ECE 8893 Final Project Report - Accelerating Ant Colony Optimization with Local Search for the Travelling Salesman Problem

Prathamesh Khare
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
pkhare8@gatech.edu

Ananth Kumar
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
ananth.kumar@gatech.edu

Maanas Purushothapu
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
mpurushothapu3@gatech.edu

Nishant Sharma
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
nsharma93@gatech.edu

I. Introduction

The Traveling Salesman Problem (TSP) is an NP-hard route finding problem; it involves finding the best route that visits a set of nodes such that the route visits each node once and only once. As it is an NP-hard problem, it cannot be solved by algorithms running parallel time (i.e. a route that is guarananteed to be the shortest route cannot be calculated in polynomial time). Metahueristics are general purpose techniques to solve the TSP and other types of NP-hard combination problems, and a well-known metaheuristic to get good solutions for TSP is Ant Colony Optimization.

Ant Colony Optimization (ACO) is a swarm-intelligence-based metaheuristic designed to solve problems that can be simplified to a graph problem, specifically that of finding the best path through a graph. ACO is based on the collaborative nature of ants to solve difficult problems, specifically their ability to find the shortest paths to food. The ants start by traveling randomly to the food source, but by depositing pheromones, which evaporate after some time, throughout their path, they are able to collectively find the shortest path to the food source, and use it. An implementation of ACO that can solve large-scale TSP instances is Focused ACO (FACO) [1].

An integral part of current ACO algorithms, including FACO, is local search, which optimizes ant routes between iterations by switching edges in the route. Local search algorithms considers pairs or triplets of edges, and determines it switching them would result in a shorter path. The highly comparitive nature of local search, as well as the potentially large search space (especially when considering more than two edges at to switch) makes it a good target for FPGA acceleration, and this was the focus of our work.

II. Background and Research

Using ACO to find solutions to the Travelling Salesman Problem (TSP) is a relatively old problem; as early as 2006, it was attempted to parallelize ACO to solve the TSP in a multicore computer, specifically by studying the impact of communication on parallelized versions of an ACO-based algorithm called the *Max-Min* Any System.

ACO has been modified to efficiently solve large instance TSP problems. This version of ACO is introduced in [1], and is called Focused-ACO (FACO). FACO mirrors the constructive behavior of regular ACO until it selects a predefined number of edges not present in the previous solution. The speed increase of this algorithm comes from the search restriction on the solution space. FACO performed well on a regular 8-core CPU against several other recent ACO approaches and is able to come within 1% of the best-known solutions for predefined TSP problems. The performance of FACO on a CPU is promising when considering the potential speedup on an FPGA.

A significant part of FACO that helps with the speed is local search. FACO uses 2-opt local search to improve the route of each ant after the ant has selected a route but before the best route is selected, as can be seen in line 30 of the FACO pseudocode in Figure 1. Rather than consider all possible pairs of edges in the route, this version of 2-opt local search has been modified to consider only edges connected to the nearest neighbors of new nodes in order to decrease time complexity.

III. Motivation

A. Problem With Parelellization

The Min-Max ACO and the FACO algorithm for the traveling salesman problem as it is implemented are not particularly good candidates for FPGA implementation. This is because of the following reasons:

- The pheromone matrix is a large data structure with the pheromone information, and it needs to be accessed often by each ant
- The communication overhead between the FPGA and CPU is high in the ACO implementation (a lot of data transfer is required)
- The tasks done by the ants are difficult to pipeline (a lot of dependencies exist preventing parallelizing certain tasks), so the kind of acceleration done before is mostly running the ants with a higher clock speed, hence using GPUs instead of FPGAs

B. Research Results and Solution

A literature review was conducted to look for different implementations of the ACO algorithm, and how others have modified the ACO algorithm in order to implement it on an FPGA. One of the papers introduces a population-based ant colony optimization (P-ACO) algorithm [2]. In the P-ACO algorithm the pheromone information is ”replaced by a small set (population) of good solutions discovered during the preceding iterations” [2]. So instead of doing pheromone updates and pheromone evaporation, there is now inserting a good solution in the population and replacing the oldest solution in the population. Another paper authored by the same people proposes a counter-based ant colony optimization (C-ACO) algorithm [3]. which even allows to systolically pipe a sequence of artificial ants through a grid of processing cells, which is very promising for an FPGA implementation. It was clear to us that the only way to successfully implement ACO on an FPGA and accelerate it is to make changes to the algorithm itself, similar to what the papers mentioned above have done.

