Algorithm Mapping to Reconfigurable Systems and Systems with Multiple Embedded Processors

by

Ioannis Mavroidis

A dissertation submitted in partial fulfillment of the
requirements of the degree of
Doctor of Philosophy

in the

Department of Electronic and Computer Engineering
of the
Technical University of Crete at Greece

Committee in charge:

Professor Dionisios Pnevmatikatos
Professor Apostolos Dollas
Professor Ioannis Papaefstathiou

December 2011

Abstract

Algorithm Mapping to Reconfigurable Systems and Systems with Multiple Embedded Processors

The Traveling Salesman Problem (TSP) is probably the most-studied combinatorial optimization problem of all time. TSP applications range from logistics, and job scheduling, to computing DNA sequences, designing and testing VLSI circuits, x-ray crystallography, and many others. Many researchers, both mathematicians and computer scientists, have attacked the TSP problem for decades resulting in a plethora of heuristics that offer a broad range of tradeoffs between running time and quality of solution. These heuristics are typically classified as either tour construction procedures that gradually build a feasible tour, or tour improvement procedures that try to optimize an existing tour by performing various tour modifications. Probably the best-known such tour modification is the 2-Opt.

In this thesis we attack the 2-Opt algorithm from a novel perspective and manage to uncover previously unknown fine-grain parallelism. We propose a baseline architecture that exploits this type of parallelism and demonstrate the implementation of various versions of this architecture on an FPGA as well as on a multi-threaded GPU. Our algorithm guarantees the 2-Optimality of the final resulting tour.

We evaluate our implementations and find that the FPGA implementation manages to outperform Concorde, the current state-of-the-art software implementation, for small-scale TSP problems, in both speed and quality of final results. The GPU implementation is able to handle bigger-scale TSP problems and achieve similar quality of final results, but lags behind Concorde in speed.

Contents

List of Figures

Chapter 1. Introduction

The Traveling Salesman Problem (TSP) is the problem of a salesman who starts from his hometown and wants to visit a specified set of cities, returning to his hometown at the end. Each city has to be visited exactly once and the requirement is to find the shortest possible tour. Stated more formally, the TSP seeks the shortest Hamiltonian cycle in a weighted graph whose vertices correspond to cities and edge weights correspond to distances between cities. In this thesis we will concentrate on the symmetric TSP, in which, going from city A to city B has the same distance/weight as going from city B to city A. More specifically, we will concentrate on very widely used fully-connected Euclidean instances of the TSP, where each city is represented by its two coordinates.

The symmetric TSP is NP-hard and is one of the best-known and most-studied combinatorial optimization problems. It has many practical applications ranging from CAD tools for VLSI chip implementation to DNA mapping and X-ray crystallography.

Since finding the tour with the minimum length is an NP-hard problem, most algorithms concentrate on finding near-optimal tours as quickly as possible. Local search algorithms start from an initial ordering of the cities and attempt to improve this ordering by performing simple tour modifications. Each such algorithm has a specified set of allowed tour modifications (or moves) that it can use to convert one tour into another, and will repeatedly perform these modifications, as long as each reduces the length of the current tour, until no further improvement can be made (i.e. a locally optimal tour has been reached). However, a locally optimal tour may not necessarily be close to the globally optimal tour. In order to escape from local minima, we may want to modify this basic scheme of pure optimization and also allow

"uphill" moves in our search for the global minimum. Simulated annealing [6] is a well-known algorithm that does just that. It allows "uphill" moves based on a carefully crafted probability function.

The most famous tour modification used in heuristic algorithms for the TSP is the 2-Opt move, described in detail in the next Chapter. The main contributions of this thesis are:

- We introduce a novel idea that uncovers fine-grain parallelism in the 2-Opt algorithm. Even though TSP is probably the most-studied combinatorial optimization problem of all time, and 2-Opt its most famous heuristic approach to solving it, to the best of our knowledge, this is the first time that this type of parallelism is uncovered and studied.
- We propose a novel architecture to exploit this newly-uncovered parallelism, and demonstrate its implementation in reconfigurable hardware. We evaluate our proposed architecture and its implementation on an FPGA using a subset of the TSPLIB benchmark. Our implementation is one of the very few implementations in reconfigurable hardware (and in hardware in general) of a TSP solver. Moreover, this is to the best of our knowledge, the most efficient TSP solver for TSP instances up to a few hundred cities using the 2-Opt algorithm, since it is on average 600% faster than Concorde, the state-of-the-art software implementation, while it also produces better quality (i.e. closer to the optimal) results.
- We modify the algorithm that we used in our hardware implementation to better adapt it to a multi-threaded implementation. In contrast to the hardware implementation that is silicon-limited since the amount of hardware used is dependent on the number of cities, the multi-threaded implementation is able to employ thousands of threads, simultaneously running in software, and thus achieve higher levels of parallelism and work on bigger-scale TSP problems. We evaluate this modified algorithm and its implementation on a contemporary GPU by NVIDIA that is able to support up to 30K threads, and present comparison results against Concorde.

Chapter 2. Background on TSP

In the traveling salesman problem, or ‘‘TSP’’ for short, we are given a set {C1, C2, …, Cn} of cities and for each pair {Ci, Cj} of distinct cities a distance d(Ci, Cj). Our goal is to find an ordering m of the cities that minimizes the quantity

$$\sum_{n-1}^{i=1} d(C_{m(i)}, C_{m(i+1)}) + d(C_{m(n)}, C_{m(1)})$$

This quantity is referred to as the tour length, since it is the length of the tour a salesman would make when visiting the cities in the order specified by the permutation, returning at the end to the initial city. We shall concentrate in this thesis on the symmetric TSP, in which the distances satisfy d(Ci, Cj) = d(Cj, Ci) for any two cities Ci, Cj.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. Many researchers, both mathematicians and computer scientists, have attacked the TSP problem for decades resulting in a plethora of heuristics that offer a broad range of tradeoffs between running time and quality of solution.

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

The TSP is NP-hard and so any algorithm for finding optimal tours must have a worst-case running time that grows faster than any polynomial. For this reason research has focused in looking for heuristics that merely find near-optimal tours, but do so quickly. Some of the most well known heuristics are presented below.

2.1 The Held-Karp Lower Bound

When evaluating the empirical performance of heuristics, we are often not allowed the luxury of comparing to the precise optimal tour length, since for large instances we typically do not *know* the optimal tour length. As a consequence, when studying large instances it has become the practice to compare heuristic results to something we can compute: the lower bound on the optimal tour length due to Held and Karp.

This bound is the solution to the standard linear programming relaxation of the TSP. For instances of moderate size it can be computed exactly using linear programming, although if one goes about this directly one is confronted with a non-trivial computation: the number of constraints in the linear program is exponential in N. A more practical approach is to solve a sequence of restricted linear programs (LP's), each involving only a subset of the constraints, and to use a separation subroutine to identify violated constraints that need to be included in the next LP. This approach has been implemented using the Simplex method to solve the linear programs. Exact values for the bound have been computed in this way for instances as large as 33,810 cities. For larger instances, we settle for an approximation to the Held-Karp bound (a lower bound on the lower bound) computed by an iterative technique proposed in the original Held-Karp papers and sped up by a variety of algorithmic tricks.

The Held-Karp bound appears to provide a consistently good approximation to the optimal tour length. From a worst-case point of view, the Held-Karp bound can never be smaller than 2/3 of the optimal length (assuming the triangle inequality). In practice, it is typically far better than this, even when the triangle inequality does not hold. A HK lower bound averages about 0.8% below the optimal tour length, even though its guaranteed lower bound is only 2/3 of the optimal tour.

2.2 Tour Construction Algorithms

Tour construction algorithms build a solution (tour) from scratch by a growth process (usually a greedy one) that terminates as soon as a feasible solution has been constructed. The four tour construction heuristics that we will briefly discuss here are Nearest Neighbor, Greedy, Nearest Insertion, and Christofides. Each of these has a particular significance in the context of local search. The first three provide plausible mechanisms for generating starting tours in a local search procedure, and interesting lessons can be learned by evaluating them in this context. The fourth represents in a sense the best that tour construction heuristics can currently do, and so it is a valuable benchmark.

2.2.1 Nearest Neighbor

Perhaps the most natural heuristic for the TSP is the famous Nearest Neighbor algorithm (NN). In this algorithm one mimics the traveler whose rule of thumb is always to go next to the nearest as-yet-unvisited location. The algorithm works as follows:

1. Select a random city.
2. Find the nearest unvisited city and go there.
3. Are there any unvisitied cities left? If yes, repeat step 2.
4. Return to the first city.

The Nearest Neighbor algorithm will often result in tours within 25% of the Held-Karp lower bound.

2.2.2 Greedy

In this heuristic, we view an instance as a complete graph with the cities as vertices and with an edge of length d(Ci, Cj) between each pair {Ci, Cj} of cities. The Greedy heuristic gradually constructs a tour by repeatedly selecting the shortest edge and adding it to the tour as long as it doesn’t create a cycle with less than N edges, or increases the degree of any node to more than 2. The algorithm works as follows:

1. Sort all edges.

2. Select the shortest edge and add it to our tour if it doesn't violate any of the above constraints.
3. Do we have N edges in our tour? If no, repeat step 2.

The Greedy algorithm normally keeps within 15-20% of the Held-Karp lower bound.

2.2.3 Insertion Heuristics

Insertion heuristics are quite straightforward, and there are many variants to choose from. The basics of insertion heuristics is to start with a tour of a subset of all cities, and then insert the rest one at a time by some heuristic. The initial subtour is often a triangle or the convex hull of the cities. One can also start with a single edge as the initial subtour. The algorithm works as follows:

1. Select the shortest edge (or the convex hull of the cities), and make a subtour of it.
2. Select a city not in the subtour, having the shortest distance to any one of the cities in the subtour.
3. Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat step 2 until no more cities remain.

2.2.4 Christofides

Most heuristics can only guarantee a worst-case ratio of 2 (i.e. a tour with twice the length of the optimal tour). Professor Nicos Christofides extended one of these algorithms and concluded that the worst-case ratio of that extended algorithm was 3/2. This algorithm is commonly known as Christofides heuristic. The algorithm works as follows:

1. Build a minimal spanning tree from the set of all cities.
2. Create a minimum-weight matching (MWM) on the set of nodes having an odd degree. Add the MST together with the MWM.
3. Create an Euler cycle from the combined graph, and traverse it taking shortcuts to avoid visited nodes.

Christofides algorithm normally keeps within 10% of the Held-Karp lower bound for random Euclidean instances.

2.3 Tour Improvement Algorithms

Once a tour has been generated by some tour construction heuristic, we might wish to improve that solution. This is what *tour improvement* algorithms, or *local search* algorithms, attempt to do. There are several ways to do this, but the most common ones are the 2-opt and 3-opt local searches.

