AE6102: Parallel Scientific Computing and Visualization

Course Project

OptiVLSI

Rohan Kalbag 20D170033

Neeraj Prabhu 200070049

Course Instructor: Prof. Prabhu Ramachandran

Abstract

Very Large Scale Integration abbreviated as VLSI involves digital circuits dealing with billions of transistors, and requires computerized design automation, design verification and testing algorithms. Digital circuits are often represented using graphs, where the logic gates are nodes and their interconnections are the edges. Since VLSI circuits have millions of logic gates, there is a need for fast and highly optimized graph algorithms. There are a few optimized graph libraries with optimized basic graph operations such as `pyEDA` and `networkx`, but there are no optimized and high performance open source libraries which build upon these to specifically cater to the VLSI computer aided design automation industry.

Contents

1 Introduction

Through this course project we aim to choose a subset of graph algorithms, visualize their operation in action, optimize them using `numba`, study the speedup with respect to input parameters by running simulations using `automan`, compare our implementation with optimized implementations from existing libraries.

The subset of problems that we decided to target in VLSI are

- **Logic Representation** - ROBDD
- **Circuit Routing/Maze Routing Problems** - Lee's Algorithm
- **Graph Operations** - Shortest Path (Dijkstra, Bellman-Ford), Minimum Spanning Tree (Prim, Kruskal)
- **Logic Simulation** - Compiled-Code Simulator, Event-Driven Simulator

The algorithms developed are hosted as open source software on a **GitHub** repository, open to collaborations from other developers in the future.

2 Logic Representation Algorithms

2.1 Need for Compact Logic Representation in VLSI Circuits

- In VLSI circuit design, Binary Decision Diagrams (BDDs) are a compact logic representation technique that can help reduce the size and complexity of logic.
- This makes it easier to design and verify circuits, leading to fewer errors and delays.
- This is because BDDs enable designers to identify issues and optimize the circuit design earlier in the process. As a result, designers can reduce the time and effort required to test and debug circuits as well as the need for re-designing circuits due to design errors.

2.2 BDD Generation: Implementation

The Reduced Ordered Binary Decision Diagram is a compact, canonical, graphical representation of a Boolean function. More details can be found in [1].

Link to the implementation on GitHub

- The python script `robdd.py` takes a circuit as input from a string and generates the binary decision diagram for the same. The time taken for the numba and pythonic implementations was recorded for multiple circuits.

  Usage Instructions

  ```
  python3 robdd.py --input input --order order
  ```

- The `--input` gives the input string consisting of the input nets and the operation to be performed among each of the nets.
  - The character `'.'` signals that there is an and operation between the 2 operands on either side of it
  - The character `'+'` signals that there is an or operation between the 2 operands on either side of it
  - The character '∼' signals that the net immediately after it is to be negated for further calculations
- The `--order` gives the priority order of the variables being taken as input for the BDD generation. The highest priority variable is at the top and the priority reduces as you go further down the BDD

2.3 BDD Generation: Results and Inferences

The script was tested for multiple input circuits, and the time taken by the pythonic and numba implementations were compared to obtain the following results

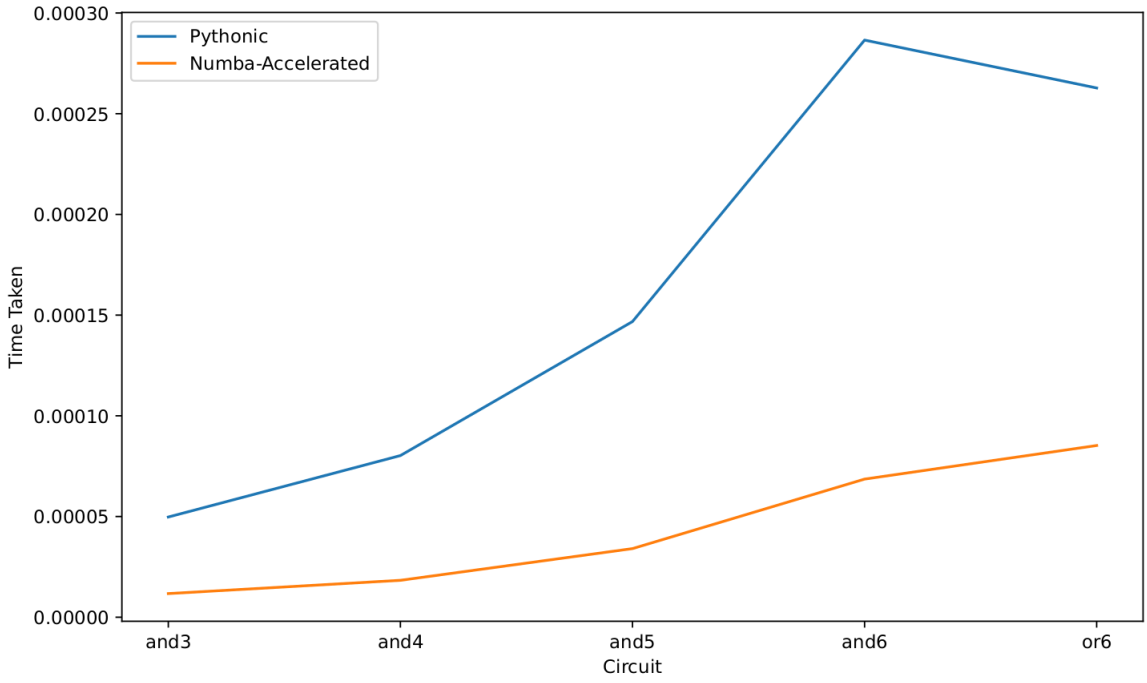

Figure 1: Time taken by the implementations for various circuit benchmarks

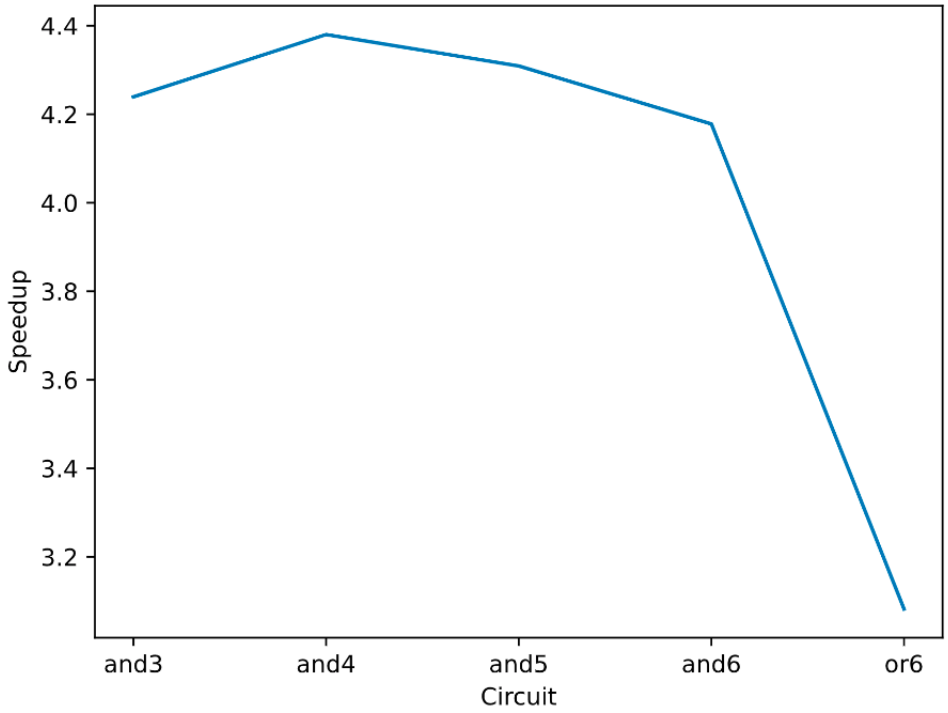

Figure 2: Speedup of numba accelerated w.r.t pythonic implementation for various benchmark

As can be seen, the numba accelerated version is approximately 4 times faster than the pythonic version for the and circuits and the speedup increases with the size of the circuit. However, the or circuit has a lesser speedup than the and circuit, giving around 3 times speedup for a 6 input or gate. This could be mainly due to how the BDD is generated.

3 Circuit/Maze Routing Algorithms

3.1 Need for Circuit Routing in VLSI

- VLSI circuits are complex integrated circuits that contain millions of transistors on a single chip. The routing process involves connecting the various components of the chip such as transistors, gates, and interconnects using metal wires to form a functional circuit.
- Improper routing can lead to delays, signal distortion, and other issues that can result in reduced performance. A well-designed routing scheme can help minimize the area of the circuit and reduce power consumption.
- Proper routing helps ensure that the chip can be manufactured correctly and reliably, reducing the risk of defects and failures.
- Lee Algorithm is used in automation of the process of routing. It is used to find the shortest path between a start point and end point (which denotes a wire connecting two

nets) in a maze which can have obstacles (which could be other routes or components like transistors, gates).