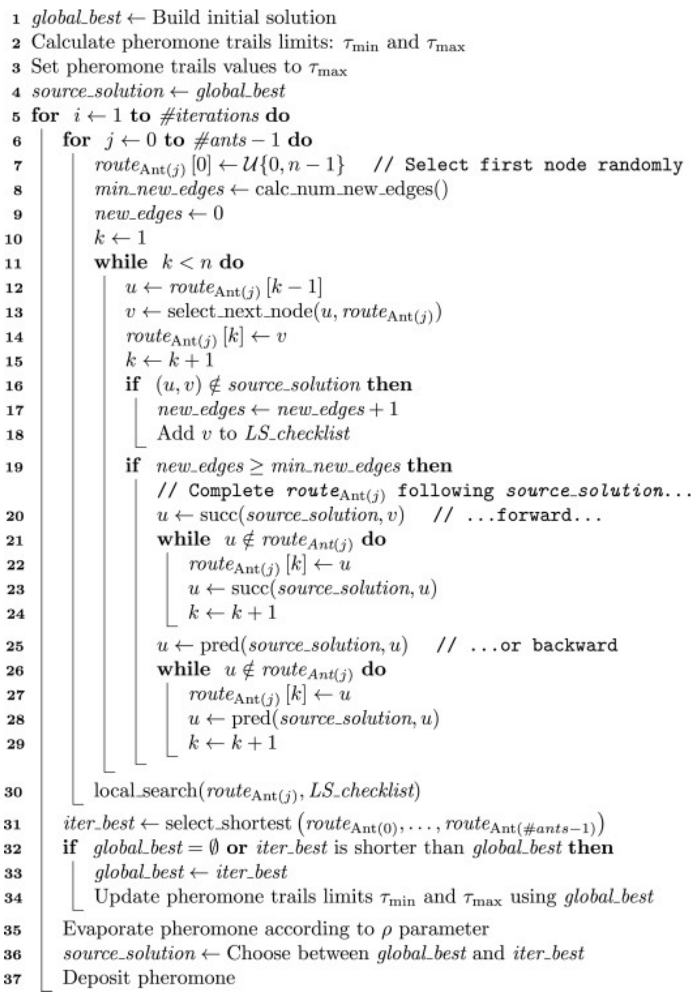
As mentioned previously two-opt Local search is another optimization algorithm done within ACO, to get a better route for each ant. So after each ant gets a route, the route is passed through the local search algorithm (two-opt-nn) which iteratively compares two edges and switches them if the distance of the route (compared to the nodes' nearest neighbors) is reduced. Local search is an optional part of the algorithm for the ACO implementation; however, it is highly recommended and most implementations of ACO have local search in one form or another. Since FACO prunes the search space, it performs significantly faster than the previous Min-Max implementation, however, local search doesn't have any major changes. The underlying algorithm of two-opt-nn is still the same, and therefore local search becomes a significant bottle-neck for FACO, making it a prime candidate for acceleration. We were able to come to this conclusion by benchmarking our code, and timing the local search algorithm and getting the ratio of it compared to the total time of the solution. As shown in table I, the total time of the algorithm is approximately 20 s, and out of that 9.6 seconds is spent on local search. Local search approximately takes up 48% of the time per ant per iteration. The ratio for Min-Max is around

TABLE I
THE PROPORTION OF TIME LOCAL SEARCH TAKES TO THE TOTAL TIME AN ANT TAKES PER ITERATION OF FACO.

Local Search Time	Total Time	Ratio
9.66177s	20.869 s	48.14%

15% on average, so the local search is a more crucial module in FACO than the Min-Max ACO implementation. Hence, a good starting point for ACO optimization could be to try and accelerate local search on the FPGA.

