Technical University of Crete

School of Electrical & Computer Engineering

Design and Implementation of the OPT-2 Algorithm in Recent Technology FPGA Logic

By Georgios Malandrakis Miller

A Thesis Submitted in Partial Fulfillment of The Requirements for The Diploma of Electrical & Computer Engineering

Thesis Committee

Professor Dollas Apostolos, *Supervisor*
Assistant Professor Papaefstathiou Yannis, *Committee Member*
Professor Pnevmatikatos Dionisios, *Committee Member*

Chania, December 2016

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere and deep gratitude towards my supervisor, Prof. Apostolos Dollas, for the chances he gave me, his continuous support and availability throughout the preparation of this thesis and last but not least, for believing in me. I would also like to specially thank Gregory Chrysos and Pavlos Malakonakis, for our valuable collaboration and their always insightful guidance through every obstacle I encountered during my work towards the completion of this thesis. Many thanks are also due to Assistant Professor Yannis Papaefstathiou and Professor Dionisios Pnevmatikatos for their participation in the committee, as well as all the staff and students of the Microprocessor & Hardware Lab, none excluded. At this point, I should obviously also thank my parents and sister for bearing up with me this whole time, and for their unconditional and inexhaustible support towards every endeavor I undertake. Finally, many, many thanks to each and every one of my friends, for all the good times we had together during our studies.

Everything that has a beginning has an end.

Every end is a new beginning.

ABSTRACT

The Traveling Salesman Problem (TSP) is one of the most well-known and thoroughly studied problems within the domain of combinatorial optimization, having attracted for many decades the attention and even life-long devotion of numerous computer scientists, ranging from algorithm researchers to High-Performance Computing specialists. The number of its practical, real-life applications is sheer: Manufacturing, logistics, telecommunications, statistics, scheduling, and even psychology are some of them, to name a few. Since it is an NP-Hard problem, solving it to optimality requires an exponentially-increasing time as its size grows, thus rendering its exact solution prohibitive for large datasets or when time constitutes a crucial factor. This has ultimately led to the development of an abundance of heuristics, specifically designed to address this issue, providing orders of magnitude reduced running times at the cost of sub- or near-optimal results. One of the oldest and most recognized such heuristic is 2-OPT. Its simplicity, combined with its effectiveness and adaptability has led to its wide adoption within the TSP community, being actively implemented by researchers worldwide even today, more than fifty years after its initial conception.

The aim of this thesis is the mapping of a 2-OPT variant based on reconfigurable logic (FPGAs). The presented work originates from research conducted nearly a decade ago in the Microprocessor & Hardware Lab at the School of ECE, Technical University of Crete. It constitutes an evolutionary extension to the latter, whose objective is to assess its performance while adapting it to next-generation FPGAs, through the utilization of both the newly-(re)emerged High Level Synthesis flow and the traditional HDL-based flow as well; the corresponding tools are part of the Xilinx Vivado Design Suite. A thorough evaluation and an in-depth comparison between the intrinsic characteristics of these two approaches to hardware design are provided. Experimental results show that the implemented hardware architectures are capable of delivering speedups ranging from 1.1x to nearly 10x for small to medium scale problem sizes, depending on the dataset used. These figures are obtained when comparing the aforementioned architectures with the Concorde TSP software package, as well as with a state-of-the-art GPU implementation.

KEYWORDS

Combinatorial optimization, Traveling Salesman Problem (TSP), 2-OPT, Parallel Implementation, Field Programmable Gate Array (FPGA), High Level Synthesis (HLS), Hardware Description Language (HDL), Comparison, Evaluation.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

This opening chapter constitutes an introduction to the scientific areas approached by the thesis, intending to familiarize the reader with some basic, related notions, and provide an early, generic insight into its main topic. Under these premises, the first section presents the concise definitions of some fundamental terms of algorithm theory encountered in this text, followed by a piece of introductory information about Reconfigurable Computing & FPGAs. The chapter then continues with the statement of the motivations that led to the development of this thesis, its scientific contribution, and ultimately concludes with the structure of the rest of the text.

1.1 CONCISE BACKGROUND IN COMPUTATIONAL COMPLEXITY

This section constitutes a short review of some key aspects of computational complexity and algorithm theory, intending to aid the comprehension of relative terms. For extensive and in-depth information about these topics, the reader is encouraged to refer to [1] and [2].

Computational complexity refers to the inherent difficulty of a computational problem solvable by an algorithm. Such a problem is considered inherently difficult if its solution requires a significant amount of a resource, be it time and/or space, regardless of the actual underlying algorithm utilized to solve it. In that notion, "time", and more formally, Time Complexity, does not necessarily translate to actual running time in the sense of seconds or minutes, since that would require the precise knowledge of several implementation-dependent parameters, but to a rather abstract concept involving the estimation of the number of a series of fixed-time elementary steps required to process the desired solution [1]. An equivalent premise holds for the case of "space", namely Space Complexity.

Within the framework of computational complexity, Time complexity typically refers to the worst-case scenario, denoted $\mathrm{T(n)}$, defined as the maximum amount of time required to solve a problem with an arbitrary-sized input of length n. When formulating the Time complexity of a problem, the former is usually provided in the form of the type of its asymptotic upper bound (Big O notation, $\mathrm{O}\sim$), such as "linear time", $\mathrm{T(n)= O(n)}$, "logarithmic time", T(n)= O(logn), "quadratic time", T(n)= O(n^2), etc.

Time/Space complexity aside, two other equally important factors characterizing the complexity of a computational problem are its type and model of computation. The most common representative of the former is decision problems, where, given a set of inputs, their solutions result in a Yes-or-No answer, while many more types exist, such as function problems, counting problems, optimization problems, promise problems etc. As far as the latter is concerned, the deterministic Turing machine is the predominant model used in computational complexity theory, some of the rest being non-deterministic Turing machines, Boolean circuits and quantum Turing machines.

These three factors mentioned above are sufficient in order to define some simple complexity classes; a set of computational problems of related resource-based complexity (time or space). The section below presents a few fundamental complexity classes, frequently appearing in algorithmic literature, based on the combination of decision problems, Turing machines, and polynomial time complexity.

P: Contains all the decision problems that can be solved by a deterministic Turing machine in polynomial time, thus implying that $\mathrm{T(n)} = 2^{\mathrm{O(logn)}} = \mathrm{poly(n)}$. As expressed in [3] and [4], a "good", in terms of efficiency algorithm, is an algorithm with polynomial execution time. This statement also appears in [1].

P-complete: A decision problem is P-complete if it is in P and every problem in P can be reduced to it by an appropriate reduction.

NP: NP stands for non-deterministic polynomial time; it is a superset of P and contains the decision problems for which an efficient, polynomial time verifier exists [1]. In other words, given an instance of a decision problem where the answer is "yes", its validity can be verified in polynomial time. Still, it is unknown whether a "good", polynomial time solution exists for any problem within the NP-P class, and this issue actually poses one of the most elusive unresolved problems in computer science, coined as the "P versus NP" problem. There is no concrete evidence pointing to one way or the other, yet the general belief within the scientific circles is that P≠NP, which implies that, possibly, there are problems in NP that are harder to compute (super-polynomial or exponential time) than to verify (polynomial time)[1].

NP-complete: NP-complete problems are considered the "hardest" decision problems within NP [5]; a problem X is classified as NP-complete if X belongs in NP and every other problem Y within NP can be polynomially reduced to X [8]. An alternate definition that highlights this inherent difficulty is the following: "Assume X is a NP-complete problem. If X is solvable in polynomial time, then P=NP" [1].

NP-hard: NP-hard is the class of problems that are at least as hard as the hardest problems in NP [8], but not necessarily in NP [6]. A more formal definition is the one found in [7], which states that a decision or optimization problem H is NP-hard when for every problem L in NP there is a polynomial time reduction from L to H [8]. Another equivalent definition is to require that there is a polynomial time reduction from an NP-complete problem G to H [7]. Consequently, if an optimization problem H has an NP-complete version G then H is NP-hard. The above imply that NP-hard class is not limited to decision problems (i.e. not limited to NP), but also includes function problems, optimization problems etc.

NP-easy: A problem which polynomially reduces to some problem in NP is called NP-easy [8].

NP-equivalent: An NP-equivalent problem is a problem that is both NP-hard and NP-easy [8].

Many more complexity classes exist, such as BPP, ZPP and RP, defined using probabilistic Turing machines; BQP and QMA are defined using quantum Turing machines; ALL is the class of all decision problems.

1.2 USEFUL THEORETICAL DEFINITIONS

Combinatorial Optimization: Combinatorial optimization is one of youngest and most active areas of discrete mathematics, having its roots in combinatorics, operations research and theoretical computer science [8]. Its main theoretical as well as practical focus is to aid the choice of a "best" (i.e. optimal) configuration of objects or set of parameters in order to achieve a specific goal, the set being finite, and typically constituting an integer set, a permutation set, or a graph. Thus, in the case of combinatorial optimization problems, the solution is chosen from among a finite set of possible solutions [9].

Local & Global Optimum: Consider an optimization problem, be it continuous or discrete, where the objective is to choose some arbitrary variables, collected into a vector x, in order to maximize or minimize an objective function $f(x)$, which in turn might or might not be subject to constraints. A vector x satisfying all the constraints set (if those exist) is called feasible. A particular feasible choice x, say x^*, is called optimum if no other feasible choice gives a higher (or equivalently lower in the case of minimization problems) value of $f(x)$, and a strict optimum if all other feasible solutions give a lower (or equivalently higher in the case of minimization problems) value of $f(x)$. An optimum x^* is called local if the comparison is restricted to other feasible choices within a sufficiently small neighborhood of x^*. If the comparison holds against all other feasible solutions, no matter how distant, the optimum is called global [10].

Metric Spaces & Metrics: A metric space is a set X that has a notion of the distance $d(x,y)$ between every pair of points $x,y \in X$. A metric on a set is a function that satisfies the minimal properties expected of a distance. A more formal definition is the following: "A metric d on a set X is a function $d : X \times X \rightarrow R$ such that for all $x,y \in X$: **(1)** $d(x,y) \geq 0$ and $d(x,y) = 0$ if and only if $x=y$, **(2)** $d(x,y)=d(y,x)$ (symmetry), and, **(3)** $d(x,y) \leq d(x,z)+d(z,x)$ (triangle inequality). A metric space (X,d) is a set X with a metric d defined on X [11][34].

Hamiltonian Cycle: A Hamiltonian cycle, also called a Hamiltonian circuit and defined on a graph $G= (V,E)$, is a graph cycle (i.e. a closed loop) through a graph that visits each one of its nodes E exactly once. A graph possessing a Hamiltonian cycle is said to be a Hamiltonian Graph [30].

1.3 ABOUT RECONFIGURABLE COMPUTING & FPGAs

Field-Programmable-Gate-Arrays (FPGAs) are (re)programmable semiconductor devices able to perform all kinds of computational operations and implement any algorithm that can be executed on a classic CPU. This is highlighted by the fact that they can be programmed to emulate the functionality of the latter; such an example is the Microblaze softcore RISC CPU from Xilinx, which, among others, is capable of running lightweight Linux distributions. FPGAs are usually employed to accelerate computationally intensive applications through the extensively parallelized mapping of the involving algorithms on the former's fabric.

Contrary to Application-Specific-Integrated-Circuits (ASICs), FPGAs are available off-the-self, completely prefabricated and ready to be customized by the end user according to the specific needs of the target application. The aforementioned customization usually comes in the form of Hardware Description Languages, such as VHDL and Verilog, enabling the user to model the desired design in the RTL abstraction, which, in turn, is provided as input to an appropriate EDA tool tasked to map the RTL model to the structural components of the FPGA fabric; this is the industry-standard, long-proven methodology to reprogrammable custom hardware design, its final step being the generation of the bitstream used to actually program the FPGA itself.

Historically, FPGAs were an evolution of PLDs, commonly termed as CPLDs, introduced in the mid-1980s as larger, compared to their PROM, PAL and GAL ancestors (all PLDs), capacity platforms for implementing digital logic. The novel and main differentiating point between PLDs and FPGAs was that the former concentrated primarily on two-level, sum-of-products implementations of functions, while the latter were optimized for multi-level circuits, thus enabling them to perform much more complex functions and at greater density. By the early 1990s, their capabilities along with their capacity and speed had largely grown, and some of their first and major usage scenarios were logic emulation and rapid system prototyping, since the offered prototyping rate was much higher than MPGAs [13,15].

Since then, their commercial acceptance has been on an ever-constant and sharp increase, rendering them the driving force of the field that has become known as Reconfigurable Computing - performing computations through the utilization of spatial, high-performance, hardware-based, programmable architectures by exploiting the inherent parallelization and flexibility traits of fast, fine-grained computing fabrics, mainly FPGAs [13,14,15]. Lately, the prominent role of FPGA-based reconfigurable systems in High Performance Computing (HPC) applications has been highlighted by the dawn of FPGA-based supercomputers, such as [16] and [17], as lower-power and higher-performance alternatives to classic x86-based multiprocessor servers.

The range of applications where the power offered by FPGAs or FPGA-based systems has been successfully harnessed and proven is literally vast: signal processing, cryptography, arithmetic/numeric computing, scientific computing, networking, military applications, financial computations, molecular dynamics modeling, telecommunications, NP-hard optimization, bioinformatics, medical applications, multimedia processing, big data processing and more [12,15,18,19,20,21,22,23].

Recently, the escalating system-on-chip design complexity combined with the huge silicon capacities, has begun pushing the design community to raise the level of abstraction beyond RTL [24]. This has led to the emergent of High Level Synthesis (HLS) tools which enable the user to capture the desired functionality in the form of a high-level language used traditionally for software programming, such as C and C++ [24,25,26,27], thus largely simplifying and most importantly speeding up the process of designing custom hardware architectures. Heterogeneous computing is also encouraged through the support of relevant software frameworks, such as Khronos' OpenCL [28].

The basic structure of an FPGA consists of the following elements: a) Look-Up Tables (LUTs), b) Flip-Flops (FFs), c) Wires and d) I/O pads; generally, the first two components, along with a few RAM-based storage elements and other auxiliary logic, form the CLB, which, in turn, constitutes the basic building block of modern FPGAs. Additional components include: Embedded memories, clocking resources for driving the FPGA fabric at different clock rates, off-chip memory controllers interfacing to external memories, DSP blocks and more [70].

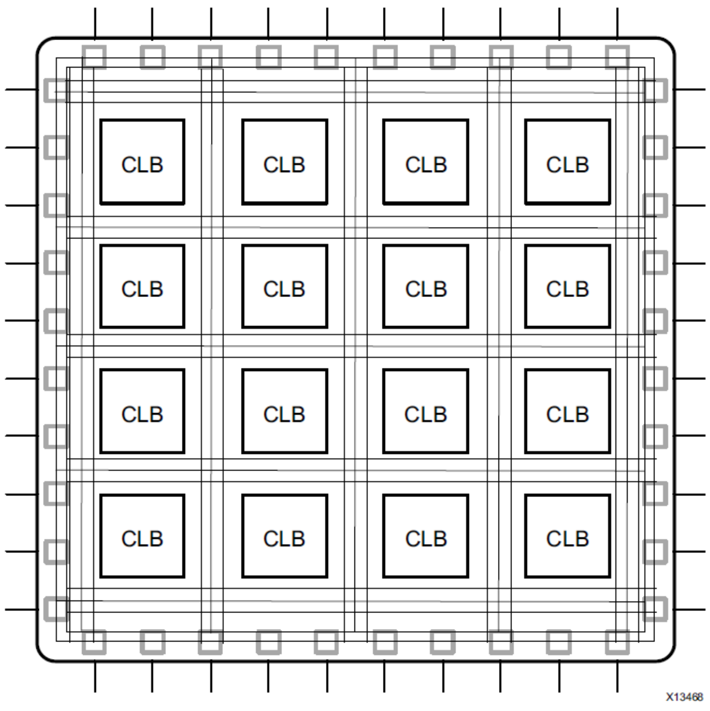


Figure 1: Basic FPGA Architecture, ©Xilinx Inc. Image extracted from Xilinx's "Introduction to FPGA Design with Vivado HLS" Guide (UG998) [70].

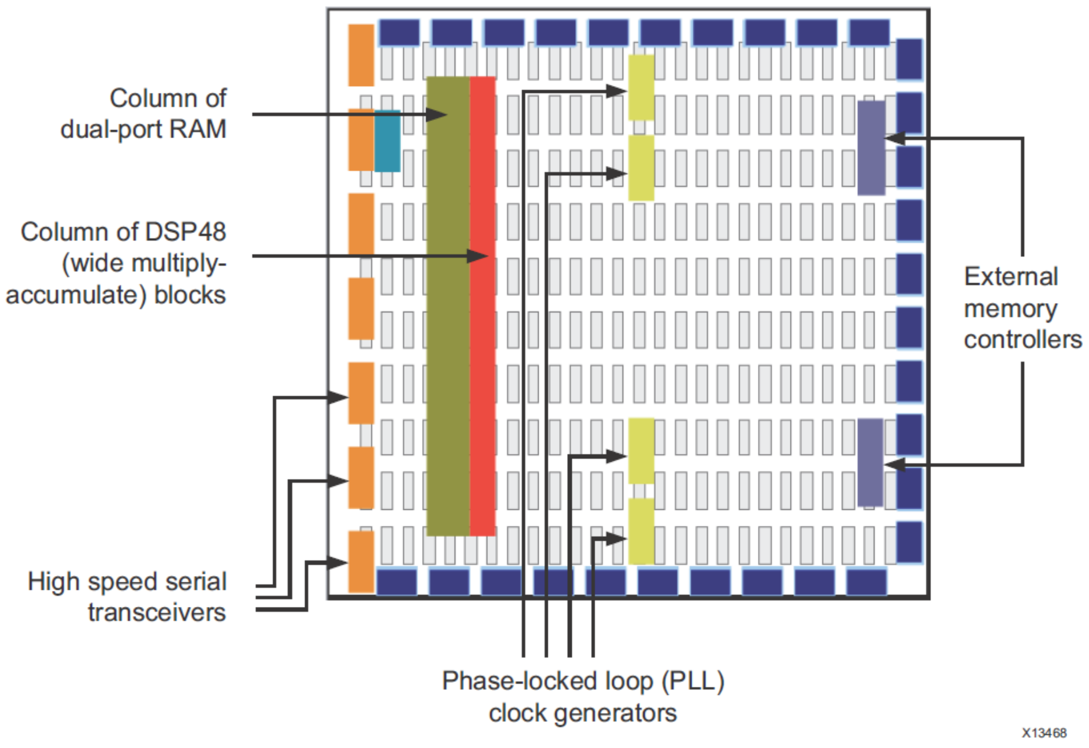

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators
X13468

Figure 2: Contemporary FPGA Architecture, ©Xilinx Inc. Image extracted from Xilinx's "Introduction to FPGA Design with Vivado HLS" Guide (UG998) [70].

Major FPGA vendors include Xilinx Inc. and Altera Corporation; the latter is a subsidiary of Intel since 2015.

1.4 MOTIVATION

Since the work presented in this thesis is based on previously conducted research [29], the motives that led to its materialization were manyfold:

- Examine how much FPGA technology has evolved in the past decade in terms of fabric capacity as well as attainable clock speeds, by conducting practical experimentations with high performance hardware architectures.
- Examine how the hardware architecture proposed in [29] scales for problem instances larger than those previously considered, as well as the performance gains obtained for previously tested instances through their implementation in next generation FPGAs and utilization of modern design suites.
- Explore the strengths and weaknesses of recently (re)introduced hardware design flows, specifically the HLS design flow, and examine how it fares against the classic HDL approach. This comparison is conducted in terms of quality of results as well as temporal and knowledge requirements, by utilizing both design flows during the implementation phase of the aforementioned architectures, always within the context of a parallel 2-OPT solver.
- Explore how the implemented architectures compete against corresponding state-of-the-art implementations of the 2-OPT algorithm, both serial (CPU based) and parallel (GPU based) as well.

1.5 CONTRIBUTION

The contribution of this thesis is as follows:

- Provides an experimentation-based comparison of the performance characteristics of multiple variations of the 2-OPT algorithm, including its original, randomized and symmetrical versions, in terms of key metrics such as quality of results, runtime, convergence rate and more; this entailed the implementation of the aforementioned 2-OPT variants in software (MATLAB), and their subsequent thorough testing against multitudes of TSP instances.
- Provides an experimentation-based assessment of the ways in which the performance characteristics of the 2-OPT algorithm are affected by the replacement of the Euclidean distance-utilized by Euclidean 2-OPT as its primary cost function-with its squared approximation (squared Euclidean distance).
- Constitutes-to the author's knowledge-one of the very few works on the mapping of 2-OPT-as well as of TSP-related heuristics in general-in reconfigurable logic, since the literature review of this thesis indicated that most of the ongoing research on the parallelization of such algorithms focuses on evolutionary-based heuristics and their optimization for execution in high-performance GPUs.
- Provides a multi-level comparison of the two most prominent approaches to hardware design, namely the HLS-flow and the RTL-to-Bitstream flow, based on the design and implementation of a 2-OPT hardware architecture along with the utilization of both the aforementioned flows; these are realized through the Vivado HLS and Vivado IDE tools respectively, which constitute part of the Xilinx Vivado Design Suite.
- Investigates the extent to which an FPGA-based 2-OPT solver can be considered as a viable and effective alternative option to equivalent high-performance solutions implemented either in software (CPU) or in hardware as well (GPU).

- Investigates the performance-wise viability of an FPGA-based TSP metaheuristic embedding the 2-OPT hardware architecture under consideration; the resulting design could be possibly realized as a software/hardware co-design mapped on a heterogeneous system such as the Convey series of hybrid supercomputers.

1.6 THESIS STRUCTURE

The rest of the thesis is organized as follows: Chapter 2 introduces the fundamentals of the Traveling Salesman Problem (TSP), such as its roots, formal definition, complexity, and solution methods in a compact, yet descriptive manner. Chapter 3 cites some of the most prominent work on parallel implementations of TSP heuristics, mainly hardware based, and more specifically GPU based implementations. Next, Chapter 4 presents the functionality of the 2-OPT algorithm upon which the implemented hardware architectures are based and compares it with some popular variations, while providing metrics representative of their intrinsic characteristics. Chapter 5 constitutes an in-depth presentation of the aforementioned hardware architectures and the design tools utilized, along with the workflow, major milestones in the development of the former and some key findings concerning the latter. Chapter 6 provides the implemented hardware architectures' evaluation along with related performance and quality results and compares them to the state-of-the-art in 2-OPT solvers. Finally, the thesis concludes with its summary along with possible future extensions and additions, Chapter 7.

CHAPTER 2: APPROACHING THE TSP

This chapter constitutes a brief overview of the fundamentals of the Traveling Salesman Problem, and as such it does not cover every aspect of it or delve into specifics, since that would be completely out of the scope of the thesis. For more, in-depth information on the TSP, the reader is encouraged to consult the references provided in this chapter.

2.1 INTRODUCTION & CHAPTER STRUCTURE

Initially, the chapter provides a brief history of the origins of the TSP as well as a descriptive formulation of the problem itself. A more formal and mathematical formulation based on graph theory follows. Next, the complexity of the TSP along with some of its most common cases is presented. The chapter concludes with a concise description of the most frequently encountered approximation algorithms (Heuristics) along with their major categorizations. A report on the practical applications of the TSP is also provided.

2.2 ORIGINS OF THE TSP

The exact origin of the name “Traveling Salesman Problem” was and remains rather obscure, since no kind of authoritative document providing any concrete evidence towards its creator exists, although it appears to have been discussed informally among members of the mathematical society for decades [3,31]. One of the earliest formulations of the problem dates back to 1832, in a German handbook made by a traveling salesman himself who roamed in the greater regions of Germany and Switzerland. Within the aforementioned manual, he describes the importance of good tours and actually presents a TSP tour through 45 German cities, which the author in [31] believes that could be optimal. While the handbook was brought into the attention of the TSP community in 1983, it is widely believed that the first work on TSP was published in 1930 by the name the “Messenger Problem” [6], while the first reference containing the actual TSP term is considered to have emerged in 1949. In [3] and [31] it is concluded that sometime around the 1935s and most probably at Princeton, the TSP took on its name officially, thus indicating the initiation of its active exploration.

2.3 WHAT IS THE TSP?

The Traveling Salesman Problem (TSP) is one of the most significant and well-studied problems in combinatorial optimization [8,31].Its definition, simply put, is the following: “The Traveling Salesman Problem is to find a routing of a salesman who starts from a home location, visits a prescribed set of cities and returns to the original location in such a way that the total distance travelled is minimum and each city is visited exactly once” [6]. A more succinct definition states that: “Given n cities and their intermediate distances, find a shortest route traversing each city exactly once” [31].

2.4 RESOLVING TSP TO GRAPHS

Aside from the existence of many forms of definitions, such as quadratic representation, integer linear representation, linear permutation representation, binary programming task and more [6,32], TSP can also be expressed using terminology and notions from graph theory, as a problem defined over a graph G. This alternative formulation also aids with the presentation of some commonly encountered cases of the TSP in the section below. The definition follows: Let $G=(V,E)$ be a complete graph (directed or undirected) and F be the family of all Hamiltonian cycles (circuits) in G. For each edge $e \in E$ a cost c_e is prescribed. Then the TSP is to find a tour (a Hamiltonian cycle) in G such as the sum of the costs of the tour edges is the least possible. Let the node set $V = \{1,2,\ldots,n\}$. The matrix $C = (c_{i,j})_{nxn}$ is called the cost matrix (or distance matrix or weight matrix), where the $(i,j)^{th}$ entry c_{ij} corresponds to the cost of the edge joining node i to node j in G [6,32,34].

2.5 TSP CASES

There are two major categorizations of the TSP: The first refers to the nature of the graph G upon which the problem is defined, or alternatively to the nature of the associated cost matrix C. This in turn distinguishes the TSP into two categories – Symmetric and Asymmetric [6,32]. The second aforementioned major categorization is based on whether the cost function utilized (and the subsequent cost matrix C) satisfies the triangle inequality or not, thus creating two more categories for the TSP, namely Metric and Non-Metric/General [8,32].

Symmetric TSP (STSP): The Symmetric TSP is one of the most common cases, where the distance between each city pair is independent of the direction the salesman travels. In the graph representation of the problem, this translates in that the associated graph $G = (V,E)$ is considered complete and undirected, or, equivalently that the cost matrix C is symmetric, i.e. $C(i,j) = C(j,i)$. [6,32].

Asymmetric TSP (ATSP): Contrary to the definition above, the distance between each city pair is dependent of the direction the salesman travels, thus implying that the associated graph $G = (V,A)$ is directed and the accompanying cost matrix C is asymmetric, i.e. $C(i,j) \neq C(j,i)$. The ATSP is a generalization of the STSP; every ATSP can be reduced to STSP simply by doubling the number of cities/nodes and while introducing distances of negative values [6,32].

Metric TSP: In Metric TSP, the inter-city distances, i.e. the cost matrix C, satisfies the triangle inequality [8], which means that $C(i,j) \leq C(i,k)+C(k,j)$ for all i, j, k. Simply put, this ensures that a direct path between two cities/vertices is at least as short as any indirect path. In particular, this is the case of planar problems for which the cities/vertices are points $Pi = (Xi,Yi,)$ within the 2D or higher-order plane. A special, widely adopted and much studied case of Metric TSP is the Euclidean TSP, where the cost function is the Euclidean distance, also known as L2 norm, other notable case being the Rectilinear TSP (Manhattan distance) [6,32,34].

Non-Metric/General TSP: In this case the above premise does not hold, and so Non-Metric TSP is the case where the cost matrix C does not obey the triangle inequality, and in fact, C does not even have to express distance; for example, G's edge weights could actually express the cost of airline tickets, where it is often the case that taking an indirect route is less expensive than a direct route.

Other Cases: Many other, not so commonly encountered variations of the TSP exist, the most notable being the MAX TSP, Bottleneck TSP, TSP with Multiple visits (TSPM), the Messenger Problem, Clustered TSP, Generalized TSP, The m-salesman TSP and others. For more information on these non-typical cases, the reader is encouraged to refer to [6].

In the work presented in the following chapters, the case considered is the Symmetric Euclidean TSP.

2.6 TSP COMPLEXITY

The Traveling Salesman Problem in its general case is an NP-hard problem [3,6,8], while its decision version, formulated in [1] as "Given a set of n cities, the distances between all city pairs and a limit D, is there a tour visiting each city exactly once of length less than D?", is NP-complete [5].Moreover, two of the former's most commonly studied special cases, namely the Metric TSP and Euclidean TSP, are also proven to be NP-hard [8,33]. Either way, a direct corollary of this fact is that, assuming the widely believed conjecture that P≠NP, any algorithm designed to find optimal tours should have a worst-case running time that grows faster than any polynomial [36], thus implying a super-polynomial or exponential growth rate, as the size of the input problem instance increases. Despite the recent advances in both the fields of High Performance Computing and combinatorial optimization theory, the aforementioned fact renders the exact solution of large or very large problem instances a non-trivial matter, especially when temporal constraints are introduced [6].

2.7 HEURISTICS AND APPROXIMATE SOLUTIONS

The premise stated above has, for many decades now, led the TSP community towards the development of an abundance of computational methods and algorithms, whose main purpose is to find near-optimal solutions/tours in a fast and efficient manner when compared to the resources required by exact solvers [35]. In fact, the research on that particular area has been so active and prolific, that an immense number of such heuristics exist, covering a broad range of the two key parameters commonly referred to in their evaluation: running time (Time Complexity) and output tour quality (distance above optimal) [36]. In the section below follows a brief presentation of the main categories of TSP heuristics along with their respective representatives.

Construction Heuristics: Construction heuristics are the simplest and weakest type of heuristics, and as such they tend to provide low quality (far from the global optimum) tours, thus rendering them as a means to generate a "good enough" starting tour for more powerful heuristics, such as Improvement Heuristics, reviewed next [32]. Still, their purpose is quite important, as the success of the latter depends heavily on the quality of the initial solution [6]. The intrinsic characteristic that all algorithms within this category share, is that they all create tours from scratch, successively adding new nodes/cities at each step and terminating when a solution (not necessarily a good one) is found, never trying to improve it [6,35,37,38]. In other words, a tour is successively built and parts already built remain in a certain sense unchanged throughout the execution of the algorithm [39]. Notable construction heuristics include Nearest-Neighbor heuristics, Insertion heuristics, Spanning Tree heuristics, Savings Methods, Greedy Algorithms, Christofides and Strip Heuristic [35,36,37,38,39].

Improvement/Local Search Heuristics: In the - admittedly vast - TSP bibliography, there are many definitions for the Improvement/Local Search Heuristics, the most descriptive being, to the author's point of view, the one found in [36]: "…Such an algorithm is specified in terms of a class of operations (exchanges or moves) that can be used to convert one tour into another. Given a feasible tour, the algorithm then repeatedly performs operations from the given class, so long as each reduces the length of the current tour, until a tour is reached for which no further operation yields an improvement (a locally optimal tour). Alternatively, this can be viewed as a neighborhood search process, where each tour has an associated neighborhood of adjacent tours, i.e. those that can be reached in a single move, and one continually moves to a better neighbor until no better neighbors exist". This native characteristic has coined Improvement heuristics the term "Local Search heuristics" [6]. It should be noted that, although commonly mistaken, Local Search is an algorithmic principle rather than an algorithm itself [8]. Notable representatives within this category are k-opt heuristics (k ∈ {2, 2.5, 3, 4+}) and their variants, Lin-Kernighan (LK) and its variants/implementations and Node & Edge Insertion [6,8,32,35,36,37,38,39].

Metaheuristics: This last category contains the most powerful, effective and complex (both algorithmically and temporally) heuristics. The main differentiating point between Improvement heuristics and Metaheuristics is that the former terminate as soon as a (not necessarily good) local optimum is reached, while the latter allow a more thorough exploration of the solution search space [6], by enabling uphill moves (i.e. moves that increase the current tour length) in order to escape from local optima. Some algorithms achieve that functionality by performing random modifications to the tour once a locally optimal one is found, while others use more elaborate schemes. In some occasions, uphill moves are allowed even before a local optimum is reached [35,36]. In general, there are two major subcategories: Local Search-based Metaheuristics and Evolutionary Metaheuristics. As their name implies, the former utilize simple Improvement/Local search heuristics and modify a single active solution at a time, while the latter are based on heuristics derived from natural evolutionary processes and generate new solutions from a collection of multiple simultaneously active solutions (the population) [6]. The most frequently encountered Metaheuristics are Simulated/Quantum Annealing, Tabu Search, Iterated Lin Kernighan, Iterated Local Search, Large-Step Markov Chains, Genetic Algorithms, Ant Colony Optimization and Min-Max Ant System [6,35,36,38,40,41].

2.8 PRACTICAL TSP APPLICATIONS

The practical, real life applications of the TSP span across a wide range of disciplines, some of which are the following: network and VLSI design, machine vision, scheduling and corporate planning, distribution management, logistics, database query design, frequency assignment, computational biology, vehicle routing, job sequencing and X-ray crystallography.

CHAPTER 3: RELATED WORK

3.1 INTRODUCTION & CHAPTER STRUCTURE

This chapter focuses on the review of related work on parallel implementations of TSP heuristics, and shows that most of the recently conducted as well as ongoing research on the aforementioned subject is based on Graphics Processing Units-GPUs, with only a few, mostly older, works utilizing FPGA-based solutions. Despite the fact that it is single-threaded, and since part of the comparisons made later in the thesis depend on it, a brief presentation of the Concorde TSP software package, the state-of-the-art in TSP solving, is also provided.

3.2 PARALLEL PROGRAMMING APIs

Parallelism is considered the future of computing since, over the past decade, processor architectures have embraced it as an essential and in some cases sole pathway to enhancing performance. Technical challenges faced in attaining ever increasing clock speeds within a certain thermal envelope, physical and financial issues regarding shrinking chip manufacturing processes further, stemming from pushing silicon beyond its capabilities and the huge R&D costs associated as well as a possible end of Moore's law in the near future, has led CPU and GPU manufacturers to turn to spatial solutions, thus signaling the rise of multicore designs. On the other hand, GPUs have evolved from fixed function rendering devices and graphics engines to a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially surpasses those offered by CPUs. Moreover, the fact that in modern high performance computer systems CPUs are usually paired with powerful GPUs, has led to the desire to harness the combined computational power of both subsystems. All these factors have necessitated the emergence of appropriate APIs and frameworks that will enable computer scientists and programmers alike to easily access the vast computing resources available in modern machines. Below follows a brief description of the most common parallel and heterogeneous programming APIs and frameworks:

OpenMP: OpenMP (Open Multi-Processing) is a portable API that supports cross-platform shared memory multiprocessing programming in C, C++ and FORTRAN, on the majority of the available computing platforms, processor architectures and operating systems, and features a set of compiler directives, library routines and environment variables that affect run-time behavior. The OpenMP Architecture Review Board (ARB), the nonprofit consortium that manages OpenMP, published the first API specifications in 1997.

CUDA: CUDA® is a general purpose parallel computing platform and programming model introduced by NVIDIA Corp. in 2006,that allows software engineers to effectively leverage the parallel computing capabilities of NVIDIA GPUs, by exposing and providing explicit and direct access to the GPU's virtual instruction set, multi-level memory architecture and computational elements. It is designed to work with a variety of general-purpose high level programming languages, such as C, C++, FORTRAN and Python.

OpenCL: OpenCL (Open Computing Language) is an open, royalty-free framework based on the C/C++ programming languages, for creating applications that execute on heterogeneous parallel processing platforms consisting of CPUs, GPUs, DSPs (Digital Signal Processors) and FPGAs, providing programmers with portable and efficient access to the computational power offered by such complex systems. OpenCL originates from Apple Inc. which submitted it in 2008 as a proposal to the non-profit technology consortium Khronos Group, responsible from then on for its maintenance.

3.3 METAHEURISTICS ENCOUNTERED IN THIS CHAPTER

Ant Colony Optimization (ACO): ACO is a metaheuristic that is inspired by the foraging behavior of real ants that enables them to find the shortest paths between food sources and their nest. In the process of finding the shortest path the ants are guided by the level of a substance called pheromone that each ant deposits on the ground while moving between food sources and the nest. Stronger pheromone concentrations indicate a short route traversed by a significant number of ants, and likely to be traversed again. [46].

Genetic Algorithms (GA): In GAs, a population of solutions evolves over time, yielding a sequence of generations. A new population is created from the old one by using a process of reproduction and selection, where the former is often performed by crossover and/or mutation and the latter decides which individuals form the next generation. A crossover operator combines the features of two parent solutions in order to create children. Mutation operators simply change one solution. The idea is that, analogous to natural evolution, the quality of the solutions in the population will increase over time [60].

Simulated Annealing (SA): SA originates from statistical mechanics and the annealing technique involving heating and controlled cooling of materials. It transposes the process of annealing to the solution of an optimization problem: the objective function of the problem, likened to the energy of a heated material, is minimized using a controllable parameter of the algorithm that is associated with a fictitious temperature level. SA is a technique that can be applied to any minimization process, based on either random or deterministic iterative steps, using simple heuristics such as 2- or 3-OPT [46].

Iterated Local Search (ILS): In ILS, one iteratively builds a sequence of solutions generated by an embedded heuristic such as 2- or 3-OPT, leading to far better solutions than if one were to use repeated random trials of that heuristic, such as in RRHC. The nature of this approach implies that there must be only a single chain (i.e. solution) that is being followed and improved throughout the execution of the metaheuristic, thus excluding population based algorithms such as GA [41].

Random Restart Hill Climbing (RRHC): RRHC is similar to ILS, with the exception that once a local minimum is reached, a random-rather than a carefully selected one-permutation is applied to the tour, before restarting the embedded heuristic. After a sufficient number of iterations, the operation comes to a stop and returns the best tour found, i.e. the one with the smallest length. RRHC can be considered as a simplified version of ILS.

3.4 PARALLEL IMPLEMENTATIONS OF TSP (META) HEURISTICS

3.4.1 SOFTWARE (CPU) BASED IMPLEMENTATIONS

One of the earliest attempts in the parallelization of TSP heuristics can be found in [42], where a parallel version of the 2-OPT algorithm was implemented on a Parsytec GCel consisting of 512 transputers. It provided “good” speedups against the classic/serial 2-OPT, while retaining the same quality of results. More recent works, such as [42], [43] and [44], exploit the highly parallelizable nature of ACO and Genetic Algorithms together with the parallelization capabilities of the OpenMP framework in order to provide better quality of results than simpler heuristics while lowering the execution time. In [42], speedups of up to nearly 6.0x are provided when compared with the serial implementation, with an efficiency factor of 0.74, running ACO on 8 processors on a SGI Origin2000 parallel machine. In the theoretical evaluation performed in [43] is estimated that the speedup achieved when running OpenMP-based ACO is linear within the range of 1 to 26 CPUs. In [44], the efficiency of the parallel computation of the TSP using GA on a multiprocessor cluster is investigated; it concludes that a ~4x speedup is attainable in a 1000-city problem, when utilizing 5 CPUs, with a reported efficiency of 83%. In 2008, [45] suggested parallel computational models for several TSP metaheuristics, such as ACO, GA and SA, that utilize multiple independent runs on a multicomputer platform comprising of 10 Intel Pentium 4 3.2GHz workstations, and exploiting a hybrid MPI + OpenMP approach. Its results, based on three TSPLIB instances of size 51, 150 and 439 cites, show that both parallel ACO and SA are capable of achieving speedups of up to 9.0x compared to serial execution, while parallel GA reached a maximum of 5.5x. It concludes that for larger problems, speedups are still attainable though decreasing, due to additional communication overhead.