- Other constraints can be added to Lee Algorithm such as restriction on the angle of the turns made by the routes, number of turns .etc, to fixate upon the best solution out of the many possibilities of paths it generates.

3.2 Lee Algorithm : Implementation

The Lee algorithm is widely used for maze routing problems based on breadth-first search. It always gives an optimal solution, if one exists. More about this algorithm can be found in [2].

Link to the implementation on GitHub

3.2.1 Script to Create Benchmark Mazes

- The `create_benchmark.py` file can be used to create a `.npz` file consisting of a benchmark maze on which the algorithm can be tested

  Usage Instructions
  `python3 create_benchmark.py -n size_of_maze -p prob_of_obstacle -f filename`

- p here denotes the probability of a particular entry of the maze $M[i][j]$ contains an obstacle, and is independent of i and j.

- The generated maze is stored as a $n \times n$ numpy array in `filename.npz`

3.2.2 Main Script containing Implementations

- The `lee_algorithm.py` contains three implementations of the algorithm, a regular pythonic implementation, its numba njit accelerated version and a networkx BFS based version, where the maze is converted to a `nx.Graph()` object and BFS is applied, the former two however directly use the matrix stored as a numpy array.

  Usage Instructions
  `python3 lee_algorithm.py --b (optional) [--c -n size -p prob]/[--f -m maze] -sx startx -sy starty -ex endx -ey endy`

- if `--c` flag is passed this script directly creates a benchmark of size n and probability p (like how in `create_benchmark.py`)

- if `--f` flag is passed then we need to specify the benchmark maze we are feeding to it by passing it filename as `-m maze` if our benchmark maze is stored in `maze.npz`

- `sx`, `sy`, `ex` and `ey` need to be passed are the start and stop points on the maze

- if `--b` flag is used then benchmarking will be done using `time.perf_counter()` and the time taken by each of the methods will be printed on three lines in the order (numba, python, networkx), if this flag is not specified, the script will produce two `.pdf` files each containing the plot of the maze generated by the script/passed to the script (in `benchmark.pdf`) and one of the possible efficient paths generated by using the algorithm (in `foundpath.pdf`)

3.2.3 Simulation Automation using Automan

- The time taken for execution, speedup were plotted for various sizes of input maze size to study scalability of the implementations using automan. The script created for the same is `automate.py`

  Usage Instructions
  `python3 automate.py`

- The plot generated by automan can be found in `/manuscripts/figures/Lee` and outputs can be found in `outputs/Lee`

3.3 Lee Algorithm : Results and Inferences

An example 10×10 maze was created with $p = 0.1$ and an optimal path was found between $(0, 0) \rightarrow (6, 6)$ with the following command to demonstrate visualization of this algorithm.

```
python3 lee_algorithm.py --c -n 10 -p 0.1 -sx 0 -sy 0 -ex 6 -ey 6
```

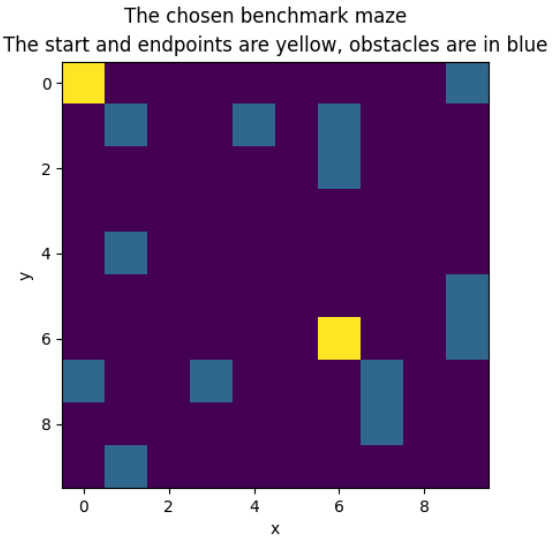

Figure 3: The benchmark maze generated

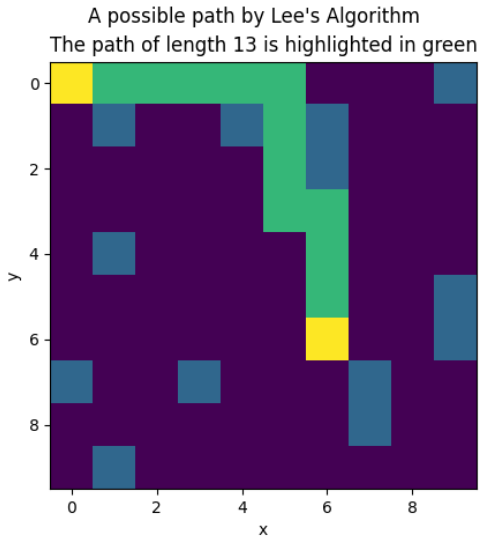

Figure 4: Final optimal path obtained

The plots obtained by the `automan` script, where the times and speedup of the three implementations were compared for varying input maze sizes can be seen below. Input sizes took values $n \in \{2, 4, 10, 20, 30, 50, 100, 150, 200, 300, 500\}$.

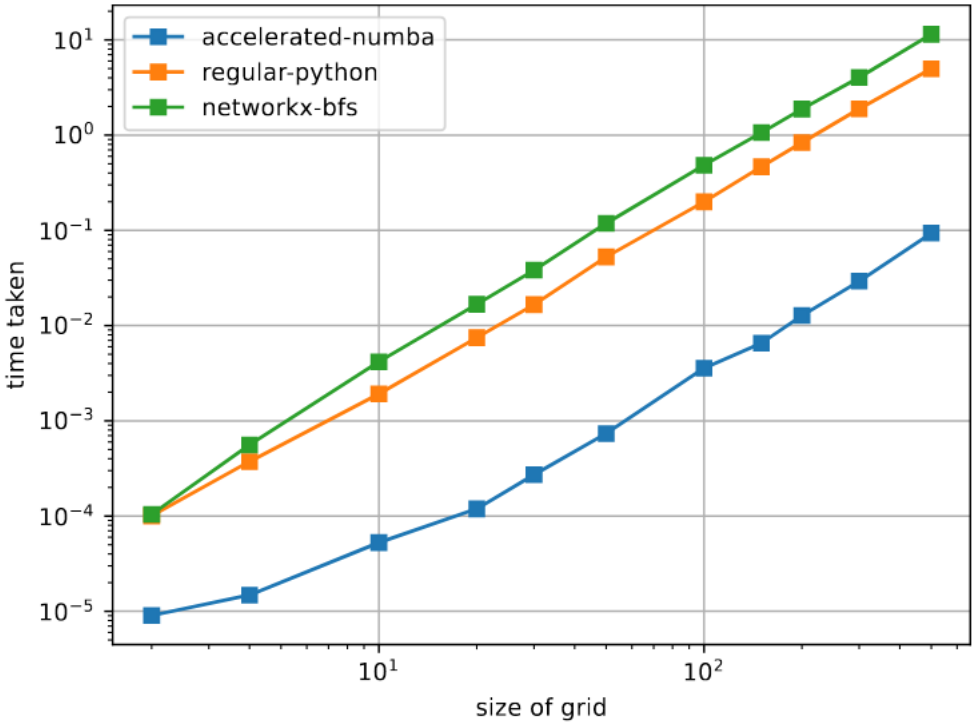

Figure 5: The timing of various implementation for varying input sizes

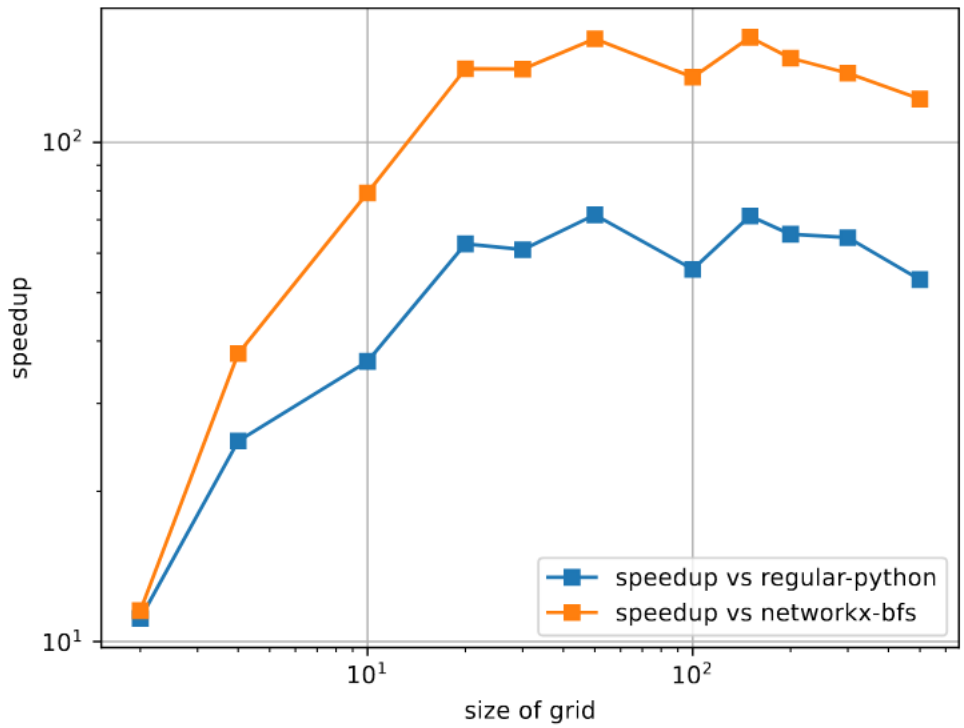

Figure 6: The speedup obtained w.r.t to numba accelerated for the other two implementations