The FACO algorithm pseudocode is shown in Figure 1. The for loop shown in Line 6 loops over all the ants that each compute their own route by adding nodes to them and finally run the local search as shown in Line 30. So the local search is run for each iteration and for each ant within each iteration.



```
1  $global\_best \leftarrow$ Build initial solution
2  Calculate pheromone trails limits: $\tau_{\min}$ and $\tau_{\max}$
3  Set pheromone trails values to $\tau_{\max}$
4  $source\_solution \leftarrow global\_best$
5  for $i \leftarrow 1$ to $\#iterations$ do
6      for $j \leftarrow 0$ to $\#ants - 1$ do
7          $route_{\mathrm{Ant}(j)}[0] \leftarrow \mathcal{U}\{0, n-1\}$   // Select first node randomly
8          $min\_new\_edges \leftarrow$ calc_num_new_edges()
9          $new\_edges \leftarrow 0$
10         $k \leftarrow 1$
11         while $k < n$ do
12             $u \leftarrow route_{\mathrm{Ant}(j)}[k-1]$
13             $v \leftarrow$ select_next_node$(u, route_{\mathrm{Ant}(j)})$
14             $route_{\mathrm{Ant}(j)}[k] \leftarrow v$
15             $k \leftarrow k + 1$
16             if $(u, v) \notin source\_solution$ then
17                 $new\_edges \leftarrow new\_edges + 1$
18                 Add $v$ to $LS\_checklist$
19             if $new\_edges \geq min\_new\_edges$ then
                   // Complete $route_{\mathrm{Ant}(j)}$ following $source\_solution$...
20                 $u \leftarrow$ succ$(source\_solution, v)$   // ...forward...
21                 while $u \notin route_{Ant(j)}$ do
22                     $route_{\mathrm{Ant}(j)}[k] \leftarrow u$
23                     $u \leftarrow$ succ$(source\_solution, u)$
24                     $k \leftarrow k + 1$
25                 $u \leftarrow$ pred$(source\_solution, u)$   // ...or backward
26                 while $u \notin route_{Ant(j)}$ do
27                     $route_{\mathrm{Ant}(j)}[k] \leftarrow u$
28                     $u \leftarrow$ pred$(source\_solution, u)$
29                     $k \leftarrow k + 1$
30         local_search$(route_{\mathrm{Ant}(j)}, LS\_checklist)$
31     $iter\_best \leftarrow$ select_shortest $(route_{\mathrm{Ant}(0)}, \ldots, route_{\mathrm{Ant}(\#ants-1)})$
32     if $global\_best = \emptyset$ or $iter\_best$ is shorter than $global\_best$ then
33         $global\_best \leftarrow iter\_best$
34         Update pheromone trails limits $\tau_{\min}$ and $\tau_{\max}$ using $global\_best$
35     Evaporate pheromone according to $\rho$ parameter
36     $source\_solution \leftarrow$ Choose between $global\_best$ and $iter\_best$
37     Deposit pheromone
```

Fig. 1. Pseudocode for the Focused ACO algorithm proposed in [1]

IV. CURRENT PROGRESS

A. Overview

Initially, our main target of parallelization was the ants themselves. Each ant is relatively independent of other ants within an iteration, and the ants are a good target for parallelization. The ants have been parallelized before in other papers and there has also been a multi-colony ACO algorithm where each processor maintains a small colony of ants [2]. The primary difference is that most of the parallelization implementations in ACO have been done using GPUs instead of FPGAs. The problem is that the ACO algorithm in its original form will not result in an efficient FPGA hardware implementation. The main issues are the pheromone matrix storage, floating point computations, high communication overhead between FPGA and CPU, as well as randomly selecting a series of points for each ant at each iteration.

We have had to pivot from our original approach and try to pinpoint an aspect of the ACO algorithm that could benefit from an FPGA implementation, or come up with new techniques to modify the ACO algorithm while preserving its effectiveness to get an efficient FPGA implementation.