3.4.2 HARDWARE BASED IMPLEMENTATIONS, FPGA-BASED

To the author's knowledge, there are few published works on the mapping of TSP heuristics on FPGAs that provide concrete evaluation results. In 2002, I. Skliarova et al. [47] presented an FPGA-based implementation of GA for the TSP and its industrial applications. The proposed architecture was evaluated on a Xilinx XCV812E Virtex FPGA, where only the most computing-intensive part of the GA was assigned to the hardware, while the rest of the algorithm's functionality was executed on an 800MHz Pentium III based host machine. The results obtained by this hybrid approach were quite impressive, achieving a maximum speedup of almost 50.0x for a 724-city instance, although it is noted that the system's performance depends heavily on the problem instance under consideration as well as some key algorithm parameters. A similar work also based on GA and using the TSP as a case study is the one found in [48]. The proposed architecture was designed using a High-Level programming language (Handel-C) and implemented in a Xilinx XCV2000E Virtex-E FPGA. The attained speedups vary from about 1.5x to almost 39x when compared to a serial implementation running on a 1.7GHz Pentium 4 machine, depending on the problem instance. It is concluded that, as far as execution time is concerned, the FPGA implementation is superior to the software one when the problem size increases or when better solutions (nearer to the optimum) must be found. In 2007, I. Mavroidis et al. [29] presented a high performance and resource-efficient FPGA-based 2-OPT solver, able to exploit the fine-grained parallelism made available by the proposed notion of Symmetrical 2-OPT moves. The hardware architecture was evaluated on a Xilinx XC2VP100-6 Virtex 2 Pro FPGA, and was shown that it was capable of achieving speedups ranging from 60x to almost 100x when compared to the corresponding software-based serial implementation, and, on average, a 6x speedup compared to the state-of-the-art Concorde TSP software package, both executed on a 3GHz Pentium 4 based machine.

3.4.3 HARDWARE BASED IMPLEMENTATIONS, GPU-BASED

As is mentioned in this chapter's introduction, the emphasis of the recently conducted and ongoing research on the parallelization of TSP heuristics has been on their mapping in highly parallel GPU kernels. Thus, there exist many such published works, ranging from simple improvement heuristics to metaheuristics such as Local Search schemes and evolutionary algorithms. For the sake of clarity, the review is presented in a categorized manner.

k-opt: In [49] and [50], K. Rocki et al. present high performance CUDA/GPU implementations of both the 2-OPT and 3-OPT improvement heuristics. The results, obtained by evaluating their proposed kernels on a 1536-streaming processor NVIDIA GeForce GTX 680 GPU using 13 TSPLIB problem instances ranging in size from 100 to 4461 cities, show that speedups up to 26x (1.53 TFLOPS) are achievable compared to a 32-core CPU or 500x when the sequential algorithm is concerned. They note that the performance gains of 3-OPT are higher than those of 2-OPT due to the fact that its increased complexity leads to more efficient utilization of the GPU's hardware. It is concluded that, as far as GPU computing is concerned, the number of streaming processors constantly increases, while the on-chip cache/shared memory limitations remain almost unchanged, thus imposing its treatment as a scarce resource.

Random Restart Hill Climbing & Iterated Local Search: In [51] (2011), MA O'neil et al. present a CUDA-based implementation of RRHC using 2-OPT moves for determining high–quality solutions to the TSP. Their approach is based on the logical assumption that by running hundreds of thousands of times the 2-OPT heuristic on randomly permutated starting tours of a given TSP instance, an optimal or near-optimal solution is bound to be found. Indeed, their results show that 10^5 random restarts are enough to yield the optimal solutions for four out of the five 100-city TSPLIB instances tested. The proposed implementation exploits both inter- (independent randomly permutated instances of the problem) as well as intra- (move evaluations within a given randomly permutated instance of the problem) 2-OPT solver, termed "climber", parallelism. The kernel was evaluated on a 448-streaming processor NVIDIA Tesla C2050 GPU capable of supporting 14336 simultaneous climbers, and yielded a throughput of 20 billion 2-OPT move evaluations per second, similar to this delivered by a corresponding 256-thread CPU-based implementation running on 32 8-core 2GHz Intel Xeon CPUs. When compared to the sequential implementation executed on a 2.53GHz Intel Xeon CPU, the speedup achieved was 62x. It is noted that the proposed implementation is memory-bound, since the limited amount of on-chip cache did not allow the evaluation of problem sizes larger than 100 cities. A refinement of the aforementioned work by the same authors can be found in [52] (2015), where a more efficient exploitation of the hierarchy of the available hardware resources combined with the utilization of a more recent and powerful NVIDIA K40 GPU with 2880 stream processors, led to a throughput increase from a maximum of 20 to 60 billion 2-OPT move evaluations per second and an 8x speedup when compared to the equivalent OpenMP implementation running on 20 2.7GHz Intel Xeon cores. This new version also removed the previous limitation on the problem size, supporting problem instances with more than 15000 cities.

K. Rocki et al. [53] (2012), based on the work conducted in [51], proposed a similar implementation in which they managed to relax the limitation imposed by the on-chip caches on the maximum supported problem size (from 100 cities to 6000 cities), by using them to store the cities' coordinates instead of pre-calculated inter-city distances, and computing the latter on-the-fly. Despite the aforementioned fact, their evaluations on the same GPU as in [51] showed that the proposed implementation was up to 90% slower than [51], due to the lack of intra-climber parallelization exploitation and the performance hit induced by the constant computation of inter-city distances. In [54] (2013), the same authors presented a refinement of their previous work [53], where they considered the case of Iterated Local Search with 2-OPT moves. It was decided to parallelize and accelerate on the GPU only the local search itself (i.e. a single run of the 2-OPT algorithm) since profiling showed that 90% of the ILS execution time is spent exclusively on it. The aforementioned limit on problem sizes was completely overcome through an appropriate ordering of the city coordinates and the utilization of extra structures, rendering the proposed implementation capable of solving arbitrarily large problem instances, in the range of 700K plus cities. The evaluation on a CUDA-enabled NVIDIA GeForce GTX 680 GPU showed that the proposed implementation was 5 to 45 times faster than the equivalent OpenCL-based parallel implementation running on a 6 core 3.3GHz Intel Core i7 3960X CPU. It is concluded that while the algorithm reaches very good solutions in less than a second for problem instances up to about 7500 cities, its main disadvantage is that it still requires a significant amount of time to reach the first local minimum in case of larger instances.

Another published work related to those already mentioned is [55], which yet again proposes parallelization strategies for the ILS metaheuristic, but this time the latter is combined with 3-OPT moves instead of 2-OPT utilized in the works previously discussed. The authors describe several CUDA-based parallelization models which were evaluated on an NVIDIA Tesla C2050 GPU. The experimental results, although highly dependent on the dataset under consideration (size-wise), showed that performance was maximized when the on-chip cache was more efficiently utilized, the accesses to global memory kept at a minimum and when more worker threads were dispatched, since the latter was necessary to hide the memory latency. Again, the small amounts of on-chip cache proved the limiting factor in the maximum supported problem instance.

Genetic Algorithm: In [56], N. Fujimoto et al. present a method to efficiently map GA on GPU using the CUDA API. It is noted that in order to maximize the performance gains, the off-chip memory access latency must be hidden, which in turn implies that, even though only about 30K can be simultaneously active, hundreds of thousands of threads must be dispatched for execution, so as when a number of them is temporarily stalled due to dependencies, others, until then inactive, can begin/resume execution. In other words, ideally, the GPU should remain constantly saturated, and the authors accomplish that by parallelizing not only the GA in its metaheuristic level, but also the underlying crossover operator as well as the embedded 2-OPT local search. The experimental results showed that for problem sizes ranging from about 100 to almost 500 cities, speedups from 9x to 24x are attainable, when compared to the equivalent serial implementation running on a 3GHz Intel Core 2 Duo E6850 CPU. The evaluation platform was an NVIDIA GeForce GTX 285 GPU. A similar work can be found in [57] by S. Chen et al. who, as stated, intended to develop a CPU-based GA metaheuristic for the TSP and then parallelize it with CUDA on GPU for performance gains. The proposed implementation was evaluated on an NVIDIA Tesla C2050 GPU, and the speedups achieved where characterized as “insignificant” due to the limitations imposed on the efficient utilization of the available computing resources by the applied synchronization solution.

Ant Colony Optimization: In [58], C. José M. et al. present a novel parallel implementation of ACO, in which the two main stages of the algorithm, namely Tour Construction and Pheromone Update, are both GPU-accelerated, contrary to the usual approach of implementing the former in GPU and the latter in CPU, thus parallelizing both. New models for both stages, aiming to provide improved mapping efficiency, are also proposed and evaluated, since the authors note that the existing ones exhibit many weaknesses, such as very low GPU utilization, serialization bottlenecks and more. The proposed kernels were evaluated on an NVIDIA Tesla M2050 CUDA-enabled GPU using a set of benchmark instances from the TSPLIB library, obtaining speedups up to 29x when compared to serial execution, depending on the dataset used. Another work on the parallelization of ACO can be found in [59] by A. Uchida et al. where an efficient CUDA-based implementation, able to tackle many of the issues and shortcomings inherent to GPU computing is proposed. Optimizations applied include the maximization of the on-chip cache usage while keeping the global, off-chip memory accesses to a minimum, as well as the introduction of a new, less computing-intensive stochastic method for determining ant movement. In order to hide the high latency of global memory and increase the attainable bandwidth, memory coalescing, the technique of simultaneously accessing consecutive addresses is applied. The proposed implementation was evaluated on a 512-streaming processor NVIDIA GeForce GTX 580 GPU, with the experimental results showing that speedups ranging from 7.5x to 44x were achieved, compared to the sequential version executed on a 2.8GHz Intel Core i7 860 CPU. The tested TSPLIB instances included problem sizes from 198 up to 2392 cities. It was also noted that some key parameters of the CUDA API, such as block size and number, play a major role in the resulting performance and thus should be chosen after careful experimentation.

A comprehensive literature review on the role of GPUs in discrete combinatorial optimization, referring to some of the works presented in this chapter plus more, can be found in [60].

3.5 THE CONCORDE TSP SOLVER

The Concorde TSP Solver is probably the most well-known and one of the best CPU-based exact solvers for the Symmetric TSP to date [32], freely available for academic research use (source code plus precompiled executables) [61]. It is written in ANSI C and consists of roughly 130000 lines of code spanning across more than 700 functions, thus rendering it highly customizable, permitting users to create specialized codes for TSP-like problems. Concorde was the result of the long-term collaboration between D. Applegate, R. Bixby, V. Chvatal and W. Cook [62], who began its development most probably during the late 1980s. Since its initial release, in the early 1990s, it gradually gained the acclaim of the TSP community, as it was the first (and in some cases the only) solver that managed to successfully obtain the optimal solutions to all 110 problem instances within the TSPLIB library of TSP datasets, the largest consisting of 85900 vertices and stemming from a VLSI application [63]. Besides from an efficient exact TSP solver, Concorde also includes a great number of highly optimized heuristics for the TSP, such as Greedy, Nearest Neighbor, Boruvka, Chained Lin-Kernighan, 2/3-OPT and more. It should be noted that, although Concorde is available both as source code as well as in a precompiled executable, the latter’s GUI interface does not provide access to the full range of the supported heuristics. The last official release of Concorde was in 2003. For an in-depth analysis of Concorde’s underlying architecture and its employed algorithms, the reader is encouraged to refer to [3], written by its creators.

CHAPTER 4: THE 2-OPT IMPROVEMENT HEURISTIC

4.1 INTRODUCTION & CHAPTER STRUCTURE

This chapter presents the 2-OPT improvement heuristic along with some very similar variations. Their functionality is explained while their performance characteristics are evaluated and analyzed through the scope of multiple metrics, such as tour quality, effectiveness and more. Next, the focus is shifted towards the Random Restart Hill Climbing (RRHC) scheme, in order to examine how the Symmetrical 2-OPT variant fares when combined with the aforementioned metaheuristic. The effect of utilizing the square of the Euclidean distance-instead of the Euclidean distance itself-as the cost function on the behavior of 2-OPT, using the Symmetrical variant as a case study, is assessed afterwards. Finally, the chapter concludes with the presentation of the fine-grained parallelism offered by the Symmetrical 2-OPT variant. As was mentioned in the end of the Chapter 2, in section 2.5, the work presented in this thesis concerns the Symmetric Euclidean TSP case, thus 2-OPT is approached and examined under that specific scope.

4.2 2-OPT AT A GLANCE (2OPT)

2-OPT is one of the oldest and probably the most basic improvement heuristic for the TSP, proposed by G. A. Croes in his 1958 paper titled “A Method for solving Traveling-Salesman Problems” [64], although the notion of the 2-OPT move itself had already been introduced in 1956 by M.M. Flood [65,34,36]. Over the years, 2-OPT managed to gain wide recognition and acceptance within the TSP community, due to the fact that it yields impressive results in terms of both tour quality and running time, considering that its functionality is exceptionally simple; it incrementally improves an initial tour by applying a series of length-reducing tour modifications, known as “2-OPT moves”, until no further such modifications can be found, i.e. a local optimum is reached, in which case the resulting tour is considered “2-optimal” [37]. A 2-OPT move deletes two edges, thus breaking the tour into two separate parts, and then reconnects those paths in the other possible way, which is equivalent to reversing the order of the cities between those edges (segment reversal) [29].

4.3 2-OPT MOVE EVALUATION

This section demonstrates the process of evaluating whether a 2-OPT move is length-reducing or not, through an example: Consider a tour T already defined over a given TSP instance represented as a graph $\mathrm{G}= (\mathrm{V},\mathrm{E})$ containing an arbitrary number of cities/vertices, which are, in turn, presented as a collection of points within the 2D Euclidean space. Now consider four specific vertices of G, namely A, B, C, and D. In the tour T, these vertices are connected through 2 separate edges, namely $\mathrm{E1}$ and $\mathrm{E2}$ as follows: Edge $\mathrm{E1}$ connects vertices A and B, while edge $\mathrm{E2}$ connects vertices C and D. These vertices are then deleted and the aforementioned nodes are reconnected the only other possible way that yields a new complete TSP tour T’: A is connected with C through edge $\mathrm{E1}'$ and B is connected with D through edge $\mathrm{E2}'$.

Next, in order to assess if the move made yields length-reducing results, the change in tour length when transitioning from T to T’ must be computed. There is no need to compute the length/weights of every edge in E though, since all vertices, with the exception of those involved in the 2-OPT move, have kept their relative distances unchanged (tour reversal). Thus, the only edges involved in the aforementioned computation are $\mathrm{E1}$ and $\mathrm{E2}$ from T and $\mathrm{E1}'$ and $\mathrm{E2}'$ from T'. Finally, the proposed 2-OPT move is length-reducing if and only if

The “Sum of lengths of proposed edges E1’ and E2’” minus the “Sum of lengths of their previous versions E1 and E2” is negative. More formally if and only if

$$\text{Delta} = (\text{length}(E1') + \text{length}(E2')) - (\text{length}(E1) + \text{length}(E2)) < 0 \quad => $$
$$\text{Delta} = (\text{distance}(A,C) + \text{distance}(B,D)) - (\text{distance}(A,B) + \text{distance}(C,D)) < 0$$

Since the 2D Euclidean case is assumed, function *distance()* refers to the Euclidean distance or L2 norm, which, considering two points K, L in the 2D Euclidean space, is calculated as

$$\text{distance}(K,L) = \sqrt{(Kx - Lx)^2 + (Ky - Ly)^2}$$

4.4 2-OPT COMPLEXITY

The worst-case running time of a straightforward 2-OPT implementation is exponential, and this holds even for Euclidean instances [8]. Thus, supposing that the number of cities/nodes within a certain TSP instance equals N, then the cost of finding a single improving move is $\Theta(N^2)$ assuming all possible N*(N-1)/2 moves must be considered (true 2-optimality) [32,38]. If the actual number of 2-OPT moves occurring in a full run of the heuristic, which is theoretically about Θ(NlogN), is taken into account, then the overall time increases to a figure more closely resembling $\Theta(N^3\log N)$ [6,36]. Many workarounds have been proposed to tackle this issue, though some of them give up the guarantee of true 2-optimality and/or increase the algorithm’s spatial complexity. Such techniques include heavy pruning of legal moves, “don’t look bits”, utilization of sophisticated structures such as k-d or black-red trees, preprocessing to generate neighbor lists and other auxiliary structures and more [29]. The experimental results conducted in the related review of Johnson and McGeoch [36] show that when such workarounds are exploited, the runtimes actually observed for the 2-optimization phase are definitely subquadratic and no worse than $O(N^{1.2})$. In this thesis the straightforward 2-OPT implementation is considered.

4.5 2-OPT TOUR QUALITY

On instances that fulfill the triangle inequality, the worst-case approximation ratio of 2-OPT is $O(4\sqrt{N})$, which means that the worst achievable local optimum is within a factor of $O(4\sqrt{N})$ of the global optimum, where N equals the number of cities within a given problem instance. In the case that Euclidean distances are considered, the aforementioned ratio becomes O(logN) [66]. Related, more experiment-oriented studies have shown that the tours produced by 2-OPT are on average about 5% above the Held-Karp bound [38]. The Held-Karp bound is itself an approximation to the length of the optimal solution of a given problem instance; very useful when evaluating the empirical performance of heuristics, as its computation is much faster than solving the instance under consideration to optimality and then computing the optimal tour length.

4.6 A GRAY AREA

Up until this point, the reader has intentionally not been provided with an explicit description of 2-OPT’s functionality, due to the fact that there seems to be a major ambiguity in the TSP bibliography concerning a key aspect of its behavior. More specifically, based on the literature review of this thesis, most of the papers that approach 2-OPT and/or the TSP in general in a theoretical manner, such as [32,34,35,38], imply more or less directly that a length-reducing move is always applied, as soon as one is found, at which point the heuristic resets and a new iteration begins. On the contrary, the papers that refer to parallel implementations of 2-OPT or to metaheuristics that embed it, such as [50] and [52], state explicitly that at every iteration all the possible moves are evaluated in order to find and apply the one that yields the best results (i.e. reduces the tour length the most). G. Reinelt [39] and M. Hahsler [37] also suggest the latter.

4.7 CLARIFICATION & 2-OPT STRATEGIES

It turns out that both the approaches described above are valid [6]. The one that seeks the best move is referred to as the "*best improvement*" strategy, while the other is referred to as "*first improvement*". It is noted that normally, practical implementations do not use the *best improvement* strategy but rather rely on the latter, since *first improvement* saves running time and often yields better results (quality-wise). These two approaches will be from now on referred to as $2OPT_{FIRST}$ and $2OPT_{BEST}$ (first improvement and best improvement respectively).

```
2OPT-FIRST
CONSIDER A TOUR T OF LENGTH N

PROCESS 2OPT_FIRST(T)
    START:
    FOR EVERY i IN RANGE 1 TO N-2
        FOR EVERY j IN RANGE i+2 TO N-1
            -> COMPUTE E1 EDGE LENGTH (DISTANCE(i,i+1)).
            -> COMPUTE E2 EDGE LENGTH (DISTANCE(j,j+1)).
            -> COMPUTE E1' EDGE LENGTH (DISTANCE(i,j)).
            -> COMPUTE E2' EDGE LENGTH (DISTANCE(i+1,j+1)).
            -> COMPUTE DISTANCE DELTA ((E1'+E2')-(E1+E2)).
            IF {DELTA IS NEGATIVE}
                -> APPLY THE LENGTH-REDUCING MOVE BY REPLACING EDGES E1 AND E2
                   WITH E1' AND E2' RESPECTIVELY.
                -> GOTO START.
            END_IF
        END_FOR
    END_FOR
END PROCESS
```

Figure 3: $2OPT_{FIRST}$ pseudocode.

```
2OPT-BEST
CONSIDER A TOUR T OF LENGTH N

PROCESS 2OPT_BEST(T)
    START:
    -> SET THE CURRENTLY MAXIMUM NEGATIVE CHANGE IN TOUR LENGTH TO ZERO.
    FOR EVERY i IN RANGE 1 TO N-2
        FOR EVERY j IN RANGE i+2 TO N-1
            -> COMPUTE E1 EDGE LENGTH (DISTANCE(i,i+1)).
            -> COMPUTE E2 EDGE LENGTH (DISTANCE(j,j+1)).
            -> COMPUTE E1' EDGE LENGTH (DISTANCE(i,j)).
            -> COMPUTE E2' EDGE LENGTH (DISTANCE(i+1,j+1)).
            -> COMPUTE DISTANCE DELTA ((E1'+E2')-(E1+E2)).
            IF {DELTA IS LESS THAN THE CURRENTLY MAXIMUM NEGATIVE CHANGE IN
               TOUR LENGTH}
                -> UPDATE CURRENTLY MAXIMUM NEGATIVE CHANGE WITH DELTA.
                -> STORE THE CORRESPONDING INDICES i, j IN BEST_i, BEST_j.
            END_IF
        END_FOR
    END_FOR
    IF {AT LEAST ONE LENGTH-REDUCING MOVE WAS FOUND DURING THE SEARCH}
        -> APPLY THE BEST LENGTH-REDUCING MOVE STORED IN BEST_j, BEST_j.
        -> GOTO START.
    END_IF
END PROCESS
```

Figure 4: $2\text{-}OPT_{BEST}$ pseudocode.

4.8 THE RANDOMIZED VARIANT (R2OPT)

As it can be inferred from the pseudocodes above, 2-OPT evaluates moves in a strictly deterministic manner, always starting from the first position of the tour-containing structure (a simple array in the case considered here) and proceeding sequentially from that point on. A contrasting approach is to choose the two edges which are candidates for deletion in a non-deterministic fashion, i.e. pick a 2-OPT move at random. The 2-OPT variant that evaluates moves in this form is termed Randomized 2-OPT (*R2OPT* from now on). R2OPT has no BEST strategy; if the randomly chosen move is length-reducing it is always applied.

```
R2OPT
CONSIDER A TOUR T OF LENGTH N

PROCESS R2OPT(T)
        START:
        -> RANDOMLY CHOOSE TWO EDGES E1 AND E2 BY GENERATING TWO UNEQUAL RANDOM INTEGERS i, j
           WITHIN AN APPROPRIATE RANGE.
        //THE PSEUDOCODE SECTION BELOW CONSIDERS THE CASE THAT (i<j). AN EQUIVALENT SECTION COVERS
        //THE OPPOSITE CASE (NOT SHOWN HERE).
        -> COMPUTE E1 EDGE LENGTH (DISTANCE(i-1,i)).
        -> COMPUTE E2 EDGE LENGTH (DISTANCE(j,j+1)).
        -> COMPUTE E1' EDGE LENGTH (DISTANCE(i-1,j)).
        -> COMPUTE E2' EDGE LENGTH (DISTANCE(i,j+1)).
        -> COMPUTE DISTANCE DELTA ((E1'+E2')-(E1+E2)).
        IF {DELTA IS NEGATIVE}
                -> APPLY THE LENGTH-REDUCING MOVE BY REPLACING EDGES E1 AND E2 WITH E1' AND E2'
                   RESPECTIVELY.
        END_IF
        IF {A LENGTH REDUCING MOVE IS NOT FOUND WITHIN N^2 CONSECUTIVE EVALUATIONS}
                -> GOTO END.
        ELSE
                -> GOTO START.
        END_IF
        END:
END PROCESS
```

Figure 5: R2OPT pseudocode.

4.9 THE SYMMETRICAL VARIANT (S2OPT)

This section describes the Symmetrical 2-OPT variant, as proposed in 2007 by I. Mavroidis et al. [29].

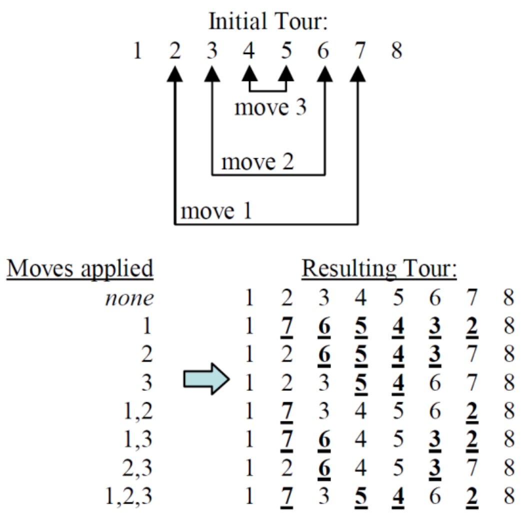
The very purpose of the design of Symmetrical 2-OPT (S2OPT from now on) was the uncovering of fine-grain parallelism and its subsequent exploitation through an efficient FPGA implementation. Having the aforementioned objective, the algorithm, besides the desired high-level functionality, is largely defined as a set of hardware-oriented structures and operations along with the schedule upon which the latter act on the former. Thus, for the time being, only an abstract overview of S2OPT is presented, as the hardware specifics are a matter of the next Chapter, Chapter 5.

S2OPT resembles 2-OPT$_{FIRST}$ in the sense that a) at each iteration any length reducing move is applied, not just the best one, and b) it evaluates moves in a strictly deterministic manner as well. The main differentiating points between these two cases are that a) S2OPT applies all the available length-reducing moves discovered within a given iteration and b) it evaluates moves in a considerably different pattern than the one utilized by 2-OPT in general. The name given to the algorithm by its authors possibly provides an insight to the intra-move relation.

The following excerpt provides an illustration and the definition/description of Symmetrical 2-OPT moves, extracted from [29].

Consider two tour segments, SegmA and SegmB, each consisting of a given ordering of the tour's cities. A set of segments (or equivalently their corresponding 2-OPT moves) are defined 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= {C1…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}, down to whichever segment is the smallest."



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 6: Example of three symmetrical 2-OPT moves. Image extracted from [29].

The figure above displays a tour consisting of 8 cities, presented as a sequence of numbers which represent their position within the starting tour, along with three possible symmetrical moves. The matrix shows the modifications incurred to the initial tour after the application of one, two or all three of the aforementioned moves.

With careful observation, two key points, largely influencing the characteristics of the derived hardware architecture, arise from the example above: a) "After the application of any number of the symmetrical moves under consideration, each city will either remain in its original position within the starting tour, or swap positions with its symmetrical city; two cities are considered symmetrical if they are at the two ends of one of the symmetrical segments", and b) "Whether the cities at the two ends of a specific segment will keep their positions or swap them, 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."

It should be noted that, as is probably evident in the example tour displayed in the figure below, reversing tour segment {C3…C7} is equivalent to reversing the remaining tour segment {C8, C1, C2}. Thus, S2OPT, during its search for length-reducing segment reversals only considers segments that consist of up to half the total number of cities.

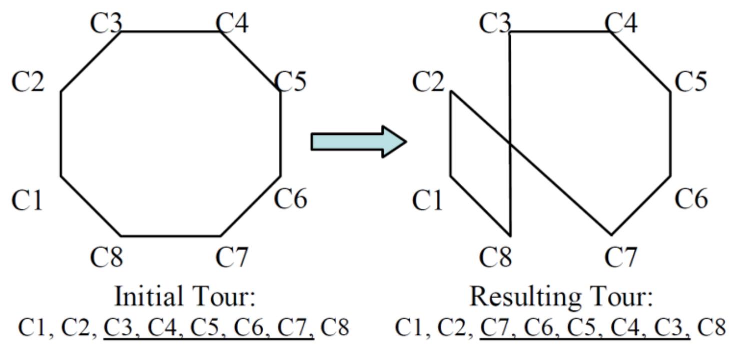

C3
C4
C2
C5
C1
C6
C8
C7
Initial Tour:
C1, C2, C3, C4, C5, C6, C7, C8
C3
C4
C2
C5
C1
C6
C8
C7
Resulting Tour:
C1, C2, C7, C6, C5, C4, C3, C8

Figure 7: Application of a 2-OPT move. Image extracted from [29].

S2OPT's serialized pseudocode is cited below. Due to the fact mentioned at the beginning of this section, its presentation is fairly abstract as it hides most of the architecture/implementation-specific functionality.

```
S2OPT
CONSIDER A TOUR T OF LENGTH N

PROCESS S2OPT(T)
        -> SET PHASE TO A.
        START:
        FOR EACH SYMMETRICAL-CITY-PAIR WITHIN THE CURRENT TOUR'S N/2 FIRST CITIES (i IN RANGE 1 TO N/4)
                -> COMPUTE E1 EDGE LENGTH (DISTANCE(i,(i-1)).
                -> COMPUTE E2 EDGE LENGTH (DISTANCE((N/2-i+2),(N/2-i+1)).
                -> COMPUTE E1' EDGE LENGTH (DISTANCE((i-1),(N/2-i+1))).
                -> COMPUTE E2' EDGE LENGTH (DISTANCE(i,(n/2-i+2)).
                -> COMPUTE DISTANCE DELTA ((E1'+E2')-(E1+E2)).
        END_FOR
        -> COMPUTE THE RESULTING LENGTH-REDUCING SYMMETRICAL-SEGMENT-REVERSALS BASED ON THE
           REVIOUSLY COMPUTED DELTAs.
        -> COMPUTE THE CORRESPONDING LENGTH-REDUCING SYMMETRICAL-CITY-PAIR SWAPS.
        -> APPLY THE AFOREMENTIONED LENGTH-REDUCING SWAPS.
        IF {PHASE EQUALS A}
                -> PERFORM PREDEFINED TOUR MODIFICATION A.
                -> SET PHASE TO A'.
                -> GOTO START.
        ELSE
                -> PERFORM THE INVERSE OF THE PREDIFINED TOUR MODIFICATION A, A'.
                -> SET PHASE TO A.
        END_IF
        -> PERFORM PREDEFINED TOUR MODIFICATION B.
        IF{A LENGTH-REDUCING MOVE IS NOT FOUND WITHIN N CONSECUTIVE ITERATIONS}
                -> GOTO END.
        ELSE
                -> GOTO START.
        END_IF
        END:
END PROCESS
```

Figure 8: S2OPT abstract pseudocode.

At this point it should be noted that in the following ***chapters*** *only the S2OPT case is considered. The rest of the variants (including the original) have been presented here for the sake of a holistic approach to the subject.*

4.10 THE TSPLIB TSP DATASET LIBRARY

Due to the significant scientific interest towards the TSP, which began to emerge right after its introduction in the 1940s, a huge number of works ranging from theoretical studies to algorithm proposals and high performance implementations started to amass. It quickly became evident that a set of standard TSP instances was needed so as to enable researchers and computer scientists worldwide to compare, assess and evaluate the performance characteristics of their proposed works with those of others. The most prominent solution to this issue originated from G. Reinelt, who, in the late 1980s, began collecting TSP instances from various sources and of various types such as actual locations within cities or countries and VLSI design layouts. His work has been widely adopted by the TSP community and is considered as the standard TSP benchmark package. G. Reinelt's TSPLIB [67] collection of TSP datasets can be found in [68].

TSPLIB's dataset naming convention obeys the following pattern: {instance name/type}{number of cities within the dataset}. For example, *berlin52* implies that the number of cities within the dataset is 52.

4.11 PERFORMANCE ANALYSIS & COMPARISONS OF 2-OPT VARIANTS

All the experimental work presented in the current as well as the following sections has been conducted by the author in MATLAB environment.

At this point, it is probably evident that there exist quite a few different approaches to the 2-OPT algorithm, even in its original formulation. Thus, a question is raised concerning their performance characteristics, especially through a comparison point of view. The aim of the current section is the evaluation of the aforementioned approaches through various criteria such as tour quality, convergence rate and more. The obtained results, along with the appropriate annotations are presented below:

4.11.1 TOUR QUALITY

The first evaluation metric is the output tour length, expressed in the form of the percentage above the optimal length (error); a smaller value translates to higher quality tour, closer to the optimal, and vice versa.

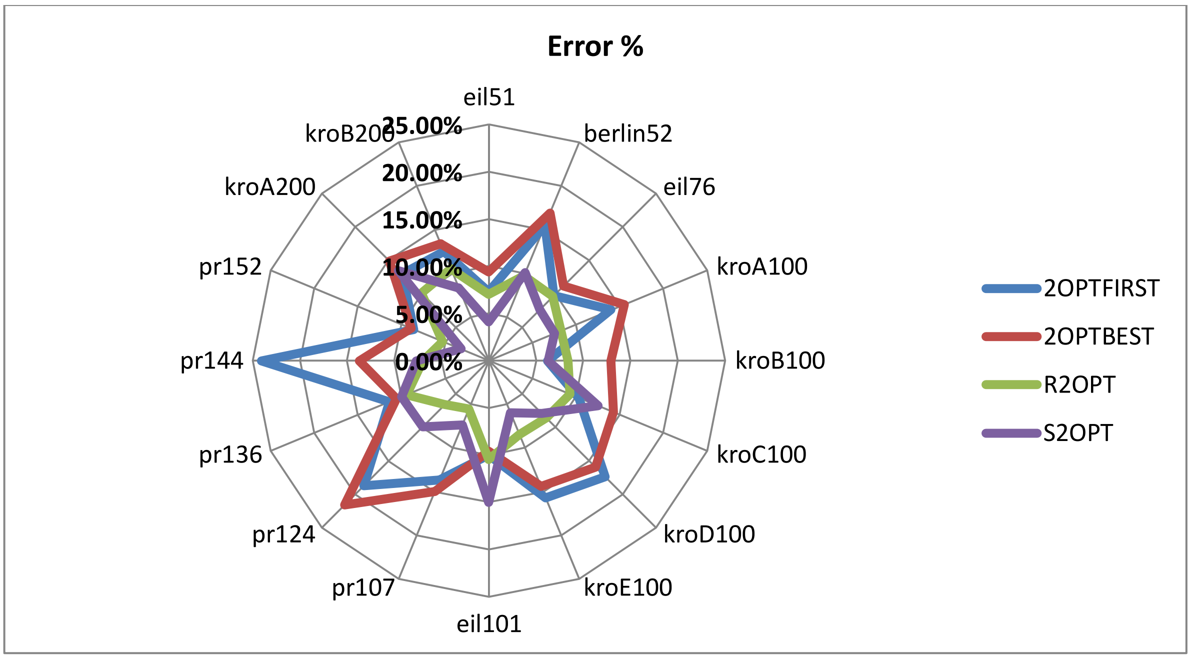


Figure 9: Comparison of the quality of the tours yielded by the four 2-OPT implementations under consideration; 16 TSPLIB datasets are considered.

Judging by the graph it is fairly evident that, as far as tour quality is concerned, the worst performer in all but one case is 2OPT in either its *FIRST* or *BEST* approaches, or even in both, while R2OPT along with S2OPT generally fare much better. On average, $2OPT_{BEST}$ yields 13.6% longer than optimal tours while $2OPT_{FIRST}$ follows with 13%. In a completely different league, S2OPT yields the second-best results, boasting an average error of 8.6%, closely trailing R2OPT which manages an average error of 8%.

Furthermore, although only by 0.6 pp.[1], the experimental results verify the notion mentioned in section 4.7, that $2OPT_{FIRST}$ often yields better results than $2OPT_{BEST}$.

4.11.2 AVERAGE TOUR CONVERGENCE RATE

The second evaluation criterion is the tour convergence rate managed by each approach. The convergence rate refers to how quickly, in terms of moves evaluated or elapsed runtime, the currently modified tour trends towards the local optimum. When a given dataset is considered, an increased convergence rate usually indicates decreased runtimes, and more importantly, it means that in the case that the heuristic is prematurely terminated, the 2-OPT implementation with the higher convergence rate will probably have managed to produce a higher quality tour than the one with the lower rate, and vice versa.

In the following chart, an approximation of the average convergence rate achieved throughout the full run of each dataset is shown. The values are obtained by computing the average gradient/steepness of the "Tour Length - Moves Evaluated" figure (not shown here), through the following formula:

[1] pp.: Percentage Point, the unit for the arithmetic difference of two percentages.

$$\frac{\sum_{i=1}^{K} \frac{(Yi-1)-Yi}{(Xi-1)-Xi}}{K}$$

- K, the total number of length-reducing moves found/applied.
- Yi, the tour length at the time when length-reducing move i is applied.
- Xi, the total number of applied moves at the time when length reducing move i is applied.

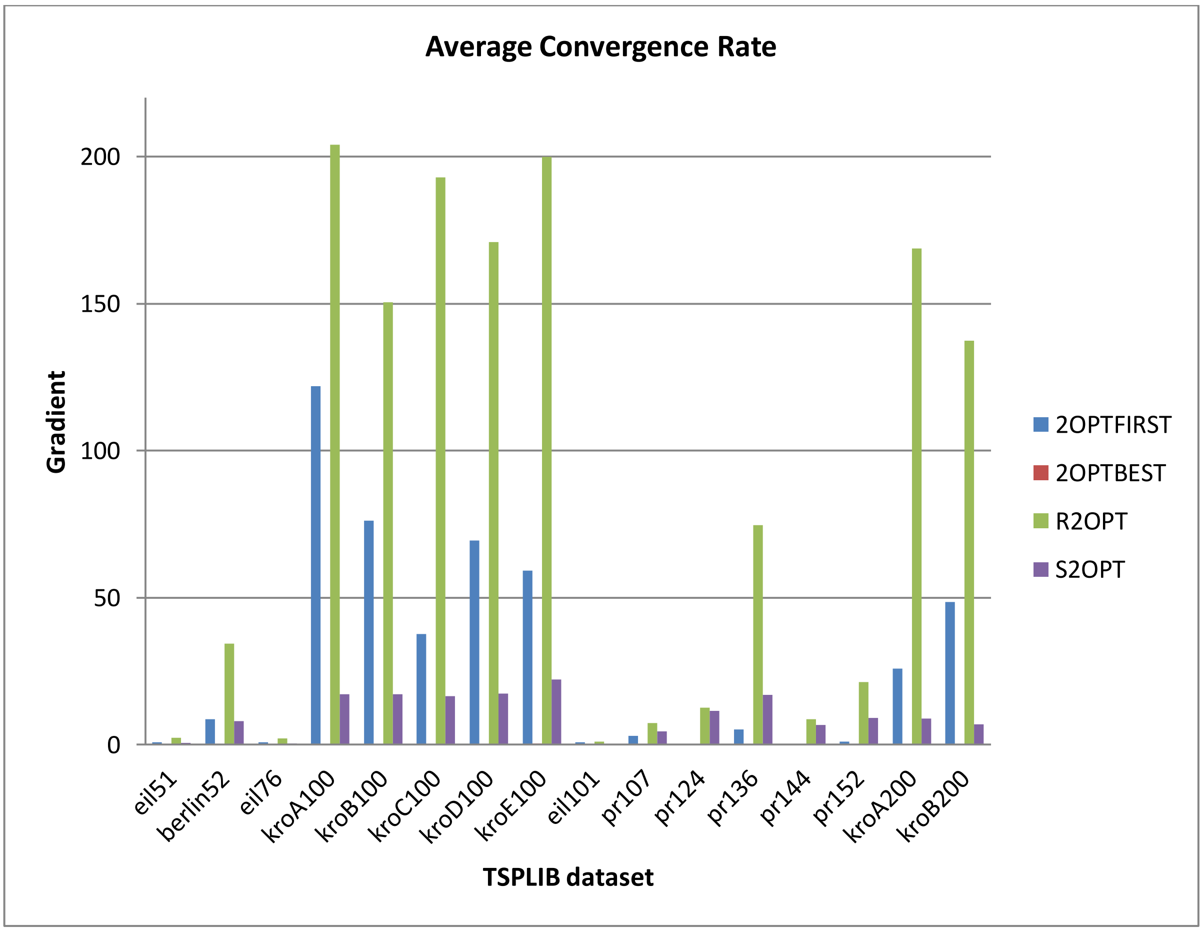


Figure 10: Comparison of the average convergence rate achieved by the four 2-OPT realizations under consideration, obtained through the gradient/steepness approximation; a high value indicates a high convergence rate and vice versa.

