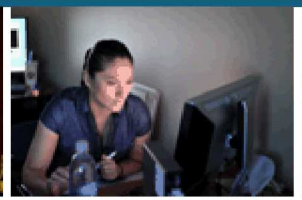
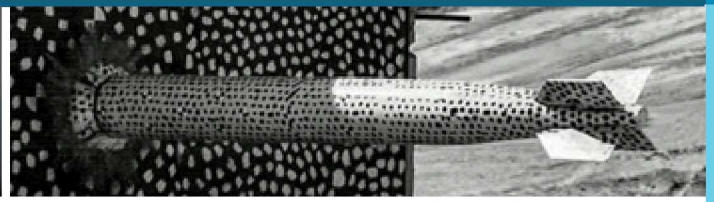
SAND2020-10733PE

Evaluation of oneAPI for FPGAs





Intern: Nicholas Miller

Mentor: Clayton Hughes

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

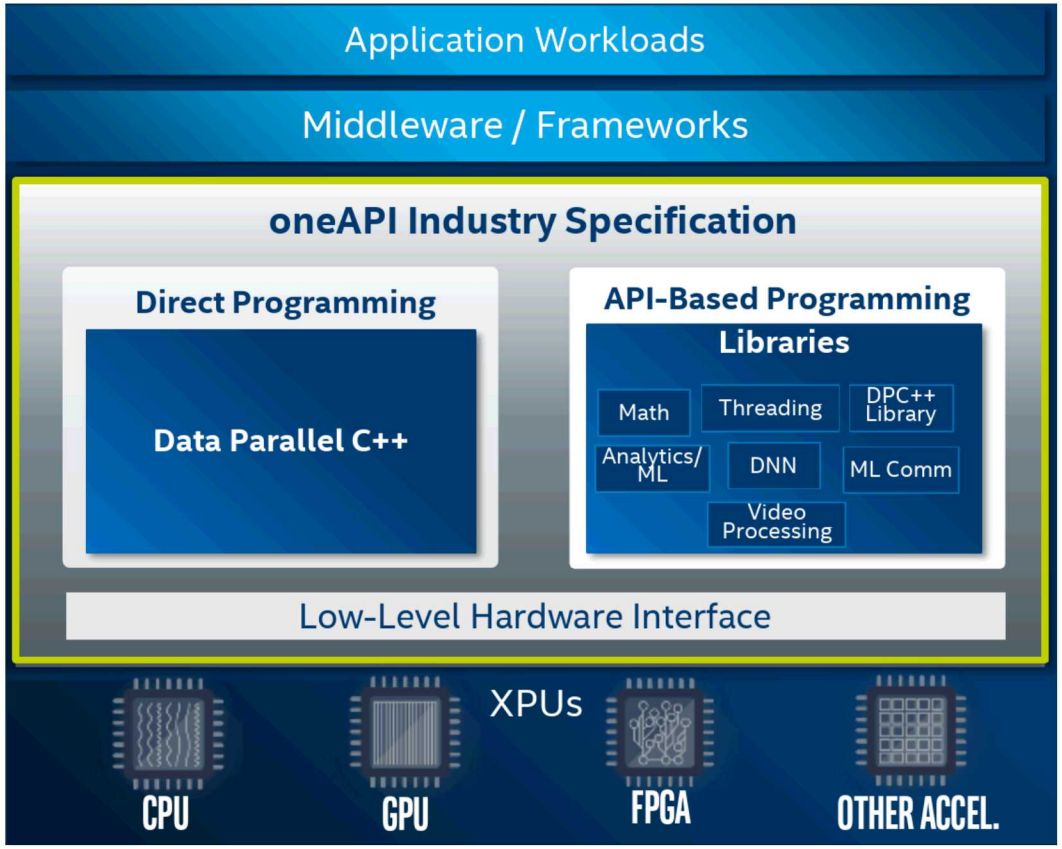
Outline

- Introduction to oneAPI
  - What is oneAPI?
  - Summer project goals
- Programming
  - DPC++ data movement
  - SYCL runtime overheads
  - Hardware configuration
- Results
  - Execution time
  - Device utilization
  - Compilation times

Introduction to oneAPI

What is oneAPI

- oneAPI is a framework for programming different accelerators using a single language
- Programming language is DPC++ which is an addition to SYCL
- Can be used to program GPU, CPU, and FPGA
- Includes libraries to accelerate certain applications
- oneAPI is an open specification
- In beta08 as of 9/10/20



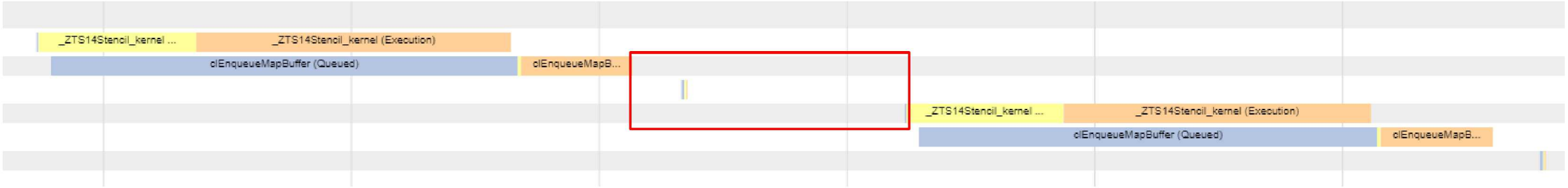
Summer Work

- Goal was to evaluate oneAPI for programmability and performance
- Studied documentation to understand the DPC++ language
- Ported the MiniAMR proxy application to DPC++ and measured its performance on an Arria 10 development board
- Explored DPC++ constructs for performance and area optimization