As such, we shifted to first focus on accelerating local search, while keeping the rest of the FACO code running on a CPU. Local search occurs between ant route iterations and optimizes each route; effective implementations of local search are able to significantly reduce convergence time. Local search involves comparing group of edges in the existing ant route and evaluating if some other order of edges in that route generates a shorter path. Common modes of local search include 2-opt, where a pair of edges are considered, and 3-opt, where a triplet of edges is considered. 3-opt produces far better results, but the complexity, and so the runtime, is higher.

```
Input : $route[0, \ldots, n-1]$
Input : $LS\_checklist$ Starting nodes for the 2-opt moves
$changes \leftarrow 0$   // # of successful 2-opt moves (changes) applied
while $LS\_checklist \neq \emptyset$ and $changes < n$ do
    $a \leftarrow \text{pop}(LS\_checklist)$   // Remove first element (node)
    $a_{\text{succ}} \leftarrow \text{succ}(route, a)$
    $a_{\text{pred}} \leftarrow \text{pred}(route, a)$
    $NN\_list \leftarrow \text{nearest\_neighbors(a)}$
    $move \leftarrow \emptyset$   // The best 2-opt move for $NN\_list$
    $gain \leftarrow 0$   // Cost change for $move$
    foreach $node\ b \in NN\_list$ do
        $b_{\text{succ}} \leftarrow \text{succ}(route, b)$
        if $get\_distance(a, a_{\text{succ}}) > get\_distance(a, b)$ then
            $cost_{\text{old}} \leftarrow \text{get\_distance}(a, a_{\text{succ}}) + \text{get\_distance}(b, b_{\text{succ}})$
            $cost_{\text{new}} \leftarrow \text{get\_distance}(a, b) + \text{get\_distance}(a_{\text{succ}}, b_{\text{succ}})$
            if $cost_{\text{old}} - cost_{\text{new}} > gain$ then
                $gain \leftarrow cost_{\text{old}} - cost_{\text{new}}$
                $move \leftarrow (a, a_{\text{succ}}, b, b_{\text{succ}})$
    foreach $node\ b \in NN\_list$ do
        $b_{\text{pred}} \leftarrow \text{pred}(route, b)$
        if $get\_distance(a_{\text{pred}}, a) > get\_distance(a, b)$ then
            $cost_{\text{old}} \leftarrow \text{get\_distance}(a_{\text{pred}}, a) + \text{get\_distance}(b_{\text{pred}}, b)$
            $cost_{\text{new}} \leftarrow \text{get\_distance}(a, b) + \text{get\_distance}(a_{\text{pred}}, b_{\text{pred}})$
            if $cost_{\text{old}} - cost_{\text{new}} > gain$ then
                $gain \leftarrow cost_{\text{old}} - cost_{\text{new}}$
                $move \leftarrow (a_{\text{pred}}, a, b_{\text{pred}}, b)$
    if $move \neq \emptyset$ then
        $(w, x, y, z) \leftarrow move$
        Flip a section of $route$ between $x$ and $y$
        Append $w, x, y, z$ to $LS\_checklist$
        $changes \leftarrow changes + 1$
```

Fig. 2. Pseudocode for the 2-opt local search algorithm used in Focused ACO [1].

B. Testbench Creation

As mentioned previously, we ran a timing benchmark to get the time that local search takes, relative to the FACO. We found that 48% of the time was spent in the local search.

The next step was to analyze the inputs and outputs of the two-opt local search. Using the 2392-instance TSP problem in FACO, we noted that the inputs to two-opt were the route, checklist, and the problem instance object. The route is ant's route that needs to be optimized, which contains 2392 32-bit integers, The coordinates and nearest neighbor information are the only things needed from the problem instance object. The coordinates are a 2D array (2392, 2), with 32-bit integers. The checklist is a list acquired in the FACO using the nearest neighbor information, which contains the nodes two-opt will use to check for a better route. The checklist is capped at a capacity of 500 integers.

In order to isolate two-opt local search for further optimization, a simpler testbench was created by recording the inputs and outputs of two-opt local search for 20 iterations. The 2392-instance TSP problem was solved using the FACO, and the inputs and the new route resulting from the two-opt for the first 20 iterations were dumped into the text file. The text file contains all the coordinates, 20 neighbors for each node, the input route and output route of local search, and a checklist for 20 iterations.

We then designed a testbench by parsing the text file and using it to manually create inputs and pass them to two-opt LS. The total distance of the route before and after the local search was recorded. The distance metric was used to check functionality. The distance of the actual route from the text file was compared with the post-two-opt LS route distance. This comparison was essential to verify functionality while modifying two-opt LS to make it more FPGA friendly.

C. Implementation of FACO Local Search for FPGA

Pseudocode for the local search implementation used in FACO is shown in Figure 2. The pseudocode depicts the essential functionality of a 2-opt local search, where pairs of edges are evaluated and, if optimal, switched. This implementation makes a couple of modifications to improve runtime. Instead of evaluating all pairs of edges, only a certain subset of pairs based on the nearest neighbors of each node are evaluated; and the maximum possible number of swaps is fixed, such that the search will stop at that limit.

The existing C++ implementation in FACO for the two-opt local search is not very FPGA-friendly. It uses std::vectors with variable sizes and adds to it in the middle of the subroutine. Additionally, there are a few functions from the C++ algorithm class namely reverse and swap that aren't HLS compliant. Additionally, the implementations flip route subroutine and inputs use a custom object called problem instance which contains the coordinates and nearest neighbors for all the nodes as vectors.

Static arrays with pre-known sizes were used instead of vectors. The checklist of nodes was the vector with a variable size, that changed in the middle of two-opt. In order to tackle this a static array of 500 was used. The 500 capacity was set by running 20 iterations of two-opt-nn and noticing that the checklist size doesn't exceed 250.

Additionally, instead of using the problem instance custom object, the coordinates, and the nearest neighbors for all the nodes were added as arrays and passed to the two-opt function. The coordinates and nearest neighbor values were moved to BRAM only during the first call to the two-opt module, as they were static values.