The chart above clearly suggests that R2OPT converges much quicker than the rest of the variants, with 2OPT$_{FIRST}$ following far behind and S2OPT even more. 2OPT$_{BEST}$'s strategy of always evaluating all possible moves at every iteration renders its convergence rate minimal compared to those of the rest. More specifically, R2OPT boasts a convergence rate of 87; 2OPT$_{FIRST}$ follows next with 29, while S2OPT ranks third, with a rate of 10. 2OPT$_{BEST}$ converges an order of magnitude slower, with a rate of 0.2.

4.11.3 TOUR LENGTH REDUCTION PER MOVE APPLIED

The third metric provides, in a sense, an insight into "how effective the moves applied by each approach are", by relating the total change in the tour length with the number of the applied moves.

This figure is obtained by calculating the change induced in the tour length by the application of each single move and then computing the average value.

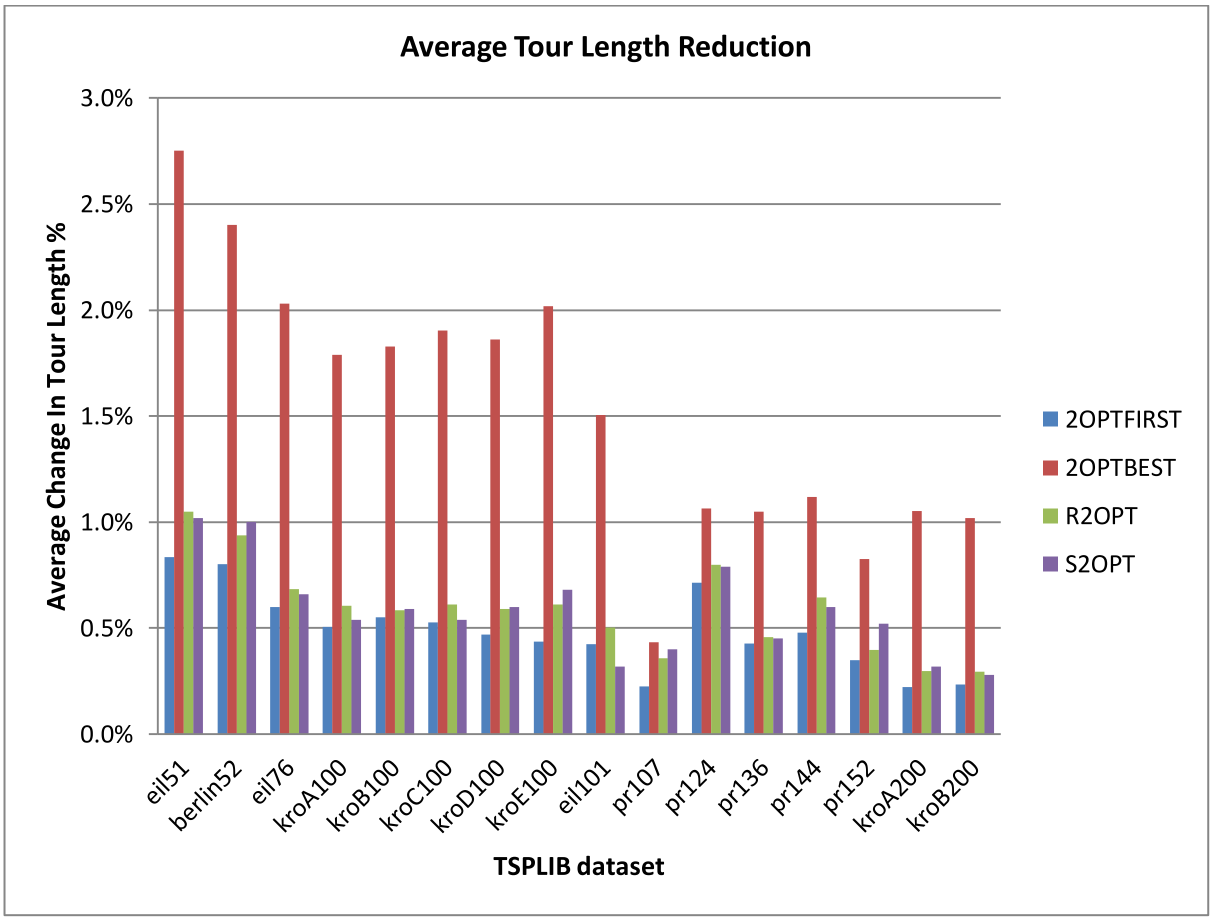


Figure 11: Comparison of the average reductions brought to the tour length by the application of each length-reducing 2-OPT move, among the four 2-OPT implementations under consideration; the reductions are expressed percent changes.

The effect of $2OPT_{BEST}$'s strategy of always seeking for the best possible move is clearly visible, as each one of the moves it applies tends to reduce the tour length in a much greater extent than those of the rest of the implementations, whose results in this aspect are roughly the same. $2OPT_{BEST}$ manages an average reduction of 1.5% per move, while R2OPT, S2OPT and $2OPT_{FIRST}$ follow equally far behind, with 0.59%, 0.58%, and 0.49% equivalently.

4.11.4 NUMBER OF EVALUATED MOVES

The fourth and final metric is the total number of Moves Evaluated, provided here as a means of assessing runtime, since actual runtime measurements would not be representative of real-world performance, due to the limitations imposed on the latter by the MATLAB environment.

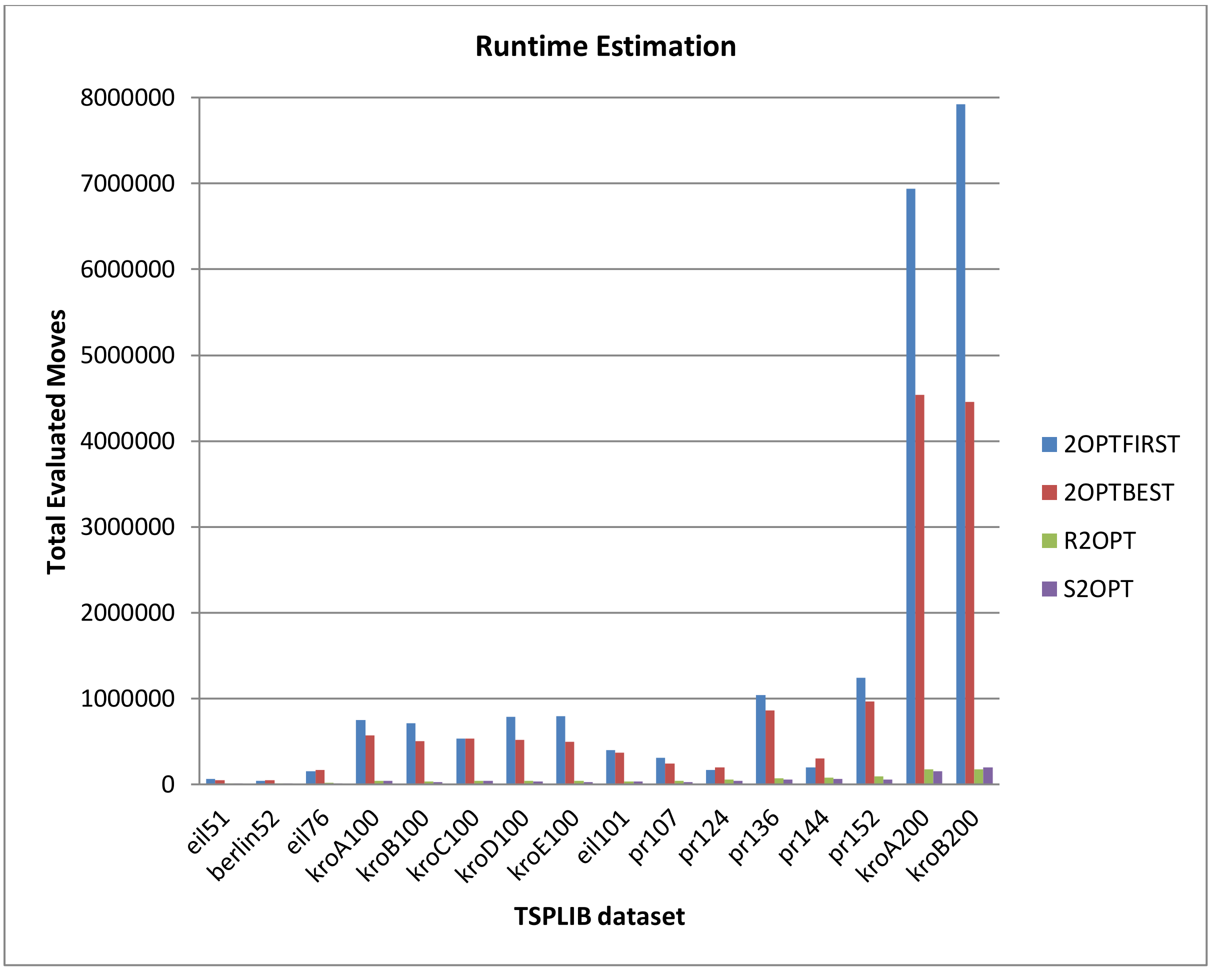


Figure 12: Comparison of the total number of 2-OPT moves evaluated by each 2-OPT implementation; a larger value indicates increased runtimes and vice versa.

Both original 2OPT variants perform significantly more move evaluations prior to termination, a trend persistent in every dataset tested. S2OPT and R2OPT lead the way, with an average of 51300 and 59273 evaluated moves respectively; $2OPT_{BEST}$ and its sibling, $2OPT_{FIRST}$, follow orders of magnitude behind, with figures of 925728 and 1377528 respectively.

An interesting result is that, contrary to the notion stated in section 4.7, $2OPT_{FIRST}$ seems to be actually almost 50% slower than $2OPT_{BEST}$, and not the other way round.

4.12 RRHC: A CASE STUDY WITH SYMMETRICAL 2-OPT (S2OPT)

This section considers the performance evaluation of S2OPT when combined with, or more precisely, when embedded within a “lightweight” yet capable metaheuristic as is Random-Restart Hill Climbing, briefly described in section 3.3 of Chapter 3. Thus, in this case, the main metric of interest is the minimum error (best tour quality) achieved within the set of the executed trials/random-restart-repetitions, and its numeric relation with the average error.

The results obtained after 10^3 iterations are presented in the opposite page.

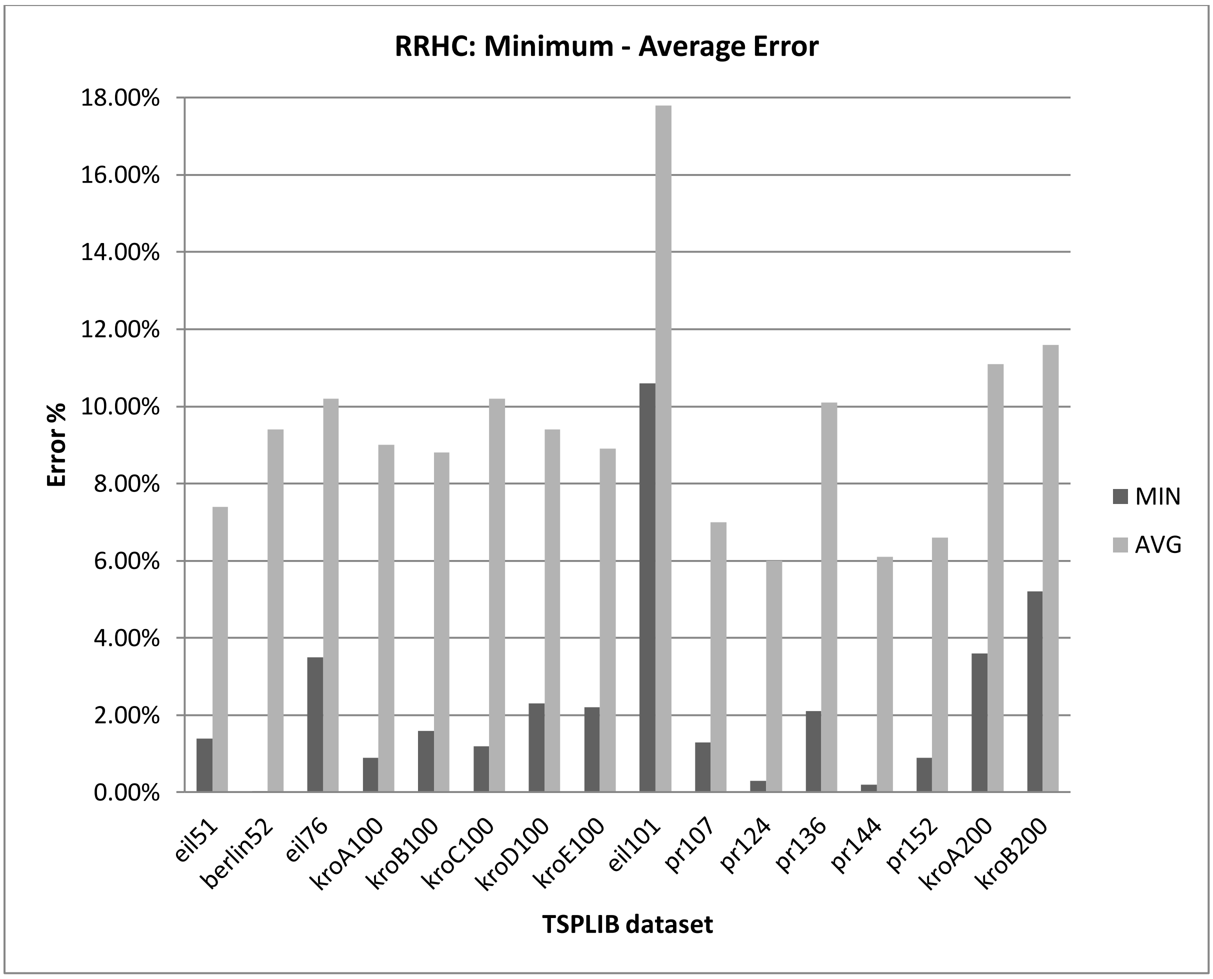


Figure 13: Performance assessment of the S2OPT-based version of the RRHC metaheuristic, in terms of tour quality; the latter is expressed as the percent-change of the final tours' length when compared to the corresponding optimal. The AVG bars refer to the average final tour length obtained in each dataset, averaged over the 10^3 performed runs; the MIN bars refer to the minimum tour length marked during the execution of the aforementioned runs. A lower error percentage translates to higher quality tour and vice versa.

Observing the Minimum and Average Error bars, it is clearly evident that the combination of RRHC with S2OPT yields much better results than those managed by the latter alone. On average, the RRHC+S2OPT combination yields nearly 24 times or almost 7 pp. smaller errors. If the metaheuristic was allowed to execute for more iterations (10^5 for example), the obtained results should be even better, yet this assumption could not be efficiently evaluated due to the long run times induced by the MATLAB environment.

As a side note, another useful inference is the experiment-wise affirmation of the fact mentioned in [39], which is that the quality of the results yielded by improvement heuristics largely depends on the quality of the corresponding input tours, thus signifying the importance of utilizing construction heuristics for the generation of "good enough" starting tours (Chapter 2, section 2.7).

4.13 THE SQUARE OF THE EUCLIDEAN DISTANCE AS THE COST FUNCTION (S2OPT)

This section is dedicated to evaluating the employment of the squared Euclidean distance as the cost function-instead of the Euclidean distance itself-using S2OPT as a case study.

The process of computing the squared Euclidean distance is the same as in the standard Euclidean distance with the exception that the final square root is not calculated. The aforementioned approach has been followed in [29] since, in the case of FPGAs, the calculation of the square root is computationally expensive and it is not easily mapped in hardware, thus requiring the utilization of complex IP cores, at least when the actual function and not an approximation of it is considered.

Using an approximation to the Euclidean distance-such as its square-in a Euclidean problem instance is rather sub-optimal, as is usually the case when approximations in general are concerned. A first major issue is that the squared Euclidean distance does not obey the triangle inequality. A second issue is that when the square root is omitted, the super-linearly-growing quadratic function ($f(x) = x^2$) utilized twice in the calculation of the Euclidean distance tends to "boost" larger values more than smaller ones.

This trend can be trivially illustrated through the graph of the aforementioned quadratic function:

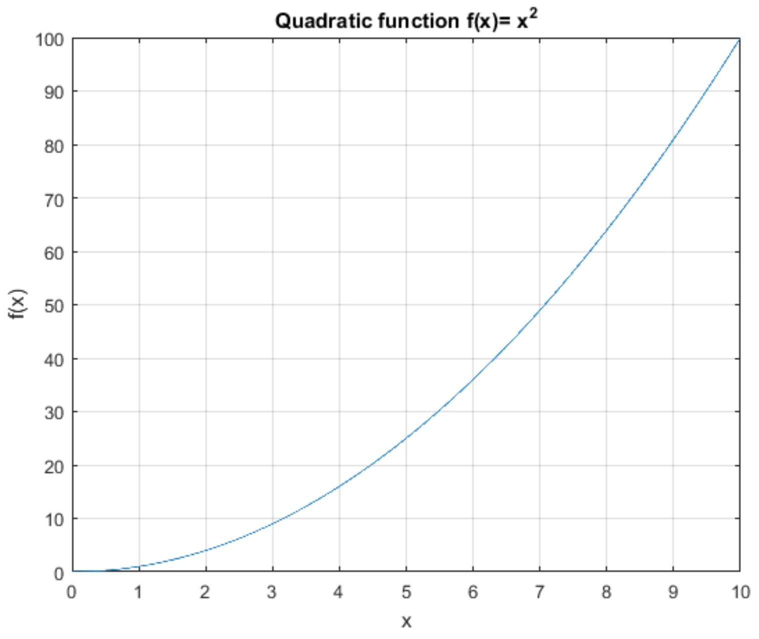


Figure 14: Illustration of the behavior of the subquadratic function $f(x)= x^2$.

In order to determine the effects of this sub-optimality induced to the move evaluation procedure, two different approaches have been taken: Approach A examines the probability that a move is deemed length-reducing under the squared Euclidean cost function but not under the standard Euclidean and vice versa, while approach B assesses the performance of S2OPT when combined with the squared Euclidean distance metric and compares it to that of the standard S2OPT.

Approach A: The evaluation performed by approach A, described above, has been realized through the utilization of the MATLAB script located in the appendix; a related arithmetic example is provided as well.

The results obtained show that out of the 10^6 total evaluated moves:

- 4.658% were applied in the standard Euclidean case but not in the squared Euclidean case. This percentage constitutes actual length-reducing moves that should have been applied but were not discovered in the squared Euclidean case.
- 4.628% were applied in the squared Euclidean case but not in the standard Euclidean case. This percentage actually constitutes length-increasing moves that were applied in the squared Euclidean case while they obviously should not have.

Both events imply a loss in tour quality.

Approach B: In order to assess the impact of the squared Euclidean metric on the overall performance of S2OPT, the exact same methodology as in the previous section (4.13) has been followed (averaging over 10^3 independent trials, resulting from the application of an equal number of random permutations to each dataset's initial tour), except of course for the change in the cost function.

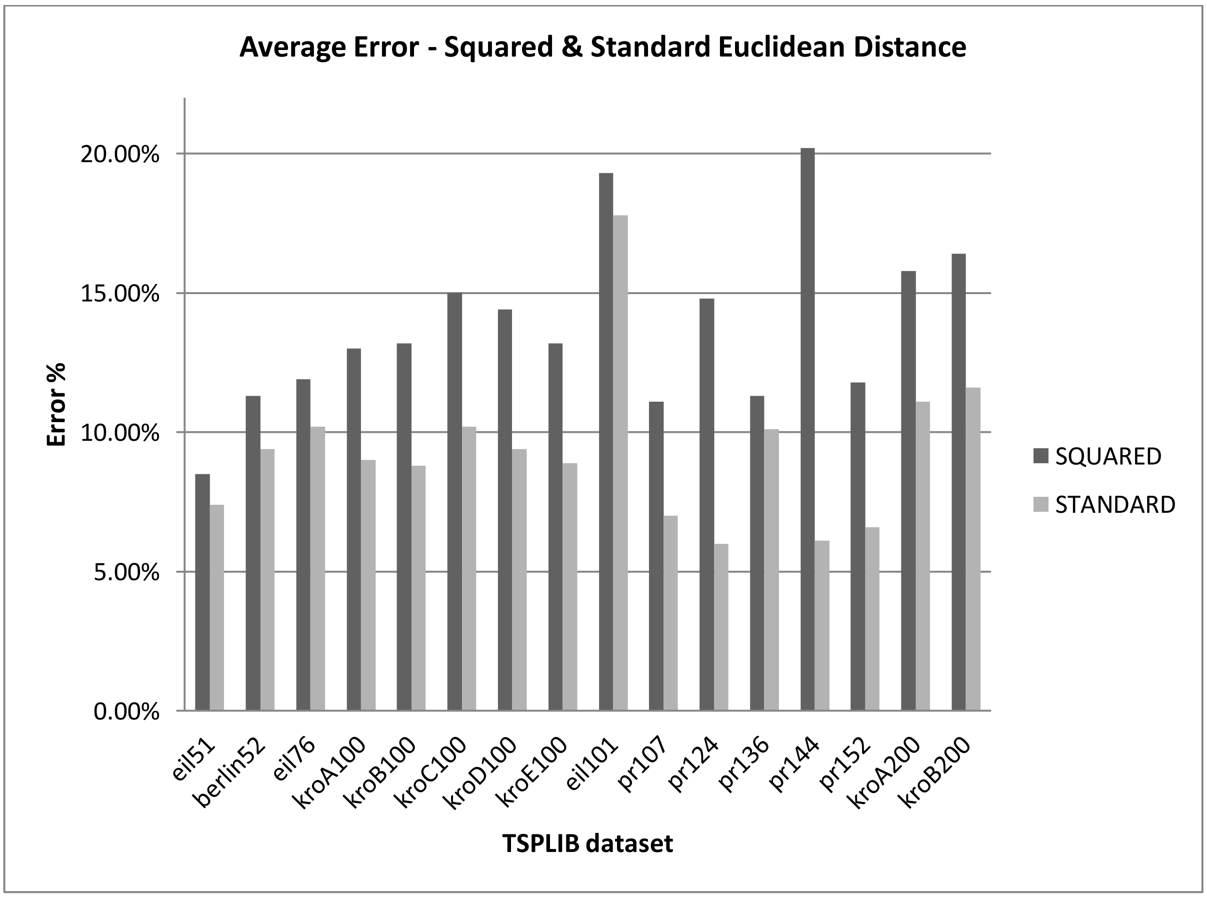


Figure 15: Comparison of the average quality of the tours obtained by each metric. The quality is expressed in terms of the percent-change of the final tours' length when compared to the corresponding optimal; a lower error percentage translates to a higher quality tour and vice versa. Results averaged over 10^3 independent runs.

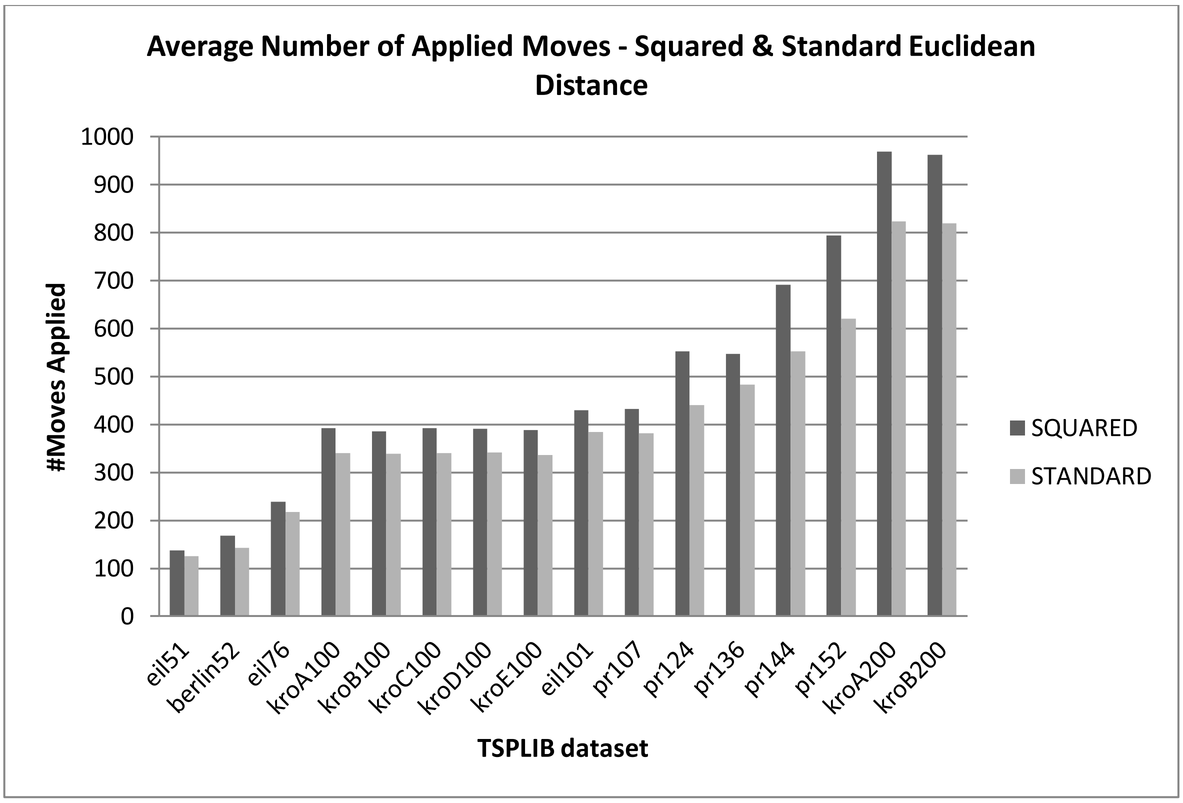


Figure 16: Comparison of the average number of length-reducing 2-OPT moves applied by each metric. Results averaged over 10^3 independent trials.

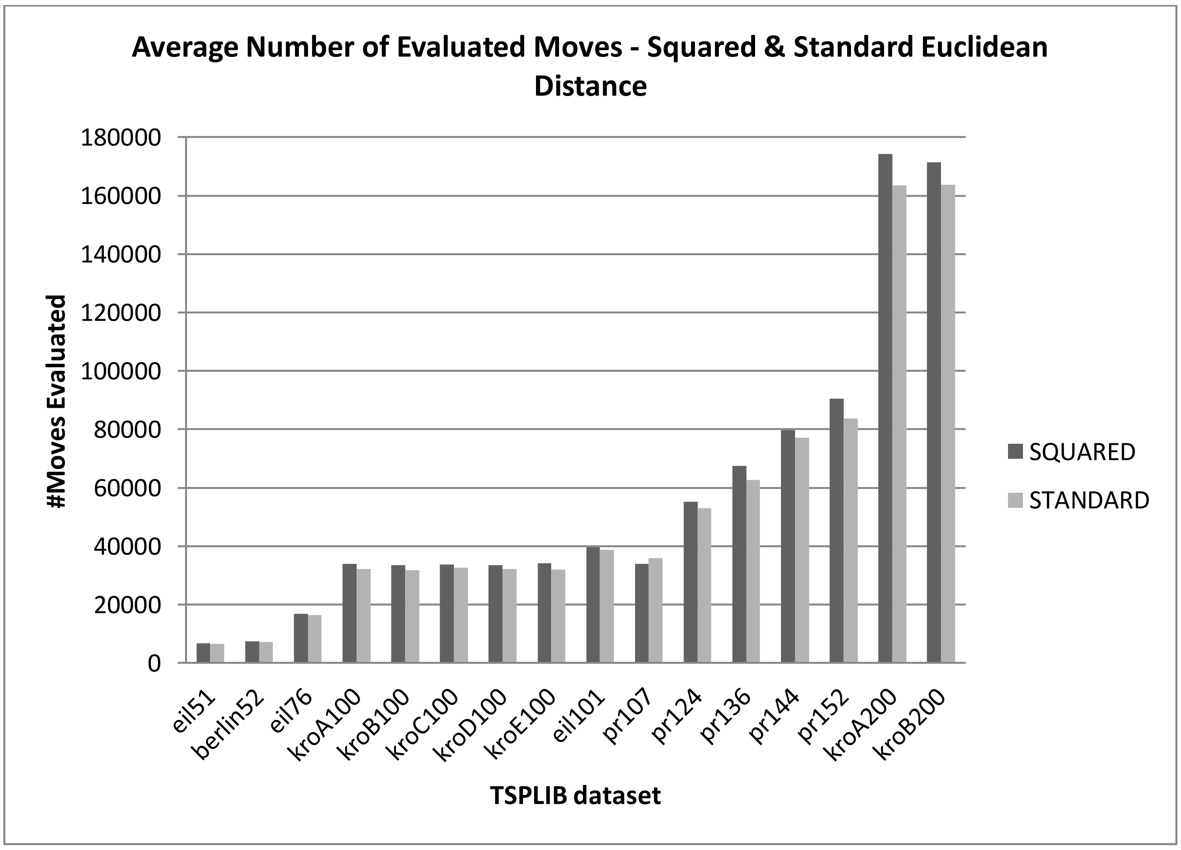


Figure 17: Comparison of the average number of 2-OPT moves evaluated by each metric. Results averaged over 10^3 independent runs.

Comments:

- As far as average tour quality is concerned, the squared Euclidean version performs notably worse than the standard version, in every problem instance; the actual severity of the error is dataset-dependent. The sole exception is the *eil101* dataset where the minimum error (not shown here) is 0.7 pp. smaller. On average, squared Euclidean S2OPT yields errors 4.4 pp. larger than standard S2OPT.
- Squared Euclidean S2OPT tends to evaluate as well as apply more moves than standard S2OPT; the former implies longer running times. The only exception to this is the *pr107* dataset where standard S2OPT evaluated 5.5% more moves; on average, squared Euclidean S2OPT applies 16.4% and evaluates 4.2% more moves than standard S2OPT.

Despite the fact that squared Euclidean S2OPT performs in every aspect slightly worse than standard S2OPT, the thesis considers the former as well, so that a direct, performance-wise comparison with the original work [29] (comparison presented in chapter 6) is made feasible.

4.14 AVAILABLE PARALLELISM OF 2-OPT VARIANTS

As far as 2-OPT parallelization strategies are concerned, there have been proposed many relevant techniques such as Geometric Partitioning or Tour-based partitioning [36]. Yet, these constitute a kind of distributed, coarse-grain approach to parallelization, and as such they are not expounded since this thesis considers fine-grain approaches only.

In this section, the parallelization options offered by the various 2-OPT implementations are discussed:

$2OPT_{BEST}$: The parallelism offered by *$2OPT_{BEST}$* lies in that the move evaluations can be performed simultaneously, since all the comparisons that need to be conducted are known a-priori due to the deterministic nature of the algorithm. The fact that at every iteration all possible moves are examined also contributes largely to its parallelization abilities, as it strengthens the predictability of its behavior. At the end of the evaluation stage, a reduction between all the length-reducing moves is required, in order to assess the one that yields the best results, apply it, and start the evaluation process again.

$2OPT_{FIRST}$: Although $2OPT_{FIRST}$'s functionality closely resembles that of $2OPT_{BEST}$'s, the fact that the former applies the first length-reducing move encountered during the search instead of the best one incurs a loss to determinism, since the location of this move within the search space cannot be known a-priori. Thus, the parallelization abilities of $2OPT_{FIRST}$ are inferior to those of $2OPT_{BEST}$.

S2OPT: The same observations that are made on *$2OPT_{BEST}$* apply in the S2OPT case as well, with the exception that all length-reducing moves are applied, not just the most efficient one. Yet, this operation can still be performed in parallel due to the symmetry of the moves.

R2OPT: Contrary to *$2OPT_{BEST}$* and S2OPT, R2OPT cannot be efficiently parallelized-on FPGA at least-due to its intrinsic, highly non-deterministic move evaluation scheme, rendering it inappropriate for such an implementation.

Final Thoughts: Based on the aforementioned remarks, only 2OPTBEST and S2OPT are eligible for subsequent consideration; further investigation into each one's intrinsic characteristics shows that the most capable-in terms of parallelization-implementation is S2OPT due to following findings:

- S2OPT utilizes a much more consistent memory/tour access pattern enabled by the symmetry of the moves, a fact that largely simplifies the memory architecture requirements, which, in turn, leads to a cleaner and more capable-performance-wise-design. Apart from that, the symmetry of the moves defines distinct and independent pairs of cities, a fact that allows all the necessary city swaps to be performed within the same single cycle.
- The reduction operation in S2OPT is computationally less demanding, since there is no need to assess the most efficient length-reducing move; if a move is length-reducing is applied anyway. This gives the ability to apply all the required city-pair swaps as soon as the evaluation stage is complete-even in a single cycle after it.
- S2OPT was designed from the very beginning with the intention of uncovering fine-grained parallelism.

The following chapter will cover, among others, the mapping of the Symmetrical 2-OPT variant in FPGA.

CHAPTER 5: S2OPT HARDWARE ARCHITECTURES & THE IMPLEMENTATION TOOLS

5.1 INTRODUCTION & CHAPTER STRUCTURE

The aim of the current chapter is the presentation of the software tools utilized throughout the preparation of this thesis along with an in-depth analysis of the implemented hardware architectures. More specifically, the chapter begins with a few words on the architecture emulator implemented in MATLAB, and then moves on to the concise description of the actual implementation tools, namely Vivado and Vivado HLS. A few remarks on their behavior and performance characteristics in relation to each other are also noted. The focus then shifts towards the comprehensive presentation of S2OPT's VHDL- and HLS-based architectures; extensive, per-module block diagrams for the former, and workflow, source code revisions and directives for the latter, are provided.

5.2 THE MATLAB-BASED HARDWARE ARCHITECTURE EMULATOR

A first major step towards the preparation of the thesis was the implementation of a software emulator of the hardware architecture under consideration. The implementation environment of choice was MATLAB, due to its extremely versatile data manipulation and viewing capabilities, convenient UI and easily accessible debugging and verification methods. On the other hand, a notable weakness of MATLAB is its reduced performance in terms of execution times when compared with the executables produced by other programming languages and associated toolchains, such as the combination of C with the highly optimized GNU GCC compiler. Yet this did not constitute a suspending factor since the emulator's main objective was the functional assessment of the aforementioned hardware architecture and not its runtime performance benchmarking.

The hardware architecture emulator's role during the development of this thesis was of great importance due to the following factors:

- Verify the correct apprehension of the hardware functionality as described in [29], and help quantify the extent to which the obtained results match those referred in the aforementioned source.
- Verify that the hardware architecture operates as intended.
- Expedite the hardware design and evaluation phase since it acted as a reference point.
- Expedite the debugging process.
- Aid in the assessment of a few parameters, important to the acceleration of the benchmarking process.
- Act as a source for the extraction of the experimental results obtained in the previous chapter, which were fundamental for the assessment of S2OPT from a purely algorithmic point of view.

5.3 THE XILINX VIVADO DESIGN SUITE

The main software tools utilized during the preparation of the thesis, namely the Vivado Integrated Design Environment (IDE) and Vivado High Level Synthesis (HLS), are both components of the Vivado

Design Suite (DS) produced by Xilinx Inc., a software package aiming at the design, integration, implementation and evaluation of HDL-based hardware systems.

The Vivado DS was introduced in April 2012 with the intention of replacing the aging, now discontinued Xilinx ISE Design Suite as the company's mainline tool chain, mostly due to the latter's inability to efficiently support the company's newer, much larger and complex reconfigurable devices (FPGAs). Thus, Vivado DS is recommended by Xilinx for new system designs with 7-series and later products, while ISE offers support for the company's legacy, pre-7-series devices.

Besides being able to handle the latest FPGAs from Xilinx, Vivado DS also brings a broad range of improvements over its predecessor, such as the integration of many-previously separate tools-into a single intuitively designed environment, improved productivity, more user-friendly, flexible and interactive UI, and a revamped underlying engine yielding much faster response/compilation times and a wide variety of available optimization strategies. At this point, it should be noted that the aforementioned enhancements were actually noticed by the author during his occupation/interaction with the tool, and do not in any way constitute an advertisement.

A short presentation of the utilized tools and design flows follows; a more elaborate discussion of the latter is found in the next chapter.

Vivado IDE: Historically, the Vivado IDE constitutes an evolution of the PlanAhead tool included in the ISE DS since version 10.1. Currently, it is a standalone tool and the main design environment of the Vivado DS, incorporating all the functionality mentioned earlier in this section along with the traditional Register Transfer Level (RTL)-to-bitstream FPGA design flow and a newer system-level integration flow focusing on Intellectual Property (IP)-centric designs.

Vivado HLS: Vivado HLS constitutes Xilinx's approach to the High-Level Synthesis paradigm, briefly described in section 1.3 of Chapter 1. It enables the transformation of programs written in a high level language such as C, C++ and SystemC as well as OpenCL API C kernels into an RTL implementation that can be directly targeted into Xilinx-powered FPGAs, thus offering a design flow that completely bypasses the complex and time-consuming process of manually specifying RTL. The aforementioned transformation process can be either fully automatic or semi-automatic, by guiding the compiler towards a single or multiple desired output characteristics/goals through the exploitation of an extensive set of directives, able to largely influence the generated design in terms of throughput, latency, I/O interfacing, resource utilization and more. It should also be noted that the tool contains a host of ready-to-use libraries involving arbitrary precision data types, math, video/image processing, linear algebra, DSP functions and more. Its capabilities are of course far more than those mentioned here, but covering every aspect of them would be completely out of the thesis' scope.

A high-level illustration of both utilized design flows, namely the C-based HLS design flow, realized through Vivado HLS and the HDL-based RTL-to-Bitstream design flow realized through the Vivado IDE, is shown in the opposite page.

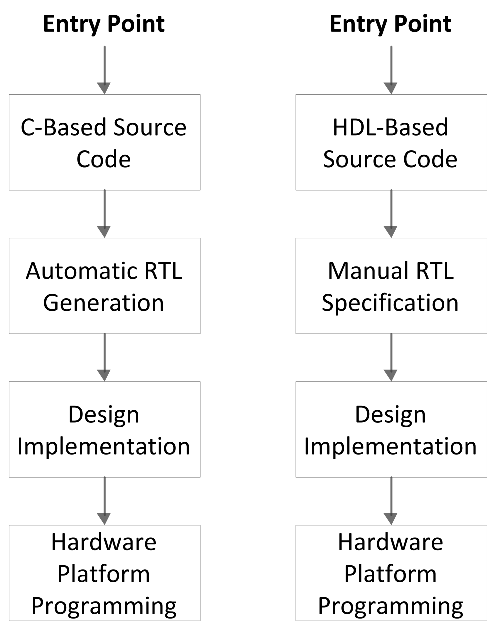


Figure 18: Abstract illustration of the two design flows under consideration. Left: HLS flow. Right: RTL-to-Bitstream/HDL-based flow.

5.4 ANALYSIS OF THE S2OPT ALGORITHM FOR FPGA IMPLEMENTATION

In the previous chapter, the reader was presented with-among others-a brief and mostly abstract description of S2OPT's functionality, mainly focusing on the notion of the symmetrical moves and the unique properties they possess. Before proceeding to the actual hardware architectures, a more elaborate study is due, so as to provide a smooth transition to the RTL representation and facilitate the correspondence between the algorithm's functionality and the hardware.