Programming

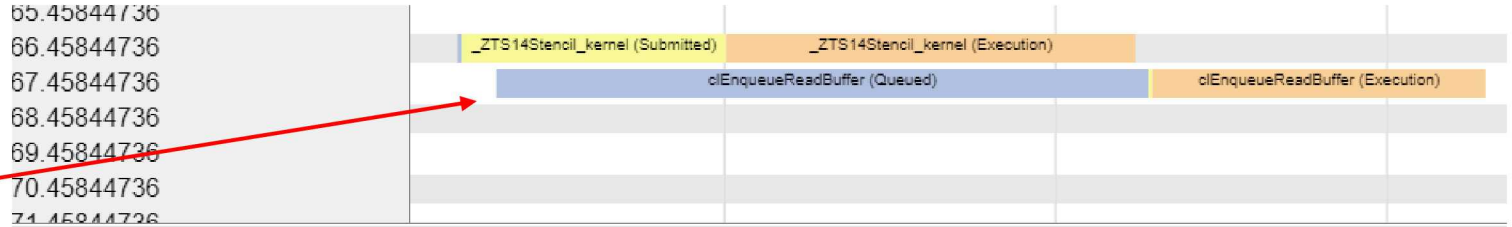
Data Movement

- Two options for data movements:
  - Buffers
    - Can be created with a host pointer to make data transfers easier
  - Unified Shared Memory (USM)
- Dependency tree created by compiler to find where synchronization events are needed during kernel executions and host code execution
- Explicit memory movement can be done through destructors or function calls like cgh.update_host(accessor);
  - Can cause unintended issues with execution like below where the executions should look like the buffered example



SYCL Runtime Overheads

- Calling the SYCL runtime to submit a task to the FPGA has high overheads
- Queue event is created at buffer destructor to read the FPGA buffer
- Queue event is blocking so processor side execution waits



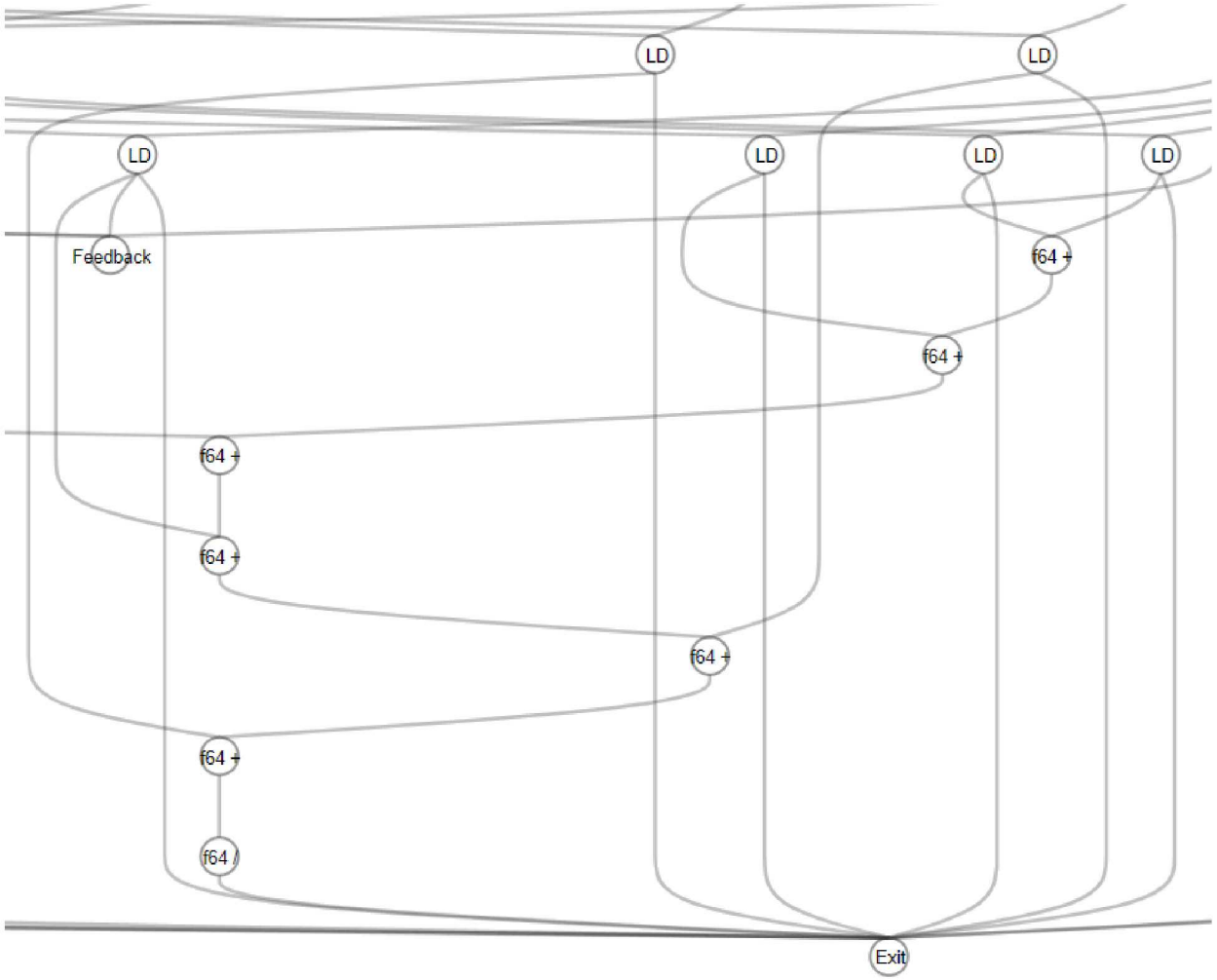
6 items selected. Slices (6)