Thus we can see that the numba-accelerated version is nearly 10^1 times faster than regular pythonic and 10^2 times faster than networkx BFS based implementation. This is a consequence of the significant overhead due to creation of large `nx.Graph()` objects in the latter. We also notice that the speedup scales with input size as well

4 Graph Operations - Shortest Path Algorithms

4.1 Need for Shortest Path Algorithms in VLSI

- VLSI circuits can be modelled as Directed Acyclic Graphs with the logic gates/transistors as nodes and wires/interconnects as weighted edges
- These wires have a cost metric (length, propogation delay) which is non negative, hence Dijkstra's/Bellman Ford Algorithm is applicable here.
- In VLSI CAD design, the placement of components on a chip is critical to the overall performance and functionality of the design. The goal is to place the components in a

way that minimizes the total distance that signals need to travel between them. This can be achieved by using Dijkstra's algorithm to find the shortest path between each pair of components and then using this information to optimize the placement of the components on the chip

- Dijkstra's algorithm can also be used to optimize the routing of connections between components on a chip. By finding the shortest path between each pair of components, this can be used to minimize the length of the connections and reduce the overall complexity of the routing

4.2 Dijkstra's Algorithm : Implementation

Dijkstra's algorithm is used for finding the shortest paths between nodes in a weighted graph. It requires the weights to be non-negative $w_{ij} \geq 0$. More details about the algorithm can be found in [3].

Link to the implementation on GitHub

4.2.1 Main Script containing Implementations

- The `dijkstra.py` contains three implementations of the algorithm, a regular pythonic implementation, its numba njit accelerated version and the networkx implementation `nx.dijkstra_path()`, where the benchmark Directed Acyclic Graph is converted to a `nx.Graph()` object and the above method is used directly to obtain the path.

  Usage Instructions
  `python3 dijkstra_algorithm.py --b (optional) [--c -n size -p prob -w1 minw1 -w2 minw2 ]/[--f] -m fileloc -s start_node -e end_node`

- if `--c` flag is passed this script directly creates a benchmark of size n and probability p using the `gnp_random_graph()` which generates erdos-renyi graphs which are DAGs (Directed Acyclic Graph), the approach in [4] was followed.

- n here denotes the number of nodes in the graph

- p here denotes the probability of a particular pair of nodes i, j to have an edge between them, and is independent of i and j and the graph is stored as numpy arrays of nodes and edgelist in `fileloc.npz`. Where edge weight is constrained to be $w_1 \leq w \leq w_2$

- if `--f` flag is passed then we need to specify the graph we are feeding to it by passing it filename as `-m fileloc` if our benchmark graph is stored in `fileloc.npz`

- if `--b` flag is used then benchmarking will be done using `time.perf_counter()` and the time taken by each of the methods will be printed on three lines in the order (numba, python, networkx), if this flag is not specified and also print the shortest path obtained all three methods one by one

- the script will produce two `.pdf` files each containing the graph generated by the script/passed to the script (in `fileloc.pdf`) and one with shortest path highlighted (in `fileloc_shortest_path.pdf`)

4.2.2 Simulation Automation using Automan

- The time taken for execution, speedup were plotted for various benchmark graph sizes to study scalability of the implementations using automan. The script created for the same is `automate.py`

  Usage Instructions
  `python3 automate.py`

- The plot generated by automan can be found in `/manuscripts/figures/Dijkstra` and outputs can be found in `outputs/Dijkstra`

4.3 Dijkstra's Algorithm : Results and Inferences

The shortest path was found for an example circuit represented by a DAG with $n = 10, p = 0.3$, and weights $5 \leq w_{ij} \leq 15$, with start node 0 and end node 9 using the following command to demonstrate visualization of the algorithm

```
python3 dijkstra.py --c -n 10 -p 0.3 -w1 5 -w2 15 -s 0 -e 9 -m benchmark
```

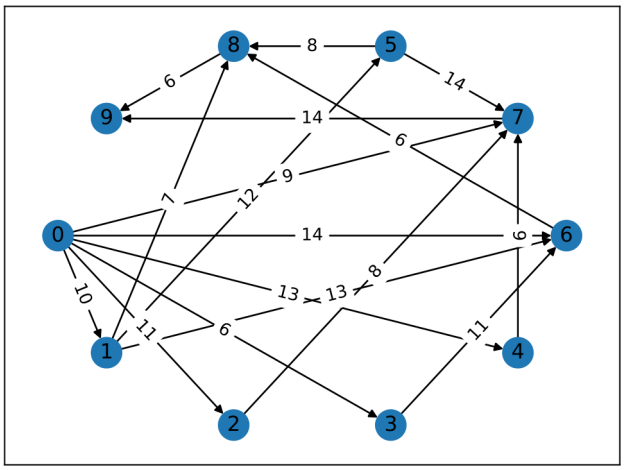

Figure 7: Generated DAG benchmark circuit

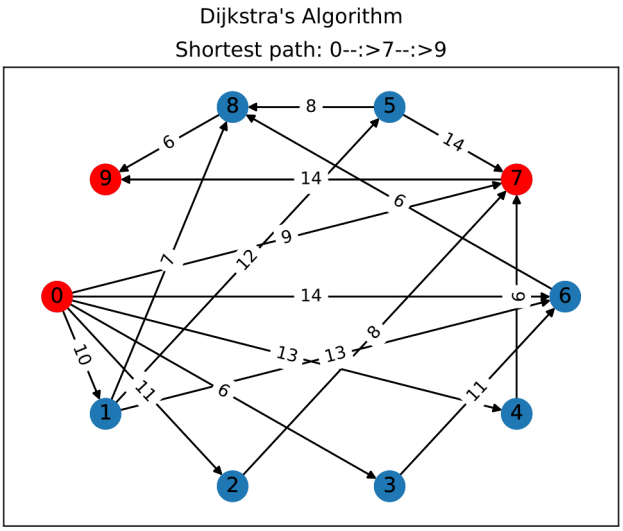

Figure 8: Shortest path found by Dijkstra's Algorithm

The plots obtained by the `automan` script, where the times and speedup of the three implementations were compared for varying input circuit sizes can be seen below. Input sizes took values $n \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 175\}$.