In order to begin, an illustration of the aforementioned functionality through the exploitation of the algorithm's flow chart is required. Each process displayed within the chart, along with its available parallelism, will then be discussed. The flow chart is based on the work done on the MATLAB-based emulator, and is intentionally shown in an analytical level, so as to partition the functionality in small, easily elaborated stages.

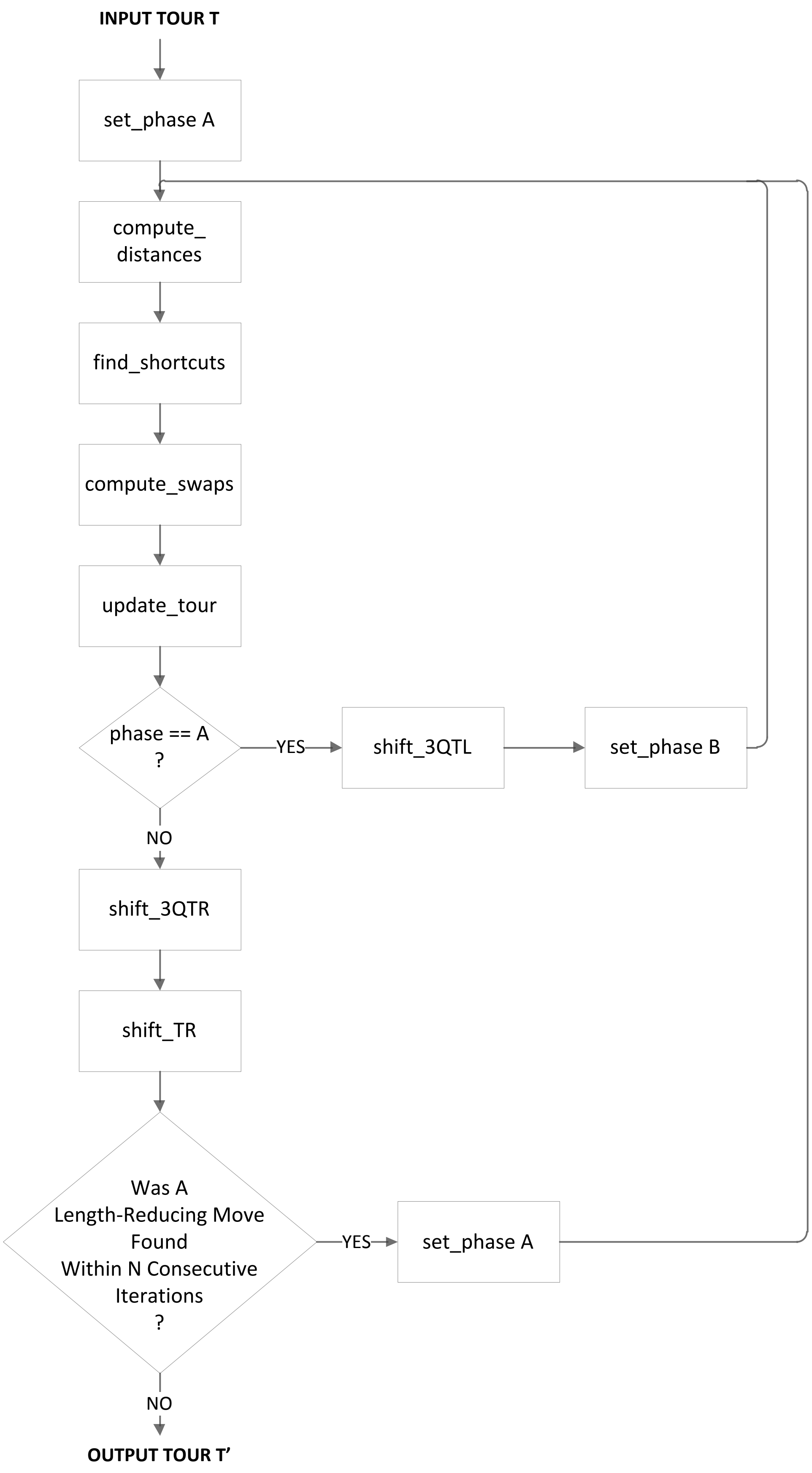


Figure 19: Flowchart of the S2OPT algorithm.

At this point, it should be noted that throughout this section, the given TSP tour is considered as an abstract, linear 1D array containing, in an equally abstract representation, the tour's nodes'/cities' Euclidean 2D coordinates in the same ordering as defined by the tour. It is also considered that the two edges of this array are actually connected, in the sense that left or right circular shifts of the cities' coordinates are possible. For the sake of vividness, an example tour of length N=16 is assumed.

The reader should keep in mind that, as stated in the previous chapter, a 2-OPT move can be equivalently viewed as a tour segment reversal, and is treated as such. In the Symmetrical 2-OPT case, in which the tour segments are symmetrical in the sense defined in that chapter, the application of a given symmetrical 2-OPT move means that the cities located at the two edges of the corresponding symmetrical segment will either keep their position or swap it with each other. The same applies to the rest of the cities within that segment. If all N/4 symmetrical segments contained within a given segment of maximum length of N/2 cities (half the total number of cities as explained in Chapter 4) are considered, then practically, all moves applied by S2OPT can be reduced to the swapping of appropriate symmetrical-city-pairs.

set_phase A/B: These do not constitute actual stages; they act as flow control flags and are utilized in order to provide a cleaner, more compact flow chart. In both cases, be it phase A or phase B, the same processes are executed. Yet there exists a distinction-from a theoretical point of view-on whether the algorithm is running phase A or B, explained in the *shift_x* sub-sections further down.

compute_distances: The compute_distances stage is responsible for the calculation of the length of the two pairs of edges (E1, E2 and E1', E2') involved in the evaluation of each symmetrical move (or equally each symmetrical segment reversal). This is performed by computing the squared Euclidean distances between the appropriate pairs of the associated quadruple of vertices/cities. The aforementioned process is repeated for every symmetrical-city-pair within the first half-the maximum considered tour segment length-(8 cities in total in the example) of the tour array (4 symmetrical-city-pairs, 4 repetitions). The move evaluations are completely independent and can be efficiently performed in parallel.

find_shortcuts: This process' functionality is fairly simple; it detects the actual length-reducing moves/tour-reversals among those evaluated in the previous stage (4 moves in the example), and marks them as such. The task is performed by computing the changes induced in the tour length from each move (the "Deltas" in the context of Chapter 4, where Delta= (length(E1') + length(E2')) - (length(E1) + length(E2))), and then checking whether the resulting quantities are negative or not. It is reminded that a negative Delta value implies a length-reducing move. As is the case with the compute_distances stage, the process discussed here can also be completely parallelized.

compute_swaps: The aim of the compute_swaps process is the "translation" of the length-reducing tour reversals, deemed as such in the previous stages, to the corresponding actually applied tour modifications, which are the symmetrical-city-pair swaps. This is made possible by the reasoning stated in section 4.9 of the previous chapter; once deemed length-reducing, a symmetrical-city-pair will swap positions if and only if the total number of tour reversals of all the symmetrical segments that the one under consideration is part of, including itself, is odd. Again, the computation of the final symmetrical-city-pair swaps can be performed in parallel, although, for reasons that will become evident in the following sections, not always efficiently.

update_tour: The functionality of the update_tour process is straightforward; it applies the previously assessed length-reducing symmetrical-city-pair swaps by performing the operations necessary to update the tour within the abstract 1D array so that it accurately reflects the modifications performed on it. The aforementioned procedure can too be efficiently parallelized.

Before proceeding to the rest of the processes/stages, a clarification on the schedule upon which S2OPT evaluates the available moves is due. Since S2OPT only considers segments of a maximum length equal to half the number of cities within the tour, the total number of available move evaluations within a single iteration is equal to (N*(N-1)/2)/2. For the given example, where N=16, the total number of move evaluations is 16*15/4= 60. The algorithm splits these evaluations into 15 groups of 4 symmetrical segments each.

At each iteration the S2OPT begins with the evaluation of symmetrical segments of even lengths.

shift_3QTL: In order to evaluate symmetrical segments of odd length, a left circular shift is performed to the tour array and more specifically to the segment consisting of the second, third, and final quarter of the array. After the completion of this process, all the steps described above are repeated. An example is shown below.

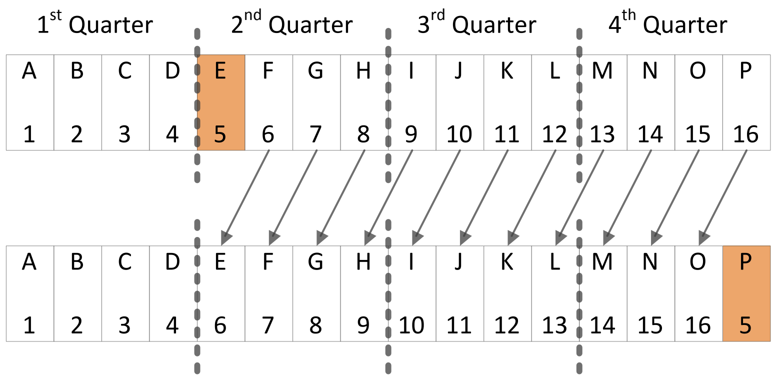


Figure 20: Example application of the shift_3QTL process on a tour of length N= 16.

shift_3QTR: After the evaluation of the symmetrical segments of odd lengths is complete, the inverse of the shift_3QTL operation, that is a right circular shift of the previously left-shifted segment, should be applied, in order to restore the cities in their original positions.

shift_TR: Finally, in order to enable the evaluation of the rest of the groups of the symmetrical moves, the tour array as a whole is circularly right shifted.

All three stages exhibit parallelism which can be exploited.

The process discussed in all of the above stages is repeated until no length-reducing move is found for N consecutive iterations, at which point the algorithm terminates.

5.5 S2OPT VHDL-BASED ARCHITECTURE

The hardware architecture discussed here is based on the corresponding description provided in [29]; the design of the former is intentionally similar to the latter due to the nature of the thesis, so as to enable a fair and an as accurate as possible comparison between the results obtained by both works.

This section considers S2OPT's VHDL-based architecture by presenting the block diagram of each hardware module implemented along with an appropriate description of its functionality, in a top-to-bottom approach. In the case of control modules, a flow chart is displayed instead.

It is noted that due to clarity considerations, the clock signal is not displayed in any of the aforementioned block diagrams; nevertheless, all of the modules discussed below are synchronous.

The hardware architectures presented here were designed, implemented and evaluated in the Xilinx Vivado IDE environment.

5.5.1 S2OPT TOP LEVEL (S2OPT MODULE)

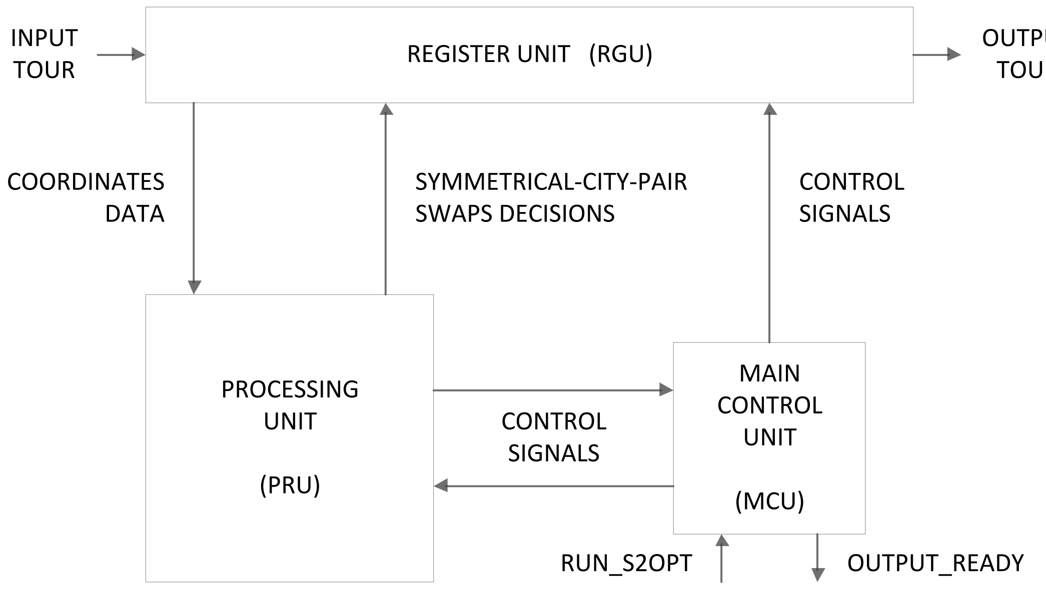
The algorithm's Top Level module is located in the highest level of the design hierarchy, connecting the basic architectural subsystems together and providing an interface between the design's I/O and external (sub)systems, such as auxiliary IP cores, larger designs, soft-core or embedded processors, a host system and more.

Design Inputs:

- clk: System clock.
- RUN_S2OPT, 1bit: Inbound control signal indicating the initiation of the hardware execution.
- DATA_IN, 30bit: The main input of the design, which are the coordinates of the tour's cities.

Design Outputs:

- DATA_OUT, 30bit: The main output of the design, which are the coordinates of the tour's cities.
- OUTPUT_READY, 1bit: Outbound control signal indicating the completion of the hardware execution.



Block Diagram 1: Illustration of the Top Level module of the hardware design.

The main components of the S2OPT module are three:

- The Register Unit (RGU)
- The Processing Unit (PRU)
- The Main Control Unit (MCU).

The RGU constitutes a set of registers whose main purpose is the storage of the coordinates of the tour's cities during the execution of the hardware. The registers are inter-connected in such a way that allows the efficient mapping of the functionality defined by the four last stages of the S2OPT's flowchart, namely the update_tour, shift_3QTL, shift_3QTR and shift_TR stages.

The PRU contains a set of Processing Element modules (PEs) which implement the functionality defined by the first two stages displayed in S2OPT's flowchart, namely the compute_distances and find_shortcuts stages. The third, the compute_swaps stage, is implemented by PRU's Swap Generation Unit (SGU).

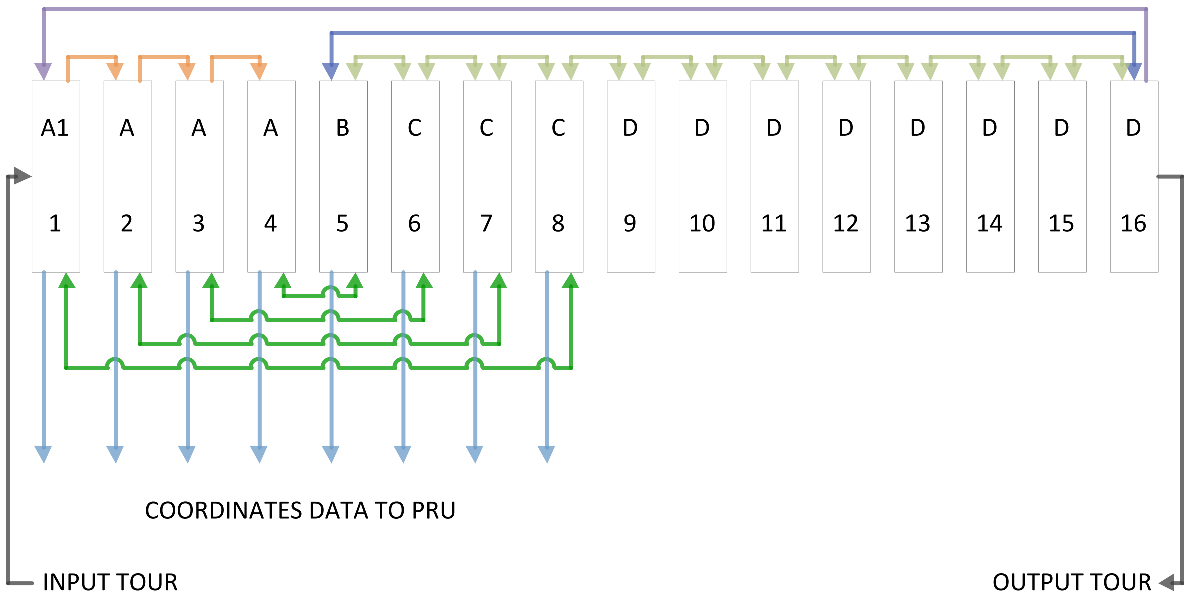
The MCU is tasked with the supervision and synchronization of the two basic components presented above. It directly manages the RGU's functionality, the communication between RGU and PRU as well as some basic parameters of the latter. Most of the operations conducted within the PRU are controlled by the PRU's dedicated Local Control Unit (LCU). MCU also handles the loading and unloading of TSP tours to/from the RGU and provides some basic control-related interfacing to/from the design.

When the MCU's *RUN_S2OPT* input is asserted, it signals the RGU to start loading the TSP tour from its DATA_IN port. As soon as the loading completes, the former notifies the PRU to start the evaluation of the first group of symmetrical moves by accessing the appropriate city coordinates stored in the RGU. The aforementioned move evaluations are performed in parallel thanks to the multitude of PEs contained within the PRU. When this process finishes, the latter signals the RGU to perform the necessary length-reducing swaps and informs the MCU that the first phase of the iteration is complete. At this point, the MCU commands the RGU to execute the shift_3QTL process and after that it notifies the PRU to begin with the second phase of symmetrical move evaluations. As soon as it finishes the MCU issues the shift_3QTR and shift_TR commands to the RGU, whose completion marks the end of an S2OPT iteration and the beginning of another. Upon successful termination of the algorithm's execution, the MCU signals the RGU to unload the final tour to its DATA_OUT port and asserts the *OUTPUT_READY* output signal.

5.5.2 REGISTER UNIT (RGU) MODULE

As mentioned in the sub-section above, the RGU's purpose is the storage of the coordinates of the tour's cities. Since 2D Euclidean space TSP instances are considered, each city's coordinates constitute of two components, namely x and y. For the datasets considered in the thesis, 15bits per component, i.e. 30bits in total are just enough for the representation of each coordinate pair; the two 15bit quantities are concatenated in a single, easily separated, 30bit entity. It does not matter which component (x or y) is mapped towards the LSB or the MSB end, as long as the decision is kept consistent throughout the design. The thesis considers a Little-endian approach and the mapping of the x component towards the MSB end.

The RGU can be visualized as a horizontal 1D array of register modules termed CITYREGs, where each such module consists of a 30bit register along with auxiliary logic, namely multiplexers. The number of CITYREGs contained within the RGU is equal to the number of cities within the given TSP tour; furthermore there is a one-to-one mapping between the order in which the city coordinates are stored within the RGU's CITYREGs and the actual order of the cities within the considered tour. The high-level block diagram shown below illustrates the aforementioned notion, where a tour of length N=16 is considered. More elaborate diagrams displaying the logic within each CITYREG along with its I/O signals and functionality are provided further down in the text.



Block Diagram 2: Abstract illustration of the internal structure of the RGU; the rectangles numbered from 1 to 16 are CITYREGs, while the (bi)directional arrows constitute their dedicated 30bit interconnections. The assignment of the color-coding to the terminology used later in the subsection is provided below:
Black Arrows: T/T', Purple Arrow: V, Orange Arrows: U
Dark Blue Arrow: X, Light Green Arrow: W, Dark Green Arrows: Y
Light Blue Arrows: Z.

CITYREG #1 holds the coordinates of the city currently first in the actual TSP tour under examination, CITYREG #2 the second, CITYREG #3 the third and so on.

The lettering on the CITYREGs represents their type; there are five types in total, although their main structure is fundamentally the same in the sense that all CITYREGs contain a 30bit register. Their differences lie in the minor variations of the number or size of the contained multiplexers, their wiring, and the required control signals, dictated by their relative, fixed location within the RGU.

The arrows represented are actual 30bit datapaths between pairs of CITYREGs; they are utilized for the efficient manipulation of the positions of the cities' coordinates within the RGU, as required by S2OPT's functionality. The direction of the arrows indicates the source and destination CITYREGs.

Both the wiring corresponding to the aforementioned datapaths as well as the characterization of a CITYREG's type are predetermined and fixed/specific for a given tour length N. The diagram above shows the exact wiring required for a tour of length 16, as well as the location of each type of CITYREGs within the RGU, relative to the tour length (displayed above the CITYREGs).

Seven sets of datapaths/wires can be discerned, referred to as T, U, V, W (and W'), X, Y and Z.

- **T/T':** These datapaths are used as the main input and output of TSP tours to/from the RGU respectively (and the hardware design as a whole), and are only utilized during the initialization and completion stages of the hardware execution of S2OPT.
- **U/V:** Datapath U is used during the shift_TR process; V is also used for the same reason, and it actually enables the RGU to act as a circular right-shift register.
- **W/X:** Datapaths W and X, which are both bi-directional, implement the shift_3QTR and shift_3QTL processes. The former is also utilized during the shift_TR stage [W: Light Green].
- **Y:** The-also bi-directional-datapath set Y is utilized during the update_tour process, enabling the swapping of the appropriate symmetrical-city-pairs.
- **Z:** Datapath set Z exposes the contents of certain CITYREGs to the PRU.

At this point the reader should be reminded that, as mentioned a few times already, the maximum symmetrical segment length that S2OPT evaluates is equal to half the length of the TSP tour under consideration (N/2= 8 in the example). The size of the rest of the symmetrical segments contained within the largest one, which are also evaluated, gets progressively smaller, until it reaches the smallest possible length of 2. There are N/4 (4 in the example) such symmetrical segments (or equivalently symmetrical-city-pair swaps, see previous section), each one involving two cities (N/2 cities in total).

The cities forming the aforementioned symmetrical-city-pairs are those found within the N/2 first/leftmost CITYREGs, hence the dense and complex wiring of that particular area of the RGU; N/4 bi-directional datapaths of type Y, and N/2 datapaths of type Z, connecting the CITYREGs that contain the symmetrical-city-pairs involved in the move evaluations/swaps with the PRU, which performs the actual computations required. A direct implication to this, is that in order to enable the PRUs to examine all the available symmetrical segments of length at most N/2, these segments need to be positioned-at some point of the hardware execution-within those special CITYREGs.

These are the CITYREGs of type A1, A, B and C. The D type CITYREGs, which are those located within the last/rightmost half of the RGU, are not accessed by the PRU; this implies that they do not perform symmetrical-city-pair swaps either. Thus their structure is simpler than the rest of the types. An in-depth presentation of the CITYREGs' structure is considered next.

As far as clock cycles are concerned, each one of the processes implemented by the RGU (shift_3QTL, shift_3QTR, shift_TR, and update_tour) execute in a single clock cycle thanks to the multitude of special purpose, dedicated inter-CITYREG datapaths.

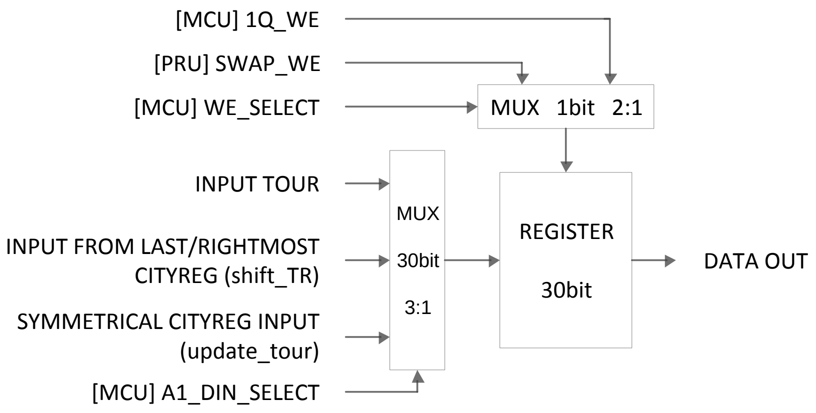
Before ending the current sub-section, a reference to the loading/unloading of TSP tours to/from the RGU is due. Both functions exploit the interconnections between the CITYREGs; the city coordinates are inserted one-by-one from T and are shifted inside the RGU using datapaths U and V and progressing one CITYREG at a time (one clock cycle). After N steps (N clock cycles), the TSP tour is completely loaded in the RGU. For a successful loading of the TSP tour, its insertion within the RGU must be performed in reverse order i.e. starting from the end of the initial tour. This certifies that when the loading process completes, the first city of the initial TSP tour under consideration will be positioned in the first/leftmost CITYREG, the second in the second and so forth. The unloading process utilizes the same method/wiring, outputting the final TSP tour one city at a time, through the output datapath T'.

5.5.2.1 CITYREG MODULES

This sub-section considers a more elaborate presentation of the CITYREG modules, focusing mostly on their internal structure and wiring. Since there is a great resemblance between the internals of all five types of CITYREGs, their common structural elements and wiring patterns will be discussed first. A type-specific discussion along with the corresponding block diagram illustrating the structures, wiring, and I/O ports follows.

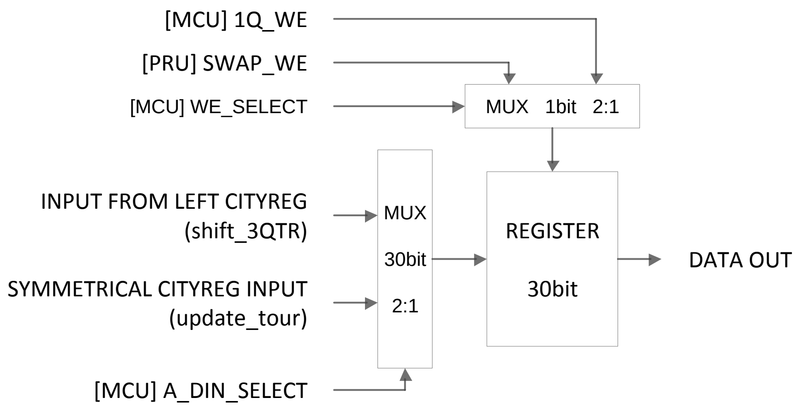
All CITYREGs contain a 30bit register which stores the city coordinates, and a 30bit 2:1 or 3:1 multiplexer which enables the register's input to interface with multiple datapaths; the latter is termed INPUT_MUX. Furthermore, all CITYREGs, with the notable exception of the D-type ones, employ a 1bit 2:1 multiplexer which renders the Write Enable (WE) port of the aforementioned register controllable by two separate sources, namely the MCU and the PRU; the former is in command during the shift_TR, shift_3QTR and shift_3QTL stages, while the latter during the update_tour stage. The 1bit multiplexer will be referred to as WE_MUX.

CITYREG_A1 module: This is the first/leftmost CITYREG within the RGU; it is a special case of the A-type CITYREGs and a special case of CITYREGs in general, in the sense that, among others, it acts as the point for the insertion of TSP tours within the RGU (and the hardware design in general). Besides the loading of tours, CITYREG_A1 is involved in the update_tour and shift_TR stages.



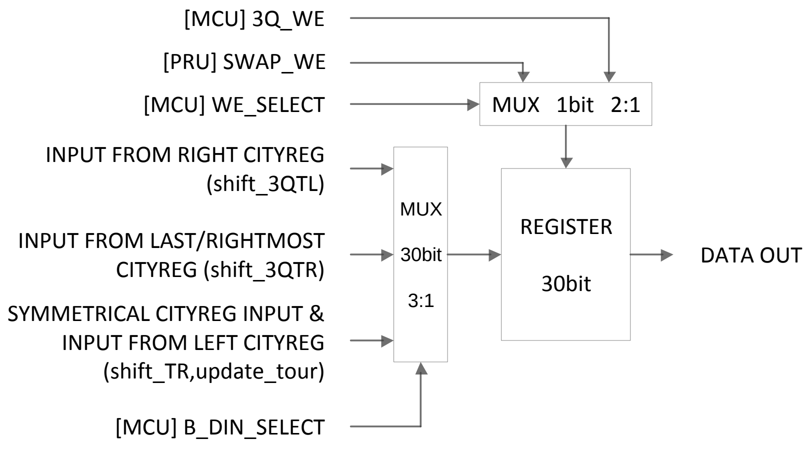
Block Diagram 3: CITYREG A1.

CITYREG_A module: This type of CITYREGs are (N/4)-1 in number, are located within the first/leftmost quarter of the RGU; they are involved in the update_tour and shift_TR stages.



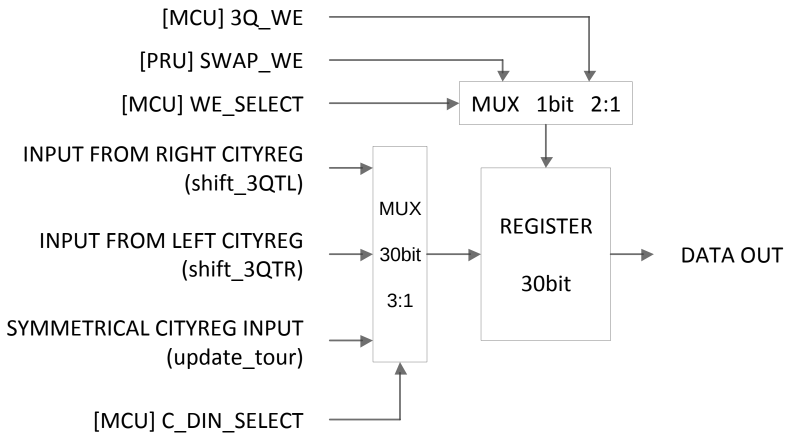
Block Diagram 4: CITYREG A.

CITYREG_B module: There is a single CITYREG of type B, located in the (N/4)+1 position of the RGU; it is involved in the update_tour, shift_TR, shift_3QTL and shift_3QTR processes.


[MCU] 3Q_WE
[PRU] SWAP_WE
[MCU] WE_SELECT
MUX 1bit 2:1
INPUT FROM RIGHT CITYREG (shift_3QTL)
INPUT FROM LAST/RIGHTMOST CITYREG (shift_3QTR)
SYMMETRICAL CITYREG INPUT & INPUT FROM LEFT CITYREG (shift_TR,update_tour)
[MCU] B_DIN_SELECT
MUX 30bit 3:1
REGISTER 30bit
DATA OUT

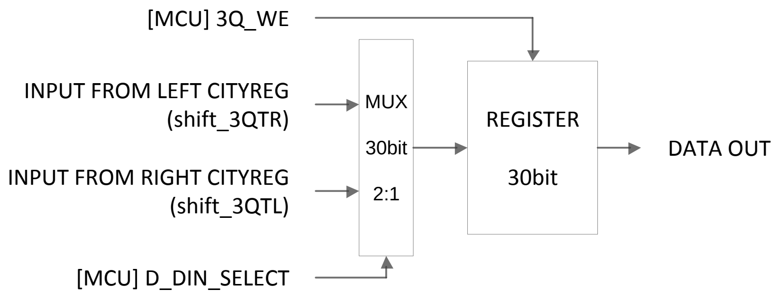
Block Diagram 5: CITYREG B.

CITYREG_C module: These CITYREGs, as is the case with A-type CITYREGs too, are (N/4)-1 in number and are located in the second-from-left quarter of the RGU; they are involved in the update_tour, shift_TR, shift_3QTL and shift_3QTR stages, similarly to the B-type ones.


[MCU] 3Q_WE
[PRU] SWAP_WE
[MCU] WE_SELECT
MUX 1bit 2:1
INPUT FROM RIGHT CITYREG (shift_3QTL)
INPUT FROM LEFT CITYREG (shift_3QTR)
SYMMETRICAL CITYREG INPUT (update_tour)
[MCU] C_DIN_SELECT
MUX 30bit 3:1
REGISTER 30bit
DATA OUT

Block Diagram 6: CITYREG C.

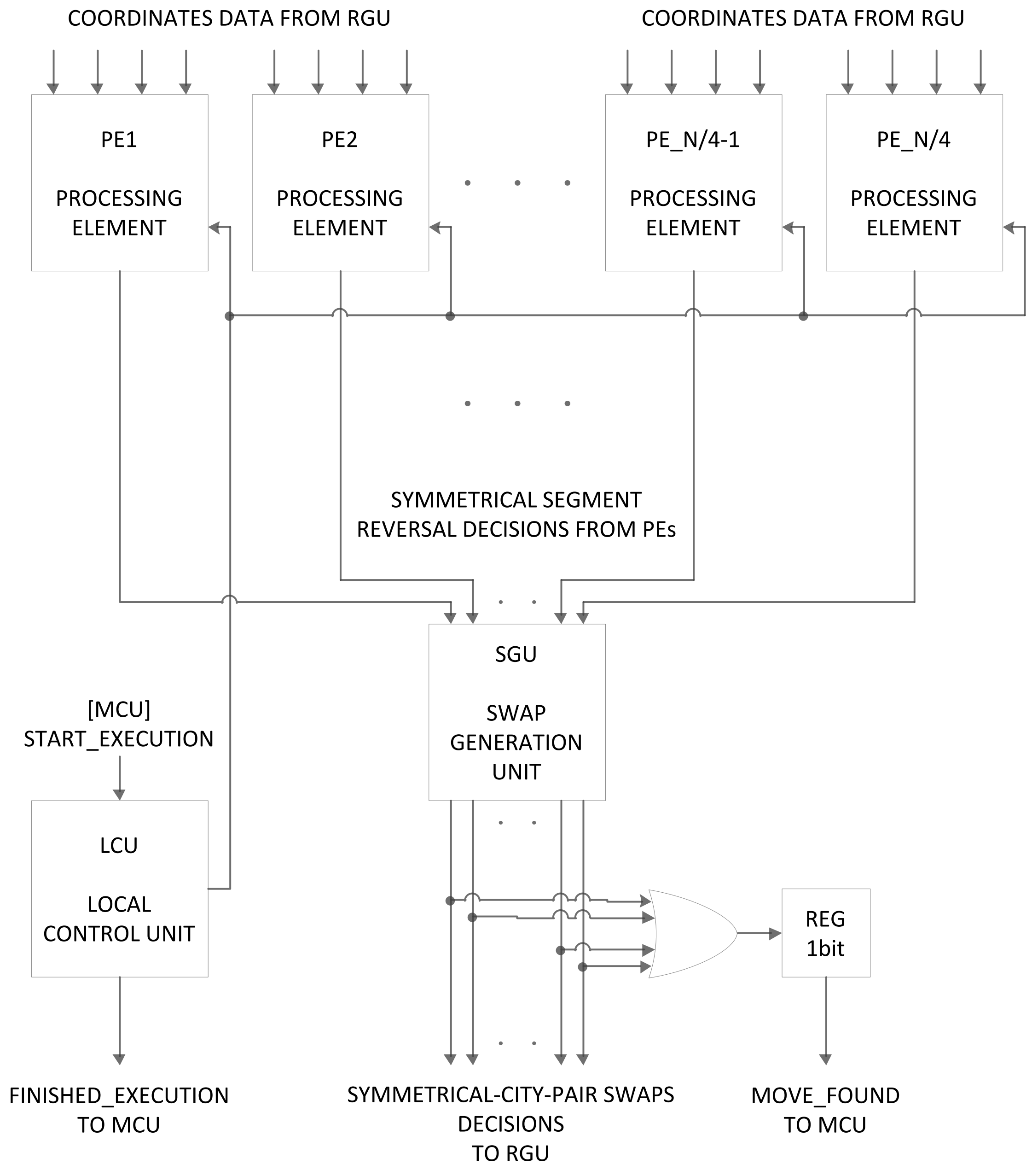
CITYREG_D module: There are N/2 D-type CITYREGs, located in the second/rightmost half of the RGU; they are structurally the simplest type of CITYREGs, involved in the shift_TR, shift_3QTR and shift_3QTL processes.


[MCU] 3Q_WE
INPUT FROM LEFT CITYREG (shift_3QTR)
INPUT FROM RIGHT CITYREG (shift_3QTL)
[MCU] D_DIN_SELECT
MUX 30bit 2:1
REGISTER 30bit
DATA OUT

Block Diagram 7: CITYREG D.

5.5.3 PROCESSING UNIT (PRU) MODULE

The Processing Unit (PRU) module is the computational module of the hardware architecture, responsible for performing the move evaluations. It reads the coordinates of the cities located in the borders of the symmetrical segments under consideration from the RGU, calculates the length-reducing symmetrical swaps and commands the former to apply them. The main structural components of the PRU are three, namely the Processing Elements (PEs), the Swap Generation Unit (SGU) and the Local Control Unit (LCU); an introduction to each one's functionality follows, while a more elaborate discussion on their internal design is also due.



Block Diagram 8: PRU.

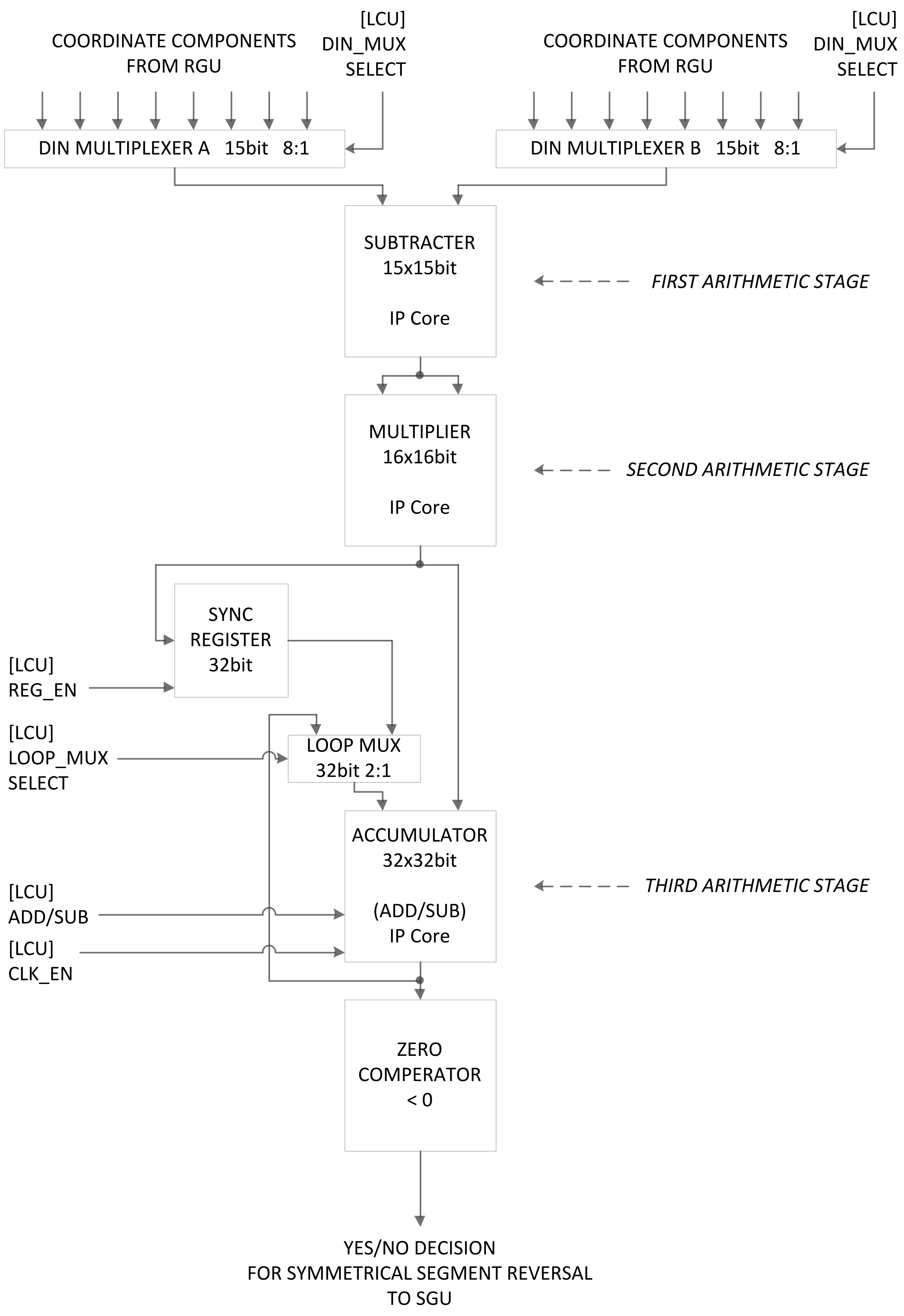
- **Processing Elements (PEs):** The PEs are the actual arithmetic units of the PRU; they execute all the numeric operations necessary for the computation of the two pairs of distances required for assessing whether a symmetrical segment reversal is length reducing or not. There are N/4 PEs within the PRU, which is the same as the number of the symmetrical segments under evaluation at every iteration of the algorithm's hardware execution, thus implying that each PE is assigned with the evaluation of a single/specific reversal. A given PE reads the coordinates of the four cities located in the borders of its allocated symmetrical segment and calculates the associated Delta() function discussed in section 4.3 of Chapter 4. The output is a Yes/No answer on whether the reversal should be applied or not. The aforementioned N/4 evaluations are performed fully in parallel.
- **Swap Generation Unit (SGU):** The SGU is responsible for converting the symmetrical tour reversal decisions made by the PEs into corresponding symmetrical-city-pair swap decisions (the two cities located at the ends of each such segment), which,-if deemed applicable-will constitute the actual length-reducing moves performed on the RGU. The SGU's functionality is of great importance to the overall design since it enables the aforementioned moves to be applied in parallel; the RGU's dedicated datapaths also contribute to this ability. It should be noted that the conversions themselves can too be performed fully in parallel, although, as will be discussed later, not always very efficiently.
- **Local Control Unit (LCU):** The LCU implements the hardware design's secondary control logic, dedicated specifically to the management of the operating parameters of the PEs' internal components. It also communicates directly with the Main Control Unit (MCU)-the primary/global control logic-for synchronization purposes.

