

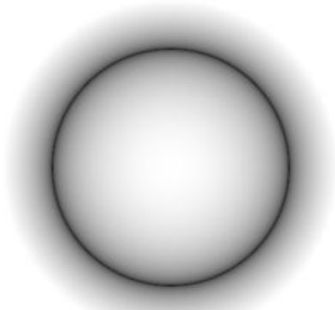
What is the Optimal Latency for Generating Signed Distance Field (SDF) given Convex Meshes Using the Ultra96v2 Board?

Team: Bo (boyings)

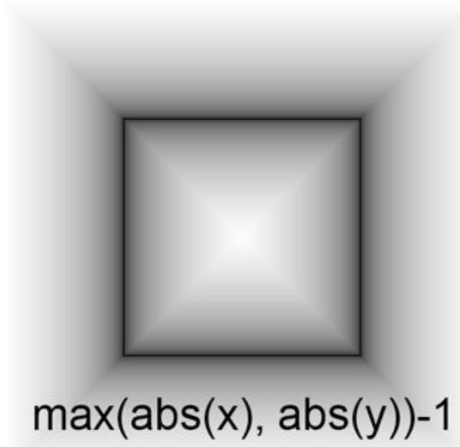
What is the Application? (1/3)

- Generate Signed Distance Field (SDF) given convex objects
- For each point in space, find the minimum distance to the closest mesh object surface

www.scratchapixel.com

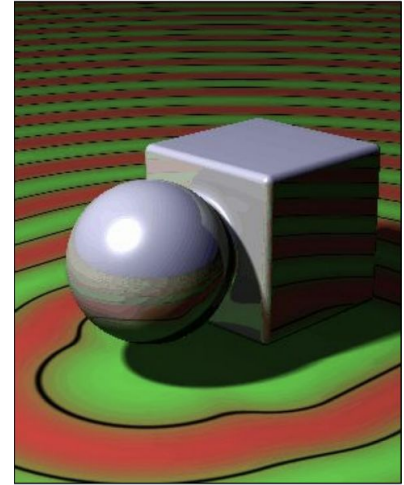


$$x^2 + y^2 - 1$$



$$\max(\text{abs}(x), \text{abs}(y)) - 1$$

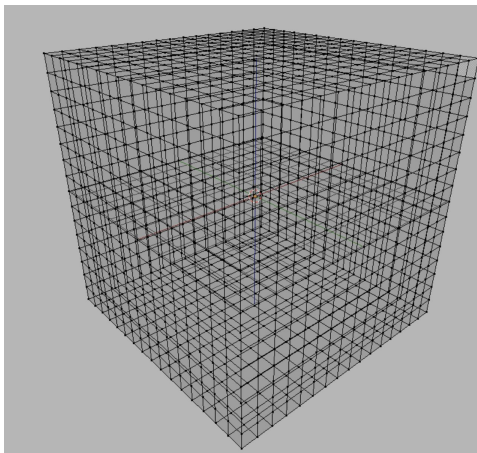
2D



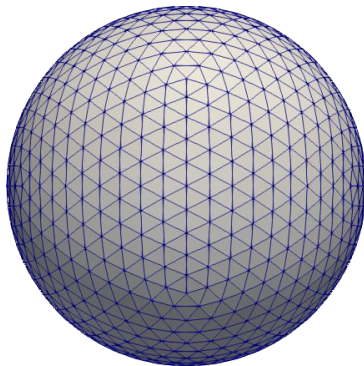
3D

What is the Application? (2/3)

- Input
 - Grid Positions (Nx3 array)
 - Graph Representation of Convex Meshes
- Output
 - Signed Distance at each Grid Positions

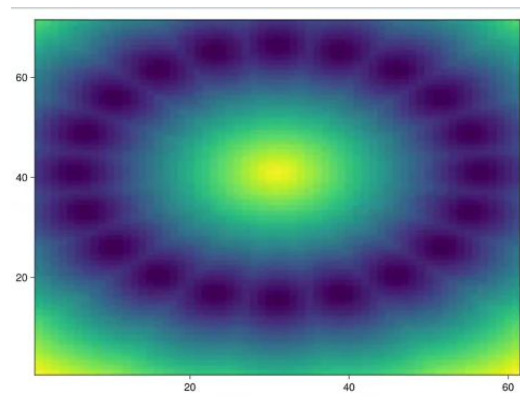


+



=

(At $z = 0$)



What is the Application? (3/3)

- Key Applications
 - Robot Motion Planning, Obstacle Avoidance
 - Computer Graphics

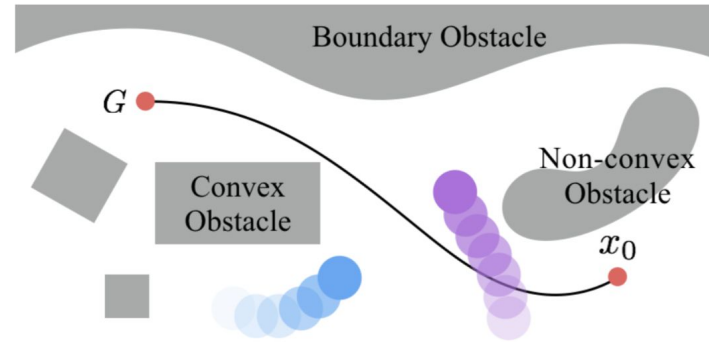
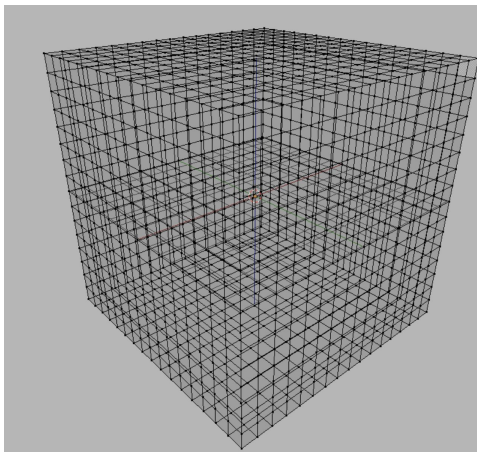


Fig. 7: The motion planning problem.

