ECE 5746 Project: kNN Engine Chip Design

Raymond Nucuta

December 2024

1 Introduction

The k-nearest neighbor (kNN) algorithm is a fundamental approach for a variety of machine learning and similarity tasks. It operates by finding the k closest data points in a database to a given query point, based on a chosen distance metric. Consider a use case involving a database of millions of geolocation points where you aim to find the nearest landmarks to a specific location. Unlike deep learning models that require extensive training and optimization of parameters, kNN is a non-parametric learning algorithm, enabling easy integration into various machine learning applications without prior training phases.

Simply put, the kNN algorithm relies on efficient computation of distance metrics such as Euclidean or Manhattan distances and sorting these distances to determine the top k closest neighbors. From a hardware design perspective, these requirements pose challenges in terms of computational efficiency and energy consumption. The pairwise distance calculations and sorting operations grow quadratically with the size of the database, making acceleration using parallel processing units quite an attractive option.

For this project, a basic version of kNN is implemented where $k = 2$, employing optimization approaches such as adding pipeline stages, implementing a parallel sorting module, and balancing a high frequency design with one which minimizes power.

2 Architecture and Design Decisions

The design requirements state that we take in the following ten inputs:

1. `search_i`: Eight 64-bit search vectors
2. `query`: One 64-bit query vector
3. `in_valid`: input validation bit

where the search and query vectors are 16-D vectors, with each dimension being represented by 4 bits. The following three values will consist of the outputs:

1. `addr_1st`: 3-bit address indicating the top-ranked nearest neighbor

2. `addr_2nd`: 3-bit address indicating the second-ranked nearest neighbor

3. `out_valid`: output validation bit

The `dist_sort` module can be split into two main sub-modules: `dist_mod` and `sort_tree`. After flopping all of the inputs, `dist_mod` is used for each (`query`, `search_v`) pair to find in parallel the pair's Euclidian distance. This step can clearly occur in parallel for each pairing. Zooming in onto an individual `dist_mod`, this module requires the following three operations to compute the Euclidian distance: 16 subtractions across each dimension, 16 multiplications on the differences, and then an accumulation sum of the 16 products. Each of these steps can be piplined to improve the throughput/increase the frequency of the design, which is the metric required to optimize for. Once having all eight Euclidian distances, they can be stored in a `coupled_dist_t` that contains the distance value and its respective 3 bit address. This datatype will serve as the input into the `sort-tree`, as seen in Figure 1.