The OR gate visible in the block diagram is utilized by the Main Control Unit in order to assess whether a length-reducing move is found after the completion of a given move evaluation run; the result is of great importance since the algorithm's termination condition depends on it.

5.5.3.1 PROCESSING ELEMENT (PE) MODULE

From this point on, the distance() function (section 2-OPT MOVE EVALUATION, Chapter 4), will refer to the squared Euclidean function; the square root is not taken into consideration.

The block diagram of the PEs is illustrated in the next page, followed by a description of each displayed sub-component.


[LCU] DIN_MUX SELECT
COORDINATE COMPONENTS FROM RGU
[LCU] DIN_MUX SELECT
COORDINATE COMPONENTS FROM RGU
DIN MULTIPLEXER A 15bit 8:1
DIN MULTIPLEXER B 15bit 8:1
SUBTRACTER 15x15bit IP Core
FIRST ARITHMETIC STAGE
MULTIPLIER 16x16bit IP Core
SECOND ARITHMETIC STAGE
SYNC REGISTER 32bit
[LCU] REG_EN
[LCU] LOOP_MUX SELECT
LOOP MUX 32bit 2:1
ACCUMULATOR 32x32bit (ADD/SUB) IP Core
THIRD ARITHMETIC STAGE
[LCU] ADD/SUB
[LCU] CLK_EN
ZERO COMPERATOR < 0
YES/NO DECISION FOR SYMMETRICAL SEGMENT REVERSAL TO SGU

Block Diagram 9: PE.

- **Din_Mux_A/B:** The two 15bit DIN_MUXes located at the module's input provide the first arithmetic stage of the PE with the appropriate coordinate-component-pairs; these are split between the aforementioned multiplexers in such a way that each input pair corresponds to an equivalent coordinate-component-pair. There are 8 such pairs involved in the computation of the Delta() function, as the latter involves the subsequent computation of the distance() function 4 times in total-each based on 2 pairs-hence the number of the multiplexers' inputs.
- **Subtracter:** The 15x15bit subtracter is the first arithmetic stage of the PE, used to calculate the differences of the city-pairs' coordinate components, as per the requirements of the distance() function. It is implemented as a highly customizable Xilinx Adder/Subtracter IP core.
- **Multiplier:** The 16x16bit multiplier is the second arithmetic stage of the PE, responsible for the computation of the squares of the aforementioned differences generated by the previous stage, hence its identical inputs. It is implemented as a highly customizable Xilinx Multiplier IP core.
- **Accumulator:** The 32x32bit accumulator is the third and final arithmetic stage; it is actually an adder/subtracter component, which performs the addition defined in the distance() function as well as all the additions and subtractions of the Delta() function. The accumulator is instantiated as a highly customizable Xilinx Adder/Subtracter IP core.
- **Sync_Register & Loop_Mux:** These two components play a key factor in the accurate initialization of the accumulator; during its initialization phase, the former delays the multiplier's (temporally) first output by one clock cycle so as to synchronize it with the second one, in order to provide the accumulator with the two inputs necessary for computing the seed value. Upon completion, the Loop_Mux multiplexer is used to loop the accumulator's output back to one of its inputs, thus setting it to normal operation.
- **Zero_Comperator** This component's functionality is rather self-explanatory; it examines the sign of the result yielded by the computation of the Delta() function and outputs an appropriate binary output.

At this point a few design- and timing-related clarifications are definitely due.

- As far as the arithmetic resources are concerned, the absolutely minimum required number is utilized by the PEs. The addition of more instances of the aforementioned resources would enable the partial parallelization of the necessary arithmetic operations, instead of serially executing them as implied by the block diagram. Such an approach would in turn decrease the overall latency, but due to the reductions needed in order to obtain the final Delta() result, the temporal gains would be largely outweighed by the subsequent increase in resource utilization. This notion was verified by practical assessments.
- The arithmetic units can be pipelined internally, with the number of pipeline stages easily controlled through the appropriate parameter of the corresponding IP core generator. An increased number of such stages usually implies a higher clock rate (if the critical path traverses the core under consideration) at the expense of higher latency, and vice versa. Extensive experimentation has been conducted in order to assess the optimal value for each arithmetic unit; the results are the following:
    - **Subtracter:** 1 pipeline stage (registered output). The addition of more pipeline stages actually yielded worse clock performance.

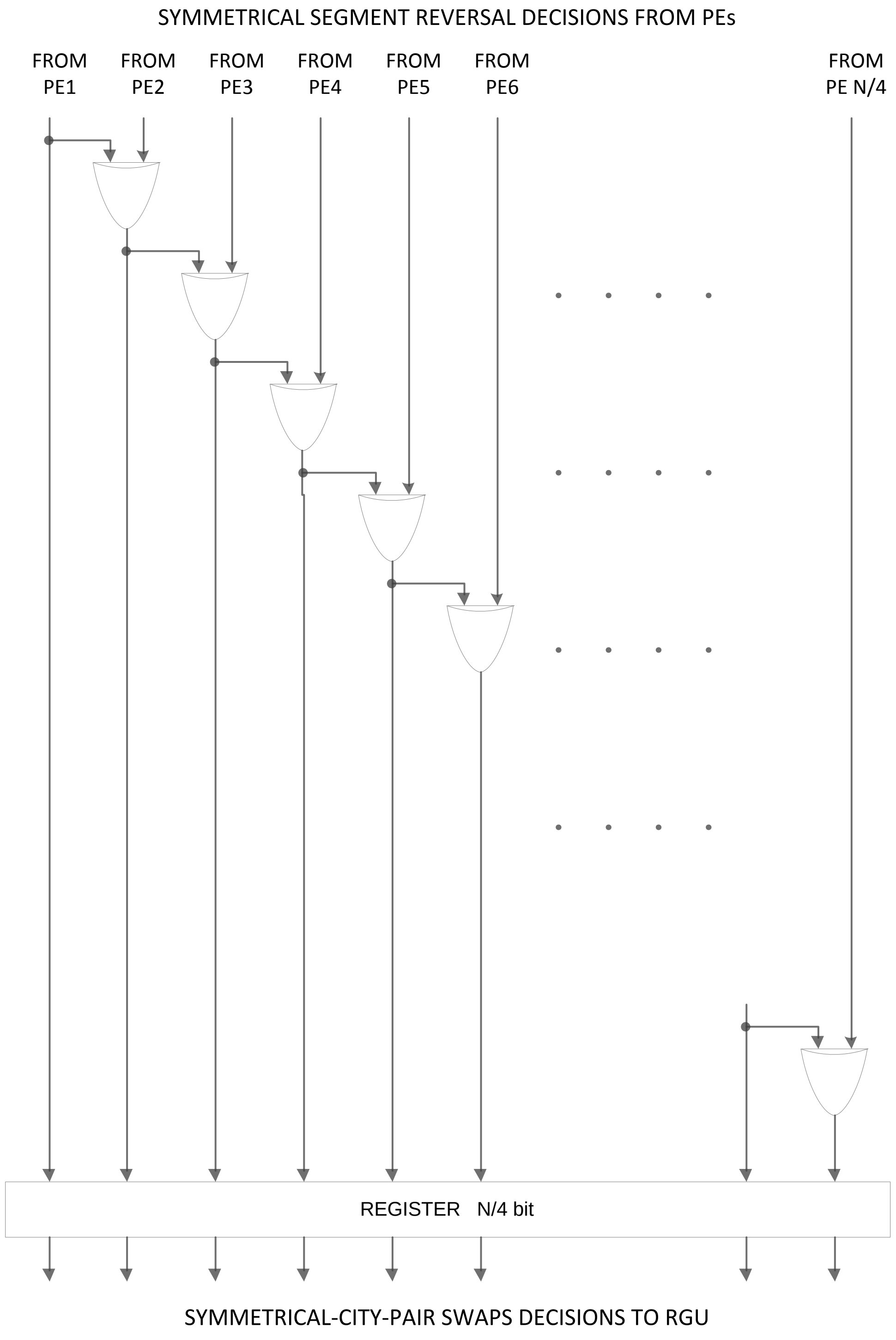
    - **Multiplier:** 1 pipeline stage (registered output) /3 pipeline stages in the event that the critical path is formed within the core under consideration. A choice of 2 pipeline stages yielded worse performance than the single stage.
    - **Accumulator:** 1 pipeline stage by default for performance reasons, due to the nature of the accumulator' functionality.
- The operation of the PEs is fully pipelined, hiding the latency induced by each arithmetic unit and processing one coordinate-component-pair (out of the 8 in total) at each clock cycle, as soon as the pipeline fills.
- The overall PE latency amounts to 12 or 14 clock cycles, depending on the multiplier's pipeline stages.

5.5.3.2 SWAP GENERATION UNIT (SGU) MODULE

The functionality of the SGU is based on the notion of the two key points mentioned in section 4.9 of Chapter 4; a symmetrical city-pair-swap is applied if the number of the length-reducing symmetrical segment reversals of the symmetrical segments that contain the one under consideration, including the reversal of itself, is odd.

The swap of the symmetrical pair of cities located on the edges of the outermost/largest symmetrical segment depends on its own reversal and only. The swap of the second pair depends on its own reversal and the previous one, the third pair depends on its own reversal and the two previous ones and so on, up until the innermost/smallest pair which depends on its own reversal as well as the (N/4)-1 previous ones.

This computational paradigm can be implemented with the utilization of appropriately interconnected (N/4)-1 XOR gates, arranged in the pattern visible in the SGU's block diagram, shown next.


SYMMETRICAL SEGMENT REVERSAL DECISIONS FROM PEs
FROM PE1
FROM PE2
FROM PE3
FROM PE4
FROM PE5
FROM PE6
FROM PE N/4
REGISTER N/4 bit
SYMMETRICAL-CITY-PAIR SWAPS DECISIONS TO RGU

Block Diagram 10: SGU.

As the block diagram makes evident, there is a diagonal path starting from the top left of the figure and ending up in the bottom right, progressively traversing every single one of the XOR gates. This is actually quite problematic, since the total delay induced by the propagation of the signal through the aforementioned gates adds up and grows linearly with the increase in problem size (city-wise), thus having a detrimental effect on clock performance.

An apparent solution to this issue is the addition of uniformly spaced pipeline registers throughout the length of the affected path, thus leading to a subsequent decrease in delay. This would be an efficient workaround if the hardware design was fully pipelined, which is not the case since the S2OPT algorithm does not follow the dataflow paradigm that suits FPGAs best; on the contrary, its nature is highly iterative.

The unavoidable consequence of this fact is that for every pipeline register added the SGU's latency increases by 1 clock cycle due to the penalty incurred by the repetitive filling of the pipeline; thus, a compromise between clock performance and latency has to be made. For the datasets evaluated in the thesis, a range of 1 to 6 pipeline stages was utilized.

In the end, the SGU proved to be the bottleneck of the hardware design, both in terms of performance as well as in its ability to efficiently support large problem sizes.

5.5.3.3 LOCAL CONTROL UNIT (LCU) MODULE

The LCU constitutes a simple Finite State Machine which manages the operational parameters of the PEs' structural components; there are five such control signals, visible in the PE block diagram and discernible from the rest due to the "FROM LCU" suffix. The LCU's functionality-abstractly defined for clarification and illustration purposes-is shown in the flow chart below.

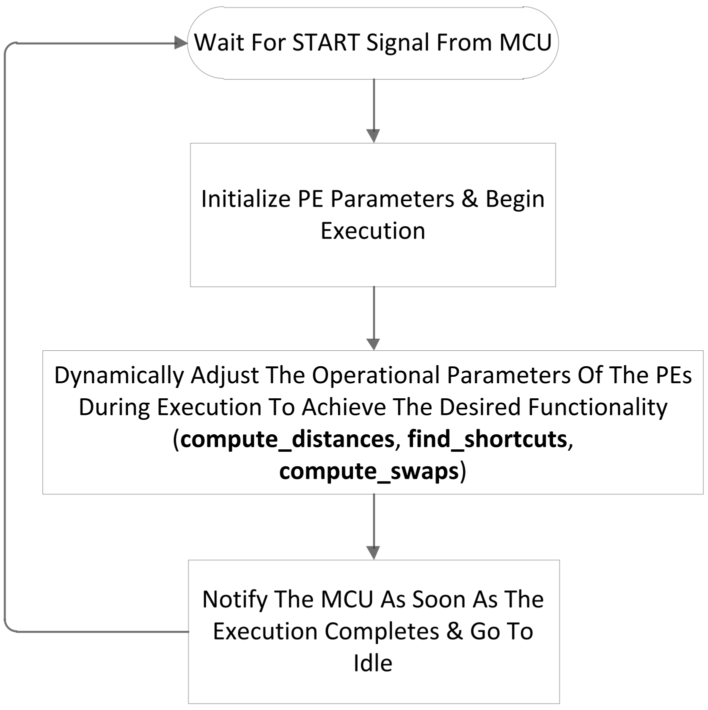

Wait For START Signal From MCU
Initialize PE Parameters & Begin Execution
Dynamically Adjust The Operational Parameters Of The PEs During Execution To Achieve The Desired Functionality (**compute_distances**, **find_shortcuts**, **compute_swaps**)
Notify The MCU As Soon As The Execution Completes & Go To Idle

Figure 21: LCU flowchart.

5.5.4 MAIN CONTROL UNIT (MCU) MODULE

The MCU constitutes the primary control unit of the hardware design, responsible for the management of the latter's I/O, the synchronization of its two main modules and the management of the RGU's operational parameters. It is implemented as a Finite State Machine, whose functionality-abstractly defined for clarification and illustration purposes-is shown in the following flow chart; most of the control signals directed or monitored by the module under consideration can be viewed in the PRU and CITYREG block diagrams, containing the MCU keyword.

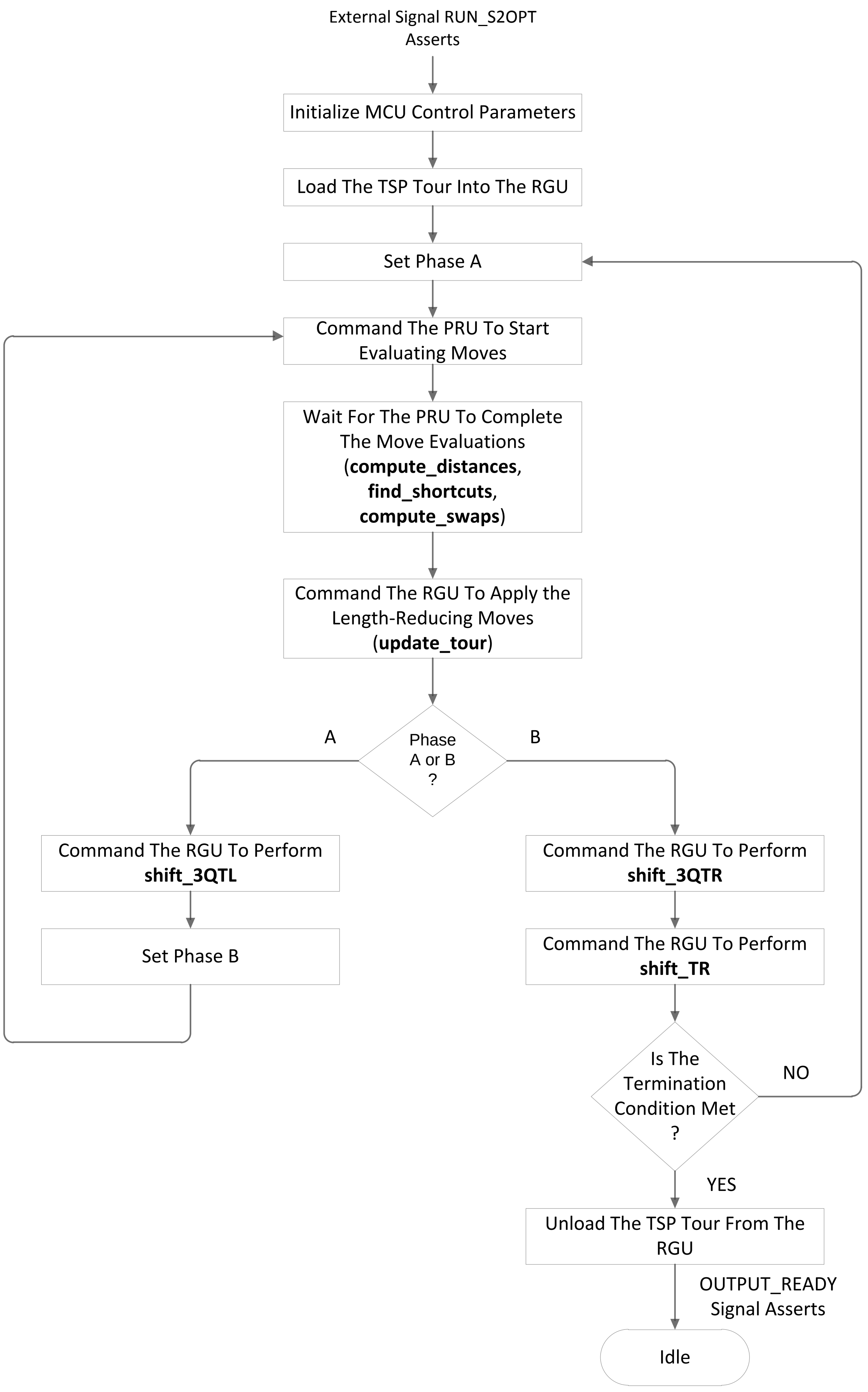

External Signal RUN_S2OPT Asserts
Initialize MCU Control Parameters
Load The TSP Tour Into The RGU
Set Phase A
Command The PRU To Start Evaluating Moves
Wait For The PRU To Complete The Move Evaluations (compute_distances, find_shortcuts, compute_swaps)
Command The RGU To Apply the Length-Reducing Moves (update_tour)
Phase A or B ?
A
B
Command The RGU To Perform shift_3QTL
Set Phase B
Command The RGU To Perform shift_3QTR
Command The RGU To Perform shift_TR
Is The Termination Condition Met ?
NO
YES
Unload The TSP Tour From The RGU
OUTPUT_READY Signal Asserts
Idle

Figure 22: MCU flowchart.

5.5.5 VHDL-BASED ARCHITECTURE LATENCY SUMMARY

This section provides a summary of the clock cycles required by the hardware architecture in order to complete a single execution iteration of the S2OPT algorithm, as illustrated in both the S2OPT FUNCTIONAL FLOWCHART as well as the MCU FLOWCHART.

- PE symmetrical segment reversal evaluations (compute_distances & find_shortcuts): 12/14 clock cycles (x2) depending on the number of pipeline stages of the PEs' multiplier core.
- SGU swap computations (compute_swaps): 1 to 6 clock cycles (x2) depending on the number of the SGU's pipeline stages.
- Registered output of the "MOVE_FOUND" OR gate within the PRU: 1 clock cycle (x2).
- RGU, apply symmetrical swaps (update_tour): 1 clock cycle (x2).
- RGU, shift_3QTL: 1 clock cycle (x1).
- RGU, shift_3QTR: 1 clock cycle (x1).
- RGU, shift_TR: 1 clock cycle (x1).

Total number of clock cycles per S2OPT execution iteration: 36 to 50.

The performance aspects of the VHDL-based architecture are covered in a more elaborate manner in chapter 6.

5.6 S2OPT HLS-BASED DESIGNS

This section is dedicated to the presentation of the workflow, the thorough experimentation and the decision-making process that ultimately led to the realization of the HLS-based S2OPT hardware designs. The main software tool employed throughout this development process was the Xilinx Vivado HLS; parts of the aforementioned work, involved in the final evaluation stages, have been conducted in the Xilinx Vivado IDE environment for the increased accuracy of the obtained performance and utilization results. The high-level language of choice for Vivado HLS's input was C.

5.6.1 FACTORS INFLUENCING THE GENERATED DESIGN CHARACTERISTICS

There are four major factors that largely affect the behavior, structure, and performance of the hardware designs (more accurately their RTL) generated by Vivado HLS:

- **Source code:** As will become evident later in the text, the structure, expressiveness, coding style and workload allocation among functions of the source code itself play a crucial role in the capabilities and behavioral characteristics of the resulting design. The importance of a "good" source code is highlighted by the fact that it determines both the applicability as well as the effectiveness of the available directives; it also facilitates the avoidance or overcoming of data dependencies.
- **Directives:** The wide range of available directives constitutes the most potent and direct method for the manipulation of the design's key parameters, in terms of performance (both latency- as well as throughput-wise), resource utilization, internal storage implementation, I/O interfacing and protocols, and execution/operation paradigms.
- **Clock constraint:** The clock constraint functions in an equally important-yet often overlooked-manner as the directives themselves, since it directly influences the performance aspects of the generated hardware design.

All three factors stated above, with the possible and partial exception of the first one (the source code), can be easily manipulated by the designer. Yet there is another one where little or no room for adjustments is available:

- **Algorithm nature:** The nature of the algorithm constitutes the sole factor that determines the success-in terms of efficiency and parallelization-of its mapping in an FPGA, and frequently poses an unavoidable bottleneck.

Finally, from the author’s practical experience, it is essential to note that in order to reap the effects of the directives to the maximum extent possible, no matter their type, a good base, i.e. a high quality source code is a definite prerequisite.

For “good” practices when performing hardware design through high-level languages, the reader is referred to the Xilinx Vivado HLS User Guide (UG902) [69].

5.6.2 THE EMPLOYED WORKFLOW PATTERN

The approach adhered to by the author in order to obtain the final versions of the HLS-based hardware design is the following:

1. First of all, the reading of the tool’s manual was mandatory for the apprehension of both the desired practices when performing hardware design through high-level languages, as well as the available directives along with their corresponding effect, parameters, applications and limitations.
2. A subset of the available directives, deemed suitable for the objectives of the work under consideration, was chosen for experimental evaluation.
3. As soon as the first version of the C source code was fully functional and ready, an iterative process of directive evaluations, interleaved with the development of new and improved source code versions began. When a performance-wise notable replacement of the source code under assessment was discovered, the subsequent directive evaluations from that point on were based on it.
4. When the point where no more coding-related C source optimizations could be applied was reached, the evaluation of the effect of multiple, simultaneously applied directives on the final source code version began. It should be noted that not all possible combinations of the candidate directives were evaluated; search-space pruning based on preliminary results was applied, in order to reduce the associated, admittedly excessive temporal requirements.
5. Three final hardware designs were chosen, each one aiming towards the fulfillment of different needs, such as low resource utilization, highest performance attainable, or a combination thereof.
6. The effect of varying clock constraint targets on the performance of the final designs was assessed afterwards; each design was implemented with a target clock constraint ranging from 100MHz to 500MHz in 50MHz increments. The clock frequency yielding the best combination of clock period and overall latency was subsequently chosen. This final part of the workflow process was conducted in the Vivado IDE environment for reasons previously discussed, and constitutes a topic that will be covered more elaborately in the next chapter.

Again, it is noted that the work described in the aforementioned steps, with the notable exception of step #6, was realized using the Vivado HLS; the dataset utilized was berlin52 from TSPLIB while the target hardware platform was a Xilinx Virtex-7 330T.

5.6.3 SOURCE CODE STRUCTURES COMMON IN THE EVALUATED REVISIONS

As mentioned in the previous subsection, a host of improvements and optimizations was applied to the initial source version, leading to the subsequent development and assessment of multiple such code revisions. Despite the modifications and/or additions from one revision to the next, most of them share a common, fundamental structure based on the results of preliminary work performed in the early phases of the HLS design, right after the porting of the MATLAB emulator code to C. These common structural elements are concisely presented below.

- ***void* S2OPT(*arguments*) function:** The S2OPT function constitutes the Top Level function of the design, implementing most or in some cases all of the algorithm's functionality (depending on the source code version) as described in section 5.4 of the current chapter. In other words, the majority of the functions discussed in the aforementioned section are embedded within the S2OPT function; yet they remain clearly discernible as they are implemented as separate code sections (see below). The function's input arguments implement the I/O of the design, namely the initial and final TSP tours; again, the exact type of the former depends on the source code revision under consideration.
    - In most source code versions, the S2OPT function contains-apart from various auxiliary variables-two core arrays, namely the CT and "swaps" arrays; the former stores the TSP tour under evaluation in the form of the cities' coordinates, while the latter stores the YES/NO decisions of the symmetrical-city-pair swaps. CT is either a 1D or 2D array, while "swaps" is always implemented as a 1D array. More arrays are found within the initial versions, such as the "distances" or "shortcuts" arrays, which were subsequently optimized away.
    - All versions include a single WHILE(1) loop; it is the function's main iteration structure which contains all of its functionality. An appropriate "break" condition is located at the end of the loop's body.
        - The WHILE(1) loop embeds 4 to 11 for-loops with standard/non-variable bounds; they are all located within the same hierarchy level, i.e. there are no nested for-loops. Each such loop implements one or more of the functions described in section 5.4 of the current chapter, while their actual number depends yet again on the given source code revision. It should be noted that there is hardly any code outside of the aforementioned loops' bounds; if any, it is necessary for initialization purposes.
- ***retval_type* cfunc(*arguments*) function:** cfunc is the sole function external to the design's top level; it is called by two of the latter's for-loops, one or more times per major (WHILE(1)) iteration. In most cases it implements auxiliary arithmetic functionality, while in others the aforementioned functionality is extended to include more complex operations. Its input arguments are cities' coordinates while the return value is either a distance in the notion defined in section 4.3 of Chapter 4, or a Boolean value signifying whether a length-reducing symmetrical segment reversal is found.

More concrete information on the various source code revisions can be found in the following section.

5.6.4 SOURCE CODE REVISIONS

This subsection considers the brief presentation of the seven plus one source code revisions implemented and evaluated during the development process of the HLS-based designs, along with the modifications, changes and improvements each such version yielded over its predecessor.

REVISION 0

This is the first major/stable revision of the C code. The functionality of cfunc() is basic and limited to the computation of the squared Euclidean distance of its input symmetrical-city-pairs. These are made available to cfunc() from S2OPT(), by passing as input arguments a pointer to the entire CT array together with two integers denoting the symmetrical-city-pairs that should be accessed (3 input arguments in total).

The TSP tour is stored within the 2D (#cities x 2) CT array as pairs of the x and y components of the cities' coordinates. The array itself is considered external to the design, in the sense that it is passed to the top level function S2OPT() (and subsequently to cfunc()) as a pointer originating from an outside source; there is no CT declaration within the design's scope, and it actually constitutes the sole input argument of S2OPT().

As far as the S2OPT() top level function is concerned, it includes a host of variable and array declarations, most of which are admittedly redundant to the smooth execution of the function. Yet this was intentional and part of the planned evaluations in order to assess the effect of the aforementioned redundancy to the resulting design. The WHILE(1) loop, the main iteration structure of the function, embeds 11 for-loops, each one implementing a single functionality as defined in section 5.4 of the current chapter.

Finally, it should be noted that the data type of all the declared variables and arrays, including the external CT array, is 32bit int.

REVISION 1

The first optimization introduced was the complete removal of the redundant arrays and variables from within S2OPT(), as well as the partial merging of the for-loops; the latter's number decreased from 11 to 7, as two of the aforementioned loops were granted more extensive functionality. This has led to a further reduction in both resource utilization and overall latency as well, although not as striking as the previous one; the improvement of the latter stems from the fact that-considering that both the overall workload per WHILE(1) iteration and for-loop bounds have remained unchanged-fewer loops equals less latency.

REVISION 2

This revision concerned the redesign of the top level function's I/O; instead of CT being accessed by S2OPT as an external array, i.e. by-reference, it is declared within S2OPT as an array internal to the design's scope. The function's input arguments were subsequently modified in order to properly support the aforementioned changes; the first argument, *IT*, is a pointer to an external 2D input TSP tour, while the second one, *FT*, is a pointer to the design's final output tour. An addition of two extra for-loops outside of the main WHILE(1) loop was also proven necessary. The first one, located at the beginning of S2OPT's body, is responsible for copying the initial TSP tour from the input pointer argument to the CT array, while the second, located at the end of S2OPT's body, copies the resulting TSP tour from CT to the output pointer argument at the end of the algorithm's execution.

The optimizations mentioned above yielded a slight increase in resource utilization yet they decreased the overall latency by almost 13%; the most noteworthy such increase was marked in the BRAM utilization where it grew from zero to four blocks, which is expected since the TSP tour under assessment had become part of the design itself.

REVISION 3

The changes introduced with the 3rd revision were twofold: The usage of arbitrary precision data types and minor modifications in the body of the cfunc() function; the latter were rather superficial and aimed at an increase in both the number and precision of applicable directives, without altering its functionality.

The arbitrary precision data types enable the declaration of variables and arrays of non-standard size, i.e. not conforming to "classic" 8bit-multiple widths. For example, the declaration of an array of 19bit-integers or a 3bit unsigned integer variable is made possible. As one can infer, this is a very powerful feature since it mitigates the unnecessary wasting of bits occurred when mapping quantities-whose exact maximum size is known or decided a-priori-to wider, quantized data types.

The aforementioned optimization yielded a further reduction in both the resource utilization and overall latency as well, due to the induced decrease in datapath and storage width requirements.

REVISION 4

The 4th revision brought a modification to the way the TSP tour is stored within the design during the algorithm's execution.

The 2D CT array, used for storing the pair of the coordinate components (x & y) of each city as separate 15bit quantities, was replaced with a 1D version storing the coordinates as a single 30bit entity, similarly to the approach taken in the VHDL-based architecture. This change necessitated the addition of bit manipulation methods such as bit-wise shifting and masking within the cfunc() function.

Although the optimization discussed above did not yield any changes in the overall latency, it did reduce the LUT utilization by a small factor, and most importantly it largely improved the legibility and succinctness of the source code. The decrease in LUT utilization is attributed to the reduction in the number of the required multiplexers; the latter is induced by the fact that the access/update of a given city entails the processing of a single value instead of two.

REVISION 5

The 5th revision introduced major alterations to the cfunc() function, which was granted more extensive functionality that was previously part of the responsibilities of some of the for-loops within S2OPT's body, namely that of the abstractly-defined compute_distances() and find_shortcuts() functions.

The new capabilities of cfunc() meant that it should be able to access four city coordinates during its call by S2OPT in order to evaluate a given symmetrical segment reversal, i.e. compute whether Delta (see section 4.3, Chapter 4) is negative or not. The previous method of passing arguments to cfunc() (see REVISION 0) would clearly prove problematic since the accesses to the required coordinates would have to be heavily serialized.

This issue was solved by providing the necessary values in parallel through four separate input arguments, a fact that enabled the tool to schedule part of the required arithmetic operations to be performed in parallel as well, thus reducing the overall latency. The reduction in the number of calls made to cfunc() by each iteration of S2OPT() also attributed towards that direction. As far as resources are concerned, there was a slight increase in DSP utilization due to the equivalent increase in the number of arithmetic units required to schedule the aforementioned operations for parallel execution.

REVISION 5.X

This version was actually an experiment encouraged by the warning message generated by the tool when attempting to apply the #ARRAY_PARTITION TYPE COMPLETE directive to the CT array; it stated that "the resulting design would be largely suboptimal due to the utilization of large multiplexers imposed by the fact that the aforementioned array was accessed through non-constant indices".

The solution that the tool suggested for that issue was either the wrapping of the array access into a function or the usage of a register file core. Since there is no such core available in the tool's library, the choice was made clear. For this reason, two self-explanatory functions were added to the source code, namely the access() and update() functions, used solely on the CT array. Apart from that, the aforementioned modification required that the array under consideration was declared as *global static.*

However, despite the numerous attempts, the proposed scheme did not manage to resolve the issue. It also failed to yield any improvements to the resource utilization or the latency, while it severely limited both the effect and the success of the subsequently applied directives.

REVISION 6

The changes in the 6th and final revision are similar to those brought by the 1st in the sense that it induced further merging to S2OPT's for-loops, reducing their number from 7 to 4. The resource utilization remained almost unchanged, while the overall latency was reduced by 24%.

SUMMARY

The changes in both resource utilization and overall latency induced by each source code revision are illustrated in the following charts, in order to highlight the major role that the source code itself plays in the final design.

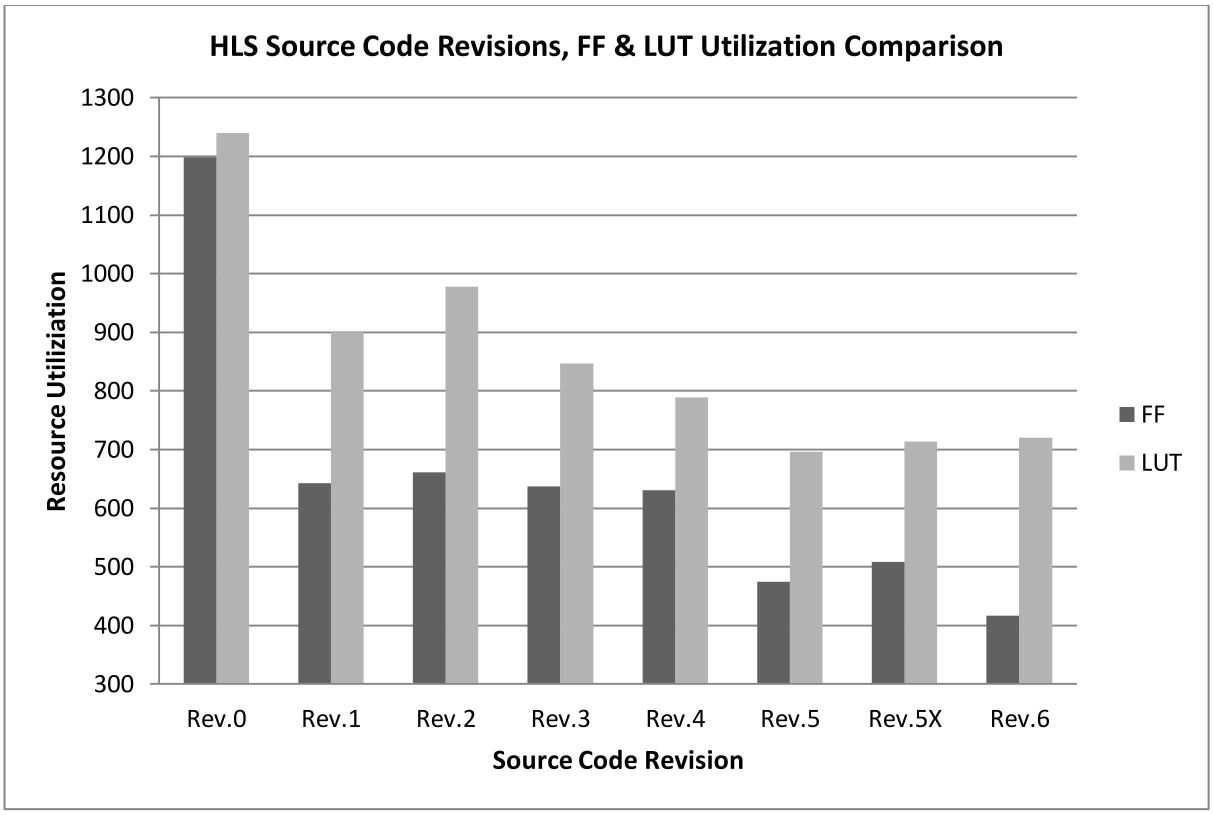


Figure 23: The changes marked in FF and LUT utilization between the evaluated source code revisions; the y-axis shows the number of actual logic blocks used. The figure does not consider the equivalent changes in BRAM and DSP utilization since they are relatively negligible. The TSPLIB dataset under consideration is berlin52.

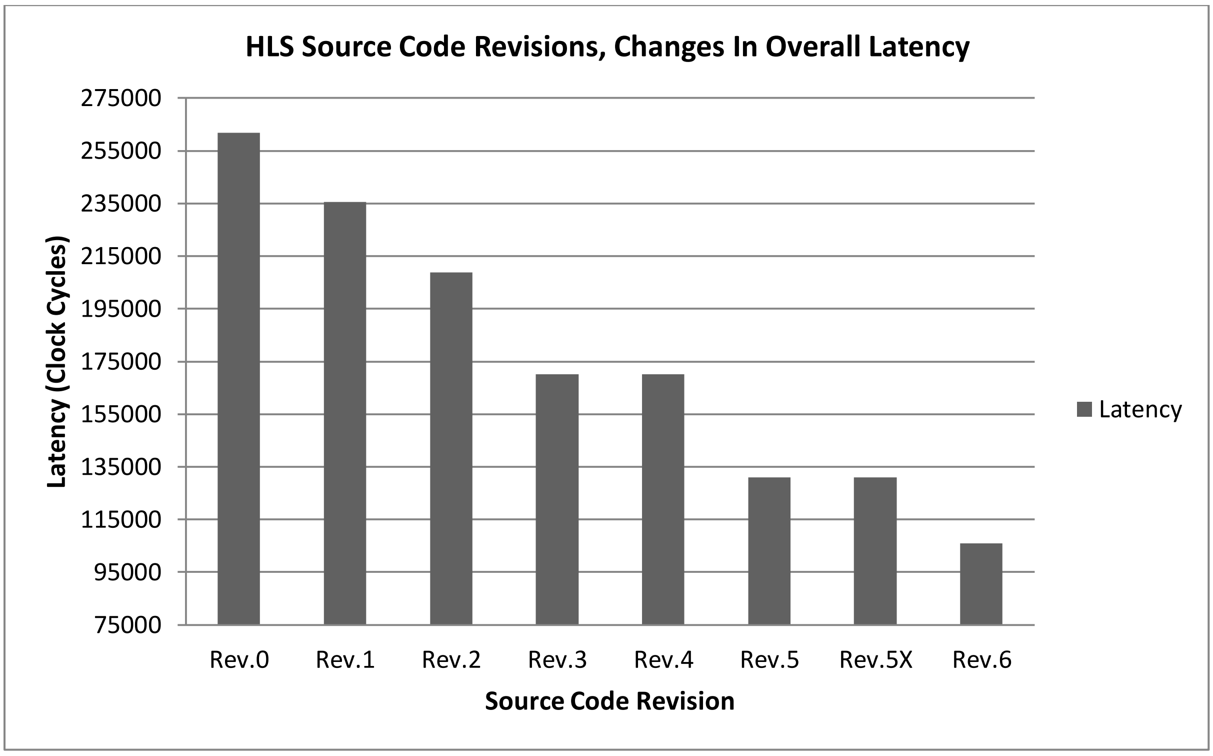


Figure 24: Changes in overall latency (clock cycles) induced by the examined source code revisions; the TSPLIB dataset under consideration is berlin52.