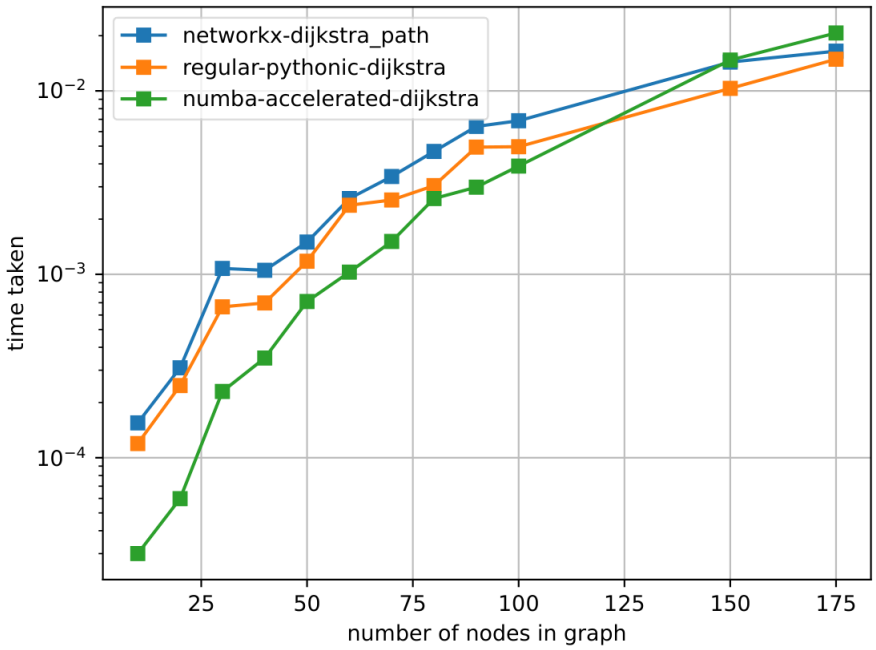

Figure 9: The timing of various implementation for varying input sizes

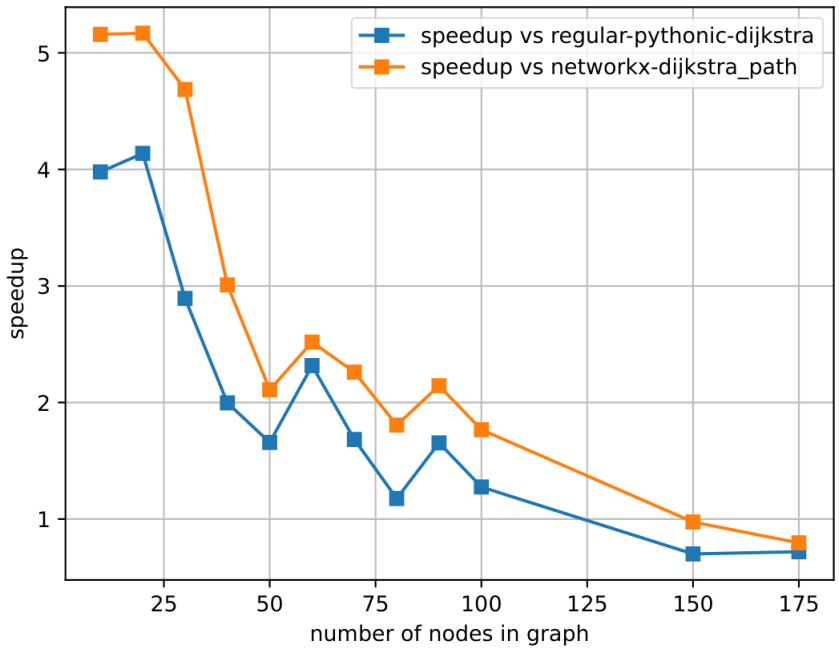

Figure 10: The speedup obtained w.r.t to numba accelerated for the other two implementations

Thus we can see that the numba-accelerated version is nearly $1-4$ times faster than regular pythonic and $1-5$ times faster than networkx implementation. But we can see that the speedup scales does not scale with input circuit size, and as the size increases all the three implementations converge, indicating the existence of a bottleneck like memory bandwidth.

4.4 Bellman-Ford Algorithm: Implementation

The Bellman–Ford algorithm computes shortest paths from a source vertex to a destination vertex in a weighted graph. It is computationally slower than Dijkstra's Algorithm but allows for negative weights $w_{ij} < 0$. For more details about the algorithm one can refer [5].

Link to the implementation on GitHub

4.4.1 Main Script containing Implementations

- The `bellman-ford.py` contains three implementations of the algorithm, a regular pythonic implementation, its numba njit accelerated version and the networkx implementation `nx.bellman_ford_path()`, where the benchmark Directed Acyclic Graph is converted to a `nx.Graph()` object and the above method is used directly to obtain the path.

  Usage Instructions

  ```
  python3 bellman-ford.py --b (optional) [--c -n size -p prob -w1 minw1 -w2 minw2
  ]/[--f] -m fileloc -s start_node -e end_node
  ```

- If `--c` flag is passed this script directly creates a benchmark of size n and probability p using the `gnp_random_graph()` which generates erdos-renyi graphs which are DAGs (Directed Acyclic Graph), the approach in [4] was followed.
- n here denotes the number of nodes in the graph
- p here denotes the probability of a particular pair of nodes i, j to have an edge between them, and is independent of i and j and the graph is stored as numpy arrays of nodes and edgelist in `fileloc.npz`. Where edge weight is constrained to be $w_1 \leq w \leq w_2$
- If `--f` flag is passed then we need to specify the graph we are feeding to it by passing it filename as `-m fileloc` if our benchmark graph is stored in `fileloc.npz`
- If `--b` flag is used then benchmarking will be done using `time.perf_counter()` and the time taken by each of the methods will be printed on three lines in the order (numba, python, networkx), if this flag is not specified and also print the shortest path obtained all three methods one by one
- The script will produce two `.pdf` files each containing the graph generated by the script/passed to the script (in `fileloc.pdf`) and one with shortest path highlighted (in `fileloc_shortest_path.pdf`)

4.4.2 Simulation Automation using Automan

- The time taken for execution and the speedup achieveed using numba were plotted for various benchmark graph sizes to study scalability of the implementations using automan. The script created for the same is `automate.py`

  Usage Instructions
  `python3 automate.py`

- The plot generated by automan can be found in `/manuscripts/figures/Bellman-Ford` and outputs can be found in `outputs/Bellman-Ford`

4.5 Bellman-Ford Algorithm : Results and Inferences

The shortest path was found for an example circuit represented by a DAG with $n = 10, p = 0.5$, and weights $5 \leq w_{ij} \leq 15$, with start node 0 and end node 9 using the following command to demonstrate visualization of the algorithm

```
python3 bellman-ford.py --c -n 10 -p 0.5 -w1 5 -w2 15 -s 0 -e 9 -m benchmark
```

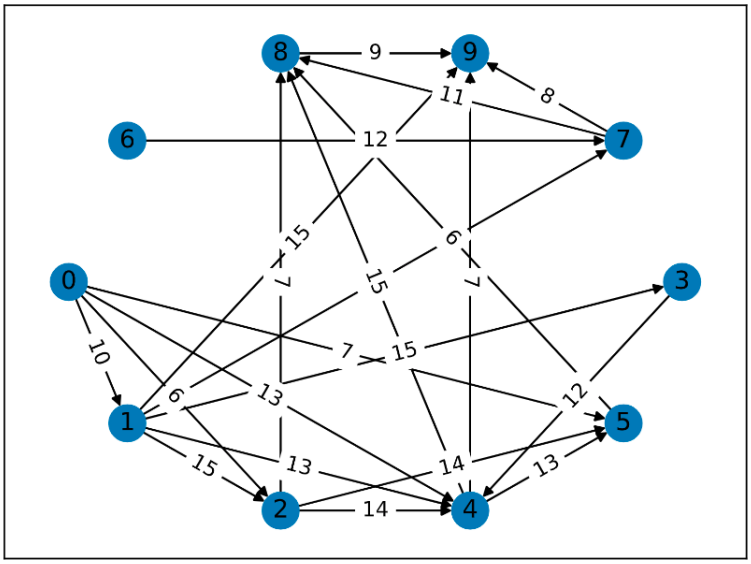

Figure 11: Generated DAG benchmark circuit

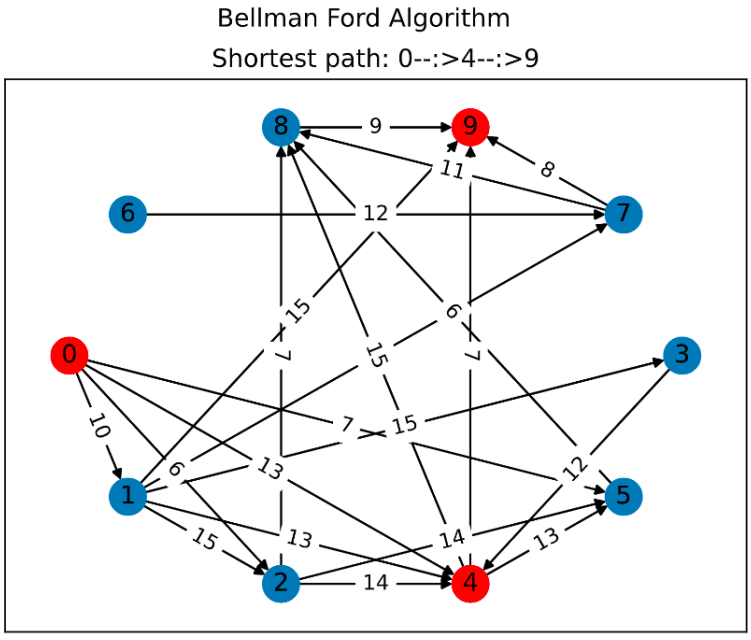

Figure 12: Shortest path found by Bellman-Ford Algorithm

The plots obtained by the `automan` script, where the times and speedup of the three implementations were compared for varying input circuit sizes can be seen below. Input sizes took values $n \in \{10, 20, 50, 100, 125, 150, 175, 200\}$.