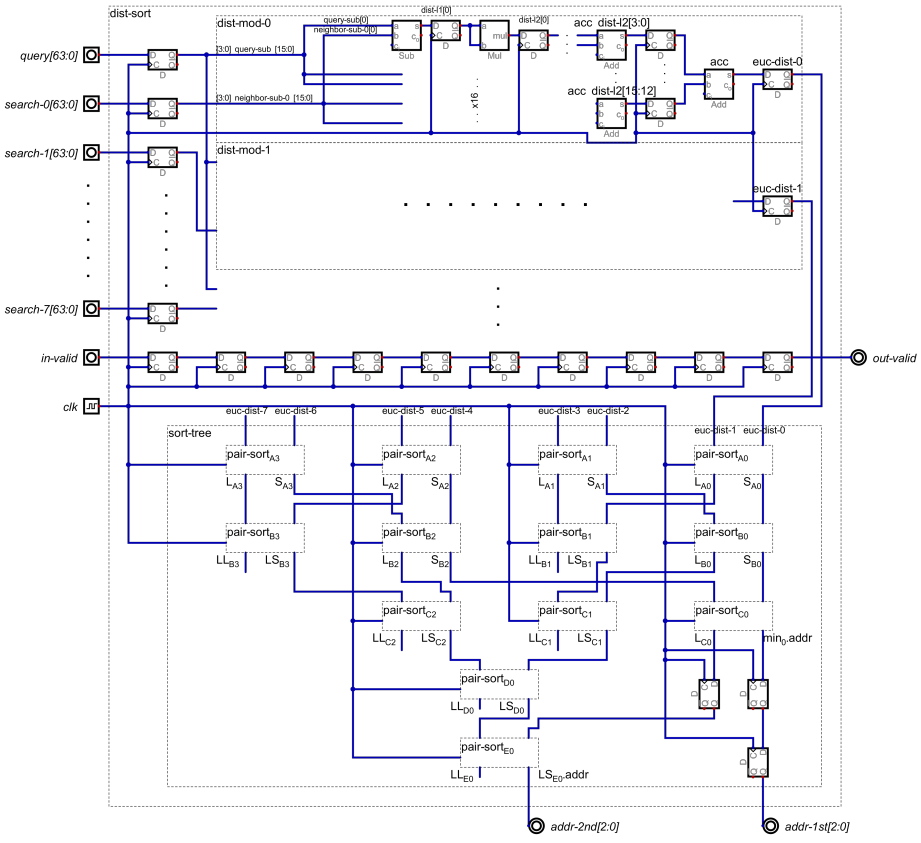

Figure 1: High-level block Diagram

Essentially, the eight values are split into pairs based on their inherent or-

dering, and `pair_sort` will output the largest and smallest value, tie-breaking based on the smallest address. The larger values (L) will be broken into another tree to find the second-smallest value. Although the tree finding the smallest value will have $\mathcal{O}(\ln(n))$ levels, the subtree finding the second smallest value will always add an additional two levels to the overall `sort-tree` as the loser of the last value of the smallest value tree must be included in the second smallest tree, and in addition, this tree starts one level after the main tree. In total, this subtree will have $\mathcal{O}(\ln(n) + 1)$ levels, but only two of these levels cannot run parallel to the main tree. This tree is reminiscent of the merge-sort algorithm, except the array of distances can initially be already broken up into pairs, and the objective is only to find the two smallest values.

2.1 Optimization

The goal of this project is to optimize the following metric:

$$\text{Quality Metric} = (\text{Tot\_latency})^2 \times \text{Power} \times \text{Area}$$

where total latency is the latency of 1000 operations that are given in consecutive clock cycles. There are inherent tradeoffs to solely optimizing just one of these metrics. For example, the clearest way to optimize total latency is to add more pipline stages, but this comes at the cost of increasing the area due to added flip flops, and subsequently increasing the power consumption.

Figure 1 shows a high-level block diagram of the design, but it does not completely reflect all optimization decisions that can be made. The majority of the optimization decisions made here will be at the RTL level, however, there is more potential for optimization. This topic is quite nuanced, and there are many other options to consider that are outside of the scope of this project, such as using all standard cells (SVT/RVT/LVT) instead of only RVT cells to improve performance, adjusting VDD to improve energy consumption without sacrificing reliability, and more.

The first consideration was minimizing the number of bits required for every operation. The first subtract operation (subtracting two 4-bit numbers to get `dist-l1`) will require a 5 bit result. However, squaring `dist-l1` to get `dist-l2` only requires 8 bits for the result as the sign will be dropped. This requires casting the output of the multiplication as an unsigned integer in Verilog. Next, when adding together 8 8-bit numbers, this importantly only requires 12 bits. By ensuring these bit operations are as minimal as possible, and there are no unused bits, the total area of this module is minimized.

The second consideration was the different critical paths. Initially, with two pipeline stages (separating the accumulation sum from the rest of `dist_sort`, and in-between `dist_sort` and `sort_tree`), the critical path is approximately 500 ps for both the entire sort tree and for the accumulation sum. To decrease the latency of these sections, I experimented with different pipline designs. Note, Figure 1 details a design with 10 pipeline stages, which would involve pipelining every possible section of each module. I found that the most reasonable middle

ground was pipelining the accumulation section into two parts, such that initially, groups of 4 8-bit numbers are added to result in a 10-bit sum, the results flopped, and then finally, 4 10-bit numbers are added to result in a 12-bit sum for `sort_tree`. Pipelining additional sections in `dist_mod` achieved minimal benefit. Consequently, I found pipelining in this fashion had the same critical path length as two levels of the `sort_tree`. Starting at the second level, I added flops to the results of every other level, such that the final level would flop the outputs. This resulted in a minimum clock period during synthesis of 325 ps (6 stages), whereas by pipelining everything, I was able to achieve 290 ps (10 stages). However, the 6 stage design used about half of the energy as the 10 stage design according to the synthesis reports, which can be attributed mostly to not having $2 \times 8 \times 64$ extra flip flops for the results of the l1 and l2 distances.

To consider power/area (which are tightly coupled), I referred to the relationships I derived in Figure 2 regarding a typical Energy v Frequency Distribution for the RVT cells in the ASAP PDK. After choosing the 6 stage design, determining the proper clock period was essential. To balance this, I chose the local minima of the total energy section of this distribution, which is approximately at $1.25 \times \text{clk\_period}_{\text{min}} = 410$ ps. Performing APR at 0.5 floorplan utilization proved to properly pass APR design rule checks and hold time violations while still having enough space for the Innovus tool to properly reroute the design. However, in PrimeTime, this clock period resulted in setup violations, so I had to increase my final clock period to **440 ps**.

