU-based PSSM design are not reported as this approach cannot reach the threshold length after spending 5 times longer search time than the CPU-based SSPM design. This indicates that the CPU-based PSSM is less capable of finding good solutions. The convergence experiments also show that this approach produces 2 times larger solution error than that of the CPU-based SSPM implementation. As a result, the analysis in this work will focus on the GPU-based PSPM and the CPU-based SSPM approaches.

The throughput measurements show that the GPU-based PSPM can achieve a 39.5 times average speedup. A maximum speedup of 223.2 times is obtained for the *tsp225* instance. Moreover, the proposed GPU-based PSPM approach is more effective in improving the solution quality. The convergence experiments show that the GPU implementation achieves quality improvement of 21.7% on average over the CPU-based SSPM design. The fluctuations in throughput and solution quality are due to the random factors in the neighbourhood search algorithm and the variations of solution space in different TSP instances.

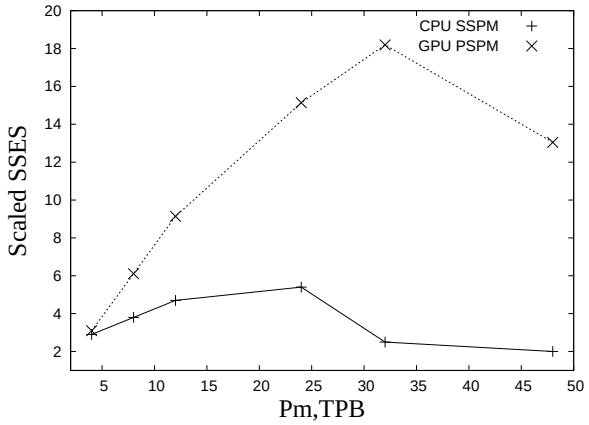

Figure 4 Scalability comparison between the CPU-based SSPM and the GPU-based PSPM implementations. The SSES coefficients of GPU implementation is divided by 50 to have a better illustration of the trend, the CPU SSES coefficients are absolute values from Table 4.

Table 4 Scalability comparison between the CPU-based SSPM and the GPU-based PSPM implementations. The SSES results are in unit of 10^6 solutions/second and the SEC results are in form of $\%$.

P_m,	CPU SSPM		GPU PSPM		SSES speedup
TPB	SSES	SEC	SSES	SEC	of GPU
4	2.9	9.0	154.8	2.2	55.2
8	3.8	6.5	305.5	2.1	82.1
12	4.7	5.9	457.1	2.2	99.0
24	5.4	4.2	756.8	1.5	142.6
32	2.5	4.0	909.7	1.4	**369.5**
48	2.0	3.6	652.2	1.1	320.6

5.4 Scalability

The scalability of the GPU-based PSPM implementation and the CPU-based SSPM implementation is compared in Table 4. In the experiments, P_m and TPB are varied from 4 to 48. Figure 4 illustrates the scalability trend of both implementations where the SSES coefficients of the GPU implementation are divided by 50 to have a better illustration and the CPU SSES coefficients are absolute values from Table 4. Figure 4 shows that the solution exploration speed of the CPU-based SSPM implementation does not change linearly with the number of parallel threads Pm. The SSES of the CPU implementation drops significantly when the number of threads exceed 24. This is because the number of active threads is larger than that of the hardware supported simultaneous threads. This excess of threads introduces frequent context switching in the CPU and thus degrades the exploration speed.

The exploration speed of the GPU implementation increases linearly with TPB when $TPB \leq 32$. However, it also drops significantly when $TPB > 32$. It is because all threads in the same block can be grouped in a warp (Section 2.1) when TPB is less than 32. Hence all threads in a block are executed in parallel and the execution time of a block is equal to that of a warp. When $TPB > 32$, the threads have to be grouped into two or more warps which are executed in sequential order. As a result, the expected execution time is increased by 200% when $TPB = 48$. Since the number of threads processing data is also increased from 32 to 48 (which is 50% increment), the expected performance degradation is 25%. The experimental results show that the degradation in GPU SSES is 28.3%, which matches the above analysis. A maximum speedup of 369.5 times is achieved at $TPB = 32$.