Name	Wall Duration	Self time	Average Wall Duration	Occurrences
_ZTS14Stencil_kernel (Queued)	0.007 ms	0.007 ms	0.007 ms	1
_ZTS14Stencil_kernel (Submitted)	0.398 ms	0.398 ms	0.398 ms	1
_ZTS14Stencil_kernel (Execution)	0.617 ms	0.617 ms	0.617 ms	1
clEnqueueReadBuffer (Queued)	0.984 ms	0.984 ms	0.984 ms	1
clEnqueueReadBuffer (Submitted)	0.006 ms	0.006 ms	0.006 ms	1
clEnqueueReadBuffer (Execution)	0.503 ms	0.503 ms	0.503 ms	1
Totals	**2.515 ms**	**2.515 ms**	**0.419 ms**	**6**

Selection start	**4,650.598 ms**
Selection extent	**1.552 ms**

Hardware Description

- Pipelines created automatically for hardware
- Load Stores Units inferred from memory access pattern
  - Different Types of LSU can be inferred
    - Burst Coalesced
    - Cached
- Will use DSPs when appropriate
  - For example f32 add operations
- Loops can be unrolled through compiler hints



Memory Description

- Memory types can be controlled by memory attributes
  - Example:

```
[[intelfpga::bankwidth(8), intelfpga::numbanks(2)]] int a[64];
```

- Allows for fine grain control of BRAM and register structures
- Cannot figure out how to specify DRAM usage yet
- If not specified, then compiler assumes structure automatically

Memory Attribute	Description
intelfpga::register	Forces a variable or array to be carried through the pipeline in registers.
intelfpga::memory("impl_type")	Forces a variable or array to be implemented as embedded memory. The optional string parameter impl_type can be BLOCK_RAM or MLAB.
intelfpga::numbanks(N)	Specifies that the memory implementing the variable or array must have N memory banks.
intelfpga::bankwidth(W)	Specifies that the memory implementing the variable or array must be W bytes wide.
intelfpga::singlepump	Specifies that the memory implementing the variable or array should be clocked at the same rate as the accesses to it.
intelfpga::doublepump	Specifies that the memory implementing the variable or array should be clocked at twice the rate as the accesses to it.
intelfpga::max_replicates(N)	Specifies that a maximum of N replicates should be created to enable simultaneous reads from the datapath.
intelfpga::private_copies(N)	Specifies that a maximum of N private copies should be created to enable concurrent execution of N pipelined threads.
intelfpga::simple_dual_port	Specifies that the memory implementing the variable or array should have no port that services both reads and writes.
intelfpga::merge("key", "type")	Merge two or more variables or arrays in the same scope width-wise or depth-wise. All variables with the same key string are merged into the same memory system. The string type can be either width or depth.
intelfpga::bank_bits(b_0,b_1,...,b_n)	Specifies that the local memory addresses should use bits (b_0,b_1,...,b_n) for bank-selection, where (b_0,b_1,...,b_n) are indicated in terms of word-addressing. The bits of the local memory address not included in (b_0,b_1,...,b_n) will be used for word-selection in each bank.

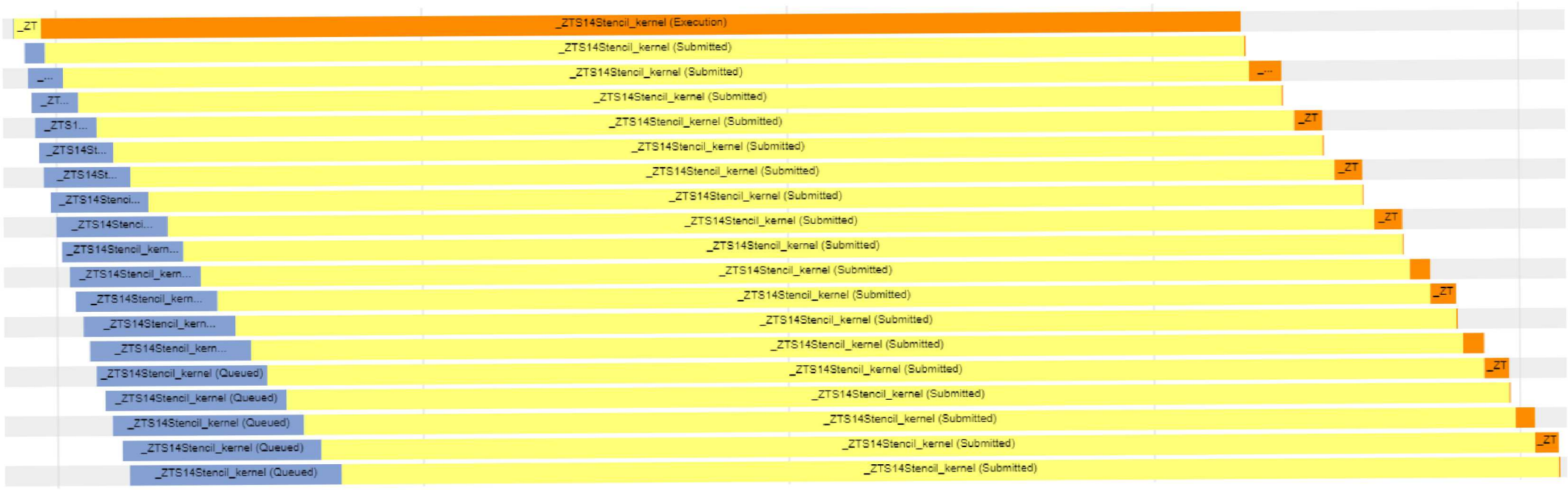
Results

Iterations of Design

- Combined Memory Transactions
  - The optimization that provided the largest performance boost was to combine all the variable computations in a block into a single communication and computation step
    - This reduced the number of calls to the SYCL runtime by 40x
- Reduced Local Memory
  - Only stores one variable (1000 elements) at a time in BRAM at any given time
- Flattened Arrays
  - Flattened all arrays to 1D accessors to make buffer creation and destruction faster
- Buffering
  - Buffers the SYCL runtimes to create overlap between kernel execution and command communication

Buffered SYCL runtime calls

- By calling the SYCL runtime for the kernel it queues and submits it while the FPGA works on the stencil calculation
- This works only if the execution of the kernel is long enough to cover up the SYCL runtime overheads

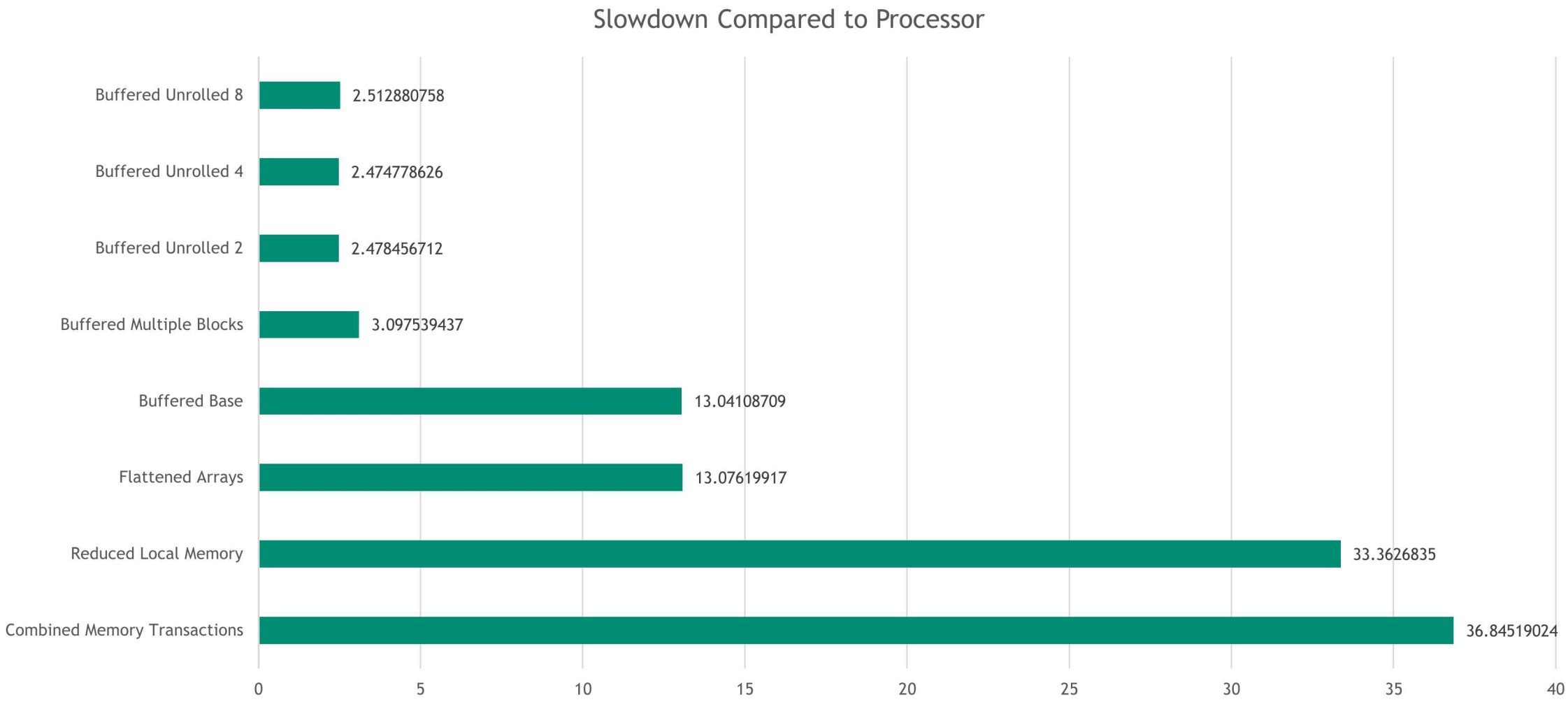


Experimental Setup

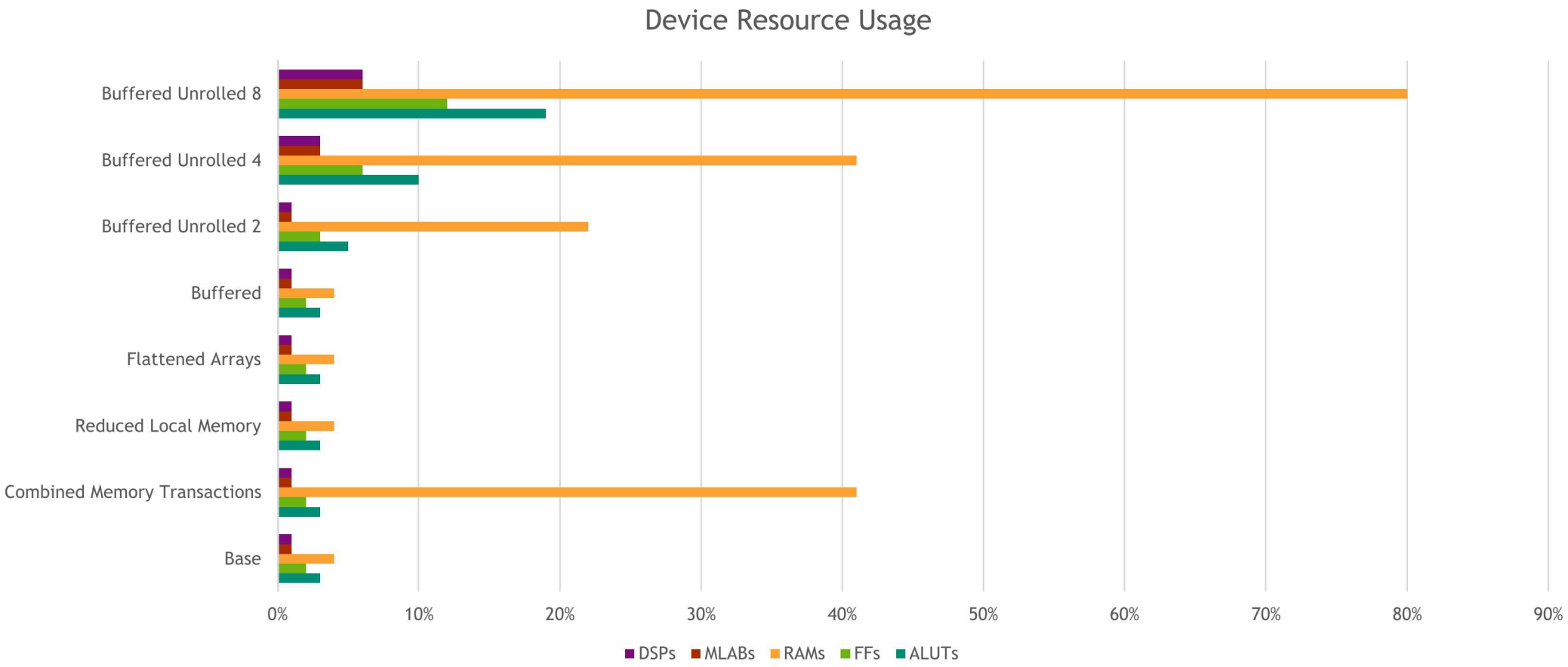
- Run on the Intel Devcloud system
- Submitted to node by OpenPBS scheduler
- Increased number of blocks in a run to compare for the buffering tests

CPU	2 x Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
FPGA Family	Arria 10
FPGA Device	10AX115S2F45I2SGES
System Memory	196 GB
Base Parameters	No Parameters
Increased Blocks Parameters	--num_refine 4 --max_blocks 9000 --num_objects 1 --object 2 0 -1.71 -1.71 -1.71 0.04 0.04 0.04 1.7 1.7 1.7 0.0 0.0 0.0 --num_tsteps 25

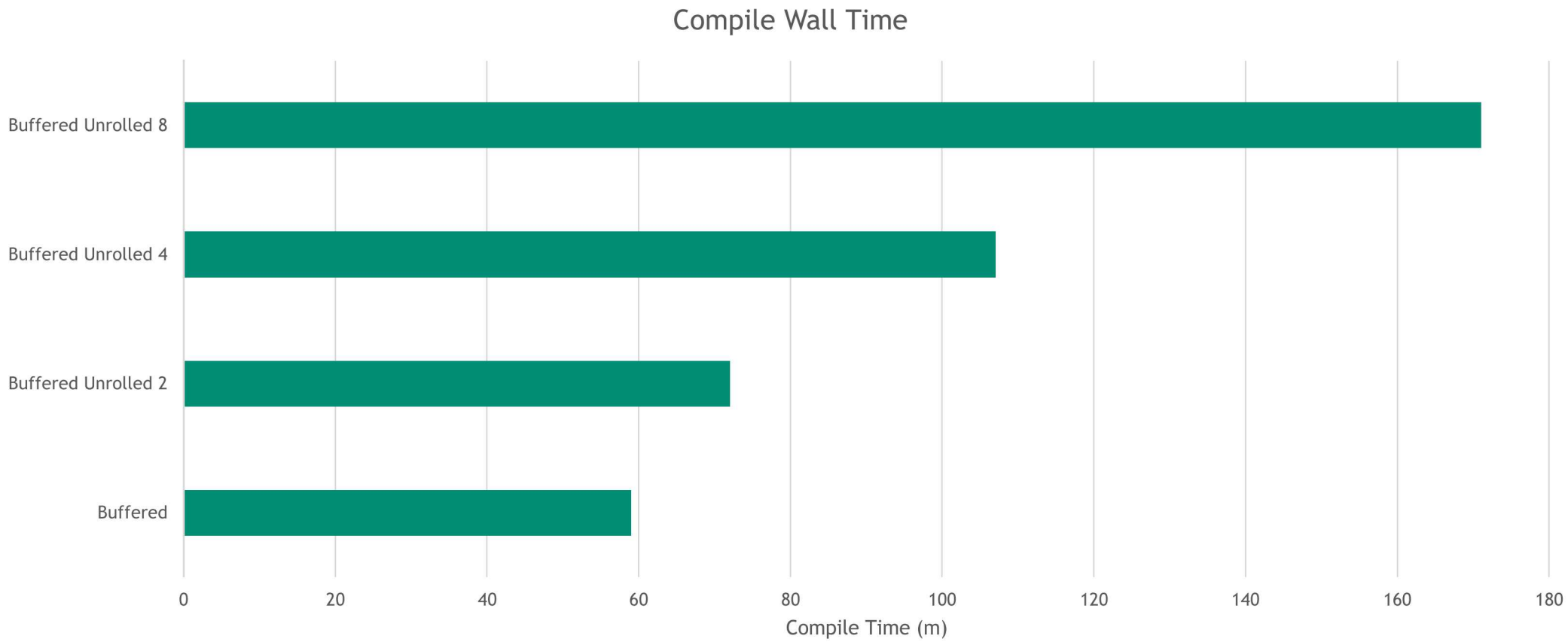
Performance



Utilization



Compile Times



Conclusions

- Greatly reduced development time and digital logic knowledge needed compared to HDL
- Best case design showed 2.4x slowdown compared to single core reference with ~22% BRAM usage
- Low transparency in device interactions
- Unable to fully customize hardware

Questions?

Appendix

Base Stencil

```
1  for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
2     bp = &blocks[sorted_list[in].n];
3     for (var = 0; var < var_max; var++) {
4        sycl::range<1> num_array{ static_cast<size_t>((x_block_size + 2) *
5           (y_block_size + 2) * (z_block_size + 2)) };
6        //create a buffer that goes to the fpga
7        double* inputArray = new double[(x_block_size + 2) *
8           (y_block_size + 2) * (z_block_size + 2)];
9        //create a buffer that comes from the fpga
10       double* outputArray = new double[(x_block_size + 2) *
11          (y_block_size + 2) * (z_block_size + 2)];
12       //flatten the 4d array to a 1d array for the buffer
13       for (int i = 0; i <= x_block_size + 1; i++)
14          for (int j = 0; j <= y_block_size + 1; j++)
15             for (int k = 0; k <= z_block_size + 1; k++)
16                inputArray[k + (z_block_size + 2) * (j + (y_block_size + 2) * i)]
17                   = bp->array[var][i][j][k];
18       sycl::buffer<double, 1> input_buffer(inputArray, num_array);
19       {
20          sycl::buffer<double, 1> output_buffer(outputArray, num_array);
21          fpga_kernel(input_buffer, output_buffer);
22       }//output_buffer detructor called here
23       //write the data back to the block array
24       for (int i = 1; i <= x_block_size; i++)
25          for (int j = 1; j <= y_block_size; j++)
26             for (int k = 1; k <= z_block_size; k++)
27                bp->array[var][i][j][k] = outputArray[k + (z_block_size + 2) *
28                   (j + (y_block_size + 2) * i)];
29    }//input_buffer detructor called here
30 }
```

Base Kernel

```
1  void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
2  sycl::buffer<double, 1>& output_buffer) {
3     //Device queue submit
4     queue_event = device_queue.submit([&](sycl::handler& cgh) {
5        //Create FPGA side accessors to the buffers
6        auto accessor_in =
7           input_buffer.get_access<sycl::access::mode::read_write>(cgh);
8        auto accessor_out =
9           output_buffer.get_access<sycl::access::mode::discard_write>(cgh);
10       cgh.single_task<class Stencil_kernel>([=]() {
11          double work[12][12][12];
12          double local_array[12][12][12];
13          for (int i = 0; i <= 11; i++)
14             for (int j = 0; j <= 11; j++)
15                for (int k = 0; k <= 11; k++)
16                   local_array[i][j][k] = accessor_in[i][j][k];
17          for (int i = 1; i <= 10; i++)
18             for (int j = 1; j <= 10; j++)
19                for (int k = 1; k <= 10; k++)
20                   work[i][j][k] = (
21                      local_array[i - 1][j][k] +
22                      local_array[i][j - 1][k] +
23                      local_array[i][j][k - 1] +
24                      local_array[i][j][k] +
25                      local_array[i][j][k + 1] +
26                      local_array[i][j + 1][k] +
27                      local_array[i + 1][j][k]) / 7.0;
28          for (int i = 1; i <= 10; i++)
29            for (int j = 1; j <= 10; j++)
30               for (int k = 1; k <= 10; k++)
31                  accessor_out[i][j][k] = work[i][j][k];
32    });
33 }
```

Combined Memory Stencil

```
1  for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
2     bp = &blocks[sorted_list[in].n];
3     sycl::range<1> num_array{ static_cast<size_t>(var_max * (x_block_size + 2) *
4        (y_block_size + 2) * (z_block_size + 2)) };
5     //create a buffer that goes to the fpga
6     double* inputArray = new double[var_max * (x_block_size + 2) *
7        (y_block_size + 2) * (z_block_size + 2)];
8     //create a buffer that comes from the fpga
9     double* outputArray = new double[var_max * (x_block_size + 2) *
10       (y_block_size + 2) * (z_block_size + 2)];
11    //flatten the 4d array to a 1d array for the buffer
12    for (var = 0; var < var_max; var++)
13       for (int i = 0; i <= x_block_size + 1; i++)
14          for (int j = 0; j <= y_block_size + 1; j++)
15             for (int k = 0; k <= z_block_size + 1; k++)
16                inputArray[(var * (x_block_size + 2) * (y_block_size + 2) *
17                   (z_block_size + 2)) + (k + (z_block_size + 2) *
18                   (j + (y_block_size + 2) * i))] = bp->array[var][i][j][k];
19    sycl::buffer<double, 1> input_buffer(inputArray, num_array);
20    {
21       sycl::buffer<double, 1> output_buffer(outputArray, num_array);
22       fpga_kernel(input_buffer, output_buffer);
23    }
24    //write the data back to the block array
25    for (var = 0; var < var_max; var++)
26       for (int i = 1; i <= x_block_size; i++)
27          for (int j = 1; j <= y_block_size; j++)
28             for (int k = 1; k <= z_block_size; k++)
29                bp->array[var][i][j][k] = outputArray[(var * (x_block_size + 2) *
30                   (y_block_size + 2) * (z_block_size + 2)) +
31                   (k + (z_block_size + 2) * (j + (y_block_size + 2) * i))];
32 }
```

Combined Memory Kernel

```
1  void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
2  sycl::buffer<double, 1>& output_buffer) {
3     //Device queue submit
4     queue_event = device_queue.submit([&](sycl::handler& cgh) {
5        //Create FPGA side accessors to the buffers
6        auto accessor_in =
7           input_buffer.get_access<sycl::access::mode::read_write>(cgh);
8        auto accessor_out =
9           output_buffer.get_access<sycl::access::mode::discard_write>(cgh);
10       cgh.single_task<class Stencil_kernel>([=]() {
11          //create a local copy of the array data for increased performance
12          double local_array[40][12][12][12];
13          for (int var = 0; var < 40; var++)
14             for (int i = 0; i <= 11; i++)
15                for (int j = 0; j <= 11; j++)
16                   for (int k = 0; k <= 11; k++)
17                      local_array[var][i][j][k] =
18                         accessor_in[(var * (12) * (12) * (12)) + (k + (12) *
19                            (j + (12) * i))];
20          for (int var = 0; var < 40; var++)
21             for (int i = 1; i <= 10; i++)
22                for (int j = 1; j <= 10; j++)
23                   for (int k = 1; k <= 10; k++)
24                      accessor_out[(var * (12) * (12) * (12)) + (k + (12) *
25                         (j + (12) * i))] = (
26                         local_array[var][i - 1][j][k] +
27                         local_array[var][i][j - 1][k] +
28                         local_array[var][i][j][k - 1] +
29                         local_array[var][i][j][k] +
30                         local_array[var][i][j][k + 1] +
31                         local_array[var][i][j + 1][k] +
32                         local_array[var][i + 1][j][k]) / 7.0;
33       });
34    });
35 }
```

Reduced Memory Kernel

```
1  void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
2  sycl::buffer<double, 1>& output_buffer) {
3     //Device queue submit
4     queue_event = device_queue.submit([&](sycl::handler& cgh) {
5        //Create FPGA side accessors to the buffers
6        auto accessor_in =
7           input_buffer.get_access<sycl::access::mode::read_write>(cgh);
8        auto accessor_out =
9           output_buffer.get_access<sycl::access::mode::discard_write>(cgh);
10       cgh.single_task<class Stencil_kernel>([=]() {
11          //create a local copy of the array data for increased performance
12          double local_array[12][12][12];
13          for (int var = 0; var < 40; var++)
14             for (int i = 0; i <= 11; i++)
15                for (int j = 0; j <= 11; j++)
16                   for (int k = 0; k <= 11; k++)
17                      local_array[i][j][k] =
18                         accessor_in[(var * (12) * (12) * (12)) + (k + (12) *
19                            (j + (12) * i))];
20             for (int i = 1; i <= 10; i++)
21                for (int j = 1; j <= 10; j++)
22                   for (int k = 1; k <= 10; k++)
23                      accessor_out[(var * (12) * (12) * (12)) + (k + (12) *
24                         (j + (12) * i))] = (
25                         local_array[i - 1][j][k] +
26                         local_array[i][j - 1][k] +
27                         local_array[i][j][k - 1] +
28                         local_array[i][j][k] +
29                         local_array[i][j][k + 1] +
30                         local_array[i][j + 1][k] +
31                         local_array[i + 1][j][k]) / 7.0;
32       });
33    });
34 }
```

Flattened Stencil

```
1  for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
2     bp = &blocks[sorted_list[in].n];
3     sycl::range<1> num_array{ static_cast<size_t>(var_max *
4        (x_block_size + 2) * (y_block_size + 2) * (z_block_size + 2)) };
5     {
6        sycl::buffer<double, 1> input_buffer(bp->array, num_array);
7        fpga_kernel(input_buffer);
8     }
9  }
```

Buffered Stencil

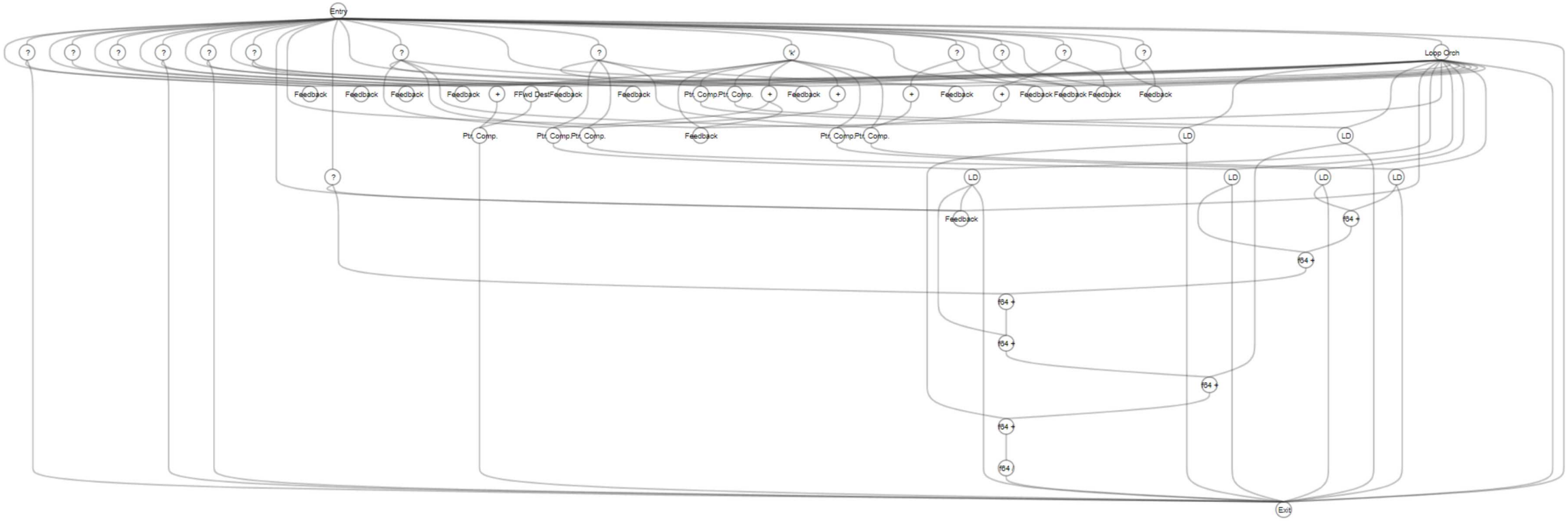
```
1  std::vector<sycl::buffer<double, 1>> input_buffer;
2
3  for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
4     bp = &blocks[sorted_list[in].n];
5     input_buffer.push_back(sycl::buffer<double, 1>(bp->array,
6        sycl::range<1>(static_cast<size_t>(var_max * (x_block_size + 2) *
7        (y_block_size + 2) * (z_block_size + 2)))));
8     fpga_kernel(input_buffer[in]);
9  }
```

Buffered Kernel

```
1  void fpga_kernel(sycl::buffer<double, 1>& input_buffer) {
2     //Device queue submit
3     queue_event[kernelCounter % 2] = device_queue.submit([&](sycl::handler& cgh)
4     {
5        //Create accessors
6        auto accessor_in =
7           input_buffer.get_access<sycl::access::mode::read_write>(cgh);
8        cgh.single_task<class Stencil_kernel>([=]() {
9           double local_array[12][12][12];
10          #pragma unroll X //replace X with the number of unrolls 0, 2, 4, or 8
11          for (int var = 0; var < 40; var++) {
12             for (int i = 0; i <= 11; i++)
13                for (int j = 0; j <= 11; j++)
14                   for (int k = 0; k <= 11; k++)
15                      local_array[i][j][k] = accessor_in[(var * (12) * (12) *
16                         (12)) + (k + (12) * (j + (12) * i))];
17             for (int i = 1; i <= 10; i++)
18                for (int j = 1; j <= 10; j++)
19                   for (int k = 1; k <= 10; k++)
20                      accessor_in[(var * (12) * (12) * (12)) + (k + (12) *
21                         (j + (12) * i))] = (
22                         local_array[i - 1][j][k] +
23                         local_array[i][j - 1][k] +
24                         local_array[i][j][k - 1] +
25                         local_array[i][j][k] +
26                         local_array[i][j][k + 1] +
27                         local_array[i][j + 1][k] +
28                         local_array[i + 1][j][k]) / 7.0;
29          }
30       });
31    });
32 }
```

Full Pipeline

```
for (int i = 1; i <= 10; i++)
   for (int j = 1; j <= 10; j++)
      for (int k = 1; k <= 10; k++)
         accessor_in[(var * (12) * (12) * (12)) + (k + (12) *
            (j + (12) * i))] = (
            local_array[i - 1][j][k] +
            local_array[i][j - 1][k] +
            local_array[i][j][k - 1] +
            local_array[i][j][k] +
            local_array[i][j][k + 1] +
            local_array[i][j + 1][k] +
            local_array[i + 1][j][k]) / 7.0;
```



Memory System

```
double local_array[12][12][12];
```