Much of the code remained the same, including the function to flip a section of the route; this function utilizes a while loop to achieve this flip, which means that standard HLS synthesis cannot give an estimation of latency.

D. *Unoptimized Two-opt Local Search Synthesis and Timing Results*

The CPU latency of the code was measured by timing the two-opt LS call. The timings were different for different runs, but usually, the first iteration was between 0.15 - 0.25 ms, and the following iterations were between 0.01 - 0.04 ms on average. Figure 3 shows an example run with the latencies recorded. As expected the first iteration latency is around **0.209 ms**, and the following are significantly lesser. Adding up all the timings, the total time for 20 calls to the two-opt function is **0.713 ms**.

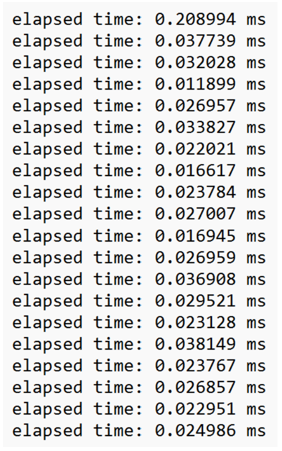

```
elapsed time: 0.208994 ms
elapsed time: 0.037739 ms
elapsed time: 0.032028 ms
elapsed time: 0.011899 ms
elapsed time: 0.026957 ms
elapsed time: 0.033827 ms
elapsed time: 0.022021 ms
elapsed time: 0.016617 ms
elapsed time: 0.023784 ms
elapsed time: 0.027007 ms
elapsed time: 0.016945 ms
elapsed time: 0.026959 ms
elapsed time: 0.036908 ms
elapsed time: 0.029521 ms
elapsed time: 0.023128 ms
elapsed time: 0.038149 ms
elapsed time: 0.023767 ms
elapsed time: 0.026857 ms
elapsed time: 0.022951 ms
elapsed time: 0.024986 ms
```

Fig. 3. CPU Timing latency for 20 2-opt Local search calls

The modified two-opt Local search code was synthesized using Vitis HLS to simulate and run on the FPGA board to get unoptimized latency values. The testbench includes the main file which parses through the inputs to two-opt for 20 iterations and calls the two-opt function. The csynth report from Vitis doesn't provide a latency report, however, it does show the unoptimized resource utilization, shown in figure 4.

The Cosim report gave us more information about the timing. Two cosim reports were run, with the first run for the first two-opt iteration, and the next run with 20 two-opt

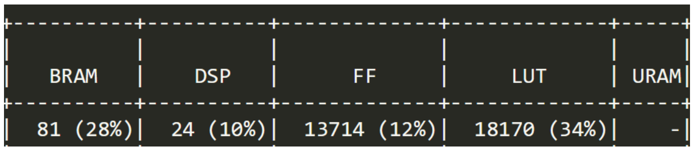

```
+----------+----------+------------+------------+-----+
|          |          |            |            |     |
|   BRAM   |   DSP    |     FF     |    LUT     | URAM|
+----------+----------+------------+------------+-----+
| 81 (28%)| 24 (10%)| 13714 (12%)| 18170 (34%)|    -|
```

Fig. 4. Vitis Csynth resource utilization report for the 2-opt module

iterations. The first iteration usually takes more time than the following iterations because the coordinates and the nearest neighbors array are copied to BRAM during the first iteration. The total execution time in clock cycles for one call of 2-opt LS is shown in figure 5, and the execution time for 20 calls is shown in figure 6. **121067 clock cycles** with a 100 MHz clock result in a one-call execution time of **1.21 ms**. **1460343 clock cycles** with a 100 MHz clock result in a twenty-call execution time of **14.6 ms**.

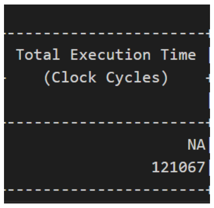

```
---------------------
 Total Execution Time
   (Clock Cycles)
---------------------
                   NA
               121067
---------------------
```

Fig. 5. Cosim latency showing total execution time for 1 call of 2-opt Local search

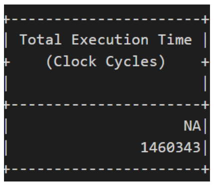
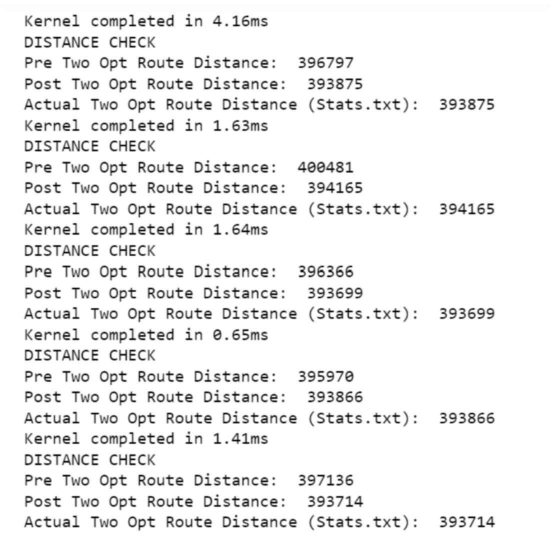

```
+--------------------+
| Total Execution Time |
+   (Clock Cycles)   +
|                    |
+--------------------+
|                  NA|
|             1460343|
+--------------------+
```

Fig. 6. Cosim latency showing total execution time for 20 calls of 2-opt Local search

Finally running the code on the Pynq board using a Jupyter notebook testbench similar to the C++ testbench created before, the onboard latencies are obtained. Figure 7 shows the Jupyter Notebook output with kernel timing and distance values. The kernel times do vary, but the result for this run shows the first call latency to be **4.16 ms**, and the following iterations are between 0.5 ms - 2 ms. Note that the output includes values for 20 iterations but only 5 are shown for brevity. The summed-up kernel execution time for 20 iterations is **30.54 ms**. Table II shows the latencies for the unoptimized two-opt LS algorithm on the CPU, Cosim, and FPGA for both a single function call 20 function calls.

TABLE II
THE LOCAL SEARCH TIMING RESULTS FOR CPU, COSIM, AND FPGA BOARD

Two-opt calls	CPU	Cosim	FPGA
1 LS call	0.209 ms	1.21 ms	4.16 ms
20 LS calls	0.713 ms	14.6 ms	30.54 ms



```
Kernel completed in 4.16ms
DISTANCE CHECK
Pre Two Opt Route Distance:  396797
Post Two Opt Route Distance:  393875
Actual Two Opt Route Distance (Stats.txt):  393875
Kernel completed in 1.63ms
DISTANCE CHECK
Pre Two Opt Route Distance:  400481
Post Two Opt Route Distance:  394165
Actual Two Opt Route Distance (Stats.txt):  394165
Kernel completed in 1.64ms
DISTANCE CHECK
Pre Two Opt Route Distance:  396366
Post Two Opt Route Distance:  393699
Actual Two Opt Route Distance (Stats.txt):  393699
Kernel completed in 0.65ms
DISTANCE CHECK
Pre Two Opt Route Distance:  395970
Post Two Opt Route Distance:  393866
Actual Two Opt Route Distance (Stats.txt):  393866
Kernel completed in 1.41ms
DISTANCE CHECK
Pre Two Opt Route Distance:  397136
Post Two Opt Route Distance:  393714
Actual Two Opt Route Distance (Stats.txt):  393714
```

Fig. 7. FPGA Onboard latency showing printing outputs including kernel timing and route distance to verify correctness for 5 calls

The jupyter notebook runs the unoptimized two-opt code and uses the respective .bit and .hwh file to do so. The kernel timings are noted to get the on-board performance of two-opt LS. The difference in latencies is understandable, the co-sim timing for 20 iterations of LS is around 14 ms, and the 20 iterations of LS on the FPGA is around 30 ms.

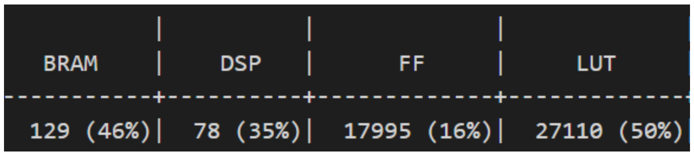
There is an increase in latency, and some of this is on-going research as to why. Possible reasons could be there maybe parts of memory access or writing operations that cause latency which is simply not captured in simulations. Requests to the memory controller for memory access may take more than one cycle and the CPU could be taking multiple cycles to send data to the FPGA.

E. HLS Optimized Two-opt Local Search Synthesis and Timing results

To be able to optimize local search (two-opt-nn) for execution on FPGA, various pragmas and optimization techniques were used. These included:

- **Function In-Lining**: rewriting, simplifying, and in-lining certain mathematical functions (e.g. max, min).
- **Loop Reformatting**: changing while loops and loops with variable loop size to typical for loops.
- **Loop Restructuring**: combining separate, redundant loops for more efficient pipelining. An example of this is combining the loops from the two-opt pseudo-code that individually considered the previous edge and the following edge.
- **Array Partitioning**: block, cyclic, and complete partitioning of coordinate, checklist, and route arrays. This allowed for loop unrolling and more parallelization! Array partitioning was also used alongside the *bind storage* pragma, enforcing certain arrays to be certain memory types (i.e. RAM with 1 write port and N read ports).
- **Loop Unrolling**: parallelizing execution of loop iterations that access the partitioned arrays. This is the most prevalent way to optimize programs for FPGA execution.
- **Loop Tiling**: used in hand with loop unrolling, this helps to unroll smaller loops (with fewer iteration counts) to allow for time-valid pipelining of these unrolled ”tiles.”
- **Pipelining**: last but not least, pipelining (executing via a pipeline) unrolled sections of code!

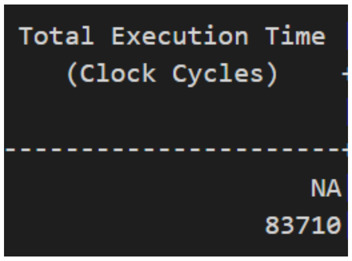
The *optimized* two-opt local search code was synthesized using Vitis HLS to simulate and run on the FPGA board to get optimized latency values. As before, the csynth report from Vitis doesn’t provide a latency report; however, it does show the optimized resource utilization, shown in Figure 8.



```
            |           |              |
   BRAM     |    DSP    |      FF      |      LUT
-----------+-----------+-------------+-------------
  129 (46%)|  78 (35%)|  17995 (16%)|  27110 (50%)
```

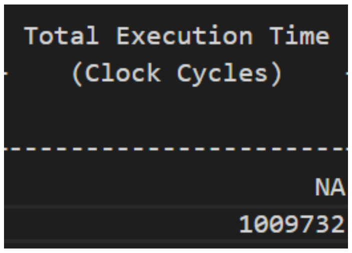
Fig. 8. Vitis Csynth OPTIMIZED resource utilization report for the 2-opt module

The co-simulation (cosim) report gave us more information about the timing. As before, two cosim reports were run, with the first run for the first two-opt iteration, and the next run with 20 two-opt iterations. The total execution time in clock cycles for one call of 2-opt LS is shown in figure 9, and the execution time for 20 calls is shown in figure 10. **83710 clock cycles** with a 100 MHz clock result in an *optimized* one-call execution time of **0.84 ms**. **1009732 clock cycles** with a 100 MHz clock result in an *optimized* twenty-call execution time of **10.1 ms**.



```
Total Execution Time
  (Clock Cycles)
----------------------
                    NA
                 83710
```

Fig. 9. Cosim latency showing total execution time for 1 call of OPTIMIZED 2-opt Local search



```
Total Execution Time
  (Clock Cycles)
----------------------
                    NA
               1009732
```

Fig. 10. Cosim latency showing total execution time for 20 calls of OPTIMIZED 2-opt Local search

Finally, running the code on the Pynq board using a Jupyter notebook testbench similar to the C++ testbench created before, the onboard latencies are obtained. As with the unoptimized results, although the kernel times do vary. Due to inabilities to secure spots in the queue to be able to run this optimized code on the physical Pynq boards, we do not have the exact latencies here for the optimized code run on-board. This is something that we intend to do immediately, before taking future steps towards implementing more optimizations.

If we were to estimate the on-board optimized latencies based on the unoptimized on-board latencies, the latency of one iteration would be something like **2.87 ms**, and the summed-up kernel execution time for 20 iterations would be around **21.12 ms**.

These times do show improvement over the unoptimized latencies. However, they are still much slower than CPU run-times achieved on the ECE server. This shows there is still plenty of room for further optimization and speed-up, given that not all resources on the FPGA have been maximally utilized either.

F. Other Experiments

1) 3-Opt Local Search: One extra experiment was to try to accelerate 3-opt local search, which considers triplets of edges. Theoretically, 3-opt local search should result in significantly better route improvements than 2-opt local search, albeit at the cost of increased time complexity.

The source code for FACO only provided a naive implementation of 3-opt local search, which considered every possible triplet of edges in the route to find a suitable exchange of edges. This implementation was isolated and rewritten to be synthesizable using techniques similar to that used to make 2-opt local search synthesizable, including function rewriting; in particular, the std::rotate function had to be rewritten. Testing the time taken to run 20 iterations of three-opt local search on the **ece-linlabsrv01** server resulted in an improvement of **15%** over 2-opt local search (i.e. the path length reduction was 35% more), but took **172 ms** per iteration (as compared to **.038 ms** per iteration for 2-opt). Attempting to optimize 2-opt local search by integrating nearest neighbor data in order to reduce the iteration latency resulted in a lower latency of **2.00 ms** per iteration, but at the cost of performance, with only a **3.2%** improvement over 2-opt.