As far as the latency chart is concerned, it should be noted that throughout the revisions the yielded clock performance remained fixed at about 8.5ns; any changes in latency imply equivalent changes in the actual runtime.

5.6.5 DIRECTIVES EVALUATED & SUBSEQUENT DISCUSSION

This subsection presents the directives that were deemed suitable for the nature of the algorithm and the desired objectives and were subsequently evaluated so as to assess their actual effect on the design's key parameters, as a prerequisite to their final adoption/application. The presentation only covers the most notable remarks made by the author during the directive experimentation phase on their behavior and/or impact on the S2OPT algorithm; thus, the aforementioned remarks can be considered case-specific to a certain extent.

For a complete list of the available directives along with the elaborate description of their functionality and parameters, the reader is referred to [69].

#PIPELINE

When applied to S2OPT's for-loops, whose bounds are non-variable, the #PIPELINE directive behaved as intended; it drastically reduced the latency of the aforementioned loops and subsequently of the design as a whole, with minimal increases in resource utilization and without affecting the clock performance, not in a negative way at least. The results obtained by pipelining such loops vary and depend on the data dependencies between consecutive loop iterations, a fact that largely determines the rate at which the implemented pipeline outputs new data (termed Initiation Interval or II in the tool's manual); an optimal II of 1 might not be always achievable. A quick possible workaround is to enable more read/write ports on the structures accessed during the execution of the loop through the usage of appropriate directives, yet this rarely is the case.

When applied to functions, the #PIPELINE directive completely unrolls all the embedded loops instead of pipelining them, having the same effect on them as if the #LOOP_UNROLL directive was applied; the latter is discussed next.

It should be noted that the #PIPELINE directive cannot be applied to loops with variable bounds, i.e. loops for which the exact number of iterations is unknown or changes dynamically during runtime; such an example is S2OPT's WHILE(1) loop. In fact, this kind of loops poses many limitations and should be avoided if the nature of the algorithm under consideration allows it; frankly, this does not apply for S2OPT, whose total number of main iterations is not known a-priori.

All in all, pipelining the for-loops proved a resource-wise efficient way of moderately enhancing performance; a testimony to this is the case of the 5th source code revision, where after the application of the directive a 100% reduction on the overall latency was achieved, followed by a small 12% increase in resource utilization.

#LOOP_UNROLL

The application of the #LOOP_UNROLL directive to the for-loops of S2OPT's WHILE(1) main loop yielded the most striking improvements to the overall latency of the design, at the expense of slightly worsened clock performance and sharply increased resource utilization; these side-effects were expected due to the creation of multiple copies of the related logic instances, necessary to facilitate the concurrent execution of the operations defined in each iteration of the affected loops.

The most notable such improvement was marked when the directive under consideration was applied to the aforementioned loops of source code revision 6, where the latency reduced by 2117%; the subsequent decrease in clock frequency was a relatively minimal 10%, while resource utilization increased by 1144% on average.

Again, the non-deterministic nature of the WHILE(1) loop proved quite limiting, as it did not support the #LOOP_UNROLL directive either.

#INLINE

The #INLINE directive proved rather inefficient; on the one hand it did manage to yield improvements in the overall latency but on the other it did so by inducing disproportionate increases in resource utilization, especially DSPs. The most noteworthy such example is the case of Rev. 6, where the DSP utilization increased by 300% while the overall latency was reduced by only 17%; the same trend is evident in other experiments as well. The rise in DSP utilization is a logical consequence of the inlining of cfunc() within the body of S2OPT(), since the arithmetic units indirectly defined by the former were instantiated as many times as S2OPT() calls cfunc() within a single iteration of the WHILE(1) loop.

To sum up, as far as the design under assessment is concerned, #INLINE turned out to be an “expensive” directive that should be enabled when aiming for maximum performance at any cost.

#RESOURCE

During the late stages of the design process, when the final designs along with their corresponding source codes and directives had been determined, the cfunc()-related resource utilization indicators of the high-performance version made evident that there was quite an imbalance between the number of implemented multipliers (DSP utilization) and the implemented adders/subtracters (LUT utilization); it seemed as if more multipliers had been instantiated. This was odd, since there is a block of eight subtractions at the beginning of the aforementioned function and another block at its end consisting of eight additions; this is a total of sixteen addition/subtraction operations compared to only eight multiplications within each cfunc() call.

Based on these two facts it was assumed that the arithmetic operations within these blocks were not being efficiently parallelized by the compiler. The application of the “#RESOURCE AddSub” directive to the block of the addition operators mitigated this issue to a certain extent, leading to an average increase in LUT utilization of 40%, followed by a subsequent decrease in overall latency by 42% on average. Another positive side-effect was the marked increase in the attainable clock frequency by about 11%.

The directive under consideration was applied to the block of subtractions as well, yet it did not yield any changes to the design’s parameters at all.

#ARRAY_PARTITION

Within the context of the design under consideration, the #ARRAY_PARTITION directive proved problematic at best. Not only did it fail to yield any substantial reductions to the overall latency-at least not without causing severely disproportional increases to the resource utilization, but nearly half the times that it was evaluated it either caused program crashes or resulted in malfunctioning or underperforming designs, despite the numerous attempts made to resolve these issues; such an attempt is described in subsection 5.7.4.6, while others include the assiduous experimentation with various parameters of the aforementioned directive.

To the author's point of view, the aforementioned issues stemmed from the fact that the multitudes of read/write ports enabled by the application of the directive under assessment on the CT array caused the compiler to either infer false parallelization options that were not actually feasible due to inherent algorithmic data dependencies or render it unable to successfully schedule/manage the increased number of available operations. As one can infer, the #ARRAY_PARTITION directive was not incorporated in any of the final design versions.

A few more directives were evaluated as well; some of them could not ultimately be used due to the fact that their application entailed specific preconditions that were impossible to be met by the source code while others proved largely underperforming or malfunctioning. These were the #LOOP_MERGE, #LOOP_FLATTEN, #ALLOCATION and #DATAFLOW directives.

Before ending this subsection, it would be a serious omission not to mention that all the experimentation conducted during the development of the HLS-based designs and discussed thoroughly within the context of the current, HLS-related section of the thesis is available in the appendix; the related spreadsheet contains the utilization and performance results of more than 400 evaluations of source codes, single and multiple directives and combinations thereof.

5.6.6 FINAL HLS-BASED DESIGNS

Upon completion of the source code revision and directive evaluation phases, three different combinations thereof were carefully selected; this resulted in an equivalent number of designs, each one aiming at different usage scenarios i.e. catering to contradicting end user needs. These are presented below; their title is largely self-explanatory.

Low Performance/Low Resource Utilization design [LP]

- **Source Code Revision:** Rev. 5.
- **Directives**
  - **#PIPELINE**, applied to the for-loops of the S2OPT() function.

The LP design is based on the 5th source code revision, although Rev. 6 is faster in terms of overall latency; this was intentional due to the for-loops of Rev. 5 being much simpler and straightforward in terms of functionality and included operations than those of Rev. 6, a fact that led to the more efficient application of the #PIPELINE directive, thus resulting in higher-performing pipelines with fewer stages and smaller Initiation Intervals.

High Performance/High Resource Utilization design [HPA]

- **Source Code Revision:** Rev. 6.
- **Directives**
  - **#LOOP_UNROLL**, applied to the for-loops of the S2OPT() function.
  - **#RESOURCE AddSub**, applied to the additions block of cfunc() function.

Very High Performance/Very High Resource Utilization design [HPB]

- **Source Code Revision:** Rev. 6.
- **Directives**
  - **#LOOP_UNROLL**, applied to the for-loops of the S2OPT() function.
  - **#RESOURCE** AddSub, applied to the additions block of cfunc() function.
  - **#INLINE**, applied to the cfunc() function.

At this point it should be noted that several attempts were made in order to introduce a balanced design, i.e. a design offering "medium" performance and proportional resource utilization; this proved quite an elusive objective as the two main factors could not be efficiently matched in order to yield a design that would pose as a substantial alternative to those already mentioned.

The chart below shows the actual runtimes achieved by the HPB design in comparison to the initial source code Rev. 0 in order to clearly illustrate the performance enhancements yielded by the previously discussed workflow process; five TSPLIB instances are considered.

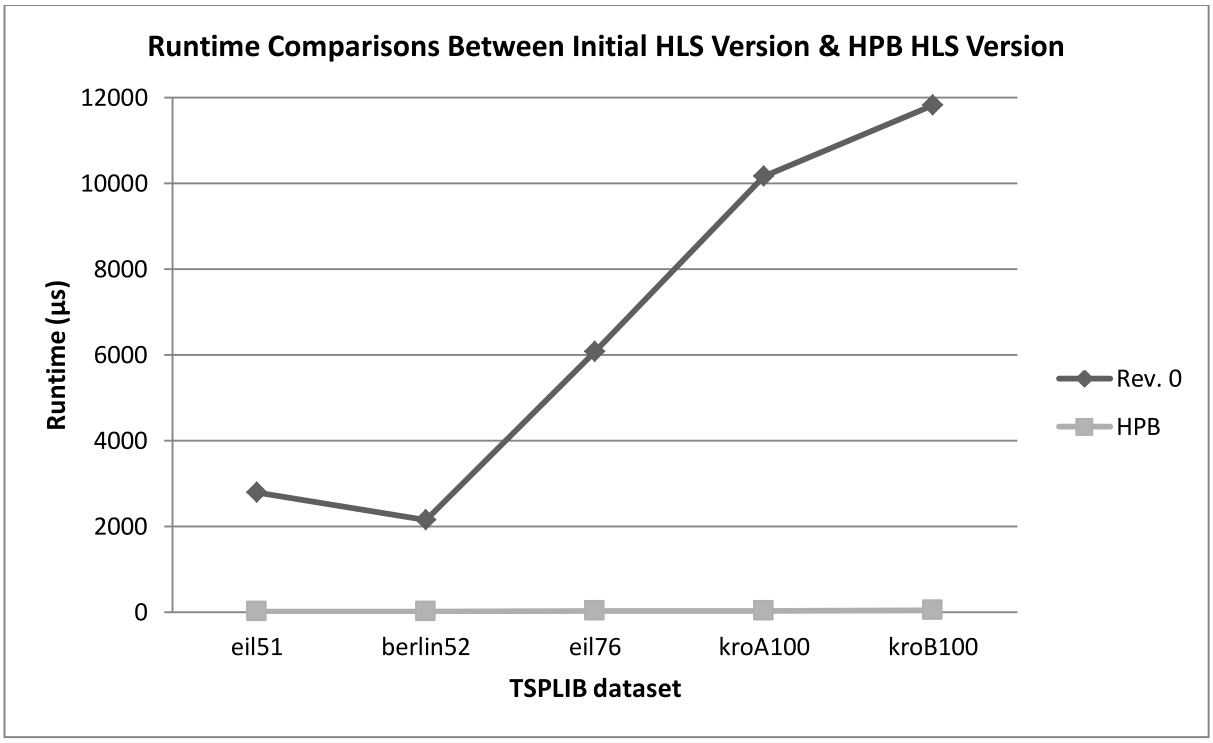


Figure 25: Comparison of the runtimes obtained by both the initial revision of the source code and the highly performance-wise optimized yet resource-intensive HPB version, in five TSPLIB datasets; the HPB runtimes are less than 50µs at most, thus appearing as nearly zero.

TSPLIB dataset	*eil51*	*berlin52*	*eil76*	*kroA100*	*kroB100*
Change %	17369	17867	21579	28971	29450

Table 1: Reductions in runtime yielded by the HPB version when compared against the initial revision of the source code, expressed in terms of Percent-Change.

The performance aspects of the HLS-based designs are covered in a more elaborate manner in the next chapter.

CHAPTER 6: EVALUTION OF THE DESIGN FLOWS, PERFORMANCE RESULTS, AND RELATED DISCUSSIONS

6.1 INTRODUCTION & CHAPTER STRUCTURE

The subject of this chapter is twofold: the discussion of the strengths and weaknesses of the utilized design flows as well as the elaborate presentation of the performance characteristics and results obtained through the implementation and the subsequent evaluation of the hardware architectures and designs examined within the context of the previous chapter. The former is based on the practical experience attained during the lengthy and extensive occupation with the corresponding design environments, while the latter derives from the thorough testing of the aforementioned designs against numerous TSPLIB datasets as well as their performance comparison with both the highly acclaimed Concorde TSP solver and a state-of-the-art GPU implementation.

6.2 DESIGN FLOWS EVALUATION

This section concerns the presentation and the subsequent evaluation of the utilized design flows.

6.2.1 PRESENTATION OF THE DESIGN FLOWS

The flow charts of each of the design flows are illustrated in the following page:

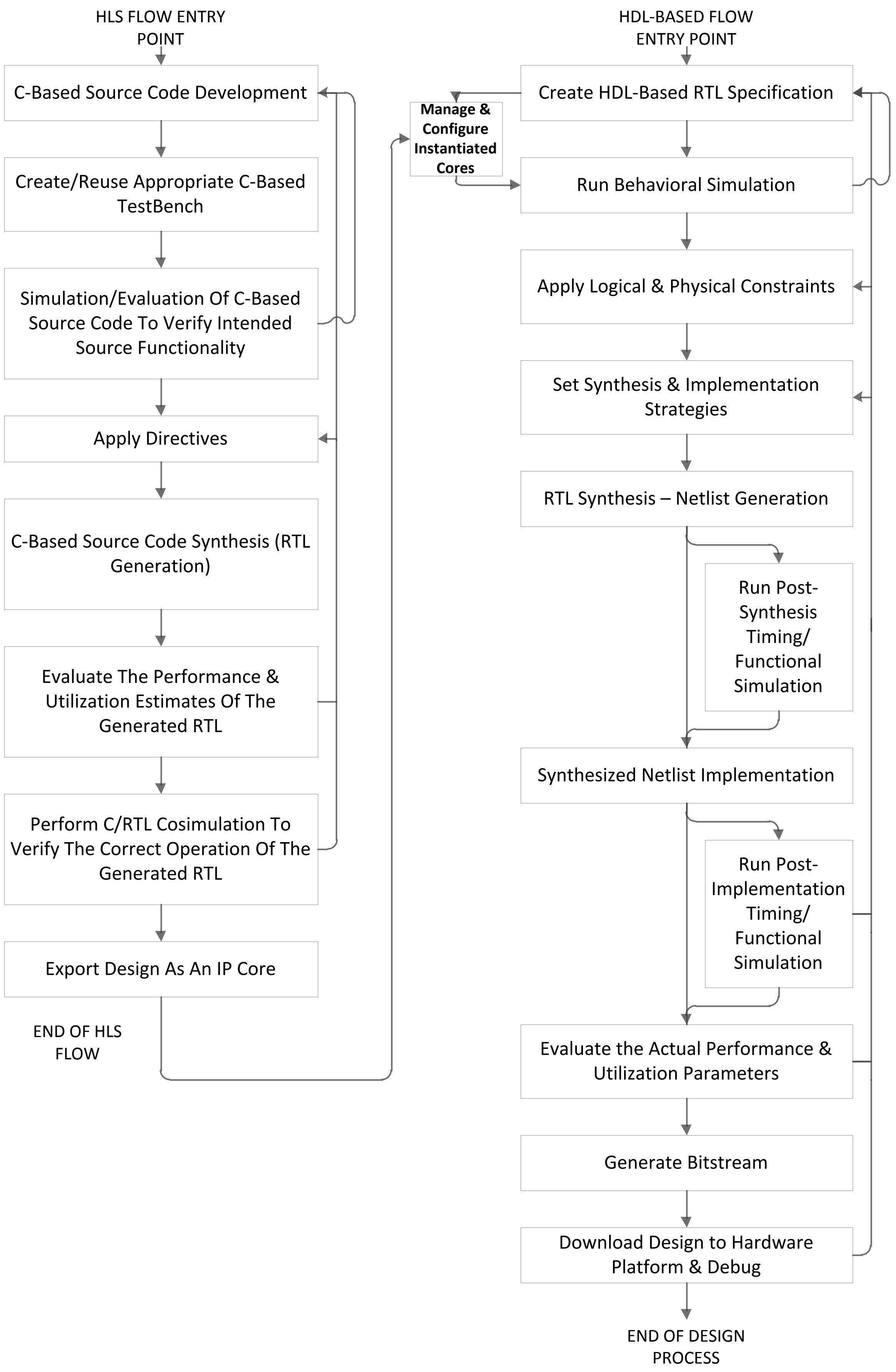

HLS FLOW ENTRY POINT
C-Based Source Code Development
Create/Reuse Appropriate C-Based TestBench
Simulation/Evaluation Of C-Based Source Code To Verify Intended Source Functionality
Apply Directives
C-Based Source Code Synthesis (RTL Generation)
Evaluate The Performance & Utilization Estimates Of The Generated RTL
Perform C/RTL Cosimulation To Verify The Correct Operation Of The Generated RTL
Export Design As An IP Core
END OF HLS FLOW
Manage & Configure Instantiated Cores
HDL-BASED FLOW ENTRY POINT
Create HDL-Based RTL Specification
Run Behavioral Simulation
Apply Logical & Physical Constraints
Set Synthesis & Implementation Strategies
RTL Synthesis – Netlist Generation
Run Post-Synthesis Timing/ Functional Simulation
Synthesized Netlist Implementation
Run Post-Implementation Timing/ Functional Simulation
Evaluate the Actual Performance & Utilization Parameters
Generate Bitstream
Download Design to Hardware Platform & Debug
END OF DESIGN PROCESS

Figure 26: HLS (left) and HDL-based/RTL-to-Bitstream (right) flowcharts.

Based on the illustrations of the flow charts, it can directly be inferred that the primary design flow that leads to the downloading and subsequent execution of the hardware design on an actual hardware platform is the RTL-to-Bitstream flow; in fact, the HLS design flow poses a preceding extension to the latter that bypasses its first step, namely the creation of the RTL specification. Hence, the differences between the two flows stem largely from the fact that the VHDL/Verilog-based design flow entails the manual specification of the design's RTL, while the HLS design flow enables the partially or fully automatic generation of the RTL as implied by the C-based input and captured by Vivado HLS' compiler. Naturally, since the two flows share a rather large number of design steps there inevitably exist many similarities among them as well, at least as far as the design implementation phase is concerned.

6.2.2 COMPARISON CRITERIA

The criteria upon which the subsequent comparisons are based are briefly mentioned below:

1. The primary target user groups.
2. Learning curve & required hardware design knowledge.
3. Time requirements.
4. Accuracy of control over the design process.
5. Quality of the obtained design.
6. Bottleneck-inducing stages.
7. Use case scenarios.

Before proceeding to the actual discussions, it is reminded that the comparisons due are approached through the author's empirical point of view, shaped during the preparation of the thesis; thus they are largely based on the experience gained during the mapping of S2OPT in hardware.

This premise holds for the majority of the aforementioned criteria, with the exception of the first two which are approached more holistically due to their theoretical-oriented nature.

For further clarification, it is stated that the HLS-part of the thesis constitutes the author's first contact with Vivado HLS and its related design flow; previous involvement with the VHDL-based design flow and the Xilinx ISE tool, based on smaller-scale works, exists.

6.2.3 COMPARING THE DESIGN FLOWS

THE PRIMARY TARGET GROUPS

The HLS design flow is relatively newly (re)introduced and caters to both software and hardware engineers alike; on the one hand software engineers often wish to map their computationally-intensive algorithmic designs to hardware platforms so as to reap the offered performance and power-consumption benefits, without the need to learn actual hardware design principles from the ground up or related HDL languages. On the other hand, hardware engineers have too many compelling reasons to get involved with the HLS flow, which will be discussed later in this section.

On the contrary, the pure RTL-to-Bitstream design flow addresses mainly to hardware engineers with decent knowledge on the fundamentals of hardware design and HDL languages, and constitutes the standard, classic approach to hardware design for many decades now.

LEARNING CURVE & REQUIRED HARDWARE DESIGN KNOWLEDGE

From the software engineer’s aspect, the HDL-based design flow has an obvious, particularly steep learning curve, thus rendering it a non-realistic path towards hardware design; the HLS flow is definitely much more suited to the programming skills that the user group under consideration usually possesses. That being said, the aforementioned flow might be relatively easy for someone with such a background to get accustomed to, yet in order to maximize the extent of the control over the desired qualities of the resulting design as well as make effective use of the available directives, a certain degree of hardware-related knowledge is a definite prerequisite.

As far as hardware engineers are concerned, the transition from the HDL-based flow to the HLS one is most probably a smooth process, albeit it involves the learning and subsequent adoption of the desired practices followed when designing hardware through high-level languages; the latter holds for the software engineers as well.

All in all, in order to make the most out of the HLS design flow, a certain amount of learning overhead is due regardless of the designer’s knowledge background, although the software engineer will probably need more preparation time.

TIME REQUIREMENTS

The time requirements of each design flow can be realized in a variety of ways, the most substantial being the following:

- **Time requirements for an initial design:** To the author’s experience, the time required in order to obtain an initial working version of a fully functional design is significantly shorter in the case of the HLS flow; one needs to simply write the C-based source code correctly and let the HLS tool handle the rest of the design process using the default settings, without applying any directives. On the contrary, when the HDL-based flow is considered, the time required to manually specify the RTL-even to a non-optimized initial state-is definitely much longer due to its inherent complexity and error susceptibility.
- **Time requirements for a highly optimized design:** In this case, the time requirements of the HLS flow rise remarkably, since both the quality of the source code as well as the careful selection of directives and their subsequent efficient/effective application must be taken into consideration, leading to what is quite a lengthy process. As previously discussed, the HDL-based flow requires copious amounts of time regardless of the intended optimization level; thus there is no marked change in the factor under consideration as the design objective alters.

Within the context of the thesis, both the C-based and the VHDL-based works required a roughly equivalent amount of time from design initiation to design closure.

ACCURACY OF CONTROL OVER THE DESIGN PROCESS

This is one aspect where the HDL-based flow markedly outshines its counterpart; the offered levels of control over various features of the design process, such as explicitly defining logic structures and module hierarchy or setting the exact type, number and parameters of a required resource or IP core, simply cannot be matched by the HLS-flow, despite its host of available directives. Not at this point at least.

QUALITY OF THE OBTAINED DESIGN

Both flows are capable of delivering equally high quality designs, provided that the time requirements dictated by each design process are adequately met. Yet there exists one factor where the HDL-based flow clearly excels: efficiency. The experiments conducted showed that the HDL-based designs tend to utilize far less resources for a given performance level than their HLS-based counterparts, especially when the two high-performing ones are concerned.

BOTTLENECK-INDUCING STAGES

Several bottleneck-inducing stages have been identified in both design flows:

- **HLS-flow:** Both the synthesis (C-to-RTL) as well as the C/RTL co-simulation phases tend to require unacceptably large amounts of time and/or system memory in certain cases; such are the cases where the design under consideration is rather complex or when resource-intensive directives are utilized. A consequence of the latter is that multitudes of logic or arithmetic elements are generated, leading to a subsequent sharp increase in the number of available operations that must be parallelized and scheduled, thus putting a heavy computational load on the compiler.
- **HDL-based flow:** As far as the HDL-based flow is concerned, the time required to perform changes in the RTL, which can vary from relatively short to significantly long depending on the severity and span of the intended modifications, poses a definite bottleneck.

USE CASE SCENARIOS

To the author's point of view, there exist numerous cases where the utilization of the HLS flow proves far more desirable than the HDL-based one; such cases include the following:

- An initial version of a working, fully functional design is needed as quickly as possible (proof of concept, rapid system prototyping).
- Quick and efficient design space exploration prior to investing significant amounts of time in the HDL-based flow.
- When limiting the time requirements is of utter importance, quite possibly at the expense of reduced resource efficiency.
- When high resource efficiency or fine-grained control over the design's parameters are considered dispensable.
- When a highly complex algorithm with limited parallelization options is under consideration, a fact that renders the HDL-based flow rather inefficient.
- When the algorithm under consideration exposes its available parallelism in an obvious manner, enabling the HLS compiler to effectively extract and exploit it so as to generate a both highly performing and resource efficient design.

For most other cases, the HDL-based flow is considered the design method of choice.

It should be noted that since this was the author's first involvement with the Vivado HLS and its corresponding design flow, he was rather pleased with its ever-increasing capabilities (worked with both the 2015.4 and 2016.2 versions), high quality of results, straightforwardness, quick-paced design iterations and clean, intuitive environment.

6.3 PERFORMANCE ASSESSMENT METHODS & PARAMETERS

Prior to proceeding to the presentation and subsequent discussion of the performance characteristics of the designs obtained by each flow, it is essential to introduce the procedure that was followed in order to extract the actual performance-defining parameters.

- **HDL-based flow:** The proper/intended functionality of the VHDL-based architectures was verified by both Behavioral and Post-Implementation Functional simulation; the Post-Implementation Timing simulation is unavailable for VHDL-based designs due to the lack of appropriate libraries supporting this feature (Verilog support only). Each one of the architectures under consideration was subsequently implemented in order to acquire the corresponding Post-Place & Route utilization and clock parameters. All the aforementioned steps were performed in the Vivado IDE.
- **HLS-flow:** As far as the C-based designs are concerned, the proper/intended functionality of the source codes was verified by "C Simulation", while the correct operation of the generated RTLs was confirmed by "C/RTL co-simulation"; both these processes constitute features of the Vivado HLS tool. As is the case with the HDL-based flow, the obtained RTLs were subsequently implemented in the Vivado IDE.

Furthermore, the utilized version of the Xilinx Vivado Design Suite was the 2016.2, while the target hardware platform of choice for both design flows was the Xilinx Virtex-7 330T, one of the smallest-logic-capacity-wise-FPGAs of Xilinx's current high-end product line. A high-end device was necessary due to the fact that the hardware designs, especially the high-performing HLS-based ones, have increased DSP requirements that only the Virtex-7 series was capable of adequately meeting.

It should also be noted that none of the designs were actually downloaded to an FPGA, due to the lengthy process involved combined with the emergence of severe, unsurpassable time limitations during the late completion stages of the thesis; yet thorough quality and correctness screening has been conducted through the simulation features of the tools, as previously mentioned.

At this point, a few words on the software, datasets as well as the related published works that were used to perform an "external" assessment of the results are due:

TSPLIB: All the datasets utilized during the experimental evaluation of the performance of both the final hardware designs and the benchmark software as well, are part of the TSPLIB library of TSP instances, presented and discussed in section 4.10 of chapter 4.

Original S2OPT: Part of the comparisons is made against the original 2007 implementation of S2OPT presented in [29]; this plays a key role to the estimation of the performance enhancements yielded by the latest generation of FPGAs, design tools and corresponding flows.

Concorde: The primary benchmark is the 2-OPT implementation included in the Concorde TSP solver software package, presented and discussed in section 3.5 of chapter 3. Concorde employs many of the optimizations discussed in the section 4.4 of chapter 4 so as to reduce the required runtimes, thus giving up the guarantee of true 2-optimality. The experiments showed that

Concorde's implementation of 2-OPT is highly non-deterministic, in the sense that a given input TSP tour does not yield a standard output TSP tour; on the contrary, the output varies largely from run-to-run, hinting that a randomized approach is at play. In order to facilitate the fairest comparison conditions attainable, the following actions have been taken:

- The source code has been tweaked in the following ways:
    - The software applies the nearest neighbor heuristic on the initial TSP tour prior to the starting of the actual 2-OPT heuristic by default; this feature has been disabled.
    - The overall running time reported by the software's built-in timer includes both the time needed to execute the heuristic and the time required to compute the length of the output tour; furthermore the utilized timing function is of very crude resolution. This feature has been modified so as to measure only the actual CPU-time of the heuristic itself while the timing function has been replaced with the clock_gettime() function which offers nanosecond-level accuracy.
- The performance results obtained by each dataset have been averaged over 100K independent runs.
- The benchmark software under consideration was executed on a latest-generation machine based on a quad-core 3.4GHz Intel Core-i7 6700 CPU with 16GB of 2.4GHz DDR4 SDRAM, running freshly-installed Ubuntu 16.04.
- The software was compiled with the latest version of gcc using the "-O3" and "-march=native" optimization flags.

GPU-based RRHC with 2-OPT: The final part of the performance comparisons revolves around the GPU-based implementation of the Random Restart Hill Climbing (RRHC) metaheuristic with embedded 2-OPT heuristic, presented in [52]; the RRHC metaheuristic is briefly discussed in section 3.3 of chapter 3. To enable such a comparison, the VHDL-based design had to undergo a minor modification that led to subsequent increases in resource utilization, thus necessitating the utilization of the high-capacity Xilinx Virtex-7 980T. This topic is further elaborated in the following section.

6.4 PERFORMANCE RESULTS & SUBSEQUENT DISCUSSIONS

6.4.1 COMPARISON OF THE HLS-BASED DESIGNS (RESOURCE UTILIZATIONS & CLOCKS)

This subsection constitutes the presentation and comparison of the performance characteristics of the three HLS-based designs in terms of target/actual clock frequency and resource utilization.

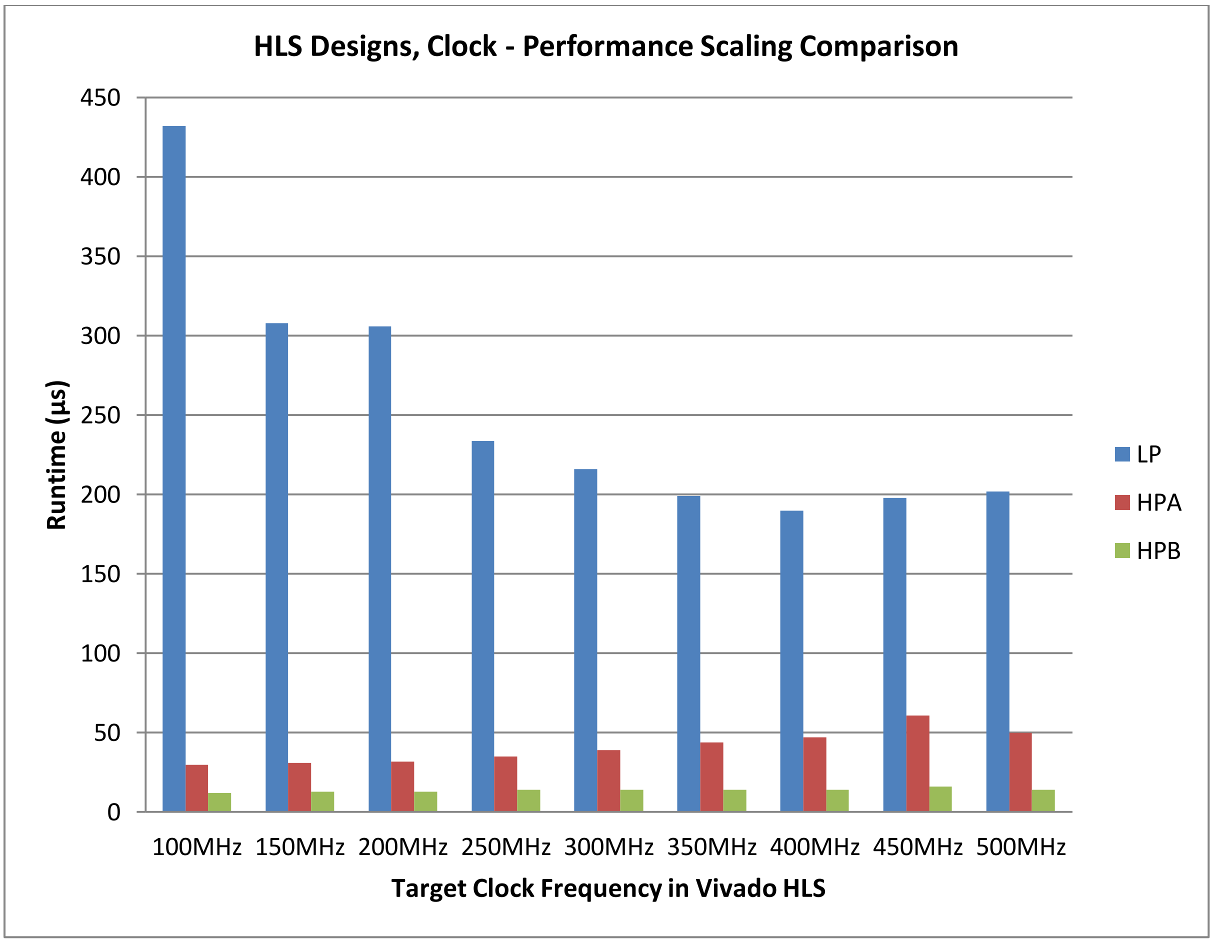


Figure 27: Illustration of the impact of various target clock frequencies to the runtime of each HLS-based design; the dataset under consideration is berlin52.

As it can be inferred from the figure, the low-performance, semi-pipelined LP design is favored by the increase in target clock frequency, despite the implied subsequent raise in the number of pipeline stages; the lowest runtime is achieved at a desired clock frequency of 400MHz. As far as the two high-performance designs are concerned, an inverse behavior is observed; although not clearly visible, the runtime-wise performance of both designs deteriorates as the target clock frequency increases. Thus, the lowest runtime is achieved at a desired clock frequency of 100MHz.

Based on these preliminary results, all datasets were subsequently tested (i.e. synthesized and implemented) in each design's appropriate clock frequency of:

- **400MHz/350MHz** for the **LP** design; the 350MHz target stems from the fact that during the dataset testing phase it was quickly noted that the aforementioned frequency could not be achieved after surpassing the 100-city mark. Thus, for datasets larger than 100 cities, a target clock frequency of 350MHz instead of the originally-planned 400MHz was used.
- **100MHz** for both **HPA** and **HPB** designs.

The actually achieved clock frequencies, obtained for each HLS-based design and TSPLIB dataset after synthesis and implementation in the Vivado IDE (post Place & Route), are displayed in the table below; the datasets under consideration are 13 in total, ranging in size from 51 to 264 cities.

TSPLIB dataset	Clock Frequency (MHz) *LP*	*HPA*	*HPB*
eil51	402	120	105
berlin52	400	126	106
eil76	397	121	105
kroA/B/C/D/E100	372	121	108
eil101	352	115	107
pr107	372	118	101
pr124	372	114	107
pr136	366	113	103
pr144	362	113	106
kroA/B150	365	112	101
pr152	364	110	100
kroA/B200	373	113	100
pr264	361	108	101

Table 2: The actual, post-Place & Route clock frequencies reported by the Vivado IDE. Target FPGA: Virtex-7 330T.

All designs proved capable of attaining their requested clock frequencies, while most of them even surpassed it.

The following table illustrates the resource utilization of the three designs in terms of Look Up Tables (LUTs), Flip-Flops (FFs) and Digital Signal Processors (DSPs); the values are obtained after the completion of the Place & Route phase of Vivado IDE and concern the aforementioned TSPLIB datasets. BRAM utilization remains minimal for all designs and datasets; thus it is not taken into consideration.

	LUT (%) (out of 204K)			FF (%) (out of 408K)			DSP (%) (out of 1120)		
TSPLIB dataset	*LP*	*HPA*	*HPB*	*LP*	*HPA*	*HPB*	*LP*	*HPA*	*HPB*
eil51	751 **1**	7033 **4**	11163 **6**	1162 **1**	5128 **1**	7117 **2**	8 **1**	56 **5**	208 **19**
berlin52	758 **1**	7077 **4**	11257 **6**	1162 **1**	5185 **1**	7182 **2**	8 **1**	56 **5**	208 **19**
eil76	814 **1**	10097 **5**	16263 **8**	1216 **1**	7566 **2**	10578 **3**	8 **1**	80 **7**	304 **27**
krox100	830 **1**	13097 **6**	21541 **11**	1216 **1**	9929 **2**	13772 **3**	8 **1**	96 **9**	400 **36**
eil101	823 **1**	13044 **6**	21677 **11**	1216 **1**	9999 **3**	13837 **3**	8 **1**	96 **9**	400 **36**
pr107	842 **1**	13856 **7**	23346 **11**	1128 **1**	10611 **3**	14688 **4**	8 **1**	96 **9**	432 **39**
pr124	854 **1**	15394 **8**	26604 **13**	1128 **1**	12136 **3**	17716 **4**	8 **1**	96 **9**	496 **44**
pr136	879 **1**	16825 **8**	29386 **14**	1182 **1**	13245 **3**	18908 **5**	8 **1**	96 **9**	544 **49**
pr144	878 **1**	17494 **9**	31241 **15**	1182 **1**	13972 **3**	19473 **5**	8 **1**	96 **9**	576 **51**
krox150	895 **1**	19178 **9**	32670 **16**	1182 **1**	14735 **4**	20835 **5**	8 **1**	112 **10**	608 **54**
pr152	888 **1**	19164 **9**	32829 **16**	1182 **1**	14856 **4**	20912 **5**	8 **1**	112 **10**	608 **54**
krox200	887 **1**	25650 **13**	43064 **21**	1182 **1**	19706 **5**	28122 **7**	8 **1**	168 **15**	800 **71**
pr264	936 **1**	32934 **16**	57165 **28**	1236 **1**	25733 **6**	37692 **9**	8 **1**	184 **16**	1056 **94**

Table 3: Post Place & Route resource utilization for each HLS-based design and TSPLIB dataset; the second number in each column is the utilized percentage of the corresponding resource. Target FPGA: Virtex-7 330T.

As it is evident by the table above, the resource which is in greatest demand-at least when the HPB design is concerned-is the DSP; this is expected due to both the large number of multiplications (squares) involved in the calculation of the (squared) Euclidean distances, which are necessary for the computation of the Delta() function for each one of the N/4 symmetrical segments, and the fact that the HPB design most probably schedules all these multiplications to be executed in parallel as well. This, in turn, significantly raises the requirements in multiplier cores which are subsequently mapped to DSPs.

The two other resources are of relatively low demand for all three designs-even for the DSP-intensive HPB design. Remarkably, the resource requirements of the LP design remain unchanged and equal to 1% for all resource types and datasets. It is noted that as far as the HPA design is concerned, larger datasets could have admittedly been examined; yet this was deemed unavailing for reasons that will soon become apparent.

6.4.2 VHDL-BASED S2OPT & COMPARISON WITH HLS-BASED DESIGNS (RESOURCE UTILIZATIONS & CLOCKS)

This subsection considers the presentation of the performance characteristics of the VHDL-based architecture in terms of resource utilization and actual clock frequency; a comparison with the HLS-based designs is also provided. Seven more TSPLIB datasets are introduced, thus increasing the upper limit of the number of cities under consideration from 264 to 2319; this applies only to the VHDL-based architecture, for reasons that will become apparent in the next subsection.