5.5 Comparison of different approaches

Table 5 compares the achievements of our work with previous efforts on mapping neighborhood search algorithms on GPU platforms. The proposed GPU-based PSPM approach, with application and platform specific optimisations, achieves the highest speedup by the knowledge of the authors. The comparison is fair since both the CPU and GPU platforms used in the experiments are up-to-date and commonly available.

The theoretical peak performance of the GPU and CPU used in this work are 1.03T FLOPS (NVIDIA, 2010) and 128G FLOPS respectively. The two Intel X5650 CPUs contain 12 cores running at $2.67GHz$. Each core can perform 4 concurrent single-precision floating point operations per clock cycle (Barker et al., 2008). The theoretical peak CPU performance of our dual-socket server is $4 \times 12 \times 2.67\text{G} = 128\text{G}$ FLOPS. A direct comparison suggests that the GPU platform should be $1.03\text{T}/128\text{G} = 8$ times faster than the CPU platform. However, the measured results show that the GPU-based PSPM design achieves much higher speedup. This indicates that the proposed GPU-based PSPM scheme can improve the performance of the neighbourhood search algorithms beyond the difference of raw processing powers of the hardware platforms.

6 Conclusions

A generic parallel search parallel move (PSPM) approach for parallelising neighbourhood search algorithms optimised for many-core platforms is presented. A simulated annealing process is parallelised using this approach and implemented on a GPU platform for solving the Traveling Salesman Problem. Using the proposed PSPM approach with application and platform specific optimisations, the GPU implementation is able to find better solutions quicker than the multi-threaded CPU design. Experimental results show that the GPU implementation can explore the solution space 99 times faster. The GPU design is very effective since it can find good solutions 39.5 times quicker or improve the solution quality by 21.7% under the same time limit. Current and future work include extending the proposed approach to cover larger TSP instances, applying the techniques to

Table 5 Comparing results from different approaches of mapping neighbourhood search algorithms on GPU platforms.

approach	algorithm	application	CPU platform	GPU platform	Speed-up
(Choong et al., 2010)	simulated annealing	placement	Intel Core 2 Quad	GeForce GTX 280	25.3
(Choi and Liu, 2010)	simulated annealing	floorplanning	Intel Core 2 Due	Quadro FX5600	15.7
(Han et al., 2011)	simulated annealing	floorplanning	Intel Core 2 Quad	GeForce GTS 250	160
(Luong et al., 2010)	Tabu search	quadratic assignment	Intel Core 2 Due	GeForce 8600M GT	6.1
(Luong et al., 2010)	Tabu search	permuted perceptron	Intel Core 2 Due	GeForce GTX 280	25.8
(Fujimoto and Tsutsui, 2011)	genetic algorithm	TSP	Intel Core 2 Due	GeForce GTX 285	24.2
this work	**simulated annealing**	**TSP**	**Two X5650 (12 cores)**	**Tesla C2050**	**99**

different neighbourhood search algorithms, and providing automated neighbourhood search algorithm mapping for many-core platforms.

Acknowledgement The support of Macao Science and Technology Development Fund (Grant No. 058/2010/A) and UK Engineering and Physical Sciences Research Council is gratefully acknowledged.

References

Ababei, C. (2009). Speeding Up FPGA Placement via Partitioning and Multithreading. *International Journal of Reconfigurable Computing 2009*.

Allwright, J. R. A. and D. B. Carpenter (1989). A Distributed Implementation of Simulated Annealing for the Traveling Salesman Problem. *Parallel Computing 10*(3), 335–338.

Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho (2008). A Performance Evaluation of the Nehalem Quad-core Processor for Scientific Computing. *Parallel Processing Letters 18*(4), 453–469.

Choi, W. H. and X. Liu (2010). Case Study: GPU-based Implementation of Sequence Pair Based Floorplanning Using CUDA. In *Proceedings of the International Symposium on Circuits and Systems*, pp. 917–920.

Choong, A., R. Beidas, and J. Zhu (2010). Parallelizing Simulated Annealing-Based Placement using GPGPU. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 31–34.

Cohoon, J. P., W. N. Martin, and D. S. Richards (1991). Genetic Algorithms and Punctuated Equilibra in VLSI. *Parallel Problem Solving from Nature, Lecture Notes in Computer Science 496*, 134–144.

Cope, B., P. Cheung, W. Luk, and L. Howes (2010). Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study. *IEEE Transactions on Computers 59*(4), 433–448.

Crainic, T. G., M. Toulouse, and M. Gendreau (1995). Synchronous Tabu Search Parallelization Strategies for Multicommodity Location-Allocation with Balancing Requirements. *OR Spectrum 17*(2-3), 113–123.

Czech, Z. J. and P. Czarnas (2002). Parallel Simulated Annealing for the Vehicle Routing Problem with Time Windows. In *Proceedings of the Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pp. 376–383.

Fujimoto, N. and S. Tsutsui (2011). A Highly-Parallel TSP Solver for a GPU Computing Platform. *Numerical Methods and Applications, Lecture Notes in Computer Science 6046*, 264–271.

Gallego, R. A., A. B. Alves, A. Monticelli, and R. Romero (1997). Parallel Simulated Annealing Applied to Long Term Transmission Network Expansion Planning. *IEEE Transactions on Power Systems 12*(1), 181–188.

Han, Y., S. Roy, and K. Chakraborty (2011). Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning. In *Proceedings of the International Symposium on Quality Electronic Design*, pp. 1–7.

Kravitz, S. A. and R. A. Rutenbar (1987). Placement by Simulated Annealing on a Multiprocessor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6*(4), 534–549.

Li, J. M., X. J. Wang, R. S. He, and Z. X. Chi (2007). An Efficient Fine-grained Parallel Genetic Algorithm Based on Gpu-accelerated. In *Proceedings of the Network and Parallel Computing Workshops*, pp. 855–862.

Luong, T. V., L. Loukil, N. Melab, and E.-G. Talbi (2010). A GPU-based Iterated Tabu Search for Solving the Quadratic 3-dimensional Assignment Problem. In *Proceedings of the International Conference on Computer Systems and Applications*, pp. 1–8.

Luong, T. V., N. Melab, and E.-G. Talbi (2010). Large Neighborhood Local Search Optimization on Graphics Processing Units. In *Proceedings of the IEEE International Symposium on Parallel and Distrubuted Processing*, pp. 1–8.

Matsumura, T., M. Nakamura, D. Miyazato, K. Onaga, and J. Okech (1997). Effects of Chromosome Migration on a Parallel and Distributed Genetic Algorithm. In *Proceedings of the IEEE International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 357–361.

Mavroidis, I., I. Papaefstathiou, and D. Pnevmatikatos (2007). Hardware Implementation of 2-Opt Local Search Algorithm for the Traveling Salesman Problem. In *Proceedings of the IEEE/IFIP International Workshop on Rapid System Prototyping*, pp. 41–47.

NVIDIA (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi v1.1. Technical report, NVIDIA.

NVIDIA (2010). TESLA C2050/C2070 GPU Computing Processor Supercomputing At 1/10 the Cost. Technical report, NVIDIA.

NVIDIA (2011). *NVIDIA CUDA Programming Guide v4.1*.

Ram, D. J., T. H. Sreenivas, and K. G. Subramaniam (1996). Parallel Simulated Annealing Algorithms. *Journal of Parallel and Distributed Computing 37*, 207–212.

Reinelt, G. (1991). TSPLIB: A Traveling Salesman Problem Library. *ORSA Journal on Computing 3*, 376–384.

Schneider, J. J. and s. Kirkpatrick (2006). *Stochastic Optimization*. Springer-Verlag Berlin Heidelberg.

Whitley, D., S. Rana, and R. B. Heckendorn (1991). Optimization Using Distributed Genetic Algorithms. *Parallel Problem Solving from Nature, Lecture Notes in Computer Science 496*, 176–185.

Wong, T. T. and M. L. Wong (2006). Parallel Evolutionary Algorithms on Consumer-level Graphics Processing Unit. *Parallel Evolutionary Computations 22*, 133–155.