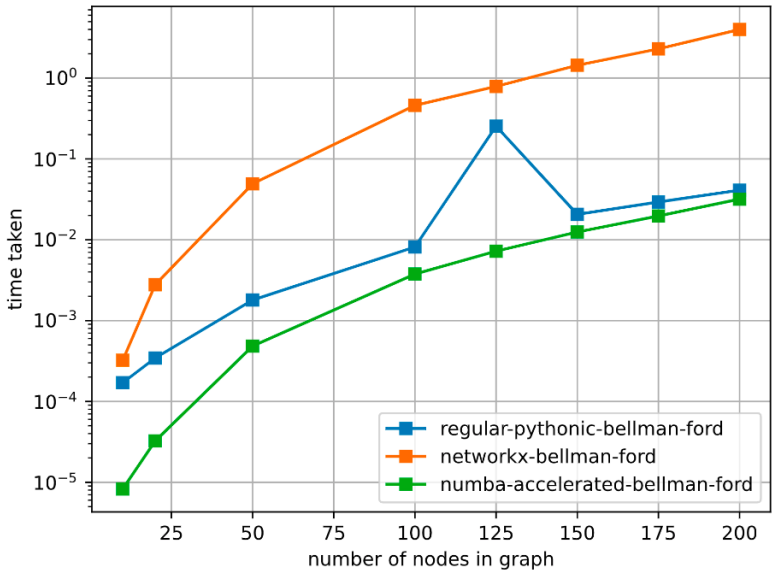

Figure 13: The timing of various implementation for varying input sizes

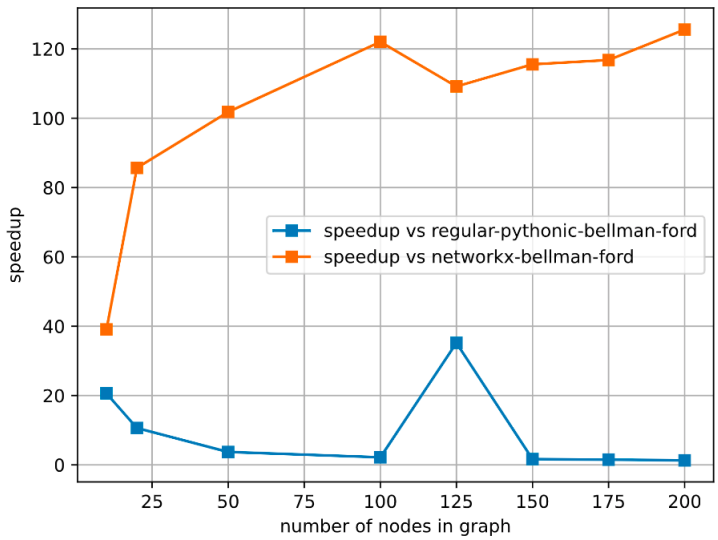

Figure 14: The speedup obtained w.r.t numba accelerated for the other two implementations

Thus we can see that the numba-accelerated version is 40+ times faster than the networkx version. As the size of the graph increases, the speedup of numba over the networkx implementation increases. This is due to the overhead of creation of the `nx.Graph()` object. The numba-accelerated implementation has a speedup of around 20 for small graph sizes. however, the speedup slowly decreases with increasing size and the time taken for both implementations slowly comes closer.

5 Graph Operations: Minimum Spanning Tree Algorithms

5.1 Need for Minimum Spanning Trees in VLSI

- This is a very powerful algorithm especially in dense VLSI circuits, especially in those with having components with lots of fan-outs and fan-ins.
- The minimum spanning tree generates a configuration of all the components present with connections reducing the total wire length.
- MSTs can be used to optimize the distribution of power in a VLSI circuit. By creating an MST of the power distribution network, the most efficient path for power delivery can be determined, reducing the overall power consumption of the circuit.
- They can also be used to place components in an optimized layout such that signal integrity is improved, especially with signals such as CLK which are needed in almost all components for synchronization of the circuit.
- For MST creation, VLSI circuits are modelled as undirected graphs as this is used to study the topology/layout of the circuit, as MSTs are only defined for undirected graphs, unlike in the case of shortest path algorithm, where circuits are modelled as directed acyclic graphs

5.2 Kruskal's Algorithm: Implementation

Kruskal's algorithm makes use of a sorted edge list and a disjoint-set-union data structure. More details can be found in [6].

Link to the implementation on GitHub

5.2.1 Main Script containing Implementations

- The `kruskal.py` contains three implementations of the algorithm, a regular pythonic implementation, its numba njit accelerated version and a networkx based version, where the undirected graph is converted to a `nx.Graph()` object.

The `nx.minimum_spanning_tree(algorithm='kruskal')` is applied, the former two however directly use the circuit stored as a numpy array.

Usage Instructions
`python3 kruskal.py --b (optional) [--c -n size -p prob -w1 minw1 -w2 minw2 ]/[--f] -m fileloc -t treeloc`

- if `--c` flag is passed this script directly creates a benchmark of size n and probability p using the `gnp_random_graph()` which generates erdos-renyi graphs which are DAGs (Directed Acyclic Graph), the approach in [4] was followed. This graph is then converted to an undirected form by adding (v, u) to $V \; \forall \; (u, v) \in V$.
- n here denotes the number of nodes in the graph
- p here denotes the probability of a particular pair of nodes i, j to have an edge between them, and is independent of i and j and the graph is stored as numpy arrays of nodes and edgelist in `fileloc.npz`. Where edge weight is constrained to be $w_1 \leq w \leq w_2$
- if `--f` flag is passed then we need to specify the graph we are feeding to it by passing it filename as `-m fileloc` if our benchmark graph is stored in `fileloc.npz`
- if `--b` flag is used then benchmarking will be done using `time.perf_counter()` and the time taken by each of the methods will be printed on three lines in the order (numba, python, networkx), if this flag is not specified and also print the shortest path obtained all three methods one by one
- Irrespective of the flags, the script will produce two `.pdf` files each containing the graph generated by the script/passed to the script (in `fileloc.pdf`) and one with shortest path highlighted (in `treeloc.pdf`) and also print the shortest path obtained all three methods one by one, both the graphs are also stored in `.npz` format as well

5.2.2 Simulation Automation using Automan

- The time taken for execution, speedup were plotted for various sizes of input maze size to study scalability of the implementations using automan. The script created for the same is `automate.py`

Usage Instructions
`python3 automate.py`

- The plot generated by automan can be found in `/manuscripts/figures/Kruskal` and outputs can be found in `outputs/Kruskal`

5.3 Kruskal's Algorithm : Results and Inferences

The minimum spanning tree was found for an example circuit represented by a DAG with $n = 6, p = 0.5$, and weights $2 \leq w_{ij} \leq 6$ using the following command to demonstrate visualization of the algorithm

```
python3 kruskal.py --c -n 6 -w1 2 -w2 5 -p 0.5 -m graph -t mst --b
```

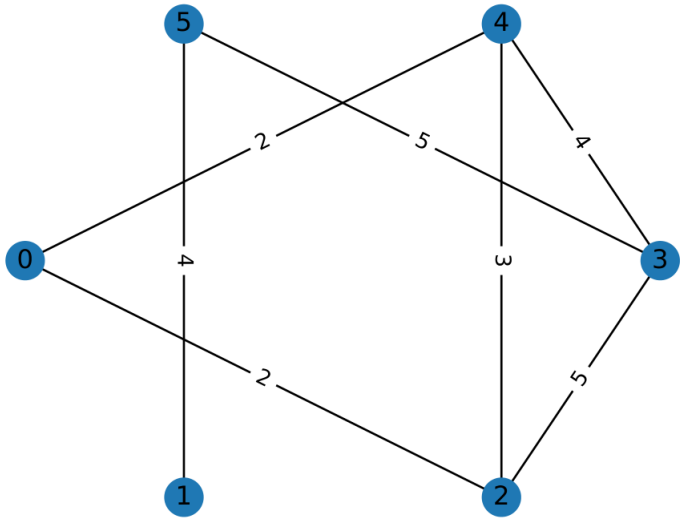

Figure 15: The benchmark undirected graph circuit generated

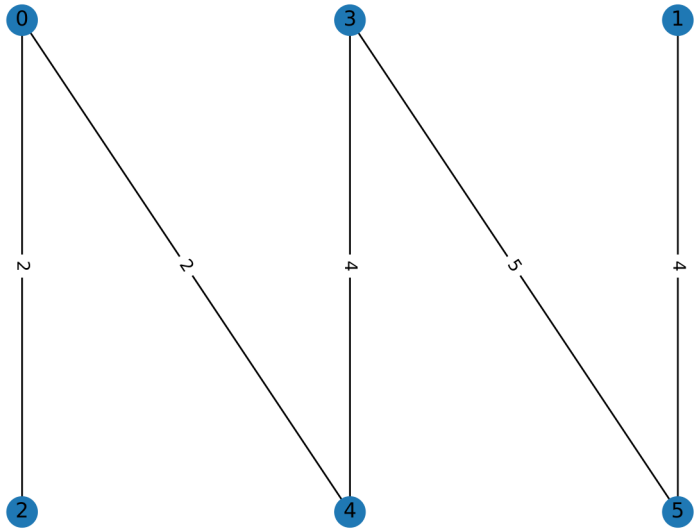

Figure 16: The minimum spanning tree obtained by using Kruskal's Algorithm

The plots obtained by the `automan` script, where the times and speedup of the three implementations were compared for varying input graph sizes can be seen below. Input sizes took values $n \in \{10, 20, 50, 100, 150, 200, 350, 400, 500\}$.

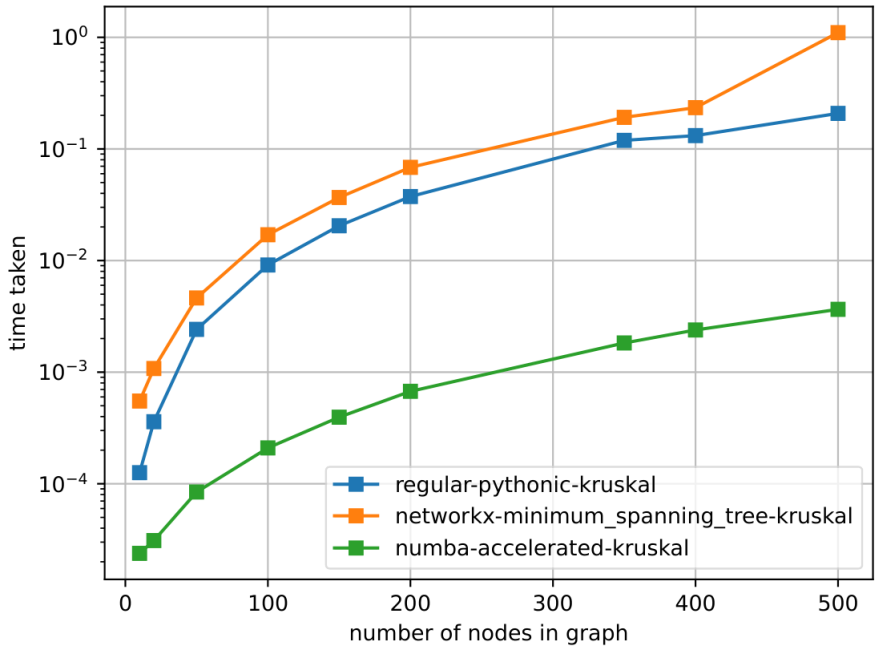

Figure 17: The timing plotted for the various implementations for varying input sizes

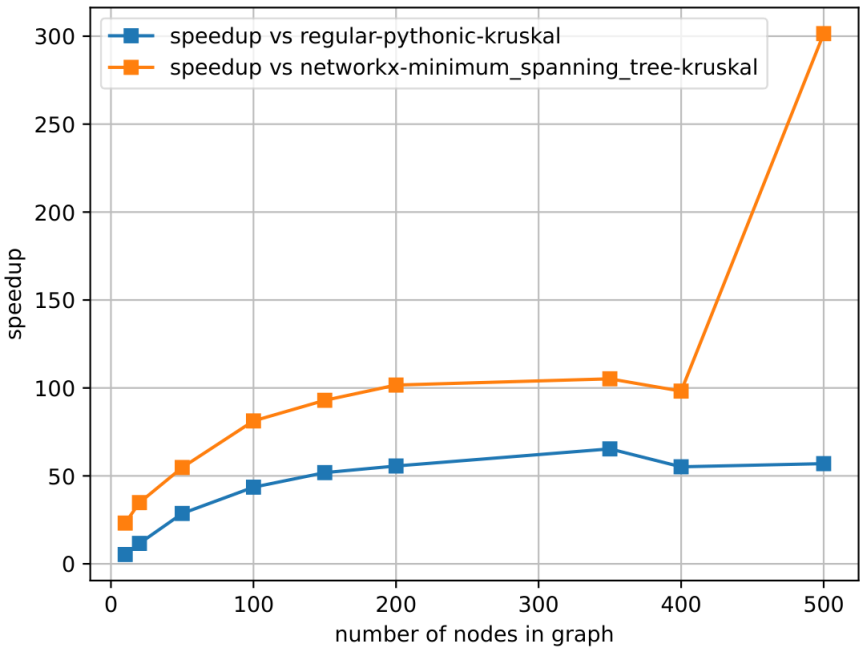

Figure 18: The speedup obtained w.r.t numba accelerated for the other two implementations

Thus we can see that the numba-accelerated version is nearly an order of 10^1 times faster than regular pythonic and an order of 10^2 times faster than networkx minimum spanning tree implementation. This is a consequence of the significant overhead due to creation of large `nx.Graph()` objects to apply the function in the latter. We also notice that the speedup gradually increases as the number of nodes in the graph increases.

5.4 Prim's Algorithm: Implementation

This algorithm grows the spanning tree, by finding the minimum-weight edge, and transfering it to the tree. More details can be found in [7].

Link to the implementation on GitHub

5.4.1 Main Script containing Implementations

- The `prim.py` contains three implementations of the algorithm, a regular pythonic implementation, its numba njit accelerated version and the networkx implementation `nx.minimum_spanning_tree(algorithm='prim')`, where the benchmark Directed Acyclic Graph is converted to a `nx.Graph()` object and the above method is used directly to obtain the path.

Usage Instructions
`python3 prim.py --b (optional) [--c -n size -p prob -w1 minw1 -w2 minw2 ]/[--f] -m fileloc -t result`

- If `--c` flag is passed this script directly creates a benchmark of size n and probability p using the `gnp_random_graph()` which generates erdos-renyi graphs which are DAGs (Directed Acyclic Graph), the approach in [4] was followed. This graph is then converted to an undirected form by adding (v, u) to $V \ \forall \ (u, v) \in V$.
- n here denotes the number of nodes in the graph
- p here denotes the probability of a particular pair of nodes i, j to have an edge between them, and is independent of i and j and the graph is stored as numpy arrays of nodes and edgelist in `fileloc.npz`. Where edge weight is constrained to be $w_1 \leq w \leq w_2$
- If `--f` flag is passed then we need to specify the graph we are feeding to it by passing it filename as `-m fileloc` if our benchmark graph is stored in `fileloc.npz`
- If `--b` flag is used then benchmarking will be done using `time.perf_counter()` and the time taken by each of the methods will be printed on three lines in the order (numba, python, networkx), if this flag is not specified and also print the shortest path obtained all three methods one by one
- The script will produce two `.pdf` files each containing the graph generated by the script/passed to the script (in `fileloc.pdf`) and one with shortest path highlighted (in `result.pdf`)

5.4.2 Simulation Automation using Automan

- The time taken for execution and the speedup achieved using numba were plotted for various benchmark graph sizes to study scalability of the implementations using automan. The script created for the same is `automate.py`

Usage Instructions
`python3 automate.py`

- The plot generated by automan can be found in `/manuscripts/figures/Prim` and outputs can be found in `outputs/Prim`

5.5 Prim's Algorithm : Results and Inferences

The minimum spanning tree was found for an example circuit represented by a DAG with $n = 10, p = 0.3$, and weights $5 \leq w_{ij} \leq 15$, by running the following command

```
python3 prim.py --c -n 10 -p 0.3 -w1 5 -w2 15 -t min_span_tree
```

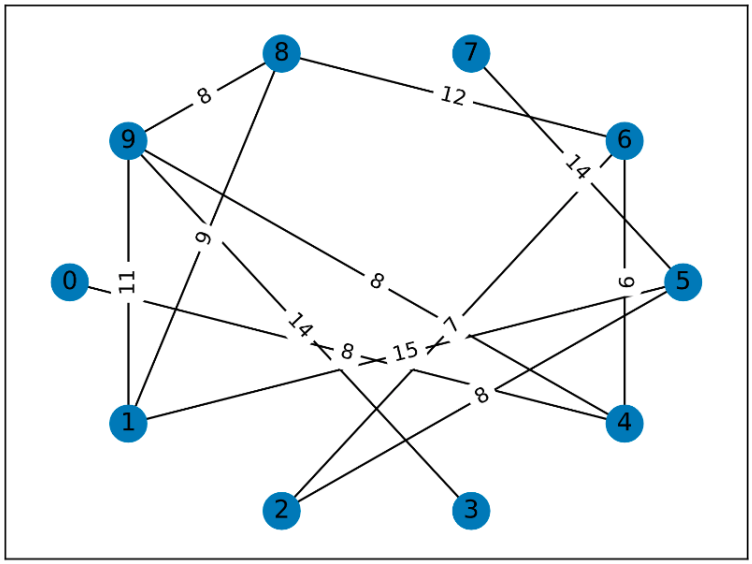

Figure 19: Generated DAG benchmark circuit

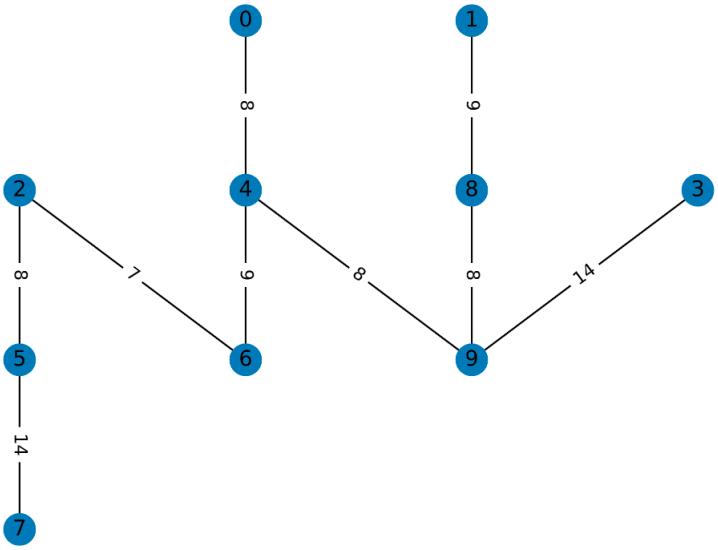

Figure 20: Minimum Spanning Tree of the above graph

The plots obtained by the `automan` script, where the times and speedup of the three implementations were compared for varying input circuit sizes can be seen below. Input sizes took values $n \in \{10, 20, 50, 100, 150, 200, 350, 400, 500\}$.

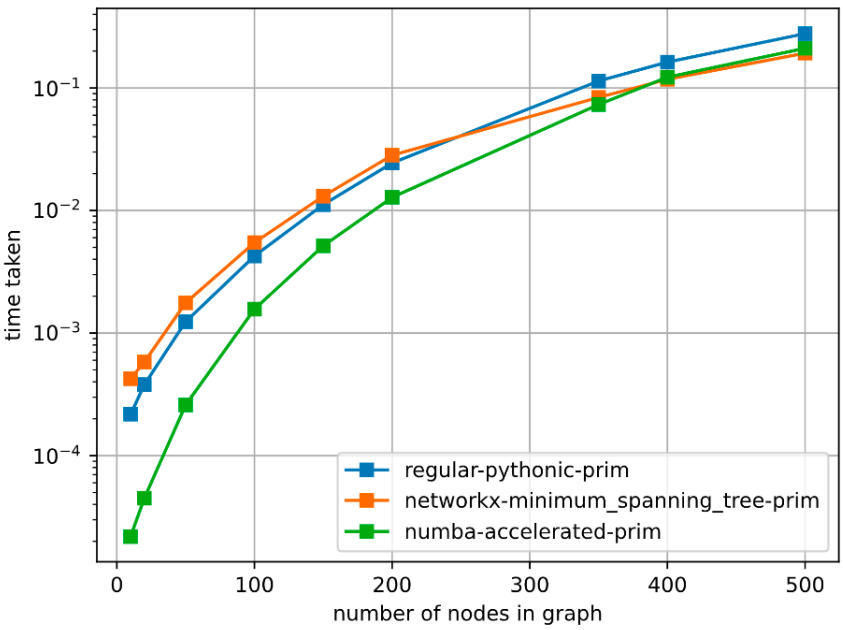

Figure 21: The timing of various implementation for varying input sizes

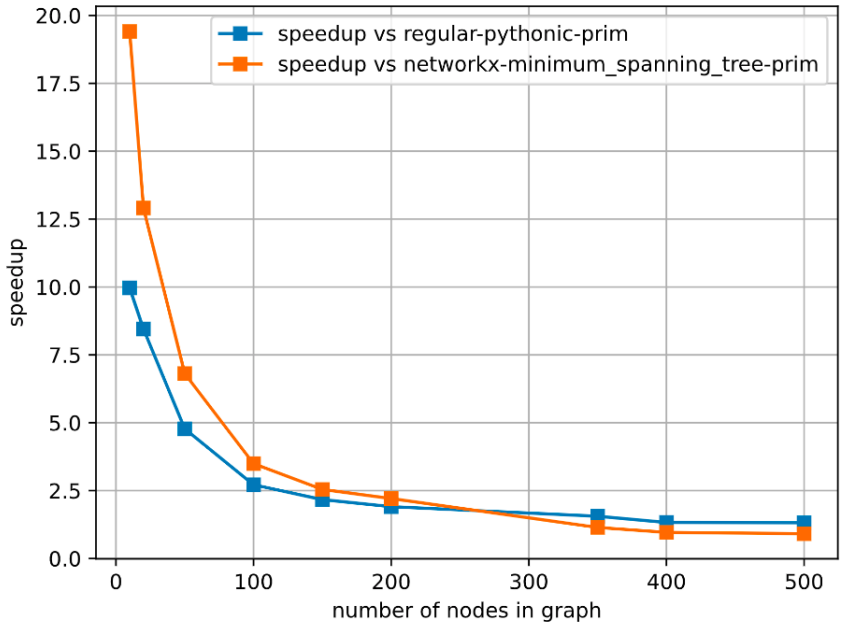

Figure 22: The speedup obtained w.r.t numba accelerated for the other two implementations

As can be seen, the numba speedup over networkx is very high (around 20) for small number of nodes and around 10 over the pythonic implementation. However, as the number of nodes increases, the speedup decreases.

6 Logic Simulation

6.1 Need for Logic Simulation in VLSI

- A logic simulator is a software tool that is used to simulate the behavior of digital circuits, they are used to verify the correctness of digital circuits before they are physically implemented.
- It allows one to test a digital circuit design against a set of test vectors, which are input signals that are used to exercise the circuit and study its outputs for correctness.
- Using a logic simulator, one can detect and fix design errors before the circuit is fabricated, saving both time and money.
- One can analyze its performance and identify potential bottlenecks or areas for improvement.

6.2 Hardware Description Language: OptiVLSIHDL

- A simulator for digital circuits is a software tool that allows you to simulate and test a digital circuit design, typically described using a hardware description language (HDL) such as **Verilog** or **VHDL**
- For the sake of this implementation, we have created our own hardware description language **OptiVLSIHDL** which is similar to **NGSPICE**, sufficient to describe any combinational digital circuit. Refer the bottom code snippet with examples on how to describe circuits

```
* this is a comment

* inp <nodes seperated by spaces> denotes input nodes to circuit

inp n1 n2 n4 n6 x1 x2 x3 x4 x5 x6


* outp <nodes seperated by spaces> denotes output nodes to circuit

outp n3 n5 n7 o1 o2 o3


and n1 n2 n3
```

```
* denotes and gate with inputs n1 and n2, output as n3


or n3 n4 n5

* denotes or gate with inputs n3 and n4, output as n5


inv n6 n7

* denotes inverter with input n6 and output n7

nand x1 x2 o1

* denotes nand gate with inputs x1 and x2, output as o1

nor x3 x4 o2

* denotes nor gate with inputs x3 and x4, output as o2

xor x5 x6 03

* denotes xor gate with inputs x5 and x6, output as o3
```

6.3 Compiled Code Simulator: Implementation

6.3.1 Script for generating n-input AND gate benchmarks

- For the generation of benchmark circuits we have the `benchmark_create.py` file that generates a OptiVLSIHDL description of a n input AND gate made of 2 input AND gates.

  Usage Instructions
  `python3 benchmark_create.py -n powerof2 -c circuit`

- `-n/--pow2` is the number of inputs to the generic n-input AND gate implemented by cascading 2 input and gates in a tree like structure, the closest power of two is picked.

- `-c/--circuit` is the description of the circuit saved in `circuit.txt`.

6.3.2 Main Script Containing Implementations

- The file `compiled_code_sim.py` contains a regular pythonic implementation of a compiled code simulator which generates a truth table of the simulated logic. The file `compiled_code_sim_numba.py` contains its numba accelerated version.