	VHDL-Based Architecture			
TSPLIB dataset	**LUT (out of 204K)**	**FF (out of 408K)**	**DSP (out of 1120)**	**Clock Freq. (MHz)**
eil51	2758 1%	2598 1%	13 1%	441
berlin52	3503 2%	2643 1%	13 1%	455
eil76	5093 3%	3844 1%	19 2%	433
kroA/B/C/D/E100	5169 3%	5019 1%	25 2%	429
eil101	5328 3%	5128 1%	26 2%	412
pr107	5577 3%	5386 1%	27 2%	407
pr124	6374 3%	6208 2%	31 3%	407
pr136	7037 4%	6838 2%	34 3%	403
pr144	7488 4%	7251 2%	36 3%	405
kroA/B150	7789 4%	7607 2%	38 3%	408
pr152	7785 5%	7667 2%	38 3%	413
kroA/B200	10257 5%	10065 3%	50 5%	394
pr264	14080 7%	13323 3%	66 6%	415
pr299	16993 8%	15096 4%	75 7%	403
pr439	29096 14%	22174 5%	110 10%	320
rat783	51211 25%	39319 10%	196 18%	323
vm1084	70920 35%	54294 13%	271 24%	319
u1432	93606 46%	71690 18%	358 32%	313
vm1748	114495 56%	87662 22%	437 39%	286
u2319	152015 75%	116005 28%	580 52%	261

Table 4: VHDL-based S2OPT architecture post Place & Route resource utilization and actual clock frequency for each TSPLIB dataset; the second number in each column is the utilized percentage of the corresponding resource. Target FPGA: Virtex-7 330T.

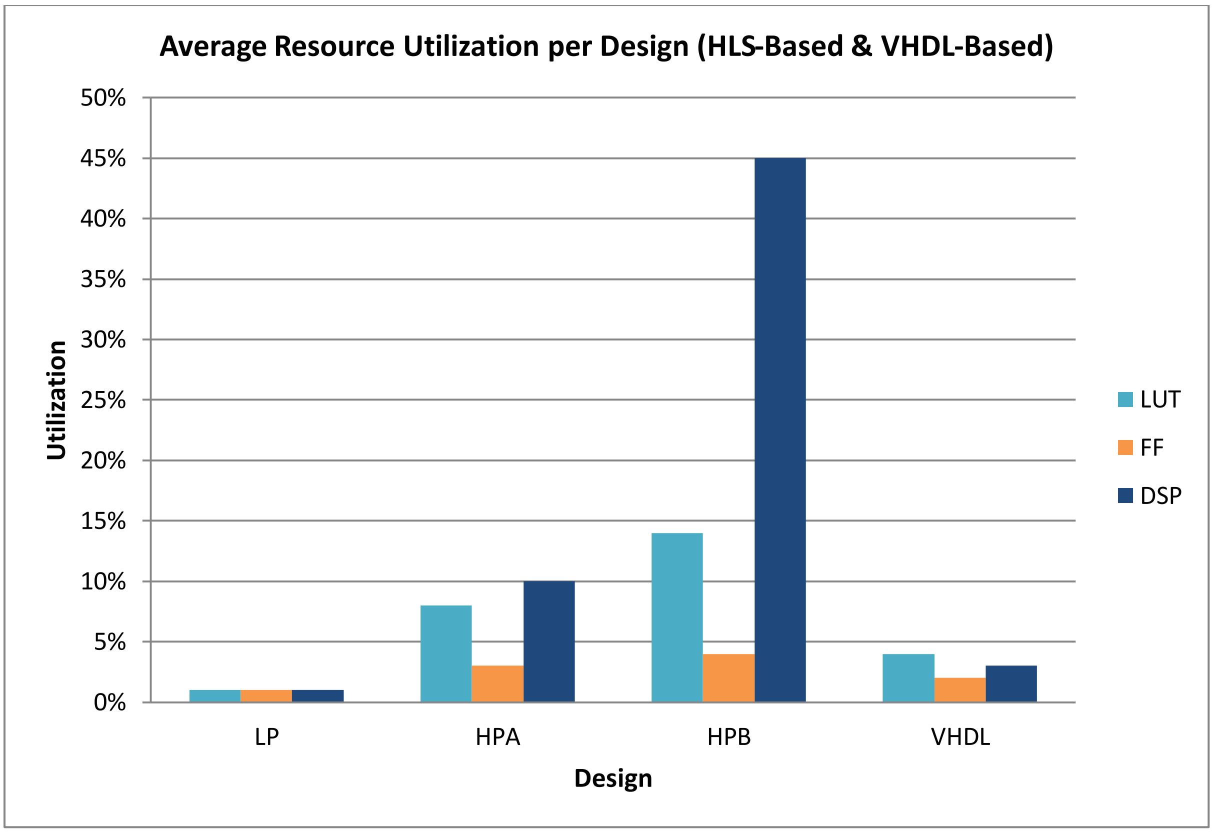


Figure 28: Post Place & Route average resource utilization per evaluated design and resource type; as far as the VHDL-based design is concerned, only the datasets tested with the HLS-based designs are taken into consideration. Target FPGA: Virtex-7 330T.

	Design							
	HLS-LP		*HLS-HPA*		*HLS-HPB*		*VHDL*	
LUT	**1%**	849	**8%**	16219	**14%**	27554	**4%**	6788
FF	**1%**	1183	**3%**	12523	**4%**	17756	**2%**	6429
DSP	**1%**	8	**9%**	103	**46%**	511	**3%**	32

Table 5: The percentages that are illustrated in the figure above, along with their actual corresponding values.

	Design			
	HLS-LP	*HLS-HPA*	*HLS-HPB*	*VHDL*
Average Clock Frequency	374MHz	116MHz	104MHz	417MHz

Table 6: Post Place & Route average clock frequency per evaluated design; as far as the VHDL-based design is concerned, only the datasets tested with the HLS-based designs are taken into consideration. Target FPGA: Virtex-7 330T.

RECAP: Based on the results presented so far on the resource utilization and attainable clock frequency of all the examined designs, the following inferences can be drawn:

- The VHDL-based design boasts the second-lowest average resource utilization after the HLS-based LP design; the two other HLS-based designs, namely the HPA and HPB designs, trail considerably far behind, achieving the 3rd and 4th position respectively.

- In all three HLS-based designs-with the partial exception of the LP design, the most sought-after resource is the DSPs; this is not the case with the VHDL-based design, as the resource which is in greatest demand turns out to be the LUTs. The aforementioned fact is mainly attributed to the large sum of multiplexers located within both the RGU and the PEs as well.
- The VHDL-based design managed to surpass all three HLS-based designs in terms of attainable clock frequency, achieving a maximum value of 455MHz in the berlin52 dataset and an average of 417MHz; the latter concerns only the 13 first datasets, i.e. those that were tested with the HLS-based designs as well.
- All four examined designs show a clear trend of reducing clock frequency as the number of cities increases; as far as the VHDL-based design is concerned, this behavior is attributed to the issue affecting the PRU's SGU module, which was discussed in subsection 5.5.3.2 of chapter 5.

6.4.3 VHDL-BASED & HLS-BASED S2OPT COMPARED TO CONCORDE (RUNTIMES, SPEEDUPS & TOUR QUALITY)

The subsection begins with the presentation of the results concerning the runtimes, speedups and tour quality obtained by all four examined designs when compared to Concorde; a subsequent, related discussion follows shortly after.

	Concorde	**HLS-LP**		**HLS-HPA**		**HLS-HPB**		**VHDL**	
TSPLIB	***Runtime (μs)***	***Runtime (μs)***	***Speedup***	***Runtime (μs)***	***Speedup***	***Runtime (μs)***	***Speedup***	***Runtime (μs)***	***Speedup***
eil51	104	244	0.4x	41	2.5x	16	6.5x	27	3.9x
berlin52	118	190	0.6x	30	3.9x	12	9.8x	19	6.2x
eil76	172	500	0.3x	60	2.9x	28	6.1x	40	4.3x
kroA100	211	857	0.2x	102	2.1x	35	6.0x	51	4.1x
kroB100	210	998	0.2x	119	1.8x	40	5.3x	60	3.5x
kroC100	216	1041	0.2x	124	1.7x	42	5.1x	62	3.5x
kroD100	223	969	0.2x	116	1.9x	39	5.7x	58	3.9x
kroE100	216	985	0.2x	118	1.8x	40	5.4x	59	3.7x
eil101	207	1222	0.2x	147	1.4x	47	4.4x	72	2.9x
pr107	170	887	0.2x	103	1.7x	37	4.6x	53	3.2x
pr124	174	924	0.2x	98	1.8x	37	4.7x	49	3.6x
pr136	225	1167	0.2x	113	2.0x	44	5.1x	56	4.0x
pr144	185	1670	0.1x	173	1.1x	57	3.2x	75	2.5x
kroA150	309	1964	0.2x	200	1.6x	68	4.5x	90	3.4x
kroB150	315	2247	0.1x	228	1.4x	78	4.0x	103	3.1x
pr152	253	2041	0.1x	208	1.2x	71	3.6x	91	2.8x
kroA200	396	4571	0.1x	410	1.0x	143	2.8x	170	2.3x
kroB200	395	4462	0.1x	400	1.0x	139	2.8x	166	2.4x
pr264	319	4799	0.1x	372	0.9x	140	2.2x	133	2.4x
pr299	391							277	1.4x
pr439	527							388	1.4x
rat783	1329							961	1.5x
vm1084	2404							2143	1.2x
u1432	1119							768	1.6x
vm1748	4442							4341	1.1x
u2319	1858							1805	1.1x

Table 7: Runtimes and speedups of the four evaluated S2OPT designs against Concorde; the results are based on the post Place & Route clock reports of the Vivado IDE.

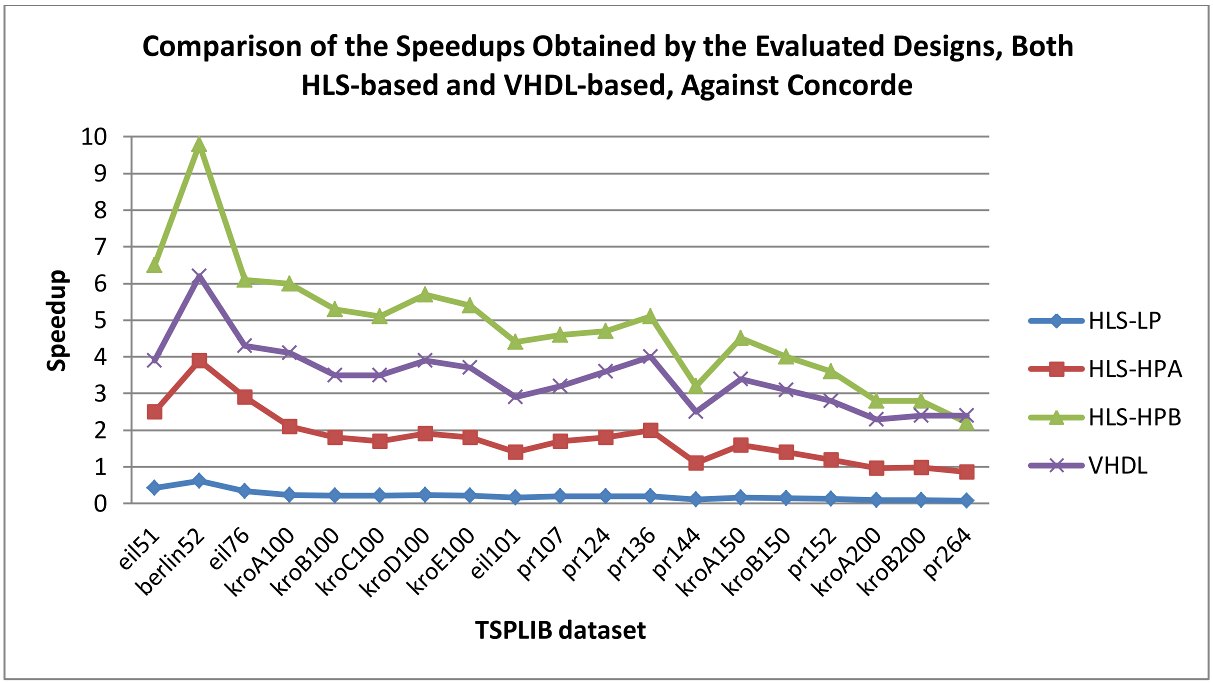


Figure 29: Graphical representation of the speedups obtained by the evaluated designs, when compared against Concorde; only the datasets that were tested with all four designs are considered.

	Design			
	HLS-LP	*HLS-HPA*	*HLS-HPB*	*VHDL*
Average Speedup	0.2x	1.8x	4.8x	3.5x

Table 8: Average speedups obtained by the evaluated designs when compared against Concorde; only the datasets that were tested with all four designs are considered.

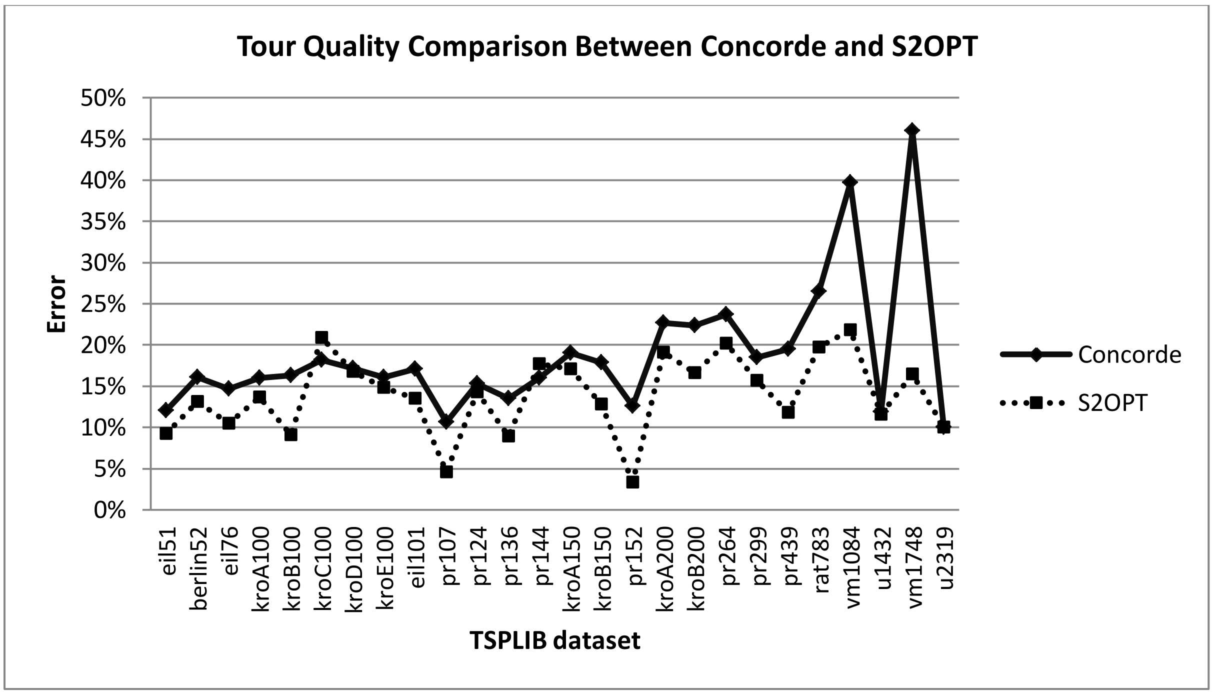


Figure 30: Comparison of the quality of the tours obtained by the evaluated designs (S2OPT) and Concorde; the quality is measured as the percentage of the output tour's length above the optimal (error). A lower error translates to higher quality (closer to optimal) tour.

	Concorde	S2OPT
Average Error	19%	14%

Table 9: Mean error of the tours yielded by both Concorde and S2OPT, averaged over the 26 tested TSPLIB datasets.

OBSERVATIONS:

- The fastest design in terms of the average speed up against Concorde is the HLS-HPB design, followed by the VHDL design and the HLS-HPA design; the HLS-LP design is completely out of competition as it is incapable of delivering any speedup, yet that was largely expected from the point of its inception.
- Throughout the range of the evaluated datasets, the VHDL design performs clearly better than the HLS-HPA design; the same holds true for the HLS-HPB design when compared to the VHDL design, with the exception of the pr264 dataset, where the aforementioned trend seems to be reversed. This could not be investigated further due to the fact that the target device (the Virtex-7 330T) had reached its full DSP-wise capacity and no larger HLS-HPB designs could be tested, at least not without using a higher-density FPGA, which in turn would distort the accuracy of the obtained results.
- Runtime-wise, the VHDL design is on average 94% faster than its HLS-HPA counterpart and 37% slower than the HLS-HPB design, yet it manages average resource utilization considerably smaller than both HLS designs under consideration, thus signifying its high levels of resource efficiency.
- The aforementioned efficiency of the VHDL design enabled the evaluation of larger datasets containing up to 2319 cities; this was infeasible in the case of the HLS-HPB design due to resource limitations and indifferent in the case of the HLS-LP and HLS-HPB designs due to the lack of speedups. Admittedly, the speedups achieved by the VHDL design within the range of 299 to 2319 cities are quite modest, averaging in 1.3x. The largest dataset that could be tested with the VHDL design in the Virtex-7 330T comprised of roughly 5000 cities, yet that was deemed completely unnecessary due to performance limitations.
- All four designs exhibit an overall decline in their speedups as the number of cities within the dataset under consideration grows; this is attributed to the fact that they all share the common bottleneck induced by the SGU, as described in subsection 5.5.3.2 of chapter 5.
- The implemented S2OPT hardware designs yield TSP tours of notably higher quality than those outputted by Concorde, with only a few exceptions to the rule; this is quite remarkable since the hardware designs do not compute the actual Euclidean distance as Concorde does, but rather an approximation to it (its square), which is suboptimal as discussed in the corresponding section of chapter 4. Yet the aforementioned designs achieve an average error of 14% compared to 19% managed by Concorde.

6.4.4 VHDL-BASED S2OPT COMPARED TO ORIGINAL S2OPT

The main topic of this subsection is the comparison of the performance characteristics of the VHDL-based design with those reported in the corresponding original implementation of 2007 [29]; only the VHDL-based design is considered as it was developed from the very beginning with the aim of largely resembling the original work for the sole purpose of this particular subsection.

	Original Implementation [29] *Virtex-2 Pro 100*		**Results Of This Implementation** *Virtex-7 330T*		
TSPLIB dataset	**Clock Frequency (MHz)**	**Runtime (μs)**	**Clock Frequency (MHz)**	**Runtime (μs)**	**Speedup**
berlin52	184	47	455	19	2.5x
eil76	178	95	433	40	2.4x
pr299	165	597	403	277	2.2x

Table 10: Comparison of the clock frequencies and runtimes achieved by the VHDL-based design developed within the context of the thesis (right) with those reported in the original implementation of 2007 (left).

Information concerning the resource utilization is intentionally omitted due to the fundamental differences in the internal structure of the FPGAs under consideration, which render the accurate comparison based on the aforementioned parameter a rather infeasible task.

The average speedup attained by the VHDL-based implementation designed within the context of the thesis against the original work of 2007 [29] is roughly 2.4x, which is exactly equal to the marked increase in clock frequency (average of 430MHz and 176MHz respectively); this attests that the two implementations under consideration are indeed quite similar, since the runtime-wise performance gains are clearly attributed to the corresponding rise in clock frequency.

6.4.5 RRHC: VHDL-BASED S2OPT COMPARED TO GPU-BASED 2-OPT

The final subsection considers the partial implementation and evaluation of the 2-OPT version of the Random Restart Hill Climbing (RRHC) metaheuristic in reconfigurable logic (FPGA); before proceeding further with the discussion, a few clarifications should be made:

- As is the case with all the implementations presented within the context of the thesis, the design under consideration has not been actually downloaded to an FPGA.
- It is assumed that the 2-OPT heuristic embedded within the RRHC metaheuristic is realized through the VHDL-based S2OPT hardware design, which was presented in the previous chapter.
- The term "partial" refers to the fact that only the underlying computing engines, responsible for the parallel execution of the embedded 2-OPT heuristic have been evaluated-not the metaheuristic as a whole.

The work discussed here was inspired by the equivalent GPU-based implementation presented in [52], which constitutes the benchmark for the subsequent evaluation of the design under consideration.

Briefly explained, the basic steps of the 2-OPT based RRHC metaheuristic are the following:

1. Compute an approximate solution to the TSP instance under consideration using the 2-OPT heuristic.
2. Store the currently best (smallest overall length) solution found by the aforementioned heuristic.
3. Apply a random permutation to the resulting tour.

4. Go to step 1 and restart the process until a solution of satisfactory quality is found.

The parallelism offered by the scheme under consideration is twofold:

1. Intra-2-OPT-move-evaluation parallelism, which is already exploited by the VHDL-based S2OPT design, and most probably by the HLS-based ones as well.
2. Intra-2-OPT-solver parallelism, available among individual 2-OPT solvers, each one working on a different random permutation of the TSP tour under consideration.

This second form of parallelism can be realized by instantiating multiple cores of the VHDL-based S2OPT design; these computing engines are completely independent since there is no form of information exchange or synchronization involved between them. As far as the application of the random permutations on the tours is concerned, this can be achieved through the utilization of the complex dedicated datapaths within each core's RGU; from a theoretical point of view, no major modifications to the internal structure of the VHDL-based design are required in order to adapt it for the scenario discussed here.

Moving on, the metric used in [52] for the evaluation of the GPU-based implementation is its throughput, expressed in terms of the number of 2-OPT move evaluations performed per second; thus the unit of measurement is the Giga-Move-Evaluations per second, GME/s. This made clear that only the VHDL-based implementation could be considered, due to the fact that the exact knowledge of its underlying architecture enables the accurate computation of the number of 2-OPT move evaluations performed per second through the following formula:

$$\mathrm{GME/s} = \frac{N\,c}{2\,y\,z}$$

The quantities corresponding to the variables are the following:

- **N**, the number of cities within the TSP instance under consideration.
- **c**, the number of the instantiated individual cores.
- **y**, the number of clock cycles required for the execution of the main iteration of the S2OPT algorithm (the WHILE(1) loop in the terminology used in the presentation of the HLS-based design in chapter 5).
- **z**, the clock period expressed in nanoseconds.

Before proceeding to presentation of the results, it should be noted that those consider the peak theoretical throughput, based on the aforementioned formula, the post Place & Route clock reports generated by the Vivado IDE, and the following assumptions:

- All #c cores operate and evaluate 2-OPT moves simultaneously, i.e. maximum utilization.
- An abstract, high speed control entity, responsible for maintaining the best tour and monitoring the computing engines, is assumed; it could be realized either as a piece of software running on an on-fabric soft/hard-core CPU or as a dedicated hardware module. An external CPU is also a possibility, i.e. a heterogeneous system such as the Convey series of hybrid supercomputers.

- The bottlenecks induced by (a) the memory subsystem accessible by both the control entity and the computing engines and (b) the buses connecting the control entity and computing engines with the memory subsystem, are considered negligible.

	GPU [52]	VHDL-Based Design (Multiple Cores)						
# Cities	Throughput (GME/s)	Throughput (GME/s)	Speedup	#Cores	Clock Frequency (MHz)	LUT (out of 612K)	FF (out of 1.2M)	DSP (out of 3600)
36	7.3	50.9	7x	300	339	94%	45%	75%
62	16.6	47.3	2.9x	165	351	94%	43%	73%
107	28.3	48.5	1.7x	100	341	96%	44%	75%
186	38.4	45.8	1.2x	58	323	96%	45%	76%
321	45.6	46	1.01x	35	327	97%	45%	79%

Table 11: Post Place & Route performance results of the evaluated FPGA-based approach to the 2-OPT version of RRHC utilizing multiple independent cores of the VHDL-based S2OPT hardware design discussed in the previous sections. Target FPGA: Xilinx Virtex-7 980T. The GPU used in [52] is an Nvidia Tesla K40 with 2880 stream processors running at about 800MHz.

OBSERVATIONS:

- The throughput of the GPU-based implementation [52] is relatively low for the smallest dataset under consideration, and increases as the number of cities rises, reaching the throughput levels achieved by the VHDL-based design near the 321-city mark; although not shown here, it is reported in [52] that the throughput continues to climb to a peak rate of around 60GME/s (8546 cities).
- On the contrary, the throughput of the VHDL-based design remains rather high within the range of the tested datasets, achieving a peak rate of 50.9GME/s and an average of 47.7GME/s; this is probably the upper limit of the design's capabilities, as increasing the number of cities further will definitely lead to performance throttling due to the previously discussed issue affecting the SGU and the fact that the design cannot be efficiently pipelined.
- Subsequently, the average speedup obtained by the VHDL-based design within the range of the evaluated datasets is 2.8x; a maximum of 7x is recorded for the smallest dataset under consideration and decreases rapidly as the number of cities grows.
- All in all the GPU-based implementation scales better than the VHDL-based implementation, even in the best-case scenario examined here, thus rendering the latter an attractive alternative only in the case where small TSP instances are considered.

CHAPTER 7: CONCLUSION & FUTURE WORK

7.1 CONCLUSION

The objectives of the thesis were manyfold, and were achieved to a great extent:

1. The evaluation and comparison of the HLS and HDL-based design flows in terms of both their inherent characteristics and the quality of the designs they produce, based on the results obtained by the implementation of Symmetrical 2-OPT (S2OPT).
2. The estimation of the performance enhancements yielded by the latest generation of FPGAs and design tools, using the work conducted in [29] (original S2OPT) as a reference point.
3. The assessment of how well the architecture presented in [29] (original S2OPT)-which was redesigned and re-implemented for the purposes of this thesis -fares against both the classic Concorde and a state-of-the-art GPU-based 2-OPT implementation as well [52], almost a decade after its inception.
4. The exploration of how effectively the architecture presented in [29] (original S2OPT) scales as the length of the TSP tour increases, a question that constituted part of the future work of the aforementioned publication.

The conclusions drawn are the following:

1. In general, both design flows are equally capable of delivering high quality of results, if their corresponding timing requirements are sufficiently met. The HLS flow, realized through the Xilinx Vivado HLS, is under constant improvement and is more flexible and time-efficient than the HDL-based flow which is realized through the Xilinx Vivado IDE. On the other hand, the latter poses the classic, long-proven approach to hardware design, with powerful capabilities and exceptional levels of control over the design under development, which simply cannot be matched by its counterpart.
2. Within the context of the application under consideration, there was a marked average increase of 2.4x in the attainable clock frequency, resulting from the switching from the venerable yet aging Xilinx Virtex-2 Pro FPGA utilized in [29] to the much newer Xilinx Virtex-7 FPGA; remarkably, this led to an exact 2.4x average reduction in the required execution times.
3. The hardware architecture under consideration proved fairly capable, achieving speedups in the range of 1.1x to nearly 10x when compared against Concorde, depending on the dataset used and the implementation version. As far as the GPU-based implementation is concerned [52], the speedups attained from the VHDL-based design varied from 1.01x to 7x; in both cases the best performance figures were obtained during the evaluation of small TSP instances, hinting that the hardware architecture is more suitable for small scale TSPs.
4. The results obtained by both design flows show a clear trend of decreasing performance as the number of cities within the TSP tour under consideration rises; this behavior is largely attested to the fact that the hardware designs cannot be efficiently pipelined due to the highly iterative nature of the Symmetrical 2-OPT algorithm that causes the aforementioned pipeline to fill and empty at every such iteration.

7.2 FUTURE WORK

Possible future extensions to the work presented in this thesis include the following:

- The inclusion of the hardware required for the computation of the square root of the Euclidean distance and the examination of its subsequent impact on the performance characteristics of the designs obtained by both design flows.
- Explore how the hardware architecture under consideration could be modified in order to support 2.5/r/3-OPT moves and provide corresponding FPGA-based implementations.
- The design of an FPGA-based implementation of a metaheuristic such as Iterated Local Search or Simulated Annealing embedding the 2-OPT hardware architecture under consideration.

APPENDIX

A.1 MATLAB SCRIPT UTILIZED IN “APPROACH A” DISCUSSED IN SECTION 4.13, CHAPTER 4.

```
%Georgios Malandrakis Miller, School of ECE, Technical University of Crete
close all;
clear all;
clc

%Set the iteration/experiment count and max 2D coordinates.
ITERS= 10^6;
MAX_COORD= 2^16;

%Initialize the required structures.
delta_EUC_SQ= inf(ITERS,1);
delta_EUC= inf(ITERS,1);

%Perform 1M iterations/experiments.
for i=1:ITERS
    %Randomly generate 4 points in the 2D Euclidean space.
    A= [randi(MAX_COORD) randi(MAX_COORD)];
    B= [randi(MAX_COORD) randi(MAX_COORD)];
    C= [randi(MAX_COORD) randi(MAX_COORD)];
    D= [randi(MAX_COORD) randi(MAX_COORD)];

    %Consider that initially the points above are connected in this sense:
    %
    %   A<->B<->C<->D<->A
    %
    %Now consider that we decide to "break" the path between points A,B
    %and C,D and connect them the only other possible way, ie.:
    %
    %   A<->C<->B<->D<->A
    %
    %thus simulating a 2-OPT move.
    %
    %The "old" distances would be dist(A,B) and dist(C,D) while the "new"
    %distances will be dist(A,C) and dist(B,D).
    %
    %Now we move on to compute these distances, first by using the square
    %of the Euclidean distance as the cost fucntion and then by using the
    %standard Euclidean distance.

    %Computing the distances using the square of the Euclidean distance.
    distOLD1_EUC_SQ= (A(1)-B(1))^2 + (A(2)-B(2))^2;
    distOLD2_EUC_SQ= (C(1)-D(1))^2 + (C(2)-D(2))^2;
    distNEW1_EUC_SQ= (A(1)-C(1))^2 + (A(2)-C(2))^2;
    distNEW2_EUC_SQ= (B(1)-D(1))^2 + (B(2)-D(2))^2;

    %Computing the distances using the standard the Euclidean distance.
    distOLD1_EUC= sqrt((A(1)-B(1))^2 + (A(2)-B(2))^2);
    distOLD2_EUC= sqrt((C(1)-D(1))^2 + (C(2)-D(2))^2);
    distNEW1_EUC= sqrt((A(1)-C(1))^2 + (A(2)-C(2))^2);
    distNEW2_EUC= sqrt((B(1)-D(1))^2 + (B(2)-D(2))^2);

    %To evaluate if the aforementioned move is length-reducing or not, we
    %need to compute the difference between the sum of the "new" distances
    %and the sum of the "old" distances. That is
    %
    %   delta= (dist(A,C)+dist(B,D)) - (dist(A,B)+dist(C,D))
    %
    %If the quantity above is negative, then the 2-OPT move is length
    %reducing and should be applied. Otherwise, it should be avoided.
    delta_EUC_SQ(i)= ((distNEW1_EUC_SQ + distNEW2_EUC_SQ) -
(distOLD1_EUC_SQ + distOLD2_EUC_SQ));
    delta_EUC(i)= ((distNEW1_EUC + distNEW2_EUC) - (distOLD1_EUC +
distOLD2_EUC));
end

%Now we compute which of the 1M moves were length-reducing, by assigning a
%logical '1' in every negative "delta" value, in both cases.
delta_EUC_SQ_NEG= (delta_EUC_SQ < 0);
delta_EUC_NEG= (delta_EUC < 0);

%We then compute the error vector by subtracting the Squared Euclidean
%results from the standard Euclidean results:
error_vector= delta_EUC_NEG - delta_EUC_SQ_NEG;

%This way we are able to distinguish between the 2-OPT moves that were
%applied in the standard Euclidean case but not in the Squared Euclidean,
%and vice-versa.
ERR1= sum(error_vector<0);
ERR2= sum(error_vector>0);

disp(['Number of 2-OPT moves applied in the standard Euclidean case but not
in the squared Euclidean: ',num2str(ERR2),' or
',num2str(ERR2/ITERS*100),'%']);
disp(['Number of 2-OPT moves applied in the squared Euclidean case but not
in the standard Euclidean: ',num2str(ERR1),' or
',num2str(ERR1/ITERS*100),'%']);
```

```
Number of 2-OPT moves applied in the standard Euclidean case but not in the
squared Euclidean: 46575 or 4.6575%

Number of 2-OPT moves applied in the squared Euclidean case but not in the
standard Euclidean: 46279 or 4.6279%

>>
```

Arithmetic Example from the berlin52 instance of TSPLIB:

A=(25,230) B=(525,1000) C=(510,875) D=(560,365)

distAB_OLD1_EUC_SQ= $(525-25)^2 + (1000-230)^2 = 500^2 + 770^2 = 250000 + 592900 =$ **842900**

distAB_OLD1_EUC= $\sqrt{842900} =$ **918**

distCD_OLD2_EUC_SQ= $(510-560)^2 + (875-365)^2 = (-50)^2 + 510^2 = 2500 + 260100 =$ **262600**

distCD_OLD2_EUC= $\sqrt{262600} =$ **513** distAC_NEW1_EUC_SQ= $(25-510)^2 + (230-875)^2 = (-485)^2 + (-645)^2 = 235225 + 416025 =$ **651250**

distAC_NEW1_EUC= $\sqrt{651250} =$ **807**

distBD_NEW2_EUC_SQ= $(525-560)^2 + (1000-365)^2 = (-35)^2 + (635)^2 = 1225 + 403225 =$ **404450**

distBD_NEW2_EUC= $\sqrt{404450} =$ **636**

delta_EUC_SQ= (distAC_NEW1_EUC_SQ **+** distBD_NEW2_EUC_SQ**) – (**distAB_OLD1_EUC_SQ **+** distCD_OLD2_EUC_SQ**)= (651250 + 404450) – (842900 + 262600)= 1055700 – 1105500= -49800 < 0 => Length reducing move.**

delta_EUC= (distAC_NEW1_EUC **+** distBD_NEW2_EUC**) – (**distAB_OLD1_EUC **+** distCD_OLD2_EUC**)= (807 + 636) – (918 + 513)= 1443 – 1431= 12 > 0 => No length reducing move**

delta_EUC_SQ ≠ delta_EUC

A.2 HLS DESIGNS EVALUATIONS

The performance and utilization results obtained by the following evaluations are based on the berlin52 TSPLIB dataset; the target hardware platform is a Xilinx Virtex-7 330T with a default clock constraint of 100MHz (10ns). The following is a "light" version of the spreadsheet used during the development of the HLS-based hardware architectures; the original spreadsheet contained extra columns displaying per-loop pipeline-related information as well as remarks made by the author on the effectiveness, impact and usefulness of the various source code revisions, directives and their combinations, used for guidance, search space pruning and decision-making purposes. The "light" version is presented instead of the original due to space, illustration and clarity concerns. The "X" visible in the results of numerous evaluations implies that the design under consideration was either implemented successfully but failed to perform as intended (single "X" in the Latency column), or could not be implemented at all (multiple "X"s) due to fatal software errors. The number within the parenthesis that follows the description of many of the evaluations denotes the source code revision upon which the given directive(s) was (were) tested.