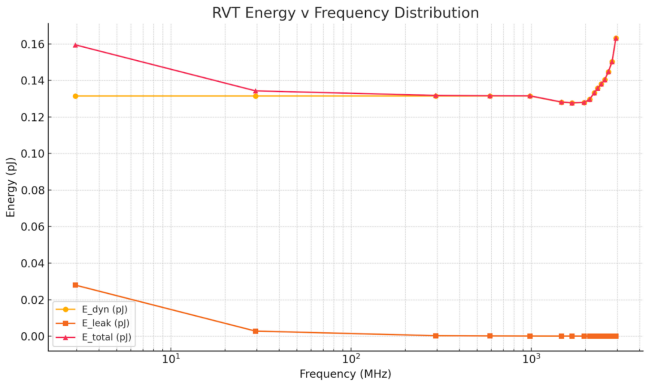

Figure 2: Typical Frequency to Energy Profile

3 Results

The final chip design can be seen below. In future iterations, it could utilize more core area for routing/standard cells.

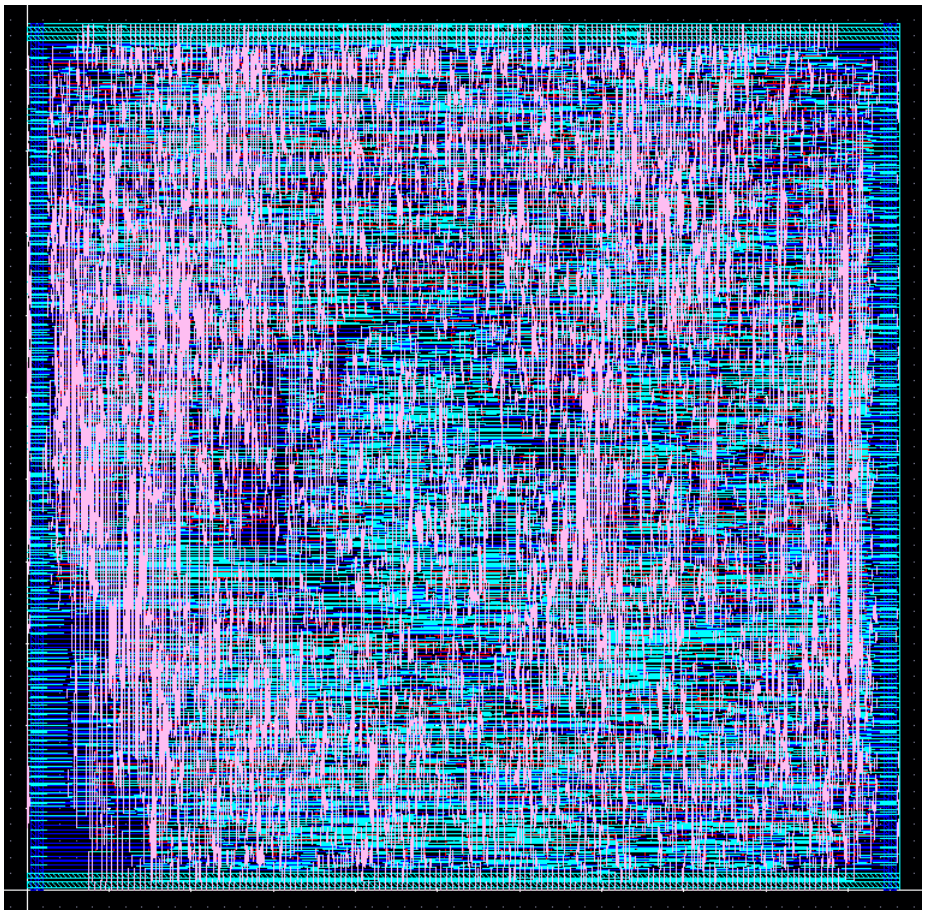

Figure 3: Final layout in Virtuoso

3.1 Total Latency

The total latency reported by the test-bench for 1000 test cases at a clock period of 440 ps is **0.00088616 ms**.

```
# Ip No:        996 Passed
# Ip No:        997 Passed
# Ip No:        998 Passed
# Ip No:        999 Passed
# ----Done with comparing against golden values------
# !!!!!!!!!!!!!!!!!!!!!!!!!END OF TB!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# ** Note: $finish    : ./tb_dist_sort.sv(218)
#    Time: 886160 ps  Iteration: 0  Instance: /tb_dist_sort
```

Figure 4: ModelSim stdout

3.2 Power

The reported power from PrimeTime was **3.118 mW**. In future iterations, this could be reduced by synthesizing at a lower clock period, or reducing the number of pipeline stages.