Usage Instructions

```
python3 [compiled_code_sim.py | compiled_code_sim_numba.py] --b (optional) -c
circuitfile -t truthtablefile
```

- `-t/--truthtable` is the filename to store the truth table of the logic simulated, it will be stored in `truthtablefile.csv`
- `-c/--circuit` is the input filename with description of digital circuit, it should be stored in `circuitfile.txt`

6.4 Compiled Code Simulator: Results and Inferences

The truthtable generated by the simulator after taking as input the description of a full adder circuit is below.

```
python3 compiled_code_sim.py -c benchmarks/fulladder -t fulladder --b

  * OptiVLSIHDL description of a full adder
  inp A B C
  outp Sum Cout
  xor A B 1
  xor 1 C Sum
  and 1 C 2
  and A B 3
  or 2 3 Cout
```

Table 1: Truth Table Obtained by Simulating Full Adder

A	B	C	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

To compare the timing performance of the numba accelerated vs regular pythonic implementation for benchmarks of various common digital circuits like 2x1 MUX, 4x1 MUX, Full Adder, 2 Bit Adder, n-input AND gate using `automan` implemented in `automate.py`

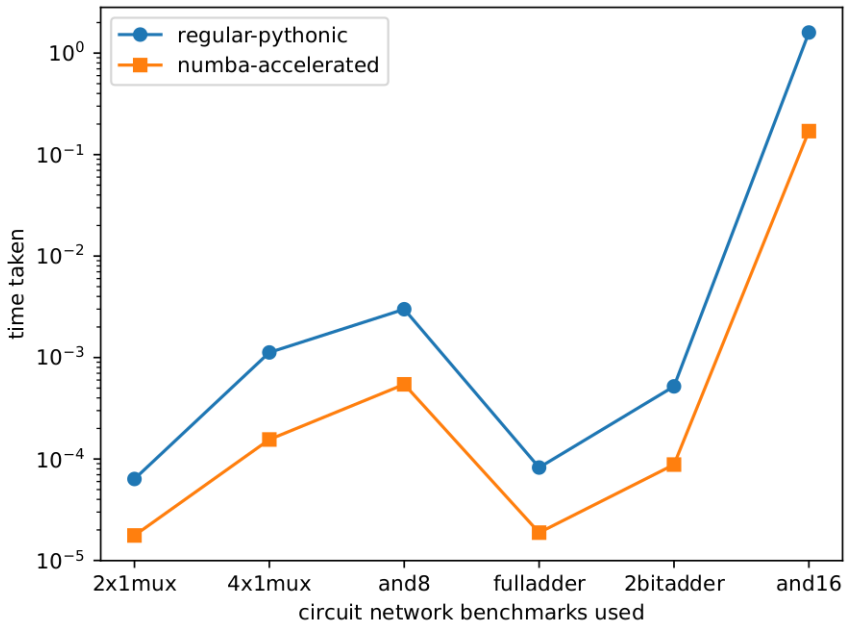

Figure 23: The timing of various implementations for different circuit benchmarks

We see that the numba accelerated code is an order of 10^1 times faster than the pythonic implementation, and the speedup scales for various circuit benchmarks.

6.5 Event Driven Simulator: Implementation

6.5.1 Script for generating n-input AND gate benchmarks

- For the generation of benchmark circuits we have the `benchmark_create.py` file that generates a OptiVLSIHDL description of a n input AND gate made of 2 input AND gates.

  Usage Instructions
  `python3 benchmark_create.py -n powerof2 -c circuit`

- `-n/--pow2` is the number of inputs to the generic n-input AND gate implemented by cascading 2 input and gates in a tree like structure, the closest power of two is picked.

- `-c/--circuit` is the description of the circuit saved in `circuit.txt`.

6.5.2 Main Script Containing Implementations

- The file `event-driven-sim.py` contains both the pythonic and numba accelerated implementations of an event-driven simulator which generates a truth table of the simulated logic.

Usage Instructions

```
python3 event-driven-sim.py --b (optional) -c circuitfile -t truthtablefile
```

- `-t/--truthtable` is the filename to store the truth table of the logic simulated, it will be stored in `truthtablefile.csv`
- `-c/--circuit` is the input filename with description of digital circuit, it should be stored in `circuitfile.txt`

6.6 Event Driven Simulator: Results and Inferences

The truthtable generated by the simulator after taking as input the description of a 2x1 Multiplexer is below.

```
python3 event-driven-sim.py -c circuit.txt -t truthtable --b

  * code to describe a 2x1 MUX
  inp S A B
  outp Z
  and A l m
  inv S l
  and S B n
  or m n Z
```

Table 2: Truth Table Obtained by Simulating 2x1 Multiplexer

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

We also compared the performance of numba over the pythonic implementation for various and tree circuits with 4, 8 and 16 inputs. The reuslts are as shown below.

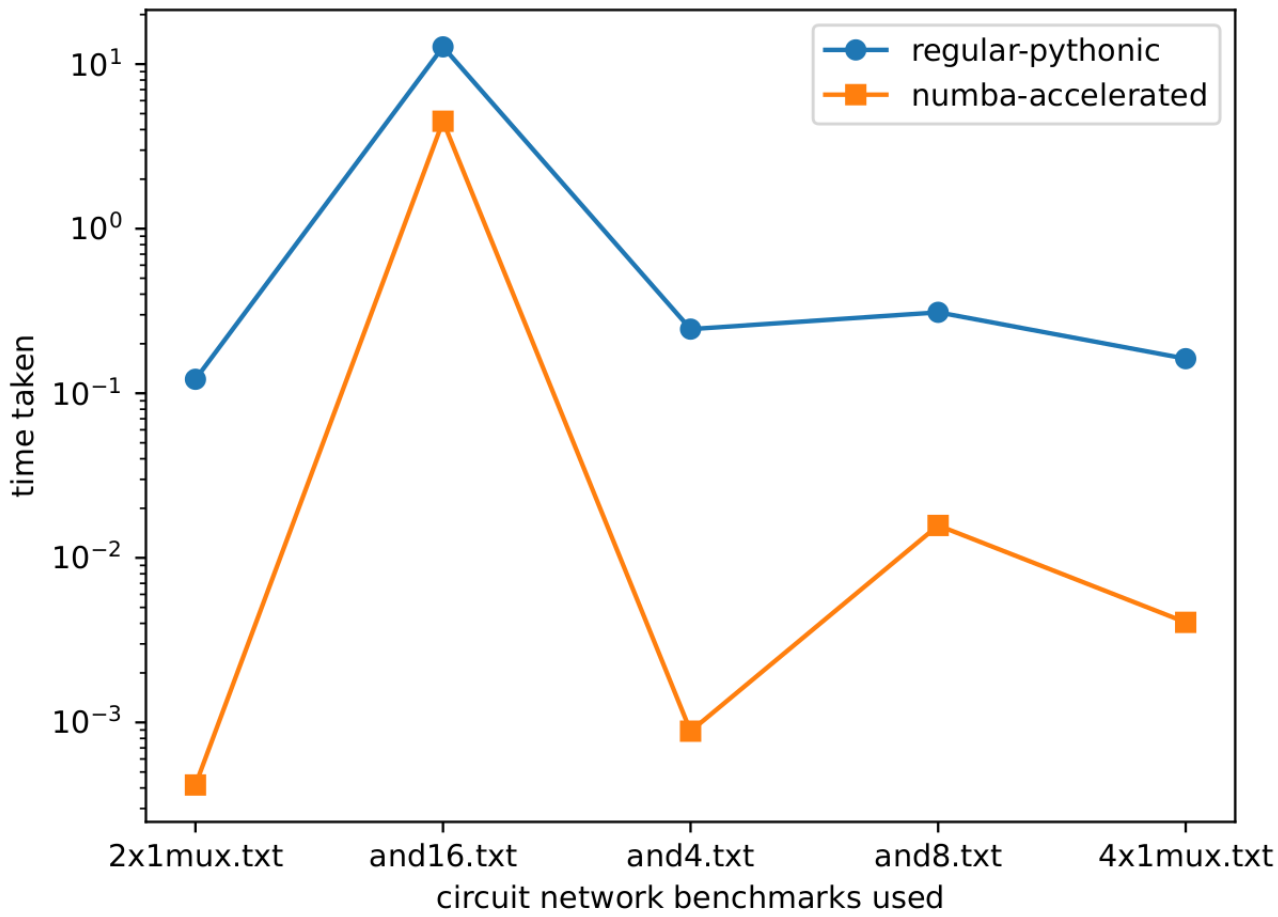

Figure 24: The timing of various implementations for different circuit benchmarks

As can be seen, the numba implementation is around 200 times faster than the pythonic version for small number of inputs. As the number of inputs increases, the speedup reduces as is evident from the time taken for running the and16 circuit.

7 Future Work

To involve all aspects of VLSI CAD, many more algorithms can be implemented in this library for different purposes. Apart from the algorithms that we have demonstrated, several more exist for getting the shortest path, spanning trees and also for automating physical design. Further, the current implementations of algorithms have been accelerated using numba. MPI and compyle can give us better performance compared to the current implementations and are improvements which can be looked into in the future. Finally, for better user knowledge, 3D

visualization of various algorithms can be carried out using mayavi and animating the process of VLSI CAD.

References

[1] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[2] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, p. 269–271, dec 1959.

[4] Aric, "Stackoverflow: How to create random single source random acyclic directed graphs in python." `https://stackoverflow.com/a/13546785`, Nov 2012. Accessed in Mar 2023.

[5] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[6] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," in *Proceedings of the American Mathematical Society, 7*, 1956.

[7] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.