	Clock Period	*BRAM_18K*	*DSP48E*	*FF*	*LUT*	*Latency*
C SOURCE REVISIONS						
0_INITIAL SOURCE VERSION	8.23ns	0	8	1198	1240	261933
1_REDUNDANT ARRAY & VARIABLE REMOVAL PLUS PARTIAL FOR-LOOP MERGING	8.23ns	0	8	643	900	235539
2_REVAMPED IO WITH FIFO SUPPORT	8.23ns	4	8	661	978	208867
3_ARBITRARY PRECISION VARIABLES & #RESOURCE DIRECTIVE COMPATIBILITY	8.25ns	2	2	637	847	170023
4_UNIFIED CITY COORDS IN SINGLE 30bit VALUE USING BIT MANIPULATION METHODS	8.25ns	2	2	631	789	170023
5_CFUNC FUNCTION REDESIGN WITH EXTENDED & MORE EXPRESSIVE FUNCTIONALITY	8.59ns	2	8	475	696	130930

6_FURTHER PARTIAL FOR-LOOP MERGING	8.59ns	2	8	417	720	106035
5X_5 PLUS WRAPPED ARRAY (CT & SWAPS) ACCESSES/UPDATES	8.59ns	2	8	508	714	130930
DIRECTIVES EVALUATIONS						
#PIPELINE (3)						
PIPELINE S2OPT function	8.25ns	2	2	3881	5998	104445
PIPELINE S2OPT & CFUNC functions	8.25ns	2	2	3881	5998	104445
PIPELINE EVERYTHING (S2OPT & CFUNC functions + WHILE/MAIN loop & FOR/SUB loops)	8.25ns	2	2	3873	5978	104445
PIPELINE FOR/SUB loops	8.25ns	2	2	714	898	129837
PIPELINE FOR/SUB loops & CFUNC function	8.25ns	2	2	714	898	129837
PIPELINE FOR/SUB loops & CFUNC function & WHILE/MAIN loop	8.25ns	2	2	3744	5047	104450
#DATAFLOW (3)						
DATAFLOW S2OPT function	8.25ns	4	2	639	835	170024
DATAFLOW S2OPT & CFUNC function	X	X	X	X	X	X
DATAFLOW WHILE/MAIN loop	X	X	X	X	X	X
DATAFLOW FOR/SUB loops	X	X	X	X	X	X
#RESOURCE (3)						
MULTIPLIER CORES						
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= Mul-> LATENCY= 0	8.25ns	2	2	673	847	170023
FUNCTION= CFUNC-> VARIABLE(S)= multA-> CORE= MuL_LUT-> LATENCY= 0	8.43ns	2	1	637	1103	170023
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= MuL_LUT-> LATENCY= 0	7.61ns	2	0	637	1389	170023
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= MuL_LUT-> LATENCY= 5	8.29ns	2	0	639	1420	273607
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= MuL_*nS-> LATENCY= 0	8.24ns	2	2	515	907	170023
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= MuL_*nS-> LATENCY= 5	8.2ns	2	2	775	908	273607
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= DSP48-> LATENCY= 0	8.25ns	2	2	637	847	170023
FUNCTION= CFUNC-> VARIABLE(S)= multA & multB-> CORE= DSP48-> LATENCY= 4	8.25ns	2	2	637	847	170023
ADDER CORES						
FUNCTION= CFUNC-> VARIABLE= add-> CORE= AddSub-> LATENCY= 0	8.64ns	2	2	517	877	170023
FUNCTION= CFUNC-> VARIABLE= add-> CORE= AddSub_*nS-> LATENCY= 0	8.69ns	2	2	517	882	170023
FUNCTION= CFUNC-> VARIABLE= add-> CORE= AddSub_*nS-> LATENCY= 5	7.35ns	2	2	544	909	293029

FUNCTION= CFUNC-> VARIABLE= add-> CORE= AddSub_DSP-> LATENCY= 0	8.43ns	2	3	517	847	170023
FUNCTION= CFUNC-> VARIABLE= add-> CORE= AddSub_DSP-> LATENCY= 5	8.43ns	2	3	517	847	170023
#ARRAY PARTITION (3)						
TYPE= COMPLETE-> DIMENSION(S)= 1 & 2-> VARIABLE(S)= IT & FT	8.25ns	2	2	639	1343	170127
TYPE= COMPLETE-> DIMENSION(S)= 1-> VARIABLE(S)= CT	8.66ns	0	8	10983	24646	115019
TYPE= COMPLETE-> DIMENSION(S)= 1 & 2-> VARIABLE(S)= CT	8.66ns	0	8	10983	24646	115019
TYPE= COMPLETE-> DIMENSION(S)= 1-> VARIABLE(S)= SWAPS	8.25ns	2	2	677	909	169525
#INLINE (5)						
CFUNC	9.81ns	2	32	1109	923	111508
#LOOP_FLATTEN (5)						
COULD NOT BE APPLIED DUE TO MULTIPLE LOOPS WITHIN A GIVEN LEVEL						
#LOOP_MERGE (5)						
COULD NOT BE APPLIED DUE TO THE EXISTENCE OF NON-TRIVIAL CODE BETWEEN LOOPS						
#LOOP_UNROLL (5)						
READIN, FACTOR= COMPLETE	8.59ns	2	8	481	937	130851
WHILE, FACTOR= COMPLETE	8.59ns	2	8	481	716	130930
CSWA, FACTOR= COMPLETE	9.07ns	3	8	552	810	118729
CSWA, FACTOR= 2	9.3ns	2	8	522	821	124954
CSWA, FACTOR= 5	8.71ns	2	8	697	1052	126448
CSWA, FACTOR= 10	9.07ns	2	8	891	1299	123958
UPDA, FACTOR= COMPLETE	8.59ns	2	8	567	883	130398
#ALLOCATION (5)						
CORE= DSP-> LIMIT= 4	X	X	X	X	X	X
CORE= DSP-> LIMIT= 1	X	X	X	X	X	X
#ARRAY_PARTITION REVISITED & VARIABLE TYPE IMPACT ASSESSMENT (5)						
TYPE= AUTOMATIC, VARIABLE= CT	8.59ns	2	8	475	696	130930
TYPE= GLOBAL, VARIABLE= CT	8.59ns	2	8	475	696	130930
TYPE= STATIC, VARIABLE= CT	8.59ns	2	8	475	696	130930
TYPE= STATIC + GLOBAL, VARIABLE= CT	8.59ns	2	8	475	696	130930
VAR. TYPE= AUTOMATIC-> VARIABLE= CT & ARRAY_PARTITION-> TYPE= COMPLETE	8.59ns	0	8	11914	14778	151377

VAR. TYPE= GLOBAL-> VARIABLE= CT & ARRAY_PARTITION-> TYPE= COMPLETE	8.59ns	0	8	3514	5328	151377
VAR. TYPE= STATIC-> VARIABLE= CT & ARRAY_PARTITION-> TYPE= COMPLETE	8.59ns	0	8	3514	5328	151377
VAR. TYPE= STATIC + GLOBAL-> VARIABLE= CT & ARRAY_PARTITION-> TYPE= COMPLETE	8.59ns	0	8	3514	5328	151377

EVALUATION OF #ARRAY_PARTITION APPLIED TO CT ARRAY + FOR/SUB LOOPS PIPELINING (5,5X)

AP: ARRAY_PARTITION, B: BLOCK TYPE, x: FACTOR [e.g. APB8]
AP: ARRAY_PARTITION, CYC: CYCLICAL TYPE, x: FACTOR [e.g. APCYC16]
AP: ARRAY_PARTITION, COMP: COMPLETE TYPE, x: FACTOR [e.g. APC]

FORPIPELINE: FOR/SUB LOOPS PIPELINING

INLINE: INLINE THE CFUNC FUNCTION

APB2 (5)	8.59ns	4	8	1036	1668	433211
APB4 (5)	8.59ns	4	12	1334	2473	X
APB8 (5)	8.68ns	6	11	2492	4356	X
APB10 (5)	8.59ns	2	13	2487	4364	486646
APB12 (5)	8.59ns	10	14	2209	4074	X
APB13 (5)	8.62ns	0	8	12626	80833	109023
APB14 (5)	X	X	X	X	X	X
APB15 (5)	8.62ns	0	8	12665	48137	121722
APB16 (5)	8.62ns	0	8	12665	48137	121722
APB32 (5)	8.62ns	0	8	12665	48137	121722
APCYC2 (5)	8.59ns	4	8	831	1322	131174
APCYC4 (5)	8.59ns	8	8	878	1959	123953
APCYC8 (5)	8.59ns	16	8	1158	5119	176031
APCYC10 (5)	8.59ns	16	20	1937	5359	X
APCYC12 (5)	8.59ns	2	20	9755	18174	X
APCYC13 (5)	8.59ns	0	14	15572	33112	446667
APCYC14 (5)	8.59ns	0	20	18748	40013	697545
APCYC15 (5)	8.59ns	0	20	11636	19455	X
APCYC16 (5)	8.59ns	0	8	15580	33941	125691
APCYC32 (5)	X	X	X	X	X	X
APC (5)	8.59ns	0	8	13906	17417	144615
FORPIPELINE (5)	8.59ns	2	8	526	745	65595
FORPIPELINE + APB2 (5)	8.59ns	4	8	1222	1744	109668
FORPIPELINE + APB4 (5)	8.59ns	4	12	1897	2881	X
FORPIPELINE + APB6 (5)	8.59ns	6	9	2549	3811	X

FORPIPELINE + APB8 (5)	8.68ns	8	11	3093	4995	X
FORPIPELINE + APB10 (5)	10.22ns	4	132	3117	5009	116159
FORPIPELINE+APB12 (5)	10.22ns	10	14	2477	4374	X
FORPIPELINE+APB13 (5)	8.59ns	0	8	13306	86514	50406
FORPIPELINE+APB14 (5)	X	X	X	X	X	X
FORPIPELINE+APB16 (5)	8.59ns	0	8	13306	53818	50406
FORPIPELINE+APB32 (5)	8.59ns	0	8	11478	41274	49659
FORPIPELINE + APCYC2 (5)	8.59ns	4	8	913	1365	103443
FORPIPELINE + APCYC4 (5)	8.59ns	8	8	946	2110	103194
FORPIPELINE + APCYC6 (5)	8.59ns	12	24	2880	5090	X
FORPIPELINE + APCYC8 (5)	9.54ns	16	8	1046	5038	103194
FORPIPELINE + APCYC10 (5)	10.9ns	20	20	2434	5739	X
FORPIPELINE + APCYC12 (5)	8.59ns	8	20	13383	24513	X
FORPIPELINE + APCYC13 (5)	8.59ns	0	14	11281	38160	X
FORPIPELINE + APCYC14 (5)	8.59ns	0	20	14455	48368	63868
FORPIPELINE + APCYC16 (5)	8.59ns	0	8	11866	40670	49161
FORPIPELINE + APCYC32 (5)						
FORPIPELINE + APC (5X)	8.59ns	0	8	10636	19056	49161
APB2 (5X)	8.59ns	0	8	1083	1056	1124513
APB4 (5X)	8.59ns	0	12	1470	2139	X
APB6 (5X)	8.59ns	0	9	1416	1591	X
APB8 (5X)	8.59ns	0	9	1346	1432	1117541
APB10 (5X)	X	X	X	X	X	X
APB12 (5X)	X	X	X	X	X	X
APB13 (5X)	8.59ns	0	8	2086	603	117178
APB14 (5X)	X	X	X	X	X	X
APB15 (5X)	X	X	X	X	X	X
APB16 (5X)	X	X	X	X	X	X
APB32 (5X)	X	X	X	X	X	X
APCYC2 (5X)	8.59ns	0	8	747	741	256697
APCYC4 (5X)	8.59ns	0	8	1255	1903	X
APCYC6 (5X)	8.59ns	0	14	1474	1528	1103555
APCYC8 (5X)	8.59ns	0	8	1113	1207	249725
APCYC10 (5X)	8.59ns	0	10	1450	1514	1064711
APCYC12 (5X)	8.59ns	0	10	2116	1370	1127252
APCYC13 (5X)	8.59ns	0	13	2657	1065	845148
APCYC14 (5X)	8.59ns	0	13	2657	1637	845148
APCYC15 (5X)	8.59ns	0	13	2657	1893	845148
APCYC16 (5X)	8.59ns	0	8	2086	1443	117178
APCYC32 (5X)	8.59ns	0	8	2086	4187	117178

APC (5X)	8.59ns	0	8	2526	9831	116680
FORPIPELINE (5X)	8.59ns	2	8	551	763	65595
FORPIPELINE + APB2 (5X)	8.59ns	0	8	1393	2134	1243134
FORPIPELINE + APB4 (5X)	8.59ns	0	12	1713	2708	X
FORPIPELINE + APB6 (5X)	8.59ns	0	9	2018	2907	1246122
FORPIPELINE + APB8 (5X)	8.59ns	0	9	1753	2580	1242636
FORPIPELINE + APB10 (5X)						
FORPIPELINE + APB12 (5X)						
FORPIPELINE + APB13 (5X)	8.59ns	0	8	2112	640	95474
FORPIPELINE + APB14 (5X)	X	X	X	X	X	X
FORPIPELINE + APB15 (5X)	X	X	X	X	X	X
FORPIPELINE + APB16 (5X)	X	X	X	X	X	X
FORPIPELINE + APB32 (5X)	X	X	X	X	X	X
FORPIPELINE + APCYC2 (5X)	8.59ns	0	8	717	766	194328
FORPIPELINE + APCYC4 (5X)	8.59ns	0	8	1177	2130	141788
FORPIPELINE + APCYC6 (5X)	8.63ns	0	14	1829	2590	1174907
FORPIPELINE + APCYC8 (5X)	8.59ns	0	8	1101	1230	193830
FORPIPELINE + APCYC10 (5X)	8.59ns	0	10	1833	2596	1171919
FORPIPELINE + APCYC12 (5X)	8.59ns	0	10	2520	2551	1170356
FORPIPELINE + APCYC13 (5X)	8.59ns	0	13	2996	1783	861913
FORPIPELINE + APCYC14 (5X)	8.59ns	0	13	2996	2351	861913
FORPIPELINE + APCYC15 (5X)	8.59ns	0	13	2996	2607	861913
FORPIPELINE + APCYC16 (5X)	8.59ns	0	8	2112	1480	95474
FORPIPELINE + APCYC32 (5X)	8.59ns	0	8	2112	4224	95474
FORPIPELINE + APC (5X)	8.59ns	0	8	3858	9862	94976

#INLINE REVISITED (5,5X)

CFUNC INLINE (5X)	9.07ns	2	32	1116	947	117982
CFUNC INLINE & FORPIPELINE (5X)	9.07ns	2	32	1137	989	64599
CFUNC INLINE (5)	9.81ns	2	32	1126	942	111508
CFUNC INLINE & FORPIPELINE (5X)	9.81ns	2	32	1165	974	63603

#UNROLL REVISITED (5,5X)

UNROLL FOR/SUB LOOPS (5)	9.07ns	0	64	6475	6018	4782
PIPELINE WHILE/MAIN LOOP (5)	24.62ns	0	104	7594	8318	1298
PIPELINE WHILE/MAIN LOOP & INLINE CFUNC (5)	58.66ns	0	208	4703	8199	302
CFUNC INLINE & UNROLL FOR/SUB LOOPS (5)	9.75ns	0	208	9149	7334	2292
UNROLL FOR/SUB LOOPS (5X)	9.52ns	1	8	1328	3490	188819

PIPELINE WHILE/MAIN LOOP (5X)	9.07ns	1	16	1513	3626	135380
PIPELINE WHILE/MAIN LOOP & INLINE CFUNC (5X)	9.07ns	1	208	5878	6578	135380
CFUNC INLINE & UNROLL FOR/SUB LOOPS (5X)	9.07ns	1	208	5908	6609	182345

#INLINE APPLIED TO CFUNC + #ARRAY_PARTITION APPLIED TO CT ARRAY (5,5X)

CFUNC INLINE + APB2 (5)	9.07ns	4	32	1565	1871	420263
CFUNC INLINE + APB4 (5)	X	X	X	X	X	X
CFUNC INLINE + APB8 (5)	X	X	X	X	X	X
CFUNC INLINE + APB16 (5)	9.75ns	0	32	11485	46901	115248
CFUNC INLINE + APB32 (5)	9.75ns	0	32	11740	34523	115248
CFUNC INLINE + APCYC2 (5)	9.75ns	4	32	1160	1566	117977
CFUNC INLINE + APCYC4 (5)	9.89ns	8	32	1183	2339	111005
CFUNC INLINE + APCYC8 (5)	9.89ns	16	32	1470	5289	163083
CFUNC INLINE + APCYC16 (5)	9.75ns	0	32	14502	32711	119217
CFUNC INLINE + APCYC32 (5)	X	X	X	X	X	X
CFUNC INLINE + APC (5)	9.75ns	0	32	12828	16157	138141
CFUNC INLINE + APB2 (5X)	9.81ns	0	32	1778	1443	1105589
CFUNC INLINE + APB4 (5X)	9.89ns	0	36	1898	2413	X
CFUNC INLINE + APB8 (5X)	10.14ns	0	33	2082	1826	1105091
CFUNC INLINE + APB16 (5X)	X	X	X	X	X	X
CFUNC INLINE + APB32 (5X)						
CFUNC INLINE + APCYC2 (5X)	9.81ns	0	32	1442	1128	237773
CFUNC INLINE + APCYC4 (5X)	9.89ns	0	32	1642	2159	X
CFUNC INLINE + APCYC8 (5X)	10.14ns	0	32	1848	1601	237275
CFUNC INLINE + APCYC16 (5X)	9.75ns	0	32	2584	1835	113194
CFUNC INLINE + APCYC32 (5X)	9.75ns	0	32	2723	3679	113692
CFUNC INLINE + APC (5X)	9.07ns	0	32	2825	10141	110206

#UNROLL APPLIED TO FOR/SUB LOOPS+ #ARRAY_PARTITION APPLIED TO CT ARRAY (5,5X)

APB2 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6476	6018	4782
APB4 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB6 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB8 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB10 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB12 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782

APB13 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB14 + UNROLL_SUB_LOOPS (5)	X	X	X	X	X	X
APB15 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB16 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB32 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC2 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC4 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC6 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC8 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC10 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC12 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC13 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC14 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC15 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC16 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APCYC32 + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APC + UNROLL_SUB_LOOPS (5)	9.07ns	0	64	6470	5998	4782
APB2 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	8	1822	4802	943301
APB4 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	10	1936	4806	943301
APB6 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	9	2057	4981	943301
APB8 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	9	2212	5090	943301
APB10 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APB12 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APB13 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	104	5770	18179	74005
APB14 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APB15 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APB16 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APB32 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APCYC2 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	8	1451	3562	189317
APCYC4 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	8	1572	3574	189317
APCYC6 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	10	2065	4952	895981
APCYC8 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	8	1813	3788	189317
APCYC10 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	10	2303	5102	895981
APCYC12 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APCYC13 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	121	8697	19753	X
APCYC14 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	123	8943	20953	X
APCYC15 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
APCYC16 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	104	5787	21558	74005
APCYC32 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	104	5787	29118	74005
APC + UNROLL_SUB_LOOPS (5X)	9.07ns	0	104	6475	4602	73507

#INLINE APPLIED TO CFUNC + #UNROLL

APPLIED TO FOR/SUB LOOPS + #ARRAY_PARTITION APPLIED TO CT ARRAY (5,X)						
INLINE + APB2 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB4 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB6 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB8 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB10 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB12 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APB13 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APB14 + UNROLL_SUB_LOOPS (5)	X	X	X	X	X	X
INLINE + APB15 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APB16 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APB32 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APCYC2 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC4 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC6 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC8 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC10 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC12 + UNROLL_SUB_LOOPS (5)	10.14ns	0	208	9020	7964	2043
INLINE + APCYC13 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APCYC14 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APCYC15 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APCYC16 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APCYC32 + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APC + UNROLL_SUB_LOOPS (5)	9.75ns	0	208	9149	7334	2292
INLINE + APB2 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	208	6376	7911	942305
INLINE + APB4 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	210	6505	7933	942305
INLINE + APB6 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	209	6628	8108	942305
INLINE + APB8 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	209	6757	8199	942305
INLINE + APB10 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB12 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB13 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	208	7476	18999	73507
INLINE + APB14 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB15 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB16 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X

INLINE + APB32 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APCYC2 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	208	5999	6671	182345
INLINE + APCYC4 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	208	6119	6683	182345
INLINE + APCYC6 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	210	6617	8061	894985
INLINE + APCYC8 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	208	6361	6897	182345
INLINE + APCYC10 + UNROLL_SUB_LOOPS (5X)	9.75ns	0	210	6855	8211	894985
INLINE + APCYC12 + UNROLL_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APCYC13 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	236	10998	22897	X
INLINE + APCYC14 + UNROLL_SUB_LOOPS (5X)	10.14ns	0	228	10099	18156	776386
INLINE + APCYC15 + UNROLL_SUB_LOOPS (5X)	10.14ns	0	228	10099	19866	776386
INLINE + APCYC16 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	208	7476	22239	73507
INLINE + APCYC32 + UNROLL_SUB_LOOPS (5X)	9.07ns	0	208	7476	29529	73507
INLINE + APC + UNROLL_SUB_LOOPS (5X)	9.75ns	0	208	7908	5874	73009
#INLINE APPLIED TO CFUNC + #ARRAY_PARTITION APPLIED TO FOR/SUB LOOPS + #PIPELINE APPLIED TO FOR/SUB LOOPS (5,X)						
INLINE + APB2 + PIPELINE_SUB_LOOPS (5)	9.07ns	4	32	1774	1928	109170
INLINE + APB4 + PIPELINE_SUB_LOOPS (5)	9.89ns	4	36	2410	3030	X
INLINE + APB6 + PIPELINE_SUB_LOOPS (5)	9.89ns	6	33	3109	4167	X
INLINE + APB8 + PIPELINE_SUB_LOOPS (5)	9.89ns	8	35	3711	5193	X
INLINE + APB10 + PIPELINE_SUB_LOOPS (5)	10.22ns	8	37	3575	5231	115661
INLINE + APB12 + PIPELINE_SUB_LOOPS (5)	10.22ns	10	38	2919	4606	X
INLINE + APB13 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	13754	86800	50406
INLINE + APB14 + PIPELINE_SUB_LOOPS (5)	X	X	X	X	X	X
INLINE + APB15 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	13754	54104	50406
INLINE + APB16 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	13754	54104	50406
INLINE + APB32 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	11948	41592	50157
INLINE + APCYC2 + PIPELINE_SUB_LOOPS (5)	9.13ns	4	32	1319	1595	102198
INLINE + APCYC4 + PIPELINE_SUB_LOOPS (5)	9.89ns	8	32	1334	2360	102198
INLINE + APCYC6 + PIPELINE_SUB_LOOPS (5)	9.89ns	12	48	3264	5186	X
INLINE + APCYC8 + PIPELINE_SUB_LOOPS (5)	9.89ns	16	32	1406	5284	102198
INLINE + APCYC10 + PIPELINE_SUB_LOOPS (5)	10.9ns	20	44	2908	6189	X
INLINE + APCYC12 + PIPELINE_SUB_LOOPS (5)	9.89ns	8	44	13731	24923	X
INLINE + APCYC13 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	38	11657	38514	X
INLINE + APCYC14 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	44	15001	48612	63370
INLINE + APCYC15 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	44	11950	23269	X
INLINE + APCYC16 + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	12332	40908	49161

INLINE + APCYC32 + PIPELINE_SUB_LOOPS (5)	X	X	X	X	X	X
INLINE + APC + PIPELINE_SUB_LOOPS (5)	9.07ns	0	32	11102	19312	49161
INLINE + APB2 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	2421	2530	1243134
INLINE + APB4 + PIPELINE_SUB_LOOPS (5X)	9.89ns	0	36	2200	3030	X
INLINE + APB6 + PIPELINE_SUB_LOOPS (5X)	9.17ns	0	33	3050	3321	X
INLINE + APB8 + PIPELINE_SUB_LOOPS (5X)	9.85ns	0	33	2785	2994	X
INLINE + APB10 + PIPELINE_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB12 + PIPELINE_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB13 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	2714	1018	95474
INLINE + APB14 + PIPELINE_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB15 + PIPELINE_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APB16 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	2714	1018	95474
INLINE + APB32 + PIPELINE_SUB_LOOPS (5X)	X	X	X	X	X	X
INLINE + APCYC2 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	1663	1173	194826
INLINE + APCYC4 + PIPELINE_SUB_LOOPS (5X)	9.89ns	0	32	1721	2400	140792
INLINE + APCYC6 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	38	2861	3004	1175405
INLINE + APCYC8 + PIPELINE_SUB_LOOPS (5X)	9.17ns	0	32	2043	1644	194826
INLINE + APCYC10 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	34	2865	3010	1175405
INLINE + APCYC12 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	34	3552	2939	1243134
INLINE + APCYC13 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	37	3737	2197	826411
INLINE + APCYC14 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	37	3556	2757	826411
INLINE + APCYC15 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	37	3556	3013	826411
INLINE + APCYC16 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	2714	1858	95474
INLINE + APCYC32 + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	2853	4638	95474
INLINE + APC + PIPELINE_SUB_LOOPS (5X)	9.07ns	0	32	4296	10156	95474
#LOOP_MERGE RETRY (5)						
ALL SOURCE CODE WITHIN THE FOR/SUB LOOPS TO ENABLE #LOOP_MERGE	8.59ns	2	8	792	969	130432
ALL SOURCE CODE WITHIN THE FOR/SUB LOOPS TO ENABLE #LOOP_MERGE + FORPIPELINE	8.59ns	3	8	900	1093	95500
BALANCED PERFORMANCE DESIGN EVALUATIONS, #UNROLL, #PIPELINE, #INLINE (5)						
UNROLL: COMPLETE UNROLL. UNROLL xZ: UNROLL WITH A FACTOR Z (PARTIAL UNROLL). THE CODENAMES WITHIN THE BRACKETS {} REFER TO CERTAIN FOR/SUB LOOPS. THE KEYWORD "REST" REFERS TO THE REST OF THE FOR/SUB LOOPS NOT MENTIONED WITHIN THE						

BRACKETS {}.IN THE CASE WHERE SUCH BRACKETS DO NOT EXIST, IT IS IMPLIED THAT THE DIRECTIVE UNDER CONSIDERATION IS APPLIED TO ALL FOR/SUB LOOPS. #INLINE APPLIES TO CFUNC						
FORPIPELINE + UNROLL {CSWAB}	9.52ns	3	8	622	899	75555
FORPIPELINE + UNROLL x2 {CSWAB}	9.30ns	2	8	557	923	88005
FORPIPELINE + UNROLL x4 {CSWAB}	9.07ns	2	8	662	1002	85017
FORPIPELINE + UNROLL x8 {CSWAB}	9.52ns	2	8	1127	1784	88503
FORPIPELINE + UNROLL {CSWAB UPDAB}	9.07ns	2	96	3671	4309	50157
FORPIPELINE + UNROLL x2 {CSWAB UPDAB}	9.30ns	2	8	685	1069	93981
FORPIPELINE + UNROLL x4 {CSWAB UPDAB}	9.07ns	2	8	928	1365	88005
FORPIPELINE + UNROLL x8 {CSWAB UPDAB}	9.52ns	2	8	1661	2527	89499
FORPIPELINE + UNROLL {CSWAB} + INLINE	9.81ns	3	208	4832	3876	56631
FORPIPELINE + UNROLL x2 {CSWAB} + INLINE	9.81ns	2	48	1559	1340	71571
FORPIPELINE + UNROLL x4 {CSWAB} + INLINE	9.81ns	2	80	2427	1943	65595
FORPIPELINE + UNROLL x8 {CSWAB} + INLINE	9.81ns	2	144	4417	3779	75555
FORPIPELINE + UNROLL {CSWAB UPDAB} + INLINE	9.07ns	2	208	5641	4997	48663
FORPIPELINE + UNROLL x2 {CSWAB UPDAB} + INLINE	9.81ns	2	48	1687	1492	77547
FORPIPELINE + UNROLL x4 {CSWAB UPDAB} + INLINE	9.81ns	2	80	2693	2306	68583
FORPIPELINE + UNROLL x8 {CSWAB UPDAB} + INLINE	9.81ns	2	144	4951	4522	76551
FORPIPELINE {CSWAB} + UNROLL REST	8.59ns	2	8	6684	6059	48171
FORPIPELINE {CSWAB} + UNROLL x2 REST	8.59ns	2	8	723	997	77296
FORPIPELINE {CSWAB} + UNROLL x4 REST	8.59ns	2	8	978	1501	76523
FORPIPELINE {CSWAB} + UNROLL x8 REST	8.59ns	2	8	1548	2513	70782
FORPIPELINE {CSWAB} + UNROLL x16 REST	8.59ns	2	8	1568	3127	66294
FORPIPELINE {CSWAB} + UNROLL x32 REST	8.59ns	2	8	2421	5567	65294
FORPIPELINE {CSWAB UPDAB} + UNROLL REST	8.59ns	2	8	6656	5911	52590
FORPIPELINE {CSWAB UPDAB} + UNROLL x2 REST	8.59ns	2	8	595	849	71320
FORPIPELINE {CSWAB UPDAB} + UNROLL x4 REST	8.59ns	2	8	712	1145	73535
FORPIPELINE {CSWAB UPDAB} + UNROLL x8 REST	8.59ns	2	8	1014	1781	69786
FORPIPELINE {CSWAB UPDAB} + UNROLL x16 REST	8.59ns	2	8	1436	2861	67290
FORPIPELINE {CSWAB UPDAB} + UNROLL x32 REST	8.59ns	2	8	2289	5333	66290
FORPIPELINE {CSWAB} + UNROLL REST + INLINE	9.81ns	2	32	7473	6011	46677
FORPIPELINE {CSWAB} + UNROLL REST x2 + INLINE	9.81ns	2	32	1345	1231	75304
FORPIPELINE {CSWAB} + UNROLL REST x4 + INLINE	9.81ns	2	32	1600	1719	74531
FORPIPELINE {CSWAB} + UNROLL REST x8 + INLINE	9.81ns	2	32	2170	2752	68970
FORPIPELINE {CSWAB} + UNROLL REST x16 + INLINE	9.81ns	2	32	2160	3297	64302
FORPIPELINE {CSWAB} + UNROLL REST x32 + INLINE	9.81ns	2	32	3013	5737	63302

FORPIPELINE {CSWAB UPDAB} + UNROLL REST + INLINE	9.81ns	2	32	7387	6831	51096
FORPIPELINE {CSWAB UPDAB} + UNROLL REST x2 + INLINE	9.81ns	2	32	1217	1089	69328
FORPIPELINE {CSWAB UPDAB} + UNROLL REST x4 + INLINE	9.81ns	2	32	1334	1385	71543
FORPIPELINE {CSWAB UPDAB} + UNROLL REST x8 + INLINE	9.81ns	2	32	1636	1999	67794
FORPIPELINE {CSWAB UPDAB} + UNROLL REST x16 + INLINE	9.81ns	2	32	2058	3100	65298
FORPIPELINE {CSWAB UPDAB} + UNROLL REST + INLINE	9.81ns	2	32	2911	5571	64298
UNROLL ALL x2	9.3ns	2	8	737	1136	99706
UNROLL ALL x4	9.07ns	2	8	1097	1763	95945
UNROLL ALL x8	9.52ns	2	8	2132	3562	93690
UNROLL ALL x16	9.07ns	2	96	4564	6443	51582
UNROLL ALL x32	9.07ns	2	96	5417	8883	50852
UNROLL ALL x2 + INLINE	9.81ns	2	48	1739	1559	83272
UNROLL ALL x4 + INLINE	9.81ns	2	80	4862	2704	76523
UNROLL ALL x8 + INLINE	9.81ns	2	144	5422	5557	80742
UNROLL ALL x16 + INLINE	9.75ns	2	208	65529	6521	50358
UNROLL ALL x32 + INLINE	9.07ns	2	208	7388	8961	49358
PARTIAL SUBFUNCTION MERGING SOURCE VERSION EVALUATION (6)						
PIPELINE FOR/SUB LOOPS	8.59ns	2	8	476	723	72069
PIPELINE FOR/SUB LOOPS + APC	9.71ns	0	8	11364	15120	37956
PIPELINE FOR/SUB LOOPS + APC + INLINE	20.43ns	0	32	8192	15463	31482
UNROLL ALL SUB/FOR LOOPS	9.07ns	0	88	7094	6715	4782
INLINE + UNROLL ALL SUB/FOR LOOPS	9.75ns	0	208	9250	7334	2043
FINAL DESIGN VERSIONS						
LOW PERFORMANCE/LOW UTILIZATION [LP]						
PIPELINE FOR/SUB LOOPS (5)	8.59ns	2	8	543	764	65595
HIGHER PERFORMANCE/MEDIUM UTILIZATION [HPA]						
UNROLL ALL SUB/FOR LOOPS + #RESOURCE= AddSub @ CFUNC ADDITIONS (6)	8.22ns	0	56	5806	7099	3786
HIGHEST PERFORMANCE/HIGHEST UTILIZATION [HPB]						
UNROLL ALL SUB/FOR LOOPS + INLINE + #RESOURCE= AddSub @ CFUNC ADDITIONS (6)	8.74ns	0	208	8383	12728	1296

BIBLIOGRAPHY

[1] Kleinberg, Jon, and Eva Tardos. Algorithm design. Pearson Education India, 2006.

[2] Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

[3] Applegate, David L., et al. The traveling salesman problem: a computational study. Princeton university press, 2011.

[4] Edmonds, Jack. "Paths, trees, and flowers." Canadian Journal of mathematics 17.3 (1965): 449-467.

[5] M. R. GAREY, AND D. S. JOHNSON, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, San Francisco, 1979.

[6] Gutin, Gregory, and Abraham P. Punnen, eds. The traveling salesman problem and its variations. Vol. 12. Springer Science & Business Media, 2006.

[7] Jan Van Leeuwen, Handbook of Theoretical Computer Science, Vol.A: Algorithms and Complexity, MIT Press Cambridge MA USA, 1990

[8] Bernhard Korte, Jens Vygen Combinatorial Optimization: Theory and Algorithms, Fourth Edition, Springe-Verlag Berling Heidelberg, 2008

[9] Papadimitriou, Christos H., and Kenneth Steiglitz. Combinatorial optimization: algorithms and complexity. Courier Corporation, 1982.

[10] Avinash K. Dixit, Department of Economics, Princeton University, "Lecture on Optimization" within the scope of "ECO305-Microeconomic Theory: A mathematical Approach"

[11] John K. Hunter, Department of Mathematics, University of California, Davis, "Lecture on Continuous Functions" within the scope of "MAT201A-Introduction to Analysis"

[12] Xilinx(R): What is an FPGA?, http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm

[13] Hauck, Scott. "The roles of FPGAs in reprogrammable systems." Proceedings of the IEEE 86.4 (1998): 615-638.

[14] Compton, Katherine, and Scott Hauck. "Reconfigurable computing: a survey of systems and software." ACM Computing Surveys (csuR) 34.2 (2002): 171-210.

[15] Tessier, Russell, Kenneth Pocek, and Andre DeHon. "Reconfigurable computing architectures." Proceedings of the IEEE 103.3 (2015): 332-354.

[16] FPGA-based Accelerators: The Convey Systems, http://www.hpcresearch.nl/euroben/Overview/web12/convey.php

[17] Maxeler Technologies, http://maxeler.com/#/

[18] Dollas, Apostolos. "Big Data Processing with FPGA Supercomputers: Opportunities and Challenges." 2014 IEEE Computer Society Annual Symposium on VLSI. IEEE, 2014.

[19] Chrysos, Grigorios, et al. "Reconfiguring the Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics Acceleration Platforms." IEEE Design & Test 31.1 (2014): 62-73.

[20] Chrysos, Grigorios, Apostolos Dollas, and Nikolaos Bourbakis. "An embedded software-reconfigurable color segmentation architecture for image processing systems." Microprocessors and Microsystems 36.3 (2012): 215-231.

[21] Papaefstathiou, Ioannis, et al. "ROTA: An Archipelago-Wide Area Network for High Speed Communication to Ships." Informatics (PCI), 2012 16th Panhellenic Conference on. IEEE, 2012.

[22] Sourdis, Ioannis, and Dionisios Pnevmatikatos. "Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system." International Conference on Field Programmable Logic and Applications. Springer Berlin Heidelberg, 2003.

[23] Kritikakis, Charalabos, et al. "An FPGA-based high-throughput stream join architecture." Field Programmable Logic and Applications (FPL), 2016 26th International Conference on. EPFL, 2016.

[24] Cong, Jason, et al. "High-level synthesis for FPGAs: From prototyping to deployment." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30.4 (2011): 473-491.

[25] LegUp High-Level Synthesis, University of Toronto, http://legup.eecg.utoronto.ca/

[26] Bambu: A Free Framework for the High-Level Synthesis of Complex Applications, http://panda.dei.polimi.it/?page_id=31

[27] Xilinx Vivado High-Level Synthesis, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[28] OpenCL: The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/
Munshi, Aaftab. "The opencl specification." 2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, 2009.

[29] Mavroidis, Ioannis, Ioannis Papaefstathiou, and Dionisios Pnevmatikatos. "A fast FPGA-based 2-Opt solver for small-scale euclidean traveling salesman problem." 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007). IEEE, 2007.

[30] Skiena, S. "Hamiltonian Cycles." §5.3.4 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 196-198, 1990.

[31] Schrijver, Alexander. "On the history of combinatorial optimization (till 1960)." Handbooks in operations research and management science 12 (2005): 1-68.

[32] Laporte, Gilbert. "A concise guide to the traveling salesman problem." Journal of the Operational Research Society 61.1 (2010): 35-40.

[33] Papadimitriou, Christos H. "The Euclidean travelling salesman problem is NP-complete." Theoretical Computer Science 4.3 (1977): 237-244.

[34] Englert, Matthias, Heiko Röglin, and Berthold Vöcking. "Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP." Algorithmica 68.1 (2014): 190-264.

[35] Johnson, David S. "Local optimization and the traveling salesman problem." International Colloquium on Automata, Languages, and Programming. Springer Berlin Heidelberg, 1990.

[36] Johnson, David S., and Lyle A. McGeoch. "The traveling salesman problem: A case study in local optimization." Local search in combinatorial optimization 1 (1997): 215-310.

[37] Hahsler, Michael, and Kurt Hornik. "TSP-Infrastructure for the traveling salesperson problem." Journal of Statistical Software 23.2 (2007): 1-21.

[38] Nilsson, Christian. Heuristics for the traveling salesman problem. Tech. Report, Linköping University, Sweden, 2003.

[39] Reinelt, Gerhard. The traveling salesman: computational solutions for TSP applications. Springer-Verlag, 1994.

[40] Antosiewicz, Marek, Grzegorz Koloch, and Bogumił Kamiński. "Choice of best possible metaheuristic algorithm for the travelling salesman problem with limited computational time: quality, uncertainty and speed." Journal of Theoretical and Applied Computer Science 7.1 (2013): 46-55.

[41] Lourenço, Helena R., Olivier C. Martin, and Thomas Stützle. "Iterated local search: Framework and applications." Handbook of Metaheuristics. Springer US, 2010. 363-397.

[42] Verhoeven, M. G. A., Emile HL Aarts, and P. C. J. Swinkels. "A parallel 2-opt algorithm for the traveling salesman problem." Future Generation Computer Systems 11.2 (1995): 175-182.

[43] Delisle, Pierre, et al. "Parallel implementation of an ant colony optimization metaheuristic with OpenMP." Proceedings of the 3rd European Workshop on OpenMP (EWOMP'01), Barcelona, Spain. 2001.

[44] Rudeanu, Laurentiu, and Mitica Craus. "Parallel implementation of ant colony optimization for travelling salesman problem." WSEASs Transactions on Systems 3.3 (2004): 1161-1166.

[45] Borovska, Plamenka. "Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster." Int. Conf. on Computer Systems and Technologies. 2006.

[46] Lazarova, Milena, and Plamenka Borovska. "Comparison of parallel metaheuristics for solving the TSP." Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing. ACM, 2008.

[47] Skliarova, Iouliia, and António B. Ferrari. "FPGA-based implementation of genetic algorithm for the traveling salesman problem and its industrial application." International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer Berlin Heidelberg, 2002.

[48] Vega-Rodriguez, Miguel A., et al. "Genetic algorithms using parallelism and FPGAs: the TSP as case study." 2005 International Conference on Parallel Processing Workshops (ICPPW'05). IEEE, 2005.

[49] Rocki, Kamil, and Reiji Suda. "High Performance GPU Accelerated TSP Solver." High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: IEEE, 2012.

[50] Rocki, Kamil, and Reiji Suda. "Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem." High Performance Computing and Simulation (HPCS), 2012 International Conference on. IEEE, 2012.

[51] O'neil, Molly A., Dan Tamir, and Martin Burtscher. "A parallel gpu version of the traveling salesman problem." 2011 International Conference on Parallel and Distributed Processing Techniques and Applications. 2011.

[52] O'Neil, Molly A., and Martin Burtscher. "Rethinking the parallelization of random-restart hill climbing: a case study in optimizing a 2-opt TSP solver for GPU execution." Proceedings of the 8th Workshop on General Purpose Processing using GPUs. ACM, 2015.

[53] Rocki, Kamil, and Reiji Suda. "An efficient GPU implementation of a multi-start TSP solver for large problem instances." Proceedings of the 14th annual conference companion on Genetic and evolutionary computation. ACM, 2012.

[54] Rocki, Kamil, and Reiji Suda. "High performance GPU accelerated local optimization in TSP." Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. IEEE, 2013.

[55] Delévacq, Audrey, Pierre Delisle, and Michaël Krajecki. "Parallel GPU implementation of iterated local search for the travelling salesman problem." Learning and Intelligent Optimization. Springer Berlin Heidelberg, 2012. 372-377.

[56] Fujimoto, Noriyuki, and Shigeyoshi Tsutsui. "A highly-parallel TSP solver for a GPU computing platform." International Conference on Numerical Methods and Applications. Springer Berlin Heidelberg, 2010.

[57] Chen, Su, et al. "CUDA-based genetic algorithm on traveling salesman problem." Computer and Information Science 2011. Springer Berlin Heidelberg, 2011. 241-252.

[58] Cecilia, José M., et al. "Parallelization strategies for ant colony optimisation on GPUs." Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE, 2011.

[59] Uchida, Akihiro, Yasuaki Ito, and Koji Nakano. "An efficient GPU implementation of ant colony optimization for the traveling salesman problem." Networking and Computing (ICNC), 2012 Third International Conference on. IEEE, 2012.

[60] Schulz, Christian, et al. "GPU computing in discrete optimization. Part II: Survey focused on routing problems." EURO journal on transportation and logistics 2.1-2 (2013): 159-186.

[61] The Concorde TSP Solver, University of Waterloo's Faculty of Mathematics http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm

[62] Applegate, David, et al. "Concorde TSP solver." (2006).

[63] Applegate, David L., et al. "Certification of an optimal TSP tour through 85,900 cities." Operations Research Letters 37.1 (2009): 11-15.

[64] Croes, Georges A. "A method for solving traveling-salesman problems." Operations research 6.6 (1958): 791-812.

[65] Flood, Merrill M. "The traveling-salesman problem." Operations Research 4.1 (1956): 61-75.

[66] Chandra, Barun, Howard Karloff, and Craig Tovey. "New results on the old k-opt algorithm for the traveling salesman problem." SIAM Journal on Computing 28.6 (1999): 1998-2029.

[67] Reinelt, Gerhard. "TSPLIB—A traveling salesman problem library." ORSA journal on computing 3.4 (1991): 376-384.

[68] TSPLIB, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[69] Xilinx Vivado Design Suite User Guide: High Level Synthesis (UG902) http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf

[70] Xilinx, Introduction to FPGA Design with Vivado High-Level Synthesis (UG998) https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf