Evaluation of oneAPI for FPGAs

Intern: Nicholas Miller
Mentor: Clayton Hughes
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
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.
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

<table>
<thead>
<tr>
<th>Memory Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intelfpga::register</td>
<td>Forces a variable or array to be carried through the pipeline in registers.</td>
</tr>
<tr>
<td>intelfpga::memory(&quot;impl_type&quot;)</td>
<td>Forces a variable or array to be implemented as embedded memory. The optional string parameter impl_type can be BLOCK_RAM or MLAB.</td>
</tr>
<tr>
<td>intelfpga::numbanks(N)</td>
<td>Specifies that the memory implementing the variable or array must have N memory banks.</td>
</tr>
<tr>
<td>intelfpga::bankwidth(W)</td>
<td>Specifies that the memory implementing the variable or array must be W bytes wide.</td>
</tr>
<tr>
<td>intelfpga::singlepump</td>
<td>Specifies that the memory implementing the variable or array should be clocked at the same rate as the accesses to it.</td>
</tr>
<tr>
<td>intelfpga::doublepump</td>
<td>Specifies that the memory implementing the variable or array should be clocked at twice the rate as the accesses to it.</td>
</tr>
<tr>
<td>intelfpga::max_replicates(N)</td>
<td>Specifies that a maximum of N replicates should be created to enable simultaneous reads from the datapath.</td>
</tr>
<tr>
<td>intelfpga::private_copies(N)</td>
<td>Specifies that a maximum of N private copies should be created to enable concurrent execution of N pipelined threads.</td>
</tr>
<tr>
<td>intelfpga::simple_dual_port</td>
<td>Specifies that the memory implementing the variable or array should have no port that services both reads and writes.</td>
</tr>
<tr>
<td>intelfpga::merge(&quot;key&quot;, &quot;type&quot;)</td>
<td>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.</td>
</tr>
<tr>
<td>intelfpga::bank_bits(b0,b1,...,bn)</td>
<td>Specifies that the local memory addresses should use bits (b0,b1,...,bn) for bank-selection, where (b0,b1,...,bn) are indicated in terms of word-addressing. The bits of the local memory address not included in (b0,b1,...,bn) will be used for word-selection in each bank.</td>
</tr>
</tbody>
</table>
Results
• 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

<table>
<thead>
<tr>
<th>CPU</th>
<th>2 x Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Family</td>
<td>Arria 10</td>
</tr>
<tr>
<td>FPGA Device</td>
<td>10AX115S2F4512GSES</td>
</tr>
<tr>
<td>System Memory</td>
<td>196 GB</td>
</tr>
<tr>
<td>Base Parameters</td>
<td>No Parameters</td>
</tr>
<tr>
<td>Increased Blocks Parameters</td>
<td>--num_refine 4 --max_blocks 9000 --num_objects 1 --object 2.0 -1.71 -1.71 -1.71 0.0 0.04 0.04 0.04 1.7 1.7 1.7 0.0 0.0 0.0 0.0 -num_tsteps 25</td>
</tr>
</tbody>
</table>
Performance

Slowdown Compared to Processor

- Buffered Unrolled 8: 2.512880758
- Buffered Unrolled 4: 2.474778626
- Buffered Unrolled 2: 2.478456712
- Buffered Multiple Blocks: 3.097539437
- Buffered Base: 13.04108709
- Flattened Arrays: 13.07619917
- Reduced Local Memory: 33.3626835
- Combined Memory Transactions: 36.84519024
Utilization

Device Resource Usage

- Buffered Unrolled 8
- Buffered Unrolled 4
- Buffered Unrolled 2
- Buffered
- Flattened Arrays
- Reduced Local Memory
- Combined Memory Transactions
- Base

- DSPs
- MLABs
- RAMs
- FFs
- ALUTs
Compile Times

Compile Wall Time

Buffered Unrolled 8
Buffered Unrolled 4
Buffered Unrolled 2
Buffered
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?
```cpp
for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
    bp = &blocks[sorted_list[in].n];
    for (var = 0; var < var_max; var++) {
        sycl::range<1> num_array = static_cast<size_t>((x_block_size + 2) * (y_block_size + 2) * (z_block_size + 2));
        //create a buffer that goes to the fpga
        double* inputArray = new double[(x_block_size + 2) * (y_block_size + 2) * (z_block_size + 2)];
        //create a buffer that comes from the fpga
        double* outputArray = new double[(x_block_size + 2) * (y_block_size + 2) * (z_block_size + 2)];
        //flatten the 4d array to a 1d array for the buffer
        for (int i = 0; i <= x_block_size + 1; i++)
            for (int j = 0; j <= y_block_size + 1; j++)
                for (int k = 0; k <= z_block_size + 1; k++)
                    inputArray[k + (z_block_size + 2) * (y_block_size + 2) * i] = bp->array[var][i][j][k];
        sycl::buffer<double, 1> input_buffer(inputArray, num_array);
        sycl::buffer<double, 1> output_buffer(outputArray, num_array);
        fpga_kernel(input_buffer, output_buffer);
        //write the data back to the block array
        for (int i = 1; i <= x_block_size; i++)
            for (int j = 1; j <= y_block_size; j++)
                for (int k = 1; k <= z_block_size; k++)
                    bp->array[var][i][j][k] = outputArray[k + (z_block_size + 2) * (y_block_size + 2) * i];
    }
}
```
```cpp
void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
                 sycl::buffer<double, 1>& output_buffer) {

  // Device queue submit
  queue_event = device_queue.submit([&](sycl::handler& cgh) {
    // Create FPGA side accessors to the buffers
    auto accessor_in =
        input_buffer.get_access<sycl::access::mode::read_write>(cgh);
    auto accessor_out =
        output_buffer.get_access<sycl::access::mode::discard_write>(cgh);

    cgh.single_task<class Stencil_kernel>([=]() {

      double work[12][12][12];
      double local_array[12][12][12];
      for (int i = 0; i <= 11; i++)
        for (int j = 0; j <= 11; j++)
          for (int k = 0; k <= 11; k++)
            local_array[i][j][k] = accessor_in[i][j][k];
      for (int i = 1; i <= 10; i++)
        for (int j = 1; j <= 10; j++)
          for (int k = 1; k <= 10; k++)
            work[i][j][k] = (
              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;
      for (int i = 1; i <= 10; i++)
        for (int j = 1; j <= 10; j++)
          for (int k = 1; k <= 10; k++)
            accessor_out[i][j][k] = work[i][j][k];
    });
  });
}
```
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) * i) *
18                        (j + (y_block_size + 2) * i))] = bp->array[var][i][j][k];
19    sycl::buffer<double, 1> input_buffer(inputArray, num_array);
20    fpga_kernel(input_buffer, output_buffer);
21    }
22    //write the data back to the block array
23    for (var = 0; var < var_max; var++)
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[(var * (x_block_size + 2) *
28                        (y_block_size + 2) * (z_block_size + 2)) +
29                        (k + (z_block_size + 2) * (j + (y_block_size + 2) * i))];
30    }
31}
void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
  sycl::buffer<double, 1>& output_buffer) {

  // Device queue submit
  queue::event = device_queue.submit([&](sycl::handler& cgh) {
    // Create FPGA side accessors to the buffers
    auto accessor_in =
      input_buffer.get_access<sycl::access::mode::read_write>(cgh);
    auto accessor_out =
      output_buffer.get_access<sycl::access::mode::discard_write>(cgh);

    cgh.single_task<class Stencil_kernel>(
      [=
        // Create a local copy of the array data for increased performance
        double local_array[40][12][12][12];
        for (int var = 0; var < 40; var++)
          for (int i = 0; i <= 11; i++)
            for (int j = 0; j <= 11; j++)
              for (int k = 0; k <= 11; k++)
                local_array[var][i][j][k] =
                  accessor_in(var * (12) * (12) * (12) + (k + (12) * (i)));

        for (int var = 0; var < 40; var++)
          for (int i = 1; i <= 10; i++)
            for (int j = 1; j <= 10; j++)
              for (int k = 1; k <= 10; k++)
                accessor_out(var * (12) * (12) * (12) + (k + (12) * (i)));

      }));
});
Reduced Memory Kernel

```cpp
void fpga_kernel(sycl::buffer<double, 1>& input_buffer,
sycl::buffer<double, 1>& output_buffer) {
    //Device queue submit
    queue_event = device_queue.submit([&](sycl::handler& cgh) {
        //Create FPGA side accessors to the buffers
        auto accessor_in = input_buffer.get_access<sycl::access::mode::read_write>(cgh);
        auto accessor_out = output_buffer.get_access<sycl::access::mode::discard_write>(cgh);
        cgh.single_task<class Stencil_kernel>() {
            //create a local copy of the array data for increased performance
            double local_array[12][12][12];
            for (int var = 0; var < 40; var++)
                for (int i = 0; i <= 11; i++)
                    for (int j = 0; j <= 11; j++)
                        for (int k = 0; k <= 11; k++)
                            local_array[i][j][k] = accessor_in[(var * (12) * (12) * (12)) + (k + (12) * (j + (12) * i))];
            for (int i = 1; i <= 10; i++)
                for (int j = 1; j <= 10; j++)
                    for (int k = 1; k <= 10; k++)
                        accessor_out[(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 + 1][k] + local_array[i][j + 1][k] + local_array[i + 1][j][k]) / 7.0;
        }
    });
}
```
### Flattened Stencil

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

### Buffered Stencil

```cpp
std::vector<sycl::buffer<double, 1>> input_buffer;
for (int in = 0; in < sorted_index[num_refine + 1]; in++) {
    bp = &blocks[sorted_list[in].n];
    input_buffer.push_back(sycl::buffer<double, 1>(bp->array,
        sycl::range<1>(static_cast<size_t>(var_max * (x_block_size + 2) *
            (y_block_size + 2) * (z_block_size + 2)))));
    fpga_kernel(input_buffer[in]);
}
```
void fpga_kernel(sycl::buffer<double, 1>& input_buffer) {
    // Device queue submit
    queue_event[kernelCounter % 2] = device_queue.submit([&](sycl::handler& cgh) {
        // Create accessors
        auto accessor_in = input_buffer.get_access<sycl::access::mode::read_write>(cgh);
        cgh.single_task<class Stencil_kernel>()[
            double local_array[12][12][12],
            #pragma unroll X // replace X with the number of unrolls 0, 2, 4, or 8
            for (int var = 0; var < 40; var++) {
                for (int i = 0; i <= 11; i++)
                    for (int j = 0; j <= 11; j++)
                        for (int k = 0; k <= 11; k++)
                            local_array[i][j][k] = accessor_in[
                                (var * (12) * (12) * (12)) +
                                (k + (12) + (12) * i)];
                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) * 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;
            }
        });
    });
}
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;
double local_array[12][12][12];