2) Population ACO: An alternative to standard ACO is Population ACO (PACO), which, instead of keeping a pheromone matrix for each edge, instead keeps a population matrix of the best routes of the last k rounds; when selecting an edge to add to the path, an edge is given more weight if it exists in one or more of the routes within the population matrix. PACO is designed to simplify ACO by reducing the amount of data needed to select a new route; however, by solely weighing edges based on occurrences in the population matrix, heuristic information (based on the inverse of edge length) is lost. [2] developed an FPGA-based implementation of PACO, which inspired us to try to implement PACO as well to address route selection acceleration.

PACO was first implemented in C++ and evaluated against the same problems as FACO. However, these problems proved to be too large for PACO, as using a sufficient number of ants and iterations resulted in very long run times for PACO. Upon significantly reducing the problem size (from 3000 nodes to 200 nodes), PACO resulted in route lengths about 8x longer than initial FACO results, even after some parameter tuning. Adding heuristic information by evaluating each node's nearest neighbors (by using preexisting code also used in FACO) still did not result in good resultant route lengths, with resultant routes still about 2x longer than FACO results. As such, PACO was shelved to potentially improve upon and implement in the future.

G. Challenges and Future

Our current challenges include:

- Accelerating local search on an FPGA is proving to be a challenge, especially when considering the (currently) much lower latency on a CPU. This is likely due to the inherent variability of some loops in the FACO implementation of 2-opt local search and the current serialization for edge switching.
- Attempts at optimizing the latency of 3-opt local search resulted in worse results compared to unoptimized 3-opt local search, even by applying the same techniques used in the FACO-optimized version of 2-opt local search. As such, further research has to be done to optimize 3-opt local search to reduce the time complexity before applying similar techniques used with 2-opt local search for FPGA implementation and acceleration.
- Accelerating route selection on an FPGA is difficult due to the random selection needed at each point of selection as well as the memory accesses needed at each point in order to weigh all considered edges. To slightly simplify this process, we tested PACO, which replaced the standard pheromone matrix with a much smaller population matrix. However, this implementation resulted in much worse routes than FACO, so efficient route finding on FPGAs needs to be implemented with another method or by highly tuning PACO.
- Integrating local search into the existing FACO algorithm would be difficult due to the multi-core nature of FACO. In FACO, ants are distributed among different cores, and so different instances of local search would have be queued if running on the FPGA. Combined with all the data transfer needed, integrating accelerated local search running on an FPGA with the rest of FACO running on a multicore processor would be a significant challenge.

Ongoing research into the TSP and optimal solvers for it led us to [4], which conducted a study and competition of TSP solvers for large scale instances (¿1M nodes). The article revealed that the best known algorithms for TSP, especially ones that solve very large instances of TSP relatively quickly, are all based on local search, as opposed to other heuristics like ACO. The best evaluated algorithm is the Lin-Kernighan-Helsgaun (LKH) algorithm, which implements the

Lin-Kernighan heuristic to solve TSP; the LKH algorithm uses up to 5-opt local search. The article also notes the importance of node localization in order to reduce the time complexity and search space for these algorithms, such as the nearest neighbors method used in FACO.

Altogether, with the comparative nature of k-opt local search, as well as the relative difficulty of route selection with ACO with an FPGA, we have decided to focus on algorithms based on local search in the future, as this has been shown to perform better and is likely more suitable for FPGA acceleration.

References

[1] R. Skinderowicz, "Improving ant colony optimization efficiency for solving large tsp instances," *Applied Soft Computing*, vol. 120, p. 108653, 2022.

[2] B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. El-Gindy, and H. Schmeck, "Fpga implementation of population-based ant colony optimization," *Applied Soft Computing*, vol. 4, no. 3, pp. 303–322, 2004. Hardware Implementations of Soft Computing Techniques.

[3] B. Scheuermann, S. Janson, and M. Middendorf, "Hardware-oriented ant colony optimization," *Journal of Systems Architecture*, vol. 53, no. 7, pp. 386–402, 2007.

[4] R. Mariescu-Istodor and P. Fränti, "Solving the large-scale tsp problem in 1 h: Santa claus challenge 2020," *Frontiers in Robotics and AI*, vol. 8, 2021.