Other ways of improving our solution is to do a *tabu search* using 2-opt and 3-opt moves. *Simulated annealing* also uses these moves to find neighboring solutions. *Genetic algorithms* generally use the 2-opt move as a means of mutating the population.

2.3.1 2-Opt

The most famous and widely used tour modifications by local search algorithms are 2-Opt, 3-Opt, and in general k-Opt moves. A 2-Opt move deletes two edges, thus breaking the tour into two parts, and then reconnects those paths in the other possible way (see Figure 1). This is equivalent to reversing the order of the cities between the two edges, thus a 2-Opt move can be seen as a segment reversal, and will be treated as such in this thesis.

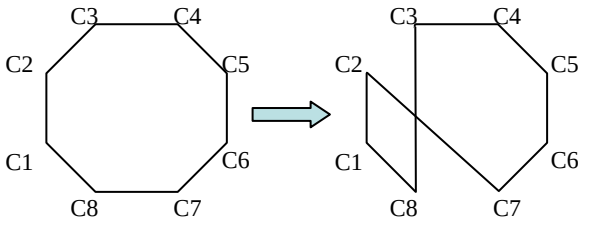


Figure 1. Applying a 2-Opt move.

Each segment reversal consideration need only take into account the four cities at the segment boundaries, no matter how long the segment is (since all internal cities keep their relative order).

For example, the difference in tour lengths that results from applying the move of Figure 1, can be calculated as:

Equation 1. Tour length difference after applying 2-opt move

$$\text{Delta(length)} = \text{dist(C2, C7)} + \text{dist(C3, C8)} - \text{dist(C2, C3)} - \text{dist(C7, C8)}$$

where dist(A, B) is the Euclidean distance between cities A and B, i.e. dist(A,B) = sqrt[(Ax – Bx)^2 + (Ay-By)^2]

If Equation 1 turns out to be negative, this is a length-reducing move and should be applied to the current tour.

The algorithm ends when there are no more length-reducing 2-opt moves to apply to the current tour, i.e. when we have reached a 2-optimal tour.

2.3.2 3-Opt, k-Opt, V-Opt

The 3-Opt algorithm works in a similar fashion, but instead of removing two edges we remove three. However in the case of 3-opt we now have two ways of reconnecting the three paths into a valid tour. A 3-opt move can actually be seen as two or three 2-opt moves.

We don’t necessarily have to stop at 3-opt, we can continue with 4-opt and so on (in general, k-opt for any positive number k), but each of these will take more and more time and will only yield a small improvement on the 2- and 3-opt heuristics.

Whereas the k-opt methods remove a fixed number (k) of edges from the original tour, the variable-opt methods do not fix the size of the edge set to remove. Instead they grow the set as the search process continues. The best known method in this family is the Lin–Kernighan method (mentioned above as a misnomer for 2-opt). Shen Lin and Brian Kernighan first published their method in 1972, and it was the most reliable heuristic for solving travelling salesman problems for nearly two decades. More advanced variable-opt methods were developed at Bell Labs in the late 1980s by David Johnson and his research team. These methods (sometimes called Lin–Kernighan–Johnson) build on the Lin–Kernighan method, adding ideas

from tabu search and evolutionary computing. V-opt methods are widely considered the most powerful heuristics for the problem.

2.3.3 Theoretical Bounds on Tour Improvement Algorithms

Concerning worst-case behavior, if we assume an adversary is allowed to choose the starting tour, the best performance guarantee possible for 2-Opt is a ratio of at least (1/4)*N^(1/2), and for 3-Opt it is at least (1/4)*N^(1/6). More generally, the best performance guarantee for k-Opt assuming the triangle inequality is at least (1/4)*N^(1/2*k). In practice, of course, we can obtain significantly better worst-case behavior simply by using a good heuristic to generate our starting tours.

However, average-case behavior, is usually much better even though at present we do not know how to prove tight bounds on the expected performance ratios for TSP heuristics such as 2-Opt and 3-Opt. If a Greedy heuristic is used to construct our starting tour, 2-Opt usually keeps within 5% of the Held-Karp lower bound, and 3-Opt within 3% of HK for random Euclidean instances.

2.4 TSPLIB and Concorde

TSPLIB [1] provides TSP instances that are often used as benchmarks to evaluate the performance (in terms of both speed and quality of results) of the different heuristics. TSPLIB contains instances with as many as 85,900 cities, including many from printed circuit board and VLSI applications, as well as geographical instances based on real cities. A good comparative survey and up-to-date picture of the state of the art in TSP heuristics can be found at the DIMACS Implementation Challenge [2] site.

Concorde [3] gathers many of these highly-optimized heuristics, including 2-Opt, in a single package. Several tricks are used to speed-up 2-Opt [4][5]. Heavy pruning of legal moves, preprocessing to construct k-d and other types of trees out of the cities, “don’t look” bits, caching etc. are some of them. As a result of all this, the algorithm runs in O(N) time (where N is the number of cities) and is actually very fast for small TSP instances.

Chapter 3. Related Work

In this thesis we uncover previously unknown fine-grain parallelism in the 2-Opt algorithm and present two implementations that take advantage of this newly-found parallelism to speed-up the 2-Opt algorithm, one on reconfigurable hardware, and one on a multi-threaded GPU.

We will now discuss previous attempts to parallelize the 2-Opt algorithm in software, or implement a TSP solver in hardware or on a multi-threaded processor such as a GPU.

3.1 Parallel 2-Opt and 3-Opt

One often-mentioned method for speeding up TSP algorithms is the use of parallelism. Various schemes have been proposed for this, and we will discuss the three most common approaches.

The first two approaches partition the problem into multiple CPUs in either a geometric or a tour-based fashion, and then combine the results back into one solution for the initial problem. This coarse-grained parallelism is unrelated and orthogonal to the fine-grain parallelism that is uncovered by our work. One could actually use our implementations to speed-up the solution of the subproblems produced by these approaches.

The third approach discusses the inherent parallelism in the search of tour-improving 2-Opt moves. Again, our solution drastically differs from this approach since, apart from searching 2-Opt moves in parallel, our idea also allows for the parallel application of any number of tour-improving 2-Opt moves.

3.1.1 Geometric Partitioning

For 2-dimensional geometric instances, a partitioning scheme proposed by Karp in 1977 can be used. It is based on a recursive subdivision of the overall region containing the cities into rectangles, with the set of cities corresponding to a given rectangle being comprised of all the cities in the rectangle's interior together with some of the cities on its boundary. Suppose we wish to construct a partition in which no set contains more than K cities. The recursive step of the partitioning scheme works as follows. Let CR be the set of cities assigned to rectangle R, and suppose ïCRï > K. One subdivides R into two subrectangles as follows. Suppose without loss of generality that the x-coordinates of the cities in CR have a larger range than the y-coordinates. Find a city c in CR whose x-coordinate has the median value for cities in CR. Divide R into two subrectangles R1 and R2 by drawing a vertical line through c, letting R1 be the rectangle to the left of the line. Each city in CR to the left of the line is assigned to R1, each city to the right is assigned to R2, and cities on the line are divided as equally as possible between the two rectangles, except that city c itself is assigned to both R 1 and R2.

Once the cities have been partitioned in this way into subrectangles, none of which has more than K cities assigned to it, one can send each subrectangle to a processor, and have that processor run 2- or 3-Opt (or any other TSP algorithm) on the corresponding set of cities. The union of the tours thus found will be a connected Eulerian subgraph, and so it can be converted to a tour for the entire instance by using shortcuts as in the Christofides algorithm.

However, even though geometric partitioning can certainly speed-up the local search algorithm, one should also expect a loss in final tour quality. The greater the number of partitions we use, the greater the loss in final tour quality. A rule of thumb seems to be that as soon as the number of cites in a subproblem drops below 1000, one can expect significant deterioration.

3.1.2 Tour-Based Partitioning

In tour-based partitioning, one begins by using a simple heuristic to generate an initial tour and then breaks that tour up into k segments of length N / k, where N is the number of cities and k is greater than or equal to the number of processors available. Each segment is then handed to a processor, which converts the segment into a tour by adding an edge between its endpoints and attempts to improve the tour by local optimization (2-Opt, 3-Opt, etc.), subject to the constraint

that the added edge cannot be deleted. The resulting tour can thus be turned back into a segment with the same two endpoints, and the improved segments can then be put back together into a new overall tour. We can then construct a revised partition where each new segment takes half its cities from each of two adjacent old segments and repeat the parallel local optimization phase. Additional phases can be performed until a time limit is exceeded or no significant further improvement is obtained.

There still appears to be a tour-quality penalty for partitioning in this way. Despite the shifting of the segment boundaries, it remains difficult to move a city very far away from its original position in the tour.

3.1.3 Using Parallelism in the Search of Improving Moves

The search for improving moves typically dominates the algorithm time and offers ample opportunities for parallelism. For example, when neighbor-list 3-Opt is applied to a random Euclidean instance, we typically evaluate 50 or more moves for every move actually made. There is no reason why these searches cannot be performed in parallel. Each processor may need access to the entire instance and all the neighbor lists, but assuming there is enough memory so that all this information can be replicated at each processor, significant reductions in running time should be possible.

3.2 Implementations on Hardware

Almost all proposed algorithms and literature focus on software implementations of TSP heuristics. As far as we know, there has been very little work done on how these algorithms could be ported to hardware. A couple of hardware implementations of genetic algorithms for the TSP can be seen in the references [7][8], but they are slower than our proposed solution since they yield significantly less parallelism.

3.3 Implementations on GPU

Recently, there has been an increased interest in implementing a parallel TSP solver on GPUs. Most GPU-based approaches to the TSP use the Max-Min Ant System (MMAS) algorithm [11].

The algorithm simulates the behavior of individual ants, which constructs tours around a graph based on the strength of evaporating pheromone trails left by other ants.

Bai et al. detail a GPU-based implementation of the parallel MMAS algorithm in which multiple ant colonies are simulated concurrently on the GPU, one for each thread block, with the tours of individual ants within each colony also parallelized [12]. This implementation achieves up to a 32x speedup over a serial CPU version under the same workload. Jiening et al. present a C++ and Cg implementation of the MMAS algorithm with up to a 1.4x speedup over the CPU implementation, which finds the optimal tour on a 30-city input [13]. You describes a CUDA implementation of a parallel ACO algorithm [14], with each thread on the GPU responsible for the travel of a single ant from a unique starting location, achieving up to a 20x speedup over a serial CPU implementation. Cecilia et al. present several GPU-based, data-parallel strategies for both the tour construction and pheromone update stages of the ACO algorithm, achieving a 28x speedup for the tour stage and a 20x speedup for the pheromone update stage over sequential CPU code [15]. However, these works focus on how to efficiently implement an Ant System on a GPU and compare the results of their GPU-based implementation only against a serial ACO implementation without providing the run times or the tour lengths. Essentially, they target to show that “ACO is a potentially fruitful area for future research in the GPU domain”.

Molly et al. explain how to parallelize the iterative hill climbing algorithm for TSP for GPU-based execution [16]. The algorithm runs on a GPU chip 62 times faster than the corresponding serial CPU code. This paper proposes to use thousands of “climbers” working independently in order to find a locally optimal solution starting from a random point and performing best moves (i.e. moves that yield the maximum possible length reduction). Each climber uses separate memory. However, a GPU has limited shared memory and, as a result, the paper presents results for small problems including 100 cities. Moreover, this work does not take into advantage of the GPU’s high sequential memory access performance. Contrary to this work, in our implementation all the threads share the same memory so as to store thousands of cities in the GPU’s shared memory (Section 6.4). Moreover, all the threads are synchronized performing coalesced memory accesses.

There also exists a recent genetic algorithm-based TSP solver in CUDA, presented by Fujimoto and Tsutsui [17]. This work parallelizes TSP using the genetic crossover operator and 2-Opt local search. Their CUDA implementation on a GTX-285 is up to 24.2x faster than a

single-core CPU version, allowing an error ratio over the optimal trip cost of up to 0.5%. However, this work also provides results for small problems (less than 500 cities) and their running times are significant longer than ours.

Chapter 4. Symmetrical 2-Opt Moves

One of the main contributions of this thesis is the introduction of the notion of *symmetrical 2-opt moves*, a novel ideal that will help us uncover fine-grain parallelism in the 2-Opt algorithm. To the best of our knowledge this is the first time that this type of parallelism is uncovered and studied.

A local search algorithm that uses 2-Opt moves typically evaluates a significant number of moves before finding a move that reduces the length of the current tour. This is especially true as the algorithm approaches a local minimum solution, where hundreds or even thousands of moves might need to be evaluated before a length-reducing move can be found. These searches can be performed in parallel.

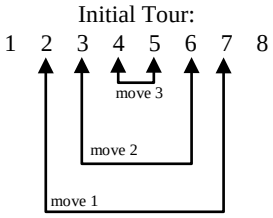
We define a set of segments (or equivalently their corresponding 2-Opt moves) as symmetrical segments (or symmetrical 2-Opt moves), if and only if:

For each segment SegmA of the set (except the smallest one), segment SegmB that consists of the same cities except the two cities at the boundaries of SegmA, is also part of the set. For example, if SegmA = {C10,...,C100} is in the set, then SegmB = {C11,...,C99} will also be in the set. Taking this further, if we assume that SegmA is the largest segment in the set, then the set will consist of segments {C10,...,C100}, {C11,...,C99}, {C12,...,C98}, etc., up to segment {C54, C56}.

Symmetrical 2-Opt moves are ideal candidates for parallel *evaluation and application* in hardware, for reasons that will become clear in the next paragraphs.

Figure 2 shows an example where three symmetrical 2-Opt moves are evaluated in parallel; the first one considers the reversal of the tour segment {2,...,7}, the second considers segment

{3,...,6}, and the third one considers segment {4, 5}. The Figure shows the starting tour, as well as the resulting tour after applying any subset of these moves. Underlined in the resulting tour are the cities that have changed positions, compared to the positions they had in the initial tour. For example, in the case where moves 1 and 2 are applied, segment {3,...,6} will be reversed twice, ending up at its initial position and orientation. Thus, in this case, only cities 2 and 7 will end up in different positions than their initial ones.



Moves applied		Resulting Tour:							
none		1	2	3	4	5	6	7	8
1		1	**7**	**6**	**5**	**4**	**3**	**2**	8
2		1	2	**6**	**5**	**4**	**3**	7	8
3	⇨	1	2	3	**5**	**4**	6	7	8
1,2		1	**7**	3	4	5	6	**2**	8
1,3		1	**7**	**6**	4	5	**3**	**2**	8
2,3		1	2	**6**	4	5	**3**	7	8
1,2,3		1	**7**	3	**5**	**4**	6	**2**	8

Figure 2. Example of three symmetrical 2-Opt moves

Looking at this Figure, the following observations can be made for a set of symmetrical 2-Opt moves:

1. After the application of any number of the symmetrical segment reversals under consideration, each city will either remain at its initial position, or swap positions with its "symmetrical" city (two cities are "symmetrical" if they are at the two ends of one of the symmetrical segments). For example, cities 2 and 7 of Figure 2 will either remain at their initial positions or swap positions (the same is true for cities 3 and 6, as well as cities 4 and 5).

2. Whether the cities at the two ends of a specific segment will stay put or will swap positions depends only on whether an even or odd number of segment reversals are applied on them. Therefore the number of the segment reversals can be counted by considering only the segments that these cities are part of. For example, in the case of Figure 2:

    - Cities 2 and 7 will swap positions iff segment {2,...,7} gets reversed.
    - Cities 3 and 6 will swap positions iff exactly one of the segments {2,...,7} and {3,...,6} gets reversed (not both).
    - Cities 4 and 5 will swap positions iff an odd number of the segments {2,...,7}, {3,…,6} and {4, 5} get reversed.

What these observations tell us is that, we are able to *apply* any subset of symmetrical 2-Opt moves in parallel, just by figuring out which cities need to be swapped. In what follows we will examine a hardware and a multi-threaded implementation that takes advantage of this *parallelism in applying* symmetrical 2-Opt moves. To the best of our knowledge, this is the first attempt to exploit this type of fine-grain parallelism in the 2-opt algorithm. Most attempts use a coarser-grain of parallelism, by partitioning the cities in a geometric or tour-based fashion among different processors, and then combining their results.

Chapter 5. Hardware Implementation

5.1 Algorithm and Architecture

In order to take advantage of the aforementioned parallelism, the proposed architecture splits all legal 2-Opt moves into groups of symmetrical 2-Opt moves. The moves of each such group are then evaluated and those that actually turn out to be length-reducing are applied in parallel.

Notice that in Figure 1, reversing tour segment {C3,…,C7} is equivalent to reversing the remaining tour segment {C8, C1, C2}. Thus, in our search for length-reducing segment reversals, we will only consider segments that consist of up to half the total number of cities.

As an example, let us consider an initial tour that consists of 200 cities. Figure 3 shows the proposed architecture for this example. Each city is represented by a register that holds its two coordinates, and the cities are put one next to the other to form a 200-entry circular shift register. The order of the cities in this shift register represents the current tour. Arrows in the Figure show all possible data movements (datapaths).

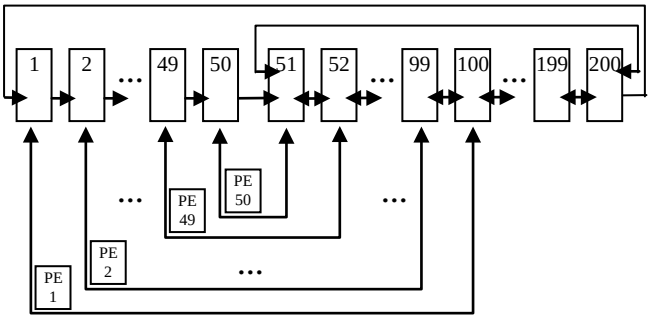


Figure 3. Architecture using 50 PEs for 200 cities

Since we have 200 cities we only need to consider segments of length at most 100. This is a total of 199*200/4 = 9950 such segments (half the combinations of picking 2 segment ends out of 200 cities). The proposed architecture will split these 9950 segments into 199 groups of 50 *symmetrical* segments each, and evaluate the segment reversals of each such group in parallel. The actual implemented algorithm executes the following tasks:

1. Firstly, it evaluates in parallel the set of 50 symmetrical segments {1,...,100}, {2,...,99}, {3,...,98}, etc., up to {50, 51}. These are all segments with even lengths of 2, 4, 6, etc., up to 100.
2. Secondly, it evaluates in parallel the set of 50 symmetrical segments {1,...,101}, {2,...,100}, {3,...,99}, etc. up to {50, 52}. These are all segments with odd lengths of 3, 5, 7, etc., up to 101.
3. Finally, the above two steps are re-applied so as to evaluate all remaining 197 groups of 50 symmetrical segments each.

For the first step, the 50 Processing Elements (PEs) shown in Figure 3 all run in parallel, each one evaluating one 2-Opt move. PE[1] evaluates whether the tour segment {1,...,100} should be reversed, PE[2] evaluates segment {2,...,99}, PE[3] evaluates segment {3,...,98}, etc., up to PE[50] which evaluates segment {50, 51}. In order for each PE to evaluate whether its 2-Opt move is length-reducing or not, as we discussed earlier, it needs to calculate Equation 1 of page 8 for the cities at its segment boundaries. For example, PE[2] calculates Equation 1 for cities 1, 2, 99 and 100. This is done in *O(1)* time (a few clock cycles) with the use of minimal hardware (a few multipliers and adders) per PE.

In this way, all PEs evaluate in parallel their 2-Opt moves, and when done, we will have 50 yes/no decisions for the 50 segment reversals under consideration. Next, we need to apply these decisions by swapping the necessary city pairs, as was explained in the previous Chapter. For example, cities 50 and 51 will swap positions iff PEs 1 to 50 have resulted in an odd number of "yes" decisions. (Similarly, cities 49 and 52 will swap positions iff PEs 1 to 49 have resulted in an odd number of "yes" decisions etc.) All necessary city swaps are done simultaneously in a single clock cycle, using the datapaths shown in the Figure.

Now, for the second step (segments with odd lengths 1, 3, 5, 7, etc., up to 101), in order to reuse the same wiring and PEs as before, we perform a left circular shift to segment {51,…,200} (see the corresponding datapaths in the Figure). In this way PE[1] will now be able to evaluate segment {1,…,101}, PE[2] will now evaluate segment {2,…,100}, PE[3] will evaluate segment {3,…,99}, etc., up to PE[50] which will evaluate segment {50,…,52}. After all these 50 new moves have been evaluated, we apply whichever turn out to be length-reducing by swapping the necessary cities as before (this time the swapping city pairs are different than before; city 50 with 52, city 49 with 53 etc.). Next, we perform a right circular shift on segment {52,…,200, 51} to restore city 51 to its original position. Notice that this scheme works since no matter how many of the segments under consideration are actually reversed, city 51 is in the centre of all these segments and will not change position.

For the third and final step, we perform a right circular shift to all 200 cities (see the corresponding datapaths in the Figure), and all the above tasks are again repeated from the beginning. In effect, the same wiring and PEs will now be used in a different position of the tour. This whole process is repeated until we can not find any length-reducing moves for 199 consecutive repetitions, since the algorithm is exhaustive and is guaranteed to search all possible tour segments in 199 repetitions (in contrast to randomized search algorithms that search 2-Opt moves at random and can not thus guarantee 2-Optimality of the final result).

Following is the pseudo-code for the control of the proposed architecture, which can be implemented with a simple Finite State Machine (FSM):

Repeat

1. PEs evaluate moves (segments with even lengths)
2. Apply length-reducing moves by swapping cities
3. Apply left circular shift to segment {51..200}
4. PEs evaluate moves (segments with odd lengths)
5. Apply length-reducing moves by swapping cities
6. Apply right circular shift to segment {52..200, 51}
7. Apply right circular shift to all cities

until done

Figure 4. Algorithm for the Hardware implementation

Notice that the proposed architecture exhibits minimal and fixed wiring with trivial control. Furthermore, notice that, since all move evaluations and applications are done in parallel, each iteration of the above pseudo-code loop (all 7 steps) is executed in *O(1)* time, independent of the total number of cities (200 in this case).

It is easy to see how this architecture scales to more cities, as long as there is enough silicon to accommodate the additional PEs and city registers. However, assuming that the PEs are expensive in terms of silicon consumption we would want to be able to scale to more cities while keeping the number of PEs constant. Thus, we would like to use the available PEs and their fixed wiring to evaluate all possible 2-Opt moves of any tour independent of its size. This in fact is possible with a small modification to our architecture. Figure 5 shows an example with 50 PEs and 1000 cities. Notice the added registers and data paths in comparison to the architecture of Figure 3.

In order to allow the PEs to work on larger segments we will use the same technique as before, i.e. perform left circular shifts to segment {51,…,1000} for a number of cycles which depends on the actual length of the segments under evaluation. For example, by performing 50 shifts, PE[1] will evaluate segment {1,…,150}, PE[2] will evaluate segment {2,…,149}, etc., up to PE[50] which will evaluate segment {50,…,101}. These segments have lengths of 150, 148, etc.,

down to 52 cities respectively. After all these new 50 moves have been evaluated and applied (by swapping the necessary cities), we need to restore segment {51,…,100} from the end of the 1000-city wide shift register back to its original position. This is done by performing 50 right circular shifts to segment {101,…,1000, 51,…,100}. This time however, there is an extra step since segment {51,…,100} might need to be reversed, in the case where cities 50 and 101 have been swapped. For this reason, we have added the extra set of registers (see Figure 5) that get loaded during the left circular shifts of segment {51,…,1000}, and will thus end-up holding the reverse of segment {51,…,100}. These registers are used as a final step to overwrite segment {51,…,100} with its reverse, if needed, in a single cycle.

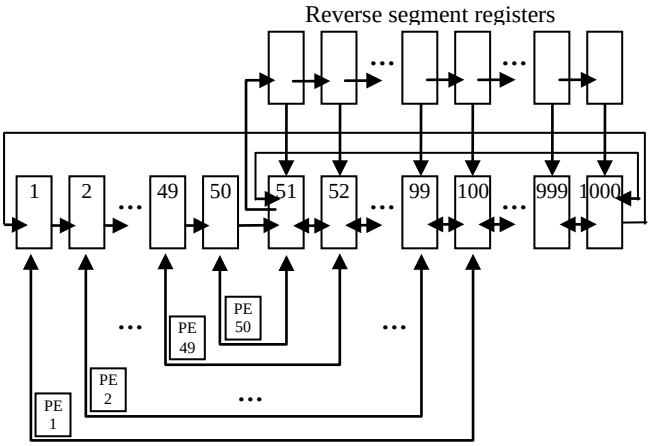


Figure 5. Architecture using 50 PEs for 1000 cities

Using the technique described we are able to use the same PEs and wiring to perform 50 2-Opt move evaluations per iteration. A total of 10 iterations would be needed to be able to evaluate moves of all segment lengths from 2 up to 500 (we do not need to evaluate bigger segments for 1000 cities). Then, a right circular shift should be applied to all the cities before the algorithm can be repeated.

5.2 Deterministic Nature of Architecture Algorithm

The proposed architecture evaluates 2-Opt moves in a strictly deterministic fashion, as opposed to a randomized local search algorithm that evaluates 2-Opt moves at random. A question that

arises is how this loss of randomness affects the time for the algorithm to converge to a locally optimal tour, as well as the quality of this resulting tour.

We implemented in software an *iteration-accurate* emulator of the proposed architecture (the version shown in Figure 3). The emulator executes the pseudo-code of Figure 4, searching the same groups of symmetrical 2-Opt moves in the exact same order as the architecture. We used this software emulator for all our performance measurements, for the following reasons:

- The final tour, as well as all the intermediate tours at the pseudo-code loop iteration boundaries (i.e. at the beginning of each iteration), matched exactly with the ones from the Verilog simulations (after turning the square root calculations of the emulator off). This validates that the emulator *evaluates and applies the exact same 2-Opt moves* as our architecture.
- Since the software emulator executes the exact same number of pseudo-code loop iterations as the implemented hardware model, and each iteration takes *O(1)* time to execute in hardware, the emulator is able to *accurately predict the performance* of a hardware implementation of the architecture, as long as it knows how long *O(1)* actually is.
- The software emulator runs orders of magnitude faster than a Verilog simulation.

In order to evaluate the effect of the deterministic nature of the architecture we also implemented in software a randomized version of the local search algorithm. This version evaluates 2-Opt moves at random, and converges after a sufficiently large number of sequential move evaluations (equal to the square of the number of cities) fail.

The two implementations were run against 70 TSPLIB instances (we used the Euclidean instances of “EUC_2D” type), *using the exact TSPLIB instances as our starting tours*, as opposed to first applying a greedy algorithm to them as is often done. The sizes of these instances ranged from 50 to around 4500 cities.

In these runs, our algorithm evaluating symmetrical 2-Opt moves converged, most of the times, slower than the algorithm that evaluated 2-Opt moves at random. Figure 6 provides a close look at the runtime behavior of the two algorithms for two specific TSPLIB instances (the results shown are typical at least for the set of TSPLIB instances that we used).

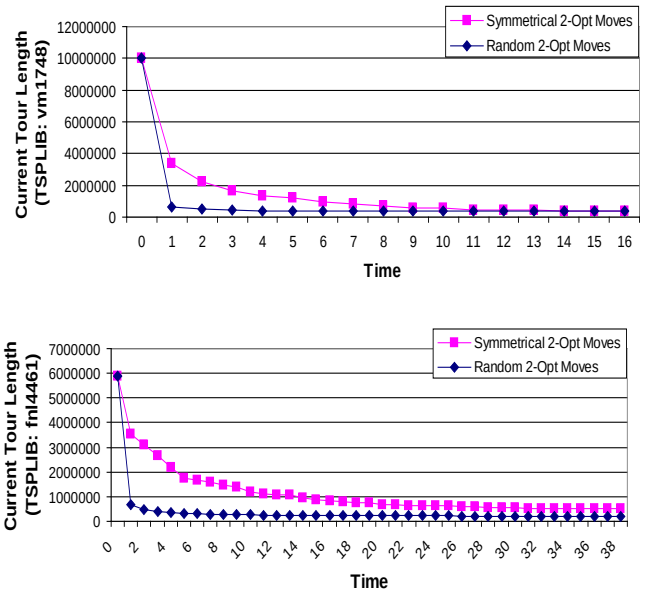


Figure 6. Runtime behavior of local search algorithm evaluating symmetrical or random 2-Opt moves

However, the running time of a hardware implementation of our deterministic algorithm, is expected to be orders of magnitude lower than the running time of a hardware implementation of the randomized algorithm (see Section 5.4 for detailed performance results). The reason is that the former was designed with such an implementation in mind.

As far as the final tour quality is concerned, the results of the two algorithms are pretty much the same. Our deterministic algorithm produced an average of 0.75% smaller final tours than the randomized algorithm.

5.3 FPGA-Based Implementation

Several instances of the first version of the proposed architecture (the one in Figure 3), with varying number of cities, were implemented in Verilog and synthesized for the Xilinx xc2vp100-6 Virtex II Pro FPGA.

For these implementations, the PEs did not include hardware to compute the square roots required for the distance calculations of Equation 1 – this is part of our ongoing and future work. For this reason, the Verilog simulations actually find the tour with the (locally) smallest sum of

squares of distances between the cities. However, as we discussed in Section 5.2, we have built an accurate software emulator of the architecture (that calculates all the required square roots) which allows us to accurately evaluate the architecture performance both in terms of running time and in terms of quality of resulting tours.

The main hardware units used, excluding pipeline registers, control and other miscellaneous logic, are the following:

- One 30-bit wide register per city to hold its X and Y coordinates (thus each coordinate can be up to 15 bits long, which was enough for all the TSPLIB instances that we use).
- One 16x16 multiplier, as well as one 16-bit and one 32-bit adder, per PE.

Having the PEs use the aforementioned hardware in a pipelined fashion, we were able to execute each iteration of the pseudo-code loop of Figure 4 in just 35 clock cycles, with each move evaluation taking 12 clock cycles.

Table 1 contains information about the area occupied by these implementations and their clock frequencies. The suffix of the TSPLIB instance name indicates the number of cities in that particular instance.

Cities (TSPLIB instance)	*Area (slices)*	*Multipliers (MULT18X18s)*	*Speed (MHz)*
berlin52	3295 (7%)	13 (2%)	184
eil76	4823 (10%)	19 (4%)	178
rd100	6332 (14%)	25 (5%)	184
ch150	9390 (21%)	37 (8%)	165
ts225	14143 (32%)	56 (12%)	171
pr299	18749 (42%)	74 (16%)	165

Table 1 Implementation Results for Xilinx xc2vp100-6 Virtex II Pro FPGA

5.4 Performance Results

As we discussed in Section 5.2, the architecture emulator is able to accurately predict the performance of a hardware implementation of the architecture, as long as it knows the duration of each iteration of the pseudo-code loop. For the performance results in this Section we have assumed that each iteration takes 35 clock cycles of a 150 MHz clock. Notice that after the hardware for the square root calculations is added to the PEs (in our future work), the number of cycles is expected to increase, which will have a negative impact to the performance numbers shown in this Section. However, the square root calculation can be performed by several estimation algorithms and thus the number of clock cycles needed is expected not to increase significantly.

We compared the performance of the hardware implementation of the architecture against two software implementations:

1. The architecture emulator.
2. A modified version of the 2-Opt local search algorithm implementation by Concorde. Concorde is widely considered as the state-of-the-art in TSP solving software, containing highly optimized implementations for the most important TSP algorithms and heuristics. We modified its 2-Opt algorithm implementation in the following way:
    - Concorde will by default first apply the greedy algorithm to the initial tour in order to obtain the starting tour for 2-Opt optimization. Since for the performance measurements of our architecture (and its software emulator) we used the exact TSPLIB instances as our starting tours, we had to modify Concorde so as not to apply the greedy algorithm to the initial tour.

Concorde runs very fast for the small TSPLIB instances that we consider. Thus, in order to more accurately measure its performance, we measured for each TSPLIB instance the average running time among 1000 runs.

Both software implementations were run on an Intel Pentium 3 GHz machine running RedHat Linux. Figure 7 shows the speed-ups obtained by the hardware implementation against its

software emulator, as well as the same speed-ups normalized to (i.e. divided by) the number of cities.

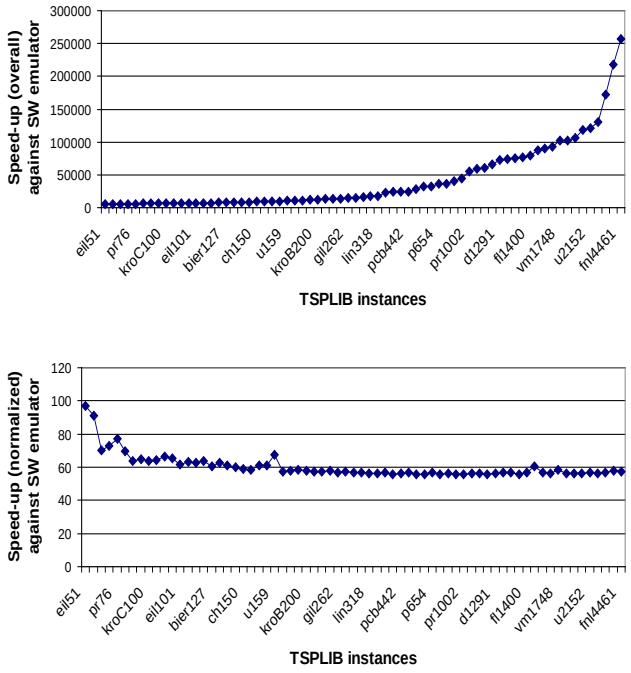


Figure 7. Overall and normalized speed-ups over software emulator (based on 35 150MHz ccs per iteration)

As expected the overall speed-up is proportional to the number of cities, since the latter reflects the number of PEs operating in parallel. The normalized speed-up of 60 seen in this Figure is attributed to the faster evaluation and application of 2-Opt moves in hardware over software.

Figure 8 compares the quality of the final tours obtained with our approach against the ones obtained by Concorde. Our architecture outperforms Concorde in final tour quality by an average of around 10%. This is expected since Concorde, in contrast to our architecture, gives up the guarantee of true 2-Optimality in favor of greatly reduced running time. The difference of 10% is exaggerated by the fact that we do not apply the greedy algorithm to our initial tours. However, it still shows the high quality of the tours produced by our architecture.

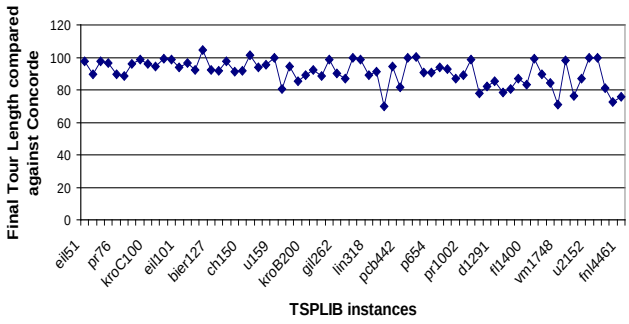


Figure 8. Final tour lengths achieved by our architecture as percentages of the ones achieved by Concorde

Figure 9 shows the speed-ups that our architecture exhibits against the state-of-the-art Concorde. We can see that our architecture exhibits an average speed-up of around 6 for the TSPLIB instances that we tried. Notice the huge difference between these speed-ups and the ones of Figure 7, which attests to the high quality of Concorde's heuristics and optimizations over our simplistic software implementations. Additionally, Concorde's performance certainly scales better with the number of cities than our implementations.

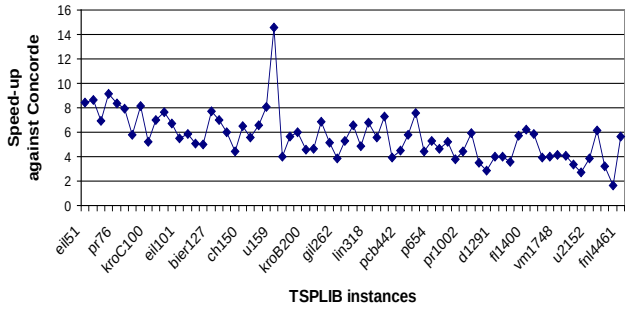


Figure 9. Speed-up over Concorde's 2-Opt implementation (based on 35 150MHz ccs per iteration)

Table 2 has a more detailed analysis of how the numbers for a small subset of the TSPLIB instances of Figure 9 were obtained.

Last but not least, we should note that greater speed-ups are achievable if, like Concorde, we give up the guarantee of true 2-Optimality. In our runs we observed that the algorithm using symmetrical 2-Opt moves converged to within 5% of the final locally optimal tour at usually

20% to 70% of its total running time. If we add a mechanism to stop running when small reductions in tour length are detected, we can gain additional speed-ups of 150% to 500%.

	Concorde	*Proposed Architecture*			
TSPLIB instance	Time (us)	Loop iterations	Clock cycles	Time (us)	Speed-up
eil51	460	234	8190	54.6	8.42
tsp225	1800	1129	39515	263.4	6.83
pr1002	5720	5576	195160	1301	4.40
rl1304	17380	18607	651245	4342	4.00
vm1748	30800	31747	1111145	7408	4.16
u2319	12330	8585	300475	2003	6.16
fnl4461	111350	84940	2972900	19819	5.62

Table 2 Comparison between Concorde and our proposed architecture

Chapter 6. Multi-Threaded Implementation

6.1 Algorithm

In comparison to the hardware implementation where each Processing Element (PE) is expensive in terms of silicon consumption, a multi-threaded implementation can provide us with thousands of threads and thus relatively cheap PE implementations. For this reason, the multi-threaded implementation uses a modified version of the hardware algorithm. The new version uses more PEs in order to achieve a higher degree of parallelism.

The hardware version of our algorithm from the previous Chapter would use (at most) N/4 PEs, where N is the number of cities, and would use them to first evaluate the segments of even length, and then the segments of odd length. In the modified version for the multi-threaded implementation we will be using N/2 PEs and evaluate all possible segments in one step. The new algorithm works as follows.

First of all, for reasons that will become clear later, the new algorithm works only for an odd number of cities. However, this is not a problem since if we have an even number of cities we can just duplicate the last one (thus resulting in an odd number of cities) and use that as our starting tour. Since our algorithm only applies length-reducing 2-opt moves, the two cities at the end of our starting tour (i.e. the last city and its duplicate) will not be separated and we can thus easily remove the duplicate city from our optimized tour once the algorithm has finished running.

To see this with an example, assume that we have an initial tour with 8 cities {1, 2, .., 8}. We append a duplicate of the last city at the end of the tour, to get {1, 2, …, 8, 8}. Now, for example, if we evaluate Equation 1 for segment {5, 6, 7, 8} we get:

Initial tour 1 2 3 4 5 6 7 8 8

Illegal move 1 2 3 4 8 7 6 5 8

$$Delta(length) = dist(4, 8) + dist(5, 8) – dist(4, 5) > 0$$

which, due to the triangle inequality, is positive. Thus reversing segment {5, 6, 7, 8} is not a length-reducing move. We can easily conclude that the two duplicate cities will remain adjacent after any number of length-reducing 2-opt moves is applied.

Since the number of cities is odd, reversing a tour segment of odd length, for example {5, 6, 7}, is equivalent to reversing the remaining tour segment {8, 8, 1, 2, 3, 4} of even length. Therefore the modified algorithm considers only segments of odd length, and considers all of them instead of considering segments that consist of up to half the total number of cities. In this way the new algorithm can evaluate N/2 symmetrical 2-Opt moves in parallel (for the 201 cities shown in Figure 10 it evaluates 100 segments in parallel) so as to support more parallelism and therefore require half the number of steps compared to the hardware implementation of the algorithm as well as to simplify its implementation.

In this Section we propose a parallel implementation of the symmetrical 2-Opt algorithm based on a generic thread-based platform that can support shared memory between the threads. This implementation will be the baseline for a multi-threaded GPU-based implementation. In order to take advantage of the aforementioned parallelism, the architecture splits all legal 2-Opt moves into groups of symmetrical 2-Opt moves of odd length. The moves of each such group are then evaluated and those that actually turn out to be length-reducing are applied in parallel.

As an example, let us consider an initial tour that consists of 201 cities. Each city is represented by a variable that holds its two coordinates, and the cities are put in an array one next to the other to form a 201-entry circular buffer as shown in Figure 10. The order of the cities in the buffer represents the current tour. Arrows at the bottom of the Figure show all possible data movements.

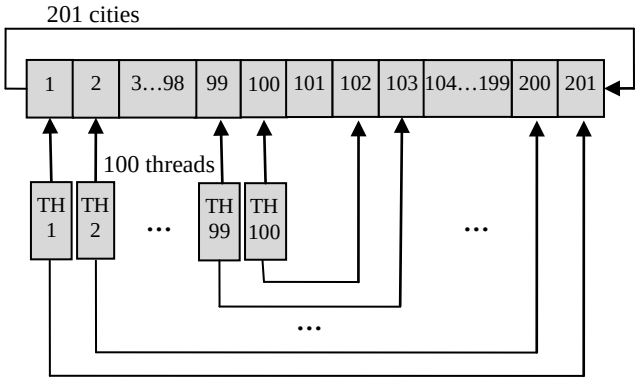


Figure 10. Implementation using 100 threads (TH1 to TH100) for 201 cities

Based on our previous reasoning, we only need to consider segments of odd length. This is a total of 200 * 200/2 = 20000 such segments (we also take into account the segments of 201 length in order to simplify the implementation). The proposed implementation will split these 20000 segments into 200 groups of 100 symmetrical segments each, and evaluate the segment reversals of each such group in parallel. The actual implemented algorithm executes the following steps at least 201 times:

1. Evaluate in parallel the set of 100 symmetrical segments {1,…,201}, {2,…,200}, {3,…,199}, etc., up to {100,…,102} using Equation 1 of page 8. These are all segments with odd lengths of 201, 199, 197, 195, etc., down to 3.
2. Combine the results of the first step performing XOR operations in order to decide whether a swapping is necessary or not for each city pair.
3. Apply in parallel the decisions from the second step by swapping the necessary city pairs.
4. Shift the circular buffer to the left in order to create a new group of 100 symmetrical segments {2,…,1}, {3,…,201}, etc., up to {101,…,103}.
5. Go to step 1, so as to evaluate the new group of segments.

Figure 11. Algorithm of the Multi-threaded implementation

For the first step, the 100 threads (TH1,…,TH100) shown in Figure 10 all run in parallel, each one evaluating one 2-Opt move. TH1 evaluates whether the tour segment {1,…,201} should be reversed, TH2 evaluates segment {2,…,200}, TH3 evaluates segment {3,…,199}, etc., up to TH100 which evaluates segment {100,…,102}. In order for each thread to evaluate whether its 2-Opt move is length-reducing or not, as we discussed earlier, it needs to calculate Equation 1 for the cities at its segment boundaries. For example, TH3 calculates Equation 1 for cities 2, 3, 199 and 200. This is done in O(1) time. In this way, all threads evaluate in parallel their 2-Opt moves, and when done, we will have 100 yes/no decisions for the 100 segment reversals under consideration.

Next, we need to apply these decisions by swapping the necessary city pairs. For example, cities 100 and 102 will swap positions if threads TH1 to TH100 have resulted in an odd number of “yes” decisions. Similarly, cities 99 and 103 will swap positions if threads TH1 to TH99 have resulted in an odd number of “yes” decisions, etc. In general, a thread will swap its city pair only if all the previous threads and itself have resulted in an odd number of “yes” decisions. If we have N threads (i.e. we have 2*N+1 cities), the final decisions for swapping the corresponding N city pairs can be derived by these threads in logN steps as shown in the example of Figure 12 for N=15. The threads should be synchronized at the end of each step before advancing to the next one.

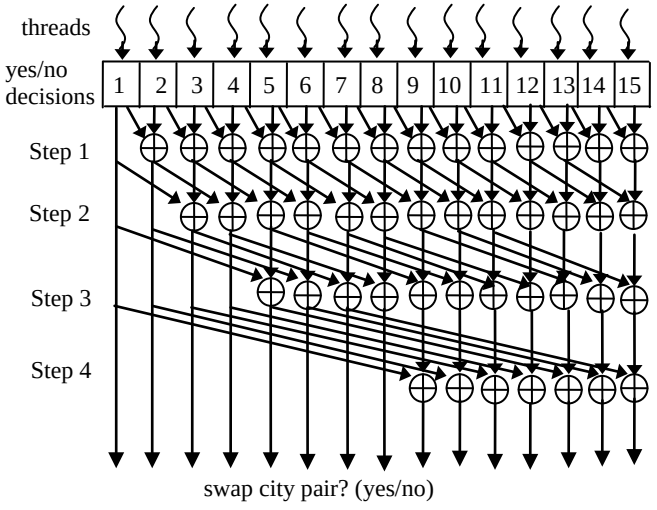


Figure 12. Parallel evaluation of the city swapping decisions in logN steps

For the last step of the algorithm, we perform a left circular shift to all 201 cities, and all the above tasks are again repeated from the beginning. In effect, all threads will point to different positions of the current tour. This whole process is repeated until we cannot find any length-reducing moves for 201 consecutive repetitions, since the algorithm is exhaustive and is guaranteed to search all possible tour segments of odd lengths in 201 repetitions (in contrast to randomized search algorithms that search 2-Opt moves at random and cannot thus guarantee 2-Optimality of the final result).

It is easy to see how this scheme scales to more cities, as long as there is enough memory to accommodate the buffer and resources for additional threads.

6.2 Background on CUDA

GPUs have evolved into highly parallel, multithreaded, manycore processors with tremendous computational horsepower and very high memory bandwidth, exceeding the capabilities of general purpose CPUs by orders of magnitude. The reason is that GPUs are specialized for compute-intensive, highly parallel computations – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

More specifically, GPUs are especially well-suited to address problems that can be expressed as data-parallel computations (i.e. the same program is executed on many data elements in parallel) with high arithmetic intensity – the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

These features make GPUs ideal candidates for our compute intensive data-parallel algorithm of Figure 11. Thus, for our multi-threaded platform we selected CUDA, a general purpose parallel computing architecture, introduced by NVIDIA in November 2006, that leverages the parallel compute engine in modern NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

Depending on the specific model, a CUDA GPU can consist of up to 512 cores, grouped into *stream processors*, with each stream processor able to execute hundreds or thousands of threads in parallel. The GPU that we used in our implementation is the GTX280 with 240 cores, grouped into 30 stream processors, and is able to run a total of 30K threads in parallel. In CUDA threads are grouped into *thread blocks*, with each block running on one processor core. At CUDA's core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. The threads in a block are allowed to cooperate when solving each sub-problem using a shared memory, while the threads from different blocks can communicate through a slower global memory.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

However, the CUDA's SIMD (Single Instruction Multiple Data) execution model can become a significant limitation for applications that employ inherently divergent tasks.

6.3 CUDA-Based Implementation

In order to follow the aforementioned architectural characteristics of CUDA the threads described in Section 6.1 are equally partitioned into blocks (the block size is configurable by the user). For example if the size of the block is 15 we will have 7 blocks {TH1,…,TH15}, {TH16,…,TH30}, …, {TH91,…,TH100} where the last block consists of 10 threads as shown in Figure 13.

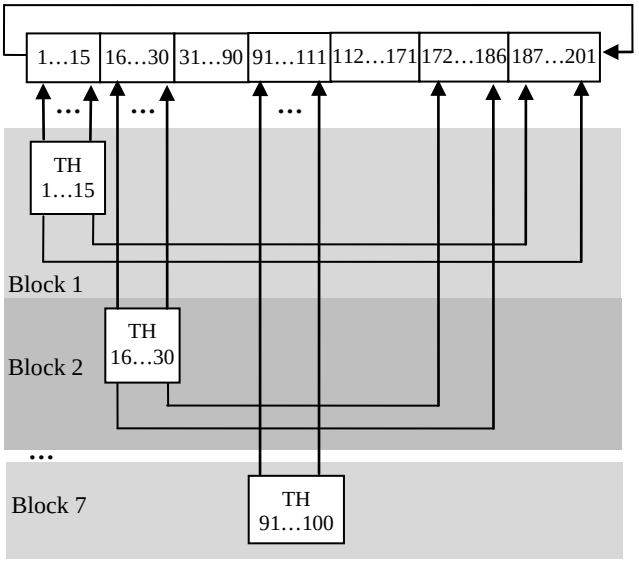


Figure 13. Partitioning of 100 threads into 7 CUDA thread blocks

The coordinates of the cities accessed by each block of threads are initially loaded from the global memory to the GPU's shared memory of the block and they are updated by the threads. Once the algorithm is done the final tour is sent back from the shared memory to the global memory. In this way the fast shared memory is used for solving the TSP instead of the slow global memory. On the other hand, the small shared memory limits the maximum number of cities to 53K, as will be seen in Section 6.4.

The aforementioned partitioning of the threads and the shared memory into blocks and the use of the device memory for inter-block communication trigger several implementation issues, such as:

- The cities at the boundaries of the blocks should be accessed by threads from different blocks that can communicate only through the global memory, in order to evaluate the 2-Opt moves.
- The "yes/no" decisions from the evaluations of the symmetrical segments reside in the shared memory of different blocks which makes difficult their processing through the scheme shown in Figure 12.

- A long circular buffer that includes the cities from all the shared memories of the blocks is required.
- A mechanism should be provided for the inter-block communication and synchronization.

These issues are discussed in detail in the following Sections.

6.3.1 Shared Memory and Inter-Block Communication

Each thread accesses the coordinates of four cities in order to calculate Equation 1 of page 8. For example, in the first iteration of the algorithm, TH16 accesses the coordinates of cities 15, 16, 186 and 187. In this way, the threads at the boundaries of say the second block TH16 and TH30 (see Figure 13) have to access eight cities including cities 15 and 31 that are also accessed by threads TH15 and TH31 of the first and third block respectively. This means that the cities at the boundaries of the blocks have to be stored in the shared memory of two blocks as shown in Figure 14.

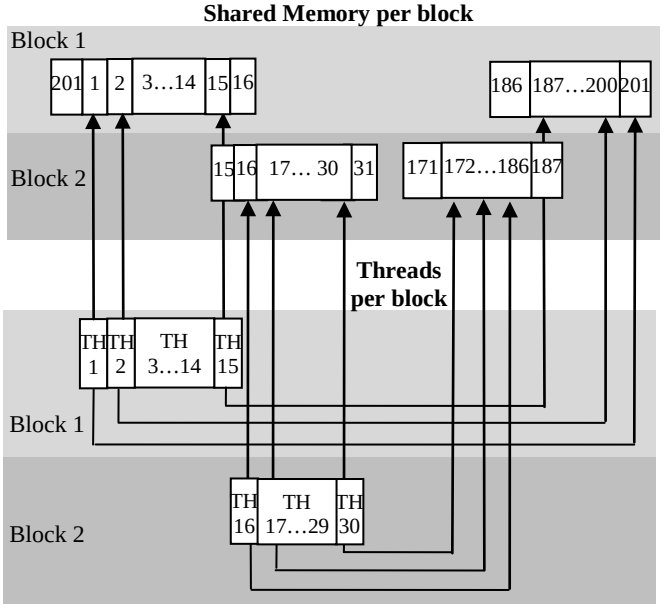


Figure 14. Shared memory of the first two blocks

In this way, in order to perform the first step of the algorithm of Figure 11 the threads of each block need only access the shared memory of their block and no inter-block communication is

required. All threads evaluate in parallel their segments, and when they are done, we will have 15 yes/no decisions in each block (except the last one that has 10 threads) for the 15 segment reversals under consideration in the block. These yes/no decisions are also stored in an array in the shared memory of the block.

Next, the threads of each block need to combine the 15 yes/no decisions in order to decide which city pairs should be reversed. The final results of the XOR operations are sent between the seven blocks following the same scheme. This is shown in Figure 15 where the "XORs" boxes denote the XOR operations performed internally in each block and the output of each box is the value derived from the last thread of the block. For example, in the "XORs" box of the second block threads TH16 to TH30 combine their yes/no decisions to derive 15 values following the scheme of Figure 12. Next, the second block sends the value from TH30 to the third block and receives the value from TH15 of the first block in order to perform the first XOR operation shown in the first step of Figure 15.

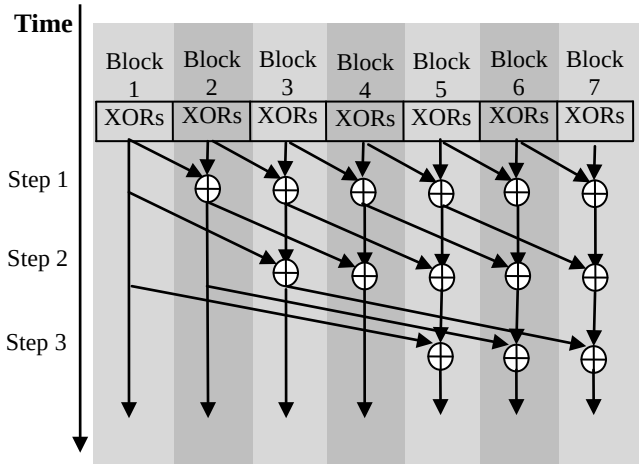


Figure 15. Parallel evaluation of the city swapping decisions among the CUDA thread blocks

The third step of the algorithm is performed in parallel by all threads. Based on the results of the previous step each thread swaps its city pair or not. However, if the first or the last thread of a block, say thread TH16 of the second block, swaps its city pair, i.e. city 16 with city 186, then we also have to update the buffer of the first block, as shown in Figure 14, since the next city of city 15 becomes city 186 and not city 16 which is stored in the first block. In the same way, if

cities 30 and 172 are swapped by thread TH30 (last thread of the block), we also have to update the buffer in the third block. Therefore, the possibly new first cities of the two buffers of each block, cities 16 and 172 in our example, are sent to the blocks holding the previous buffers.

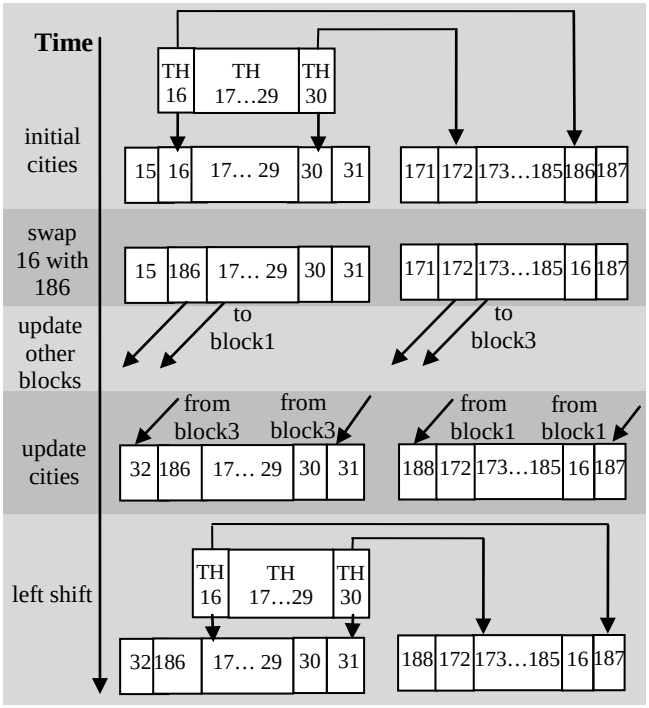


Figure 16. Example inter-block communication and left shift of the buffers of the second block

Finally, in order to implement the left shift of the circular buffer in the final step of the algorithm we have to a) copy the second city of each buffer from one buffer to the previous one and b) shift all the buffers stored in the shared memory of each block. At the end of the previous step we had to copy the first city of each buffer to the previous one while at the beginning of this step we have to copy the second city of each buffer to the previous one. Therefore, combining these actions together, the first two cities of each buffer are sent to the previous buffer after the city swaps, as shown in the example of Figure 16. Both cities are sent over a single inter-block transaction in order to save time.

6.3.2 Inter-Block Synchronization

CUDA provides a synchronization mechanism for the threads within a block but no support of any message passing protocol between threads from different blocks is provided. Therefore, the only way to transfer data from a block to another is to use the slow global memory which is accessible by all threads of any block. Since in the CUDA programming model, the execution of a thread block is non-preemptive, care must be taken to avoid deadlocks in the GPU synchronization design.

We have tried to minimize and parallelize the inter-block transactions between threads from different blocks, as explained in the previous Section, in order to increase the system performance. If the total number of blocks is N, we will have 4N inter-block transactions (2N for the second step and another 2N for the third and fourth steps) in each iteration of the algorithm which will be performed in parallel in logN+2 steps.

A communication protocol based on split transactions has been employed in order to increase the system throughput and reduce the stall times of the communication operations. The split transaction protocol allows the sender block to initiate a new transaction (i.e. write the data to the global buffer) while transactions for other receivers are not completed (i.e. other data is still pending for other receivers). Each sender block keeps synchronization flags stored in the global memory for each receiver block. In particular, a transaction between sender block *A* and receiver block *B* consists of the following actions:

- A thread in block *A*, responsible for the transaction, waits until the synchronization flag that corresponds to block *B* is deasserted in order to make sure that there is no pending transaction from *A* to *B*.
- The thread in block *A* writes the data to the global memory and asserts the synchronization flag that corresponds to block *B*.
- In parallel another thread in the receiver block *B*, responsible for the transaction, reads periodically the aforementioned synchronization flag until it is asserted which means that the data is ready for reading.
- The thread in block *B* reads the data and deasserts the flag showing that the transaction is done.

A global synchronization between the threads of the blocks is guaranteed in the final step of the symmetrical 2-Opt algorithm where each block receives the coordinates of four cities from two other blocks. In this way, each block is stalled until two other blocks have performed all the steps of the current iteration of the algorithm and then it can receive the new coordinates. Having read the correct data the block can continue with the next iteration of the algorithm.

6.3.3 Simulated Annealing

The symmetrical 2-Opt algorithm performs length reducing 2-Opt moves until no further improvement can be made (i.e. a locally optimal tour has been reached). However, a locally optimal tour may not necessarily be close to the globally optimal tour. In order to escape from local minima, we may want to modify this basic scheme of pure optimization and also allow "uphill" moves in our search for the global minimum.

Simulated Annealing (SA) is a well-known algorithm that does just that. It allows "uphill" moves based on a carefully crafted probability function. The algorithm can perform length increasing 2-Opt moves based on a probability that depends on the length difference of the tour and on a global parameter “temperature” *T*, that is gradually decreased during the process. After lowering the temperature several times, the process accepts only length reducing moves in order to find a local minimum.

In order to incorporate simulated annealing to our algorithm, we modified the first step. In particular, every thread derives a “yes” decision for a length increasing segment (i.e. it reverses its segment) if the following formula is true:

$$e^{-\Delta L/T} > R(0,1)$$

ΔL is the change in the length of the tour if the 2-Opt move is applied, T is the temperature and R(0,1) is a random number in the interval [0,1]. In this way, “uphill” moves are permitted since a thread can derive a “yes” decision even when $\Delta L > 0$ (i.e. approve a length increasing move). In order to calculate faster the outcome of this formula we used a pseudo-random function instead of the random number R(0,1) and pre-calculated values for the power function.

6.4 Performance Results

We implemented the algorithm of Section 6.1 on two different platforms: a) in software on an Intel Pentium 3 GHz machine running RedHat Linux, b) in CUDA on an Nvidia GeForce GTX280 card.

The GTX280 card supports 240 processing cores with a GPU clock of 602MHz, partitioned into 30 stream processors (each one holding 8 cores). Each stream processor has 16KB of shared memory and can execute up to two blocks of 512 threads. Therefore we can have up to 30*2*512 = 30K threads running in parallel and a total of 30*16K = 480K of shared memory.

If the coordinate of a city is an integer of 4 bytes, each city occupies 8 bytes in the shared memory of a stream processor. Taking also into account 1 byte overhead per city pair in order to hold the "yes/no" decisions, we can hold up to 16KB/9B=1.77K cities in the shared memory of each stream processor which corresponds to 906 threads (half the number of cities), or two thread blocks of 906/2=453 threads each. Therefore the maximum number of cities the GTX280 card of 30 stream processors can hold is 30*1.77K≈53K cities.

In order to evaluate and measure the performance of our 2-Opt algorithm, both with and without Simulated Annealing (SA), we used the TSPLIB instances (we used the Euclidean instances of "EUC_2D" type); notice that we used the exact TSPLIB instances were used as our starting tours, as opposed to first applying a greedy algorithm to them as is often done. The sizes of these instances range from 50 to around 18K cities.

We first analyze the parameters that affect the performance of the GPU-based implementation and the efficiency of the simulated annealing algorithm. Next, we follow with some performance results for our implementations.

6.4.1 Block Size of GPU

The following equation derives the number of blocks as a function of the block size and the number of cities.

$$Blocks_{number} = |Cities_{number} /(2* Block_{size})|$$

Figure 17 shows the number of blocks as a function of the block size using an example TSPLIB instance with 2103 cities.

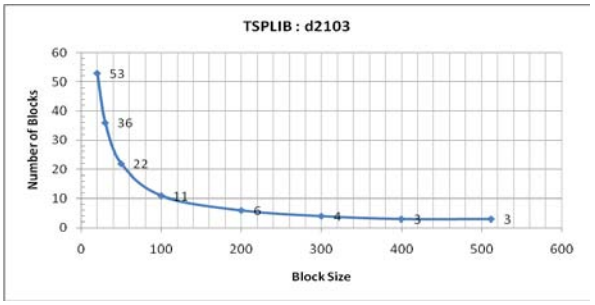


Figure 17. Number of thread blocks as a function of the block size

The threads of a block run on the same stream processor of the GPU card. By having many blocks (i.e. reducing the block size) the programmer can increase the parallelism since the blocks can be distributed and executed in parallel by different stream processors. On the other hand increasing the number of blocks will also increase the inter-block communication overhead which can eventually deteriorate the performance.

Figure 18 shows how the running time is affected by the size of the block for the TSPLIB instance d2103. When the block size is close to 35 we have the fastest solution. In this case we have the maximum possible parallelism, since we have 30 blocks (see Figure 17) running on the 30 stream processors. By lowering the block size below 35, we essentially increase the inter-block communication.

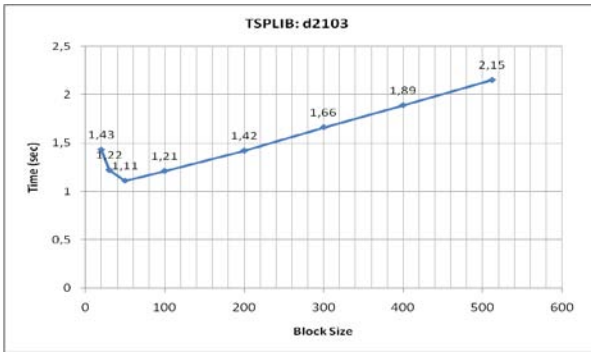


Figure 18. Runtime behavior as a function of the block size

6.4.2 Cooling Schedule of Simulated Annealing

The cooling schedule of a simulated annealing algorithm consists of three components: a) the starting temperature, b) the temperature decrement step, and c) the number of iterations at each temperature. The values of these components essentially determine the rate that the temperature *T* decreases over time. This rate affects both the execution time and the quality of the results.