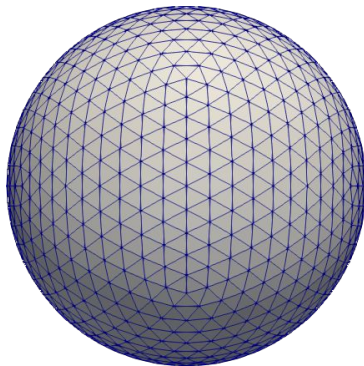
The Convex Feasible Set (CFS) motion planning algorithm requires Signed Distance and Convexified Object Meshes

Optimization Metric (1/1)

- End to end Latency
- A simple scene with a Sphere Mesh
 - 3442951 Grid Positions (151*151*151) in 3D
 - 1 Sphere Mesh with 162 vertices and 960 edges centered in the grid

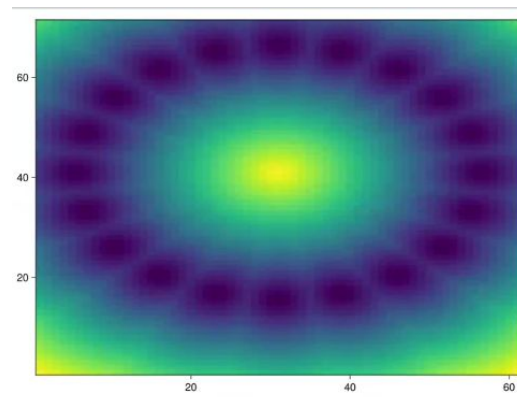


+



=

(At $z = 0$)



Design space (1/1)

1. Algorithm

- Naive or Hill Climb?

2. Local Memory

- Whether or not to load {Grid Position?, CSR of the Mesh?} to FPGA local memory?

3. Tiling

- Tile size for grid positions input and sdf output

4. Reordering Loops

- Traverse the Mesh Graph or iterate the tile in the inner or outer loop?

5. Pipelining

- Whether or not to pipeline the loop when iterating over vertex neighbors?

6. Unrolling

- # of threads processing a tile

7. Load Balancing

- Whether or not to load balance the threads over tile grid positions

8. Temporal Locality

- Whether or not to take advantage of temporal locality by warm-starting traversal using previous vertex

Tool and Platform (1/1)

- Hardware
 - Ultra96-V2 with AMD Xilinx Zynq UltraScale+™ MPSoC
- Software
 - Vitis HLS

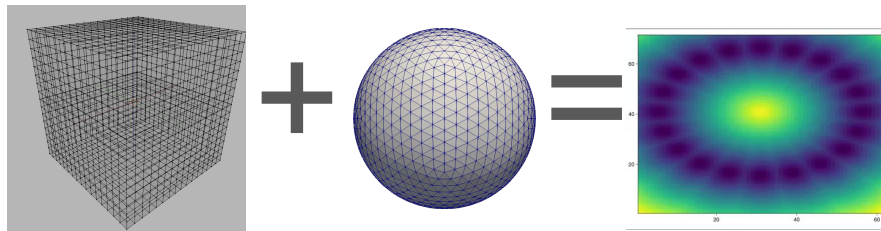
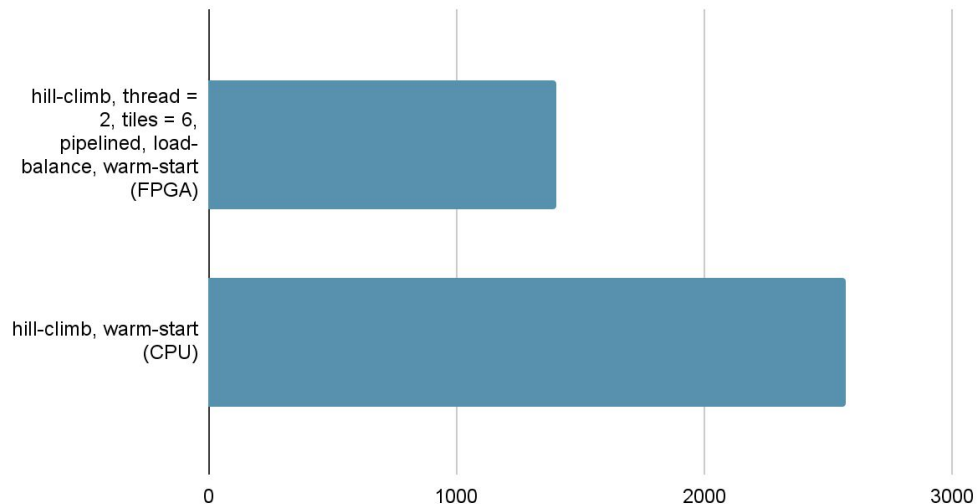
Resources

Resource	Total
BRAM	216
DSP	360
LUT	70560
FF	141120
REG	127214

What we learned

- What is the Optimal Latency for Generating Signed Distance Field (SDF) given Convex Meshes Using the Ultra96v2 Board?
 - 1406.481ms
- 45.4% lower latency on Ultra96v2 Board @ 150 MHz compared to sequential CPU @1.5 GHz

Latency (ms) <Lower the Better>



Work Performed

Design Point 1: Algorithm

What I did

Naive Solution

- For each grid position, compare it to every mesh vertices and take $\min()$

```
FOR each width value w from 0 to grid_w
  FOR each height value h from 0 to grid_h
    FOR each depth value d from 0 to grid_d
      Initialize sdf to a very large value (e.g., 999999)
      FOR each vertex i from 0 to v_num
        Initialize distance dist to 0
        FOR each dimension j from 0 to 2 (representing x, y, z)
          Get vertex position vtx from vertices_list
          Get grid position gd from grid_positions
          Calculate offset off as the difference between gd and vtx
          Accumulate the squared offset in dist
        END FOR
        Update sdf to be the minimum of itself and dist
      END FOR
      Store sdf in sdf_list
    END FOR
  END FOR
END FOR
```

Embarrassingly parallel
over mesh vertices and grid

vs

Hill Climb Solution

- For each grid position, follow the $\min()$ of neighbor vertex

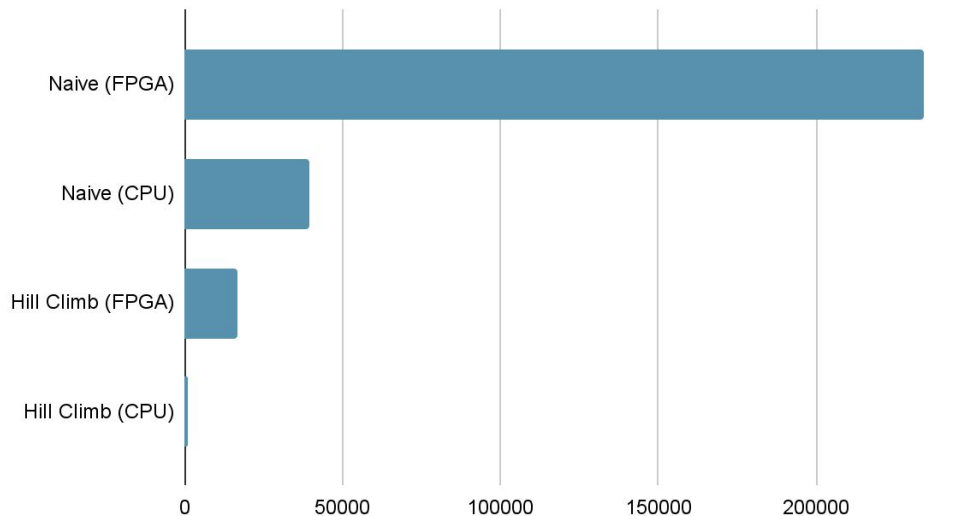
```
Reset x to 0
Set improving to true
WHILE improving is true
  Set prev to x
  FOR each neighbor i from vertex_adj_list[prev] to vertex_adj_list[prev + 1]
    Get neighbor index y from adj_list[i]
    Reset dist to 0
    FOR each dimension j from 0 to 2
      Get vertex position vtx from vertices_list at position (3 * y + j)
      Get grid position gd from grid_positions at same position as before
      Calculate offset off as before
      Accumulate the squared offset in dist
    END FOR
    IF dist is less than sdf
      Set sdf to dist
      Update x to y
    END IF
  END FOR
  IF x equals prev
    Exit the while loop
  END IF
END WHILE
Store sdf in sdf_list at position (w * grid_h * grid_d + h * grid_d + d)
```

Strictly sequential over mesh vertices
Parallelizable over grid

How does the metric respond?

- Hill Climbing Significantly faster on both CPU and FPGA
- The gap between Naive and Hill Climb on CPU (39x) is larger than on FPGA (14x)
- Hill Climb on FPGA is faster than Naive on CPU

Latency (ms) <Lower the Better>

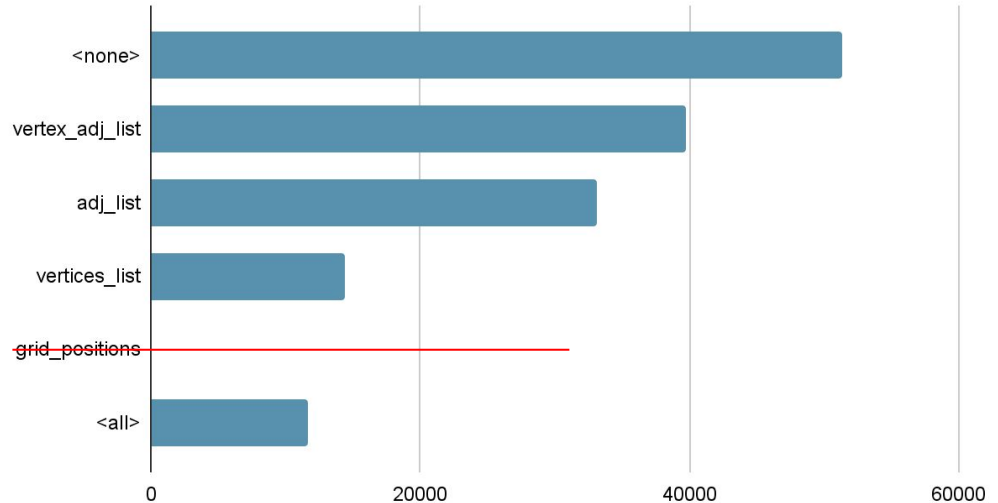


Design Point 2: Local Memory

How does the metric respond? (loading on-chip)

- Loading the entire mesh to FPGA at the beginning gives very significant speedup (4.4x)
- Grid positions are too large to fit on FPGA
- Loading Vertices to FPGA is most significant, as this array is randomly-accessed in each iteration and the position vector being 3 floats makes accessing this very memory intensive

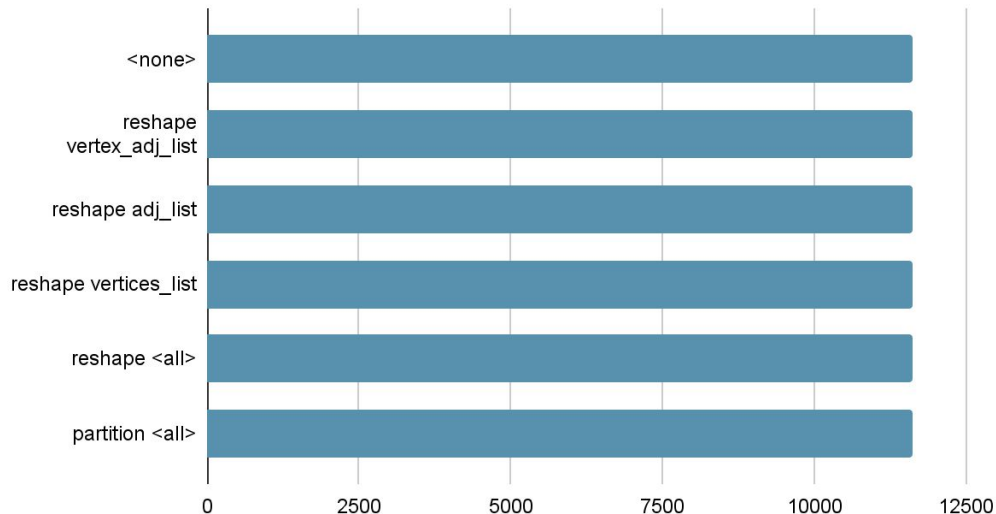
Latency (ms) <Lower the Better>



How does the metric respond? (Reshaping)

- We observed no improvements in latency after reshaping the arrays. This is due to the purely sequential code incurring resource conflicts.

Latency (ms) <Lower the Better>



Design Point 3: Tiling

What I did

- Tile the outermost loop such that a tile size of Grid Position Vectors are loaded to on-chip memory in batch
- Tile the outermost loop such that writing SDF to ram is buffered on on-chip memory and written to ram in batch

```
for (int nt = 0; nt < numgrid; nt += t_num * numtile) {  
    // ignore last loop  
    if (nt + t_num * numtile > numgrid) {  
        break;  
    }  
#pragma HLS PIPELINE II=1 rewind  
    // load from memory  
    for (int t = 0; t < t_num * numtile; t++) {  
        for (int j = 0; j < 3; j++) {  
            grid_position1[t][j] = grid_positions[(nt + t) * 3 + j];  
        }  
    }  
}
```

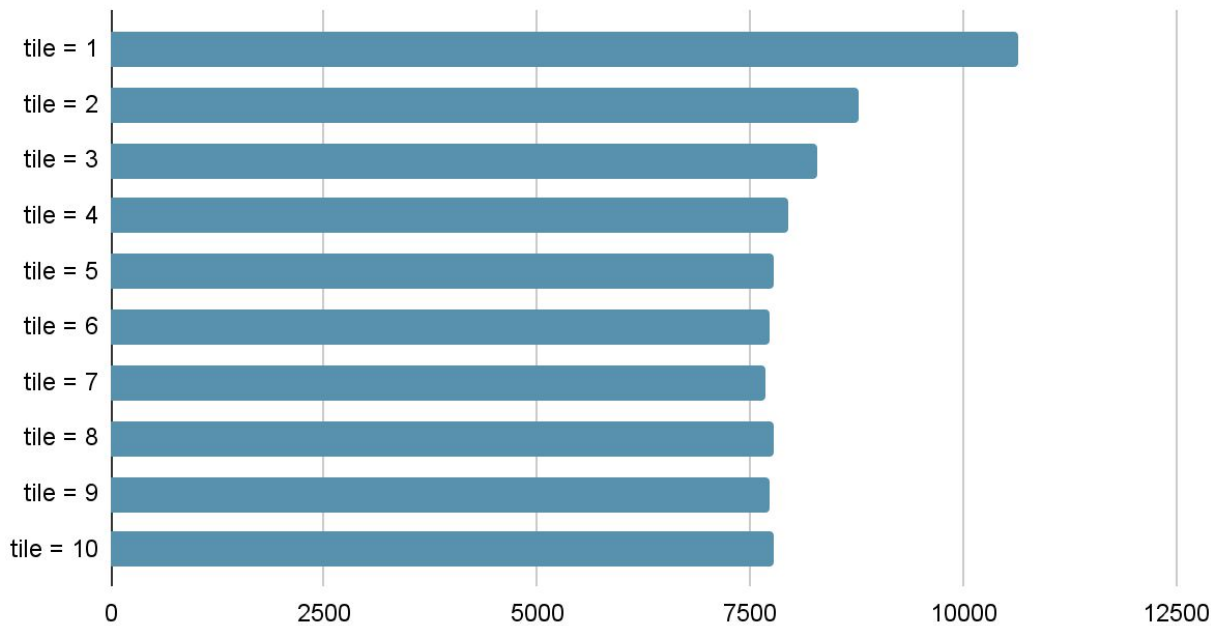
⋮

```
// store to memory  
for (int t = 0; t < t_num * numtile; t++) {  
#pragma HLS PIPELINE II=1 rewind  
    sdf_list[nt + t] = result0[t];  
}  
}
```

How does the metric respond?



- Tiling the reads and writes from and to memory help the latency a lot
- The improvement plateaued at tile size = 5
- In each cycle, we access $3 \text{ floats} * \text{tile size}$, which the $3 * 4 * 5 = 60$ bytes, just a little less than the AXI burst size

Latency (ms) <Lower the Better>



Design Point 5: Pipelining

Pipelining tiles? Pipelining neighbors?

- Tile loop 
- In each iter step for a vertex, we have to loop over its neighbors to find the best next vertex 
- Should we pipeline these 2 loops?

```
for (int ni = 0; ni < numtile;) {  
#pragma HLS PIPELINE II=1
```

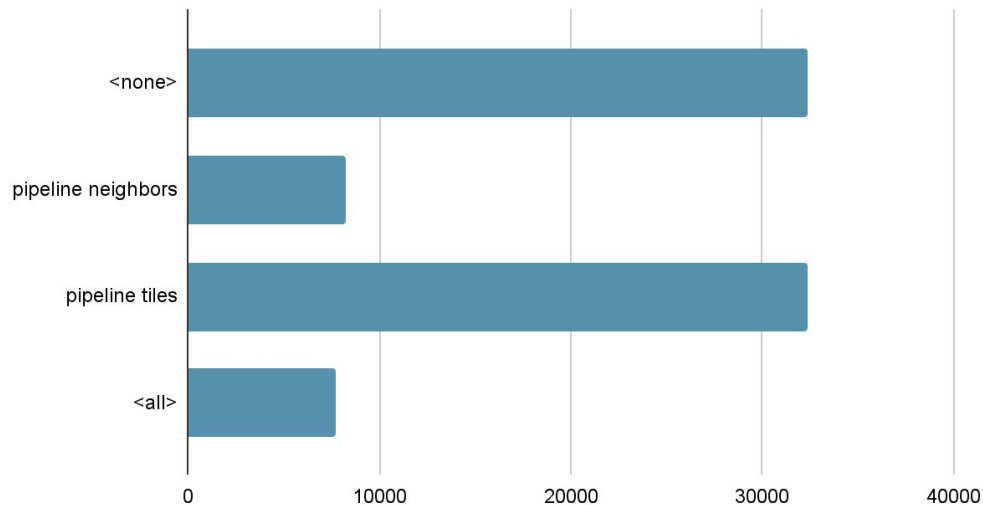
```
idxdata_t neighborstart = local_vertex_adj_list[xp];  
idxdata_t vtxmax = neighborstart + f_num;  
idxdata_t neighborend = MIN(vtxmax, local_vertex_adj_list[xp + 1]);
```

```
for (int i = neighborstart, k = 0; i < neighborend; i++, k++) {  
#pragma HLS PIPELINE II=1  
    sdfdata_t dist = (local_vertices_list[local_adj_list[i]][0] - p[0]) *  
                    (local_vertices_list[local_adj_list[i]][0] - p[0]) +  
                    (local_vertices_list[local_adj_list[i]][1] - p[1]) *  
                    (local_vertices_list[local_adj_list[i]][1] - p[1]) +  
                    (local_vertices_list[local_adj_list[i]][2] - p[2]) *  
                    (local_vertices_list[local_adj_list[i]][2] - p[2]);  
  
    if (dist < d) {  
        d = dist;  
        x = local_adj_list[i];  
    }  
}
```

How does the metric respond?

- No effect in pipelining tiles
- Significant latency improvements after pipelining neighbor loop

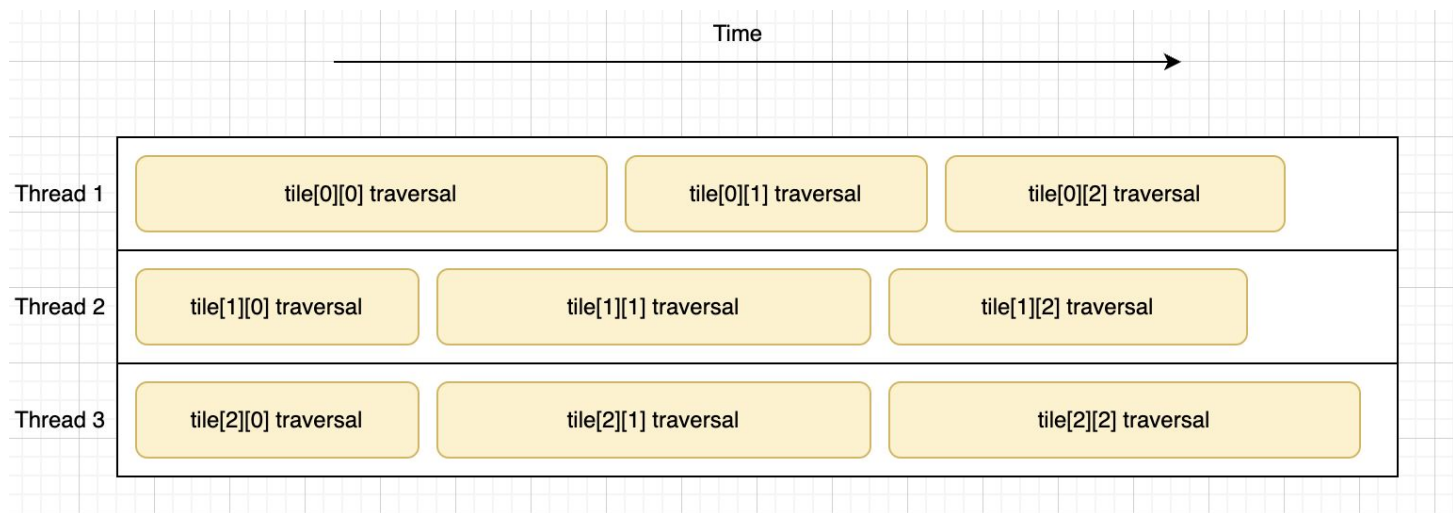
Latency (ms) <Lower the Better>



Design Point 6: Unrolling

What I did

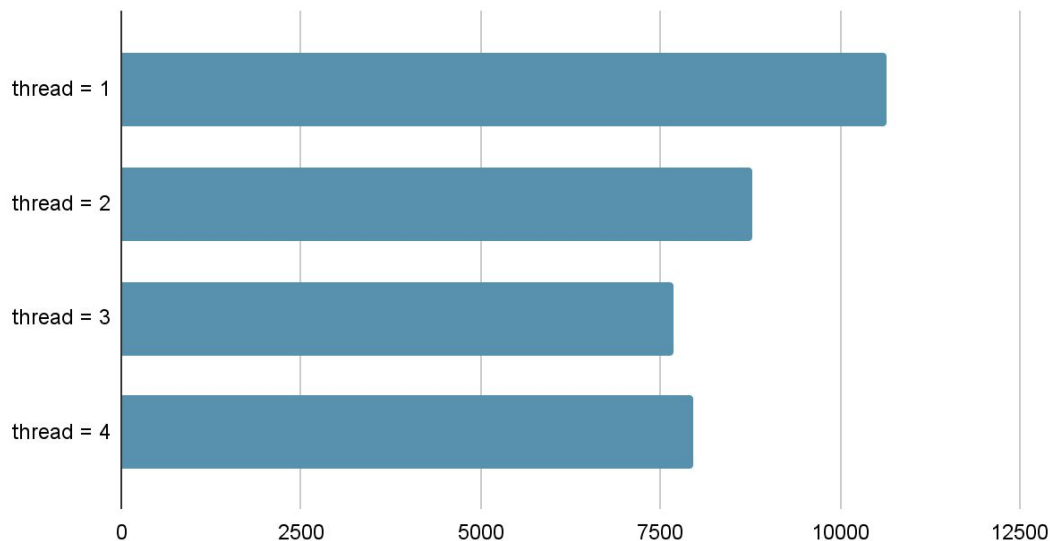
- Create a fixed # of “threads”
- Each thread processes a different set, but same # of tiles at each outer loop



How does the metric respond? (1/2)

- More threads gives significant latency improvements
- Plateaued at 3 threads
- Limited by LUT utilization, have to find a balance between # of threads and tile size

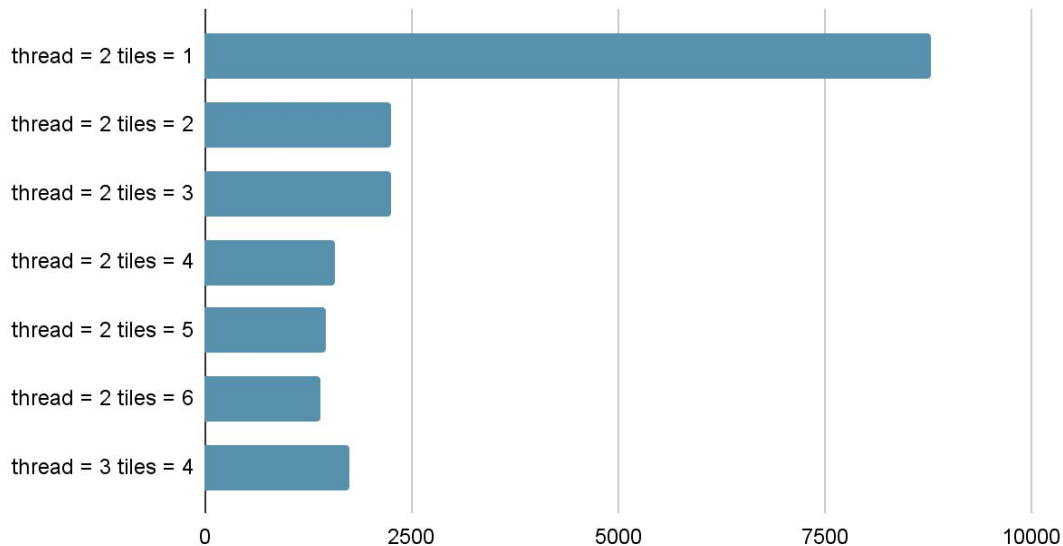
Latency (ms) <Lower the Better>



How does the metric respond? (2/2)

- Either # of threads and tile size can give latency improvements
- But they compete LUT resources.

Latency (ms) <Lower the Better>

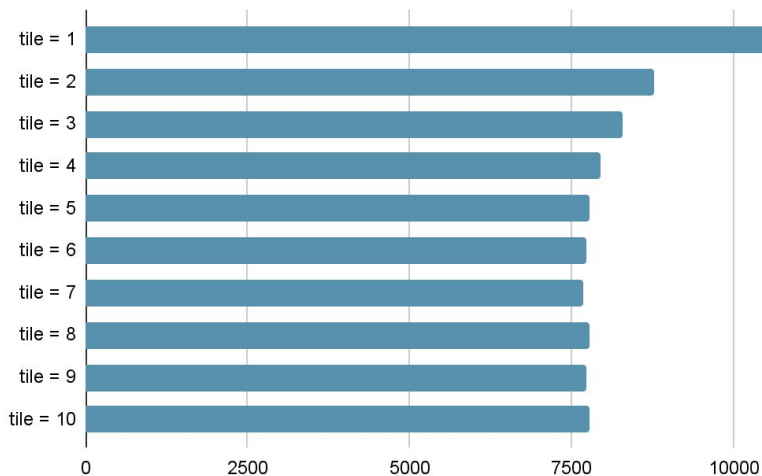


What is optimally achievable?

- # of threads = 2
- Tile size = 6

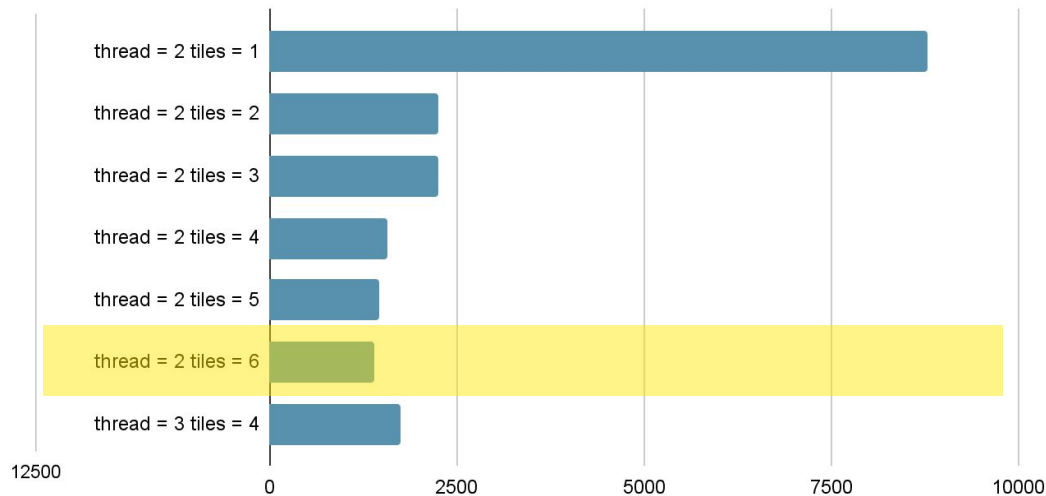
Threads = 1

Latency (ms) <Lower the Better>



Threads = 2+

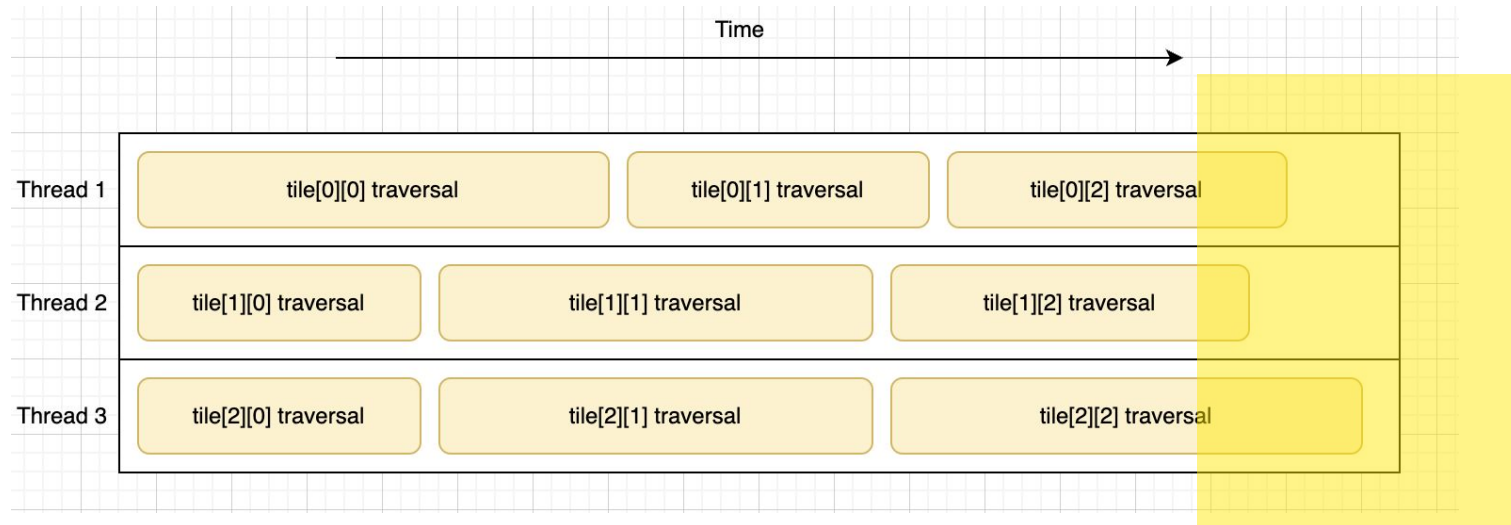
Latency (ms) <Lower the Better>



Design Point 7: Load Balancing

What I did (1/2)

- Having multiple thread processing is good
- However, the # of iterations for each grid position is unknown, causing early finished threads to stall.



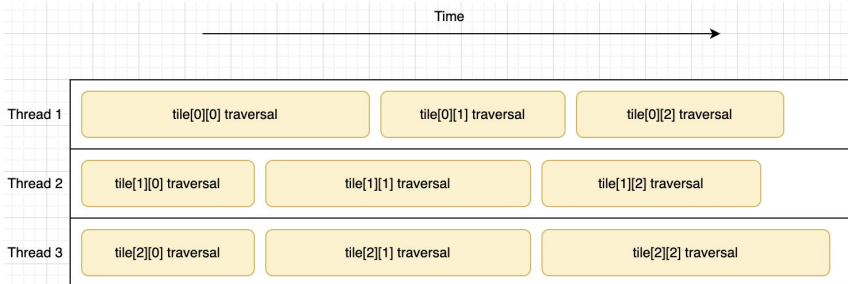
What I did (2/2)

- Multi-consumer queue
- For each thread t

```
idxdata_t tileidx = 0;
idxdata_t tileremaining = t_num * numtile;
idxdata_t tilecnts = t_num * numtile;
while (1) {
#pragma HLS PIPELINE II=1
    if (tileremaining == 0) {
        break;
    }
    for (int t = 0; t < t_num; t++) {
#pragma HLS UNROLL
        if (g[t] == -1 && tileidx < tilecnts) {
            idxdata_t assigned = tileidx++;
            g[t] = assigned;
            //
            p[t][0] = grid_position1[assigned][0];
            p[t][1] = grid_position1[assigned][1];
            p[t][2] = grid_position1[assigned][2];
        }
        xp[t] = x[t];
    }
}
```

Loop until all tile idx done

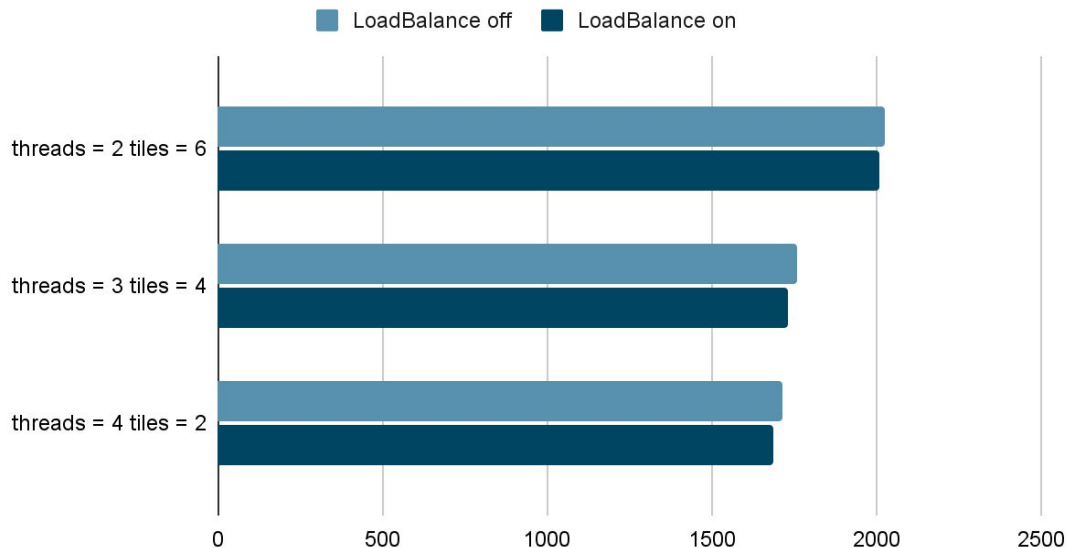
Assign next unprocessed
grid position



How does the metric respond?

- Latency improvements by enabling Load Balancing is consistent over # of threads, although being small
- Tile size is maximized while not exceeding the resource limits

Latency (ms) <Lower the Better>



Important outcome

(Design Point 4)

Loop Reordering

- This actually does not improve latency, but is an important design decision
- Given the loaded tiles, do we
 - For each tile, traverse the graph
 - or
 - For each traversal step, loop over tiles

```
Initialize x to 0
FOR each tile
  Initialize dist to 0
  Set sdf to dist
  Initialize prev
  Reset x to 0
  Set improving to true
  WHILE improving is true
    Set prev to x
    FOR each neighbor
      ...
    END FOR
    IF x equals prev
      Exit the while loop
    END IF
  END WHILE
  Store sdf in sdf_list
END FOR
```

```
Initialize x to 0
Initialize dist to 0
Set sdf to dist
Reset x to 0
Initialize prev
WHILE not all tiles done
  FOR each tile
    Set improving to true
    Set prev to x
    FOR each neighbor
      ...
    END FOR
    IF x equals prev
      Set tile_done to true
    END IF
  END FOR
END WHILE
Store sdf in sdf_list
```


Why is it Better?

- Given the loaded tiles, if we
 - For each tile, traverse the graph
 - Due to the sequential nature of the graph traversal, data dependency occurs in the while loop (expand)

```
Initialize x to 0
FOR each tile
  Initialize dist to 0
  Set sdf to dist
  Initialize prev
  Reset x to 0
  Set improving to true
  WHILE improving is true
    Set prev to x
    FOR each neighbor
      ...
    END FOR
    IF x equals prev
      Exit the while loop
    END IF
  END WHILE
  Store sdf in sdf_list
END FOR
```

Why is it Better?

- Given the loaded tiles, if we
 - For each traversal step, loop over tiles
 - Each tile is data independent, the operations can be pipelined or unrolled
 - Only thing we have to worry about is array partitioning/reshaping

```
Initialize x to 0
Initialize dist to 0
Set sdf to dist
Reset x to 0
Initialize prev
WHILE not all tiles done
  FOR each tile
    Set improving to true
    Set prev to x
    FOR each neighbor
      ...
    END FOR
    IF x equals prev
      Set tile_done to true
    END IF
  END FOR
END WHILE
Store sdf in sdf_list
```

Why is it Better?

- Disadvantages of
 - For each traversal step, loop over tiles
 - Number of traversal steps is unknown and is different across tiles
 - Outer loop iters # of times = the longest path among all tiles
 - However, this can be mitigated by load balancing described later
- Decision
 - For each traversal step, loop over tiles

```
Initialize x to 0
Initialize dist to 0
Set sdf to dist
Reset x to 0
Initialize prev
WHILE not all tiles done
  FOR each tile
    Set improving to true
    Set prev to x
    FOR each neighbor
      ...
    END FOR
    IF x equals prev
      Set tile_done to true
    END IF
  END FOR
END WHILE
Store sdf in sdf_list
```

Limitations and Weaknesses

Limitations

- We did not perform grid search across the entire parameter space, instead we assume them to be independent, greedily picking the optimal parameter in each design point
- We ensure that experiments in each degrees of freedom are perform in the same environment (temperature, cooling etc.). Not across experiments from the other DOF

Weaknesses

- We haven't explored how meshes of different geometry affect the design points
- We haven't explored how the order of grid_positions being computed affect the design points
- We didn't push the FPGA to a higher clock frequency, instead we fix it to 150MHz
- We didn't explore how more DRAM ports improve the latency

Artifacts

Vitis HLS Kernel

Details in speaker notes

Summary

- What is the Optimal Latency for Generating Signed Distance Field (SDF) given Convex Meshes Using the Ultra96v2 Board?
 - 1406.481ms
- 45.4% lower latency on Ultra96v2 Board @ 150 MHz compared to sequential CPU @1.5 GHz

Latency (ms) <Lower the Better>