3.3 Area

The total area of my core (excluding power rings) was 40699.896 square microns, or **0.040699 square mm**. This can also be calculated from Figure 5, keeping in mind that the power grid uses 5 microns on each side.

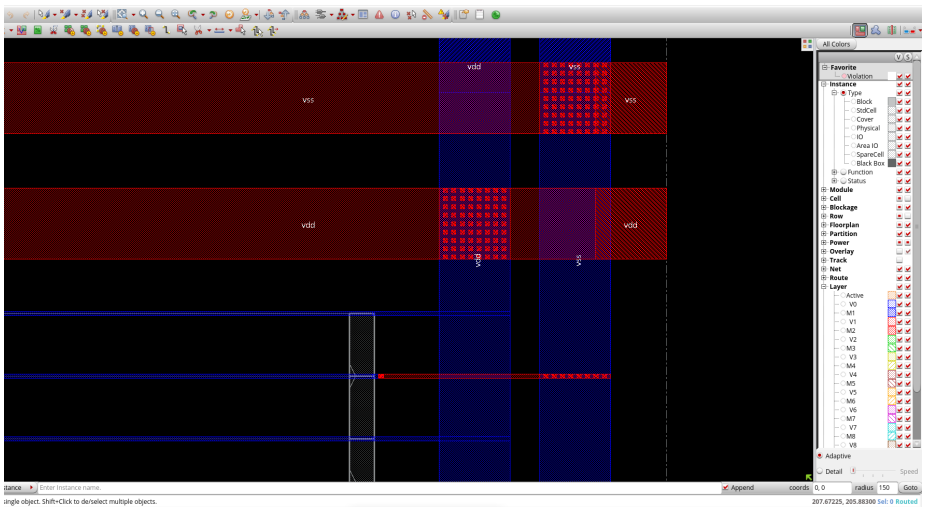

Figure 5: Max x-y coordinate in core area

3.4 Density

To allow the Innovus tool to have ample room for routing, a **0.5** utilization ratio was used for the floorplan. This resulted in a **0.58** core density before filler insertion.

3.5 Gate Count

The output of `reportGateCount` is

```
Gate area 0.6998 um^2
[0] dist_sort Gates=37270 Cells=16351 Area=26083.0 um^2
```

which indicates the core area used by standard cells.

4 Conclusion

This project has demonstrated the design and optimization of a kNN engine, achieving efficient distance computation and sorting. As a potential next step, integrating locality-sensitive hashing could significantly enhance the scalability of the engine for high-dimensional and large-scale datasets, offering approximate solutions with reduced computational overhead. Alternatively, implementing lookup tables could enable faster response times for repeated or fixed queries by precomputing and storing results. Both approaches could expand the applicability of the module in real-world scenarios, balancing trade-offs between accuracy and efficiency. In a massive database, these techniques implemented in hardware would greatly improve the performance of kNN for practical scenarios.

4.1 Bottlenecks

This project has successfully designed and optimized a kNN engine with efficient distance computation and sorting, achieving a balance between performance, power, and area. However, certain bottlenecks remain. The primary bottleneck in terms of performance is the critical path in the sorting module, specifically within the final stages of the `sort_tree`, where data dependencies limit further parallelism. While additional pipelining could reduce the critical path and improve clock frequency, this would come at the cost of increased power consumption due to the added flip-flops.

In terms of power, the pipeline stages contribute significantly to dynamic power, with the trade-off being necessary to achieve acceptable latency. Reducing power further would require optimizing the pipeline depth or exploring alternative cell libraries, such as LVT cells, which offer lower power at higher frequencies but may introduce reliability challenges. In addition, increasing the floorplan utilization ratio could be explored to lower the effects of parasitics and slew from long data paths.

Future work could focus on alternative algorithms to address these bottlenecks. An exciting option would include using the initial `dist_sort` module as a sub-module for more scalable approaches such as LSH or LUTs to improve usage for large datasets.