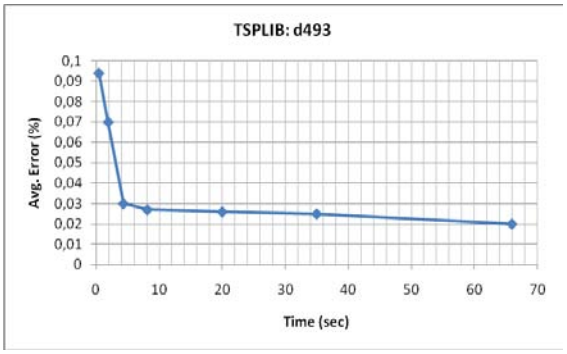


Figure 19. SA trade-off between execution time and quality of results

By adjusting the cooling schedule we could trade off between execution time and higher quality results. This is shown in Figure 19, where we varied the cooling rate for an example TSPLIB instance with 493 cities running on the GTX280 card.

6.4.3 Performance Results

We compare the performance of the following implementations of the 2-Opt algorithm in terms of quality of results and speed:

1. A modified version of the 2-Opt local search algorithm implementation by Concorde. Concorde is widely considered as the state-of-the-art in TSP solving software, containing highly optimized implementations for the most important TSP algorithms and heuristics. We modified its 2-Opt algorithm implementation in the following way:
    - Concorde will by default first apply the greedy algorithm to the initial tour in order to obtain the starting tour for 2-Opt optimization. Since for the performance

measurements of our architecture (and its software emulator) we used the exact TSPLIB instances as our starting tours, we had to modify Concorde so as not to apply the greedy algorithm to the initial tour.

2. The multi-threaded symmetrical 2-Opt algorithm, described in Section 6.1, running either on an Intel CPU, or on the GTX280 GPU card.
3. The multi-threaded symmetrical 2-Opt algorithm with Simulated Annealing (SA) enabled, as described in Section 6.3.3, running on the GTX280 GPU card.

TSPLIB instance	Concorde (2-Opt)		Symmetrical 2-Opt without SA			Symmetrical 2-Opt with SA	
	avg. error (%)	CPU_{time} (**msec**)	avg. error (%)	CPU_{time} (**msec**)	GPU_{time} (**msec**)	avg. error (%)	GPU_{time} (**msec**)
pr124	5.3	0.78	3.1	1230	9.7	1.4	69
bier127	19.7	1.40	10	1430	18.7	3.8	94
ch130	21	1.38	10	1600	22.2	2.1	99
pr152	8	1.30	2.8	1490	25.6	1.9	175
d198	11.2	1.54	6.8	3110	26.2	3.9	400
kroA200	32.6	2.14	12.3	4130	48.4	2.5	421
tsp225	20	1.32	8.2	3400	27.2	3.2	720
lin318	27.3	3.00	17.1	7890	78.8	4.2	2690
d493	13.2	4.80	10.5	23280	155	4.3	5470
p654	18.5	5.30	14.1	37390	377	4.3	33200
pcb1173	35.8	9.10	11.9	204980	704	4.8	318400
rl1323	44.7	17.6	16.5	372630	858	4.7	559300
u1817	16.8	9.00	16.2	223610	740	4.9	728160

Table 1. Execution times and average errors for three TSP algorithms running on software or hardware platforms

Table 1 shows the efficiency and the performance of the aforementioned algorithms for several problem instances from the TSPLIB benchmark. TSPLIB provides the optimal lengths of the tours. The *average error* columns show the difference from the absolute optimal solutions for these instances.

We adjusted the cooling schedule of the simulated annealing algorithm in order to succeed an error ratio below 5%. Concorde's results range between 5% and 45% from the optimal solution while the symmetrical 2-Opt algorithm is always below 17%. This is expected since Concorde, in contrast to our architecture, gives up the guarantee of true 2-Optimality in favor of greatly reduced running time. This shows the high quality of the tours produced by our architecture.

In terms of speed, Concorde comes first followed by the GPU and finally the CPU. Concorde runs very fast for the TSPLIB instances that we considered. Thus, in order to more accurately measure its performance, we measured for each TSPLIB instance the average running time among 1000 runs.

The data-parallel features of the symmetrical 2-Opt algorithm give significant speedup when the algorithm is implemented in a multi-threaded system such as CUDA as opposed to a general purpose CPU. Figure 20 shows the speed-ups obtained by the CUDA-based implementation of the symmetrical 2-Opt algorithm against the CPU-based version.

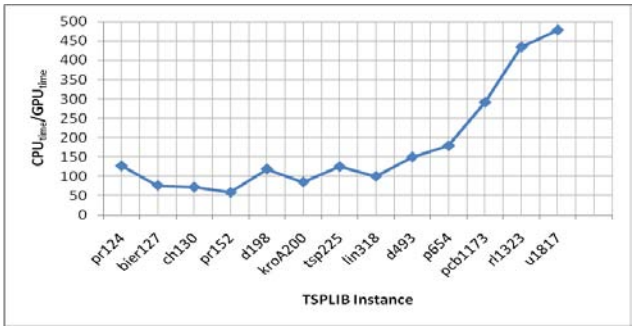


Figure 20. Performance comparison between GPU and CPU

As expected the overall speed-up is roughly proportional to the number of cities, since the latter reflects the number of threads operating in parallel.

By adjusting the cooling schedule of the simulated annealing we could succeed an error ratio below 5%. Figure 21 shows the execution time of our symmetrical 2-Opt algorithm running on the GPU with Simulated Annealing enabled over the execution time when Simulated Annealing is not performed.

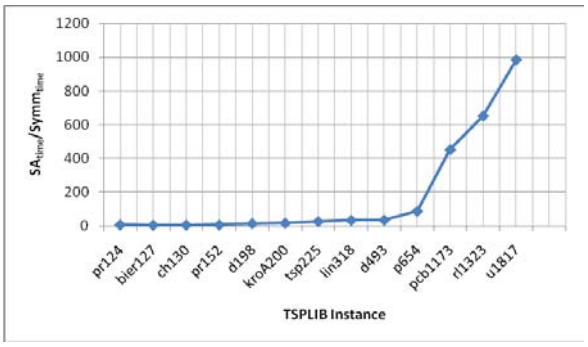


Figure 21. Simulation annealing significantly delays bigger TSP instances

For small TSPLIB instances up to 200 cities performing SA results in 10 to 15 times increase in run time. However, for bigger TSPLIB instances the error ratio of the symmetrical 2-Opt algorithm is high (above 10%) and SA requires a significant slower cooling process in order to lower the error ratio below 5%.

Chapter 7. Conclusions and Future Directions

In this thesis we have uncovered for the first time fine-grain parallelism in the application of 2-Opt moves for the Traveling Salesman Problem. We also investigated how this newly-found parallelism can be exploited to speed-up 2-Opt. We implemented our approach on two types of platforms, an FPGA and a multi-threaded GPU, and evaluated both in terms of both speed and quality of final results. The hardware implementation outperformed Concorde, the current state-of-the-art software implementation.

7.1 Future Directions

Regarding our future work, we will focus on the following tasks:

- Add hardware to the PEs for the square root calculations and adjust our performance numbers accordingly. In order not to add major hardware resources nor significantly increase the number of clock cycles needed for the PEs we will use a square root estimation technique such as the one in [9].
- Examine how our architecture scales for TSP instances larger than the ones considered in this thesis, and explore the possibility of using external RAM for both implementations.
- Explore the cost and performance of Simulated Annealing in our hardware implementation.

- Explore how we could handle 3-Opt moves.
- Explore how our scheme can be combined with other parallel solutions for the TSP such as the ones discussed in Section 3.1.

Bibliography

[1] TSPLIB is a database of instances for the TSP and is currently available from http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[2] DIMACS Implementation Challenge on the STSP at http://www.research.att.com/~dsj/chtsp/

[3] Concorde is computer code for the TSP and is currently available for download from http://www.tsp.gatech.edu/concorde.html

[4] D. Johnson and L. McGeoch, ``*Experimental analysis of heuristics for the STSP*" chapter of "*The Traveling Salesman Problem and Its Variations"*, Boston 2002, pp. 369-443. Draft of chapter currently available from http://www.research.att.com/~dsj/papers/stspchap.pdf

[5] D. Johnson and L. McGeoch, "*The Traveling Salesman Problem: A Case Study in Local Optimization*" chapter of "*Local Search in Combinatorial Optimization*", London 1997, pp. 215-310

[6] "*Simulated Annealing Methods*" chapter 10.9 of "*Numerical Recipes in C*". Chapter available from http://www.nrbook.com/a/bookcpdf.php

[7] I. Skliarova and A. Ferrari, "*FPGA-Based Implementation of Genetic Algorithm for the Traveling Salesman Problem and Its Industrial Application*", IEA/AIE 2002, LNAI 2358, pp. 77-87, 2002

[8] P. Graham and B. Nelson, "*A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash 2*", FPL August 1995, pp 352-361

[9] Xiaojun Wang, Brent Nelson, "*Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs*", FCCM 2003

[10]Ioannis Mavroidis, Ioannis Papaefstathiou, Dionisios Pnevmatikatos, "*Hardware Implementation of 2-Opt Local Search Algorithm for the Traveling Salesman Problem*", Technical University of Crete, Greece, Rapid System Prototyping 2007

[11]Stutzle, T. and Hoos, H.H. "MAX-MIN Ant System." *Future Gen. Comput. Syst.*, vol. 16, no. 9, pp. 889-914. June 2000

[12]Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, Haihon Yu, "MAX-MIN Ant System on GPU with CUDA", icicic, pp.801-804, 2009 Fourth International Conference on Innovative Computing, Information and Control, 2009

[13]Wang Jiening, Dong Jiankang, Zhang Chunfeng, "Implementation of Ant Colony Algorithm Based on GPU", Sixth International Conference on Computer Graphics, Imaging and Visualization, 2009

[14]You, Y.-S. "Parallel Ant System for Traveling Sales-man Problem on GPUs", Eleventh Annual Conference on Genetic and Evolutionary Computation, July 2009

[15]Cecilia, J.M., Garcia, J.M., Ujaldon, M., Nisbet, A., and Amos, M. "Parallelization Strategies for Ant Colony Optimisation on GPUs", 14th International Workshop on Nature Inspired Distributed Computing, May 2011

[16]M. A. O'Neil, D. Tamir, and M. Burtscher, "A Parallel GPU Version of the Traveling Salesman Problem", International Conference on Parallel and Distributed Processing Techniques and Applications. July 2011

[17]Fujimoto, N. and Tsutsui, S. "A Highly-Parallel TSP Solver for a GPU Computing Platform", Lecture Notes in Computer Science, Vol. 6046, pp. 264-271, 2011