OpenCL backend for FPGA

Kazutaka Morita
NTT Software Innovation Center
Motivation

• **Why HLS?**
  • Debugging software is much faster than hardware
  • Easier to specify functions in software

• **Why OpenCL?**
  • We can share codebase among different HW accelerators
    • GPGPU
    • Intel FPGA
    • Xilinx FPGA
AOCL backend example

- **AOCL backend**
  - OpenCL backend for Intel FPGA hardware
  - Supported target name
    - aocl: Target for actual hardware
    - aocl_sw_emu: Software emulation mode

- **Example**
  - Target board: DE10-Nano
    - Cyclone V SoC with Dual-core ARM Cortex-A9
  - Target graph: Conv2d layer in the Tiny YOLO v2 model

```python
(batch, in_channel, in_height, in_width) = (1, 1024, 15, 15)
(num_filter, kernel) = (1024, 3)
A = tvm.placeholder((batch, in_channel, in_height, in_width), name='A', dtype="int8")
W = tvm.placeholder((num_filter, in_channel, kernel, kernel), name='W', dtype="int8")
B = conv2d_nchw(A, W, stride=1, padding=0)
```

- 3.8 sec with ARM CPU of DE10-nano
- Let's offload this layer to FPGA with OpenCL backend!
Current generated code

```c
__kernel void conv2d_kernel0(__global char* A,
                             __global char* W,
                             __global char* compute) {
    for (int ff = 0; ff < 1024; ++ff) {
        for (int yy = 0; yy < 13; ++yy) {
            for (int xx = 0; xx < 13; ++xx) {
                for (int rc = 0; rc < 1024; ++rc) {
                    for (int ry = 0; ry < 3; ++ry) {
                        for (int rx = 0; rx < 3; ++rx) {
                            compute[((((ff * 13) + yy) * 13) + xx)]
                                += A[rc * 225 + (yy + ry) * 15 + (xx + rx)]
                                * W[ff * 9216 + rc * 9 + ry * 3 + rx];
                    }
                }
            }
        }
    }
}
```

Performance results

- CPU (LLVM): 3.8 sec
- FPGA (AOCL): 7.9 sec

All the loop are pipelined and their II equals 1. Why so slow?
Why so slow?

• Reading global memory redundantly
  • Global memory is DRAM and reading data from it is slow and stallable
  • It’s also bad idea to access global memory in deeply nested loop.

• Too many iteration counts
  • 1,594,884,096 iterations
  • Unroll loops as far as resource is available
__kernel void conv2d_kernel0(__global char* A,
                          __global char* W,
                          __global char* compute) {

    for (int ff = 0; ff < 1024; ++ff) {
        for (int yy = 0; yy < 13; ++yy) {
            for (int xx = 0; xx < 13; ++xx) {

                for (int rc = 0; rc < 1024; ++rc) {
                    for (int ry = 0; ry < 3; ++ry) {
                        for (int rx = 0; rx < 3; ++rx) {
                            compute[(((ff * 13) + yy) * 13) + xx]]
                                += A[rc * 225 + (yy + ry) * 15 + (xx + rx)]
                                * W[ff * 9216 + rc * 9 + ry * 3 + rx];
                        }
                    }
                }
            }
        }
    }
}
__kernel void conv2d_kernel0(__global char* A,
                        __global char* W,
                        __global char* compute) {
    for (int ff = 0; ff < 1024; ++ff) {
        for (int rc = 0; rc < 1024; ++rc) {
            for (int yy = 0; yy < 13; ++yy) {
                for (int xx = 0; xx < 13; ++xx) {
                    for (int ry = 0; ry < 3; ++ry) {
                        for (int rx = 0; rx < 3; ++rx) {
                            compute[(((ff * 13) + yy) * 13) + xx]] +=
                                A[rc * 225 + (yy + ry) * 15 + (xx + rx)] 
                                * W[ff * 9216 + rc * 9 + ry * 3 + rx];
                        }
                    }
                }
            }
        }
    }
}
Current generated code

__kernel void conv2d_kernel0(__global char* A,
   __global char* W,
   __global char* compute) {
  for (int ff = 0; ff < 1024; ++ff) {
    for (int rc = 0; rc < 1024; ++rc) {
      for (int yy = 0; yy < 15; ++yy) {
        for (int xx = 0; xx < 15; ++xx) {
          A_local[yy * 15 + xx] = A[rc * 225 + yy * 15 + xx];
        }
      }
    }
    for (int yy = 0; yy < 13; ++yy) {
      for (int xx = 0; xx < 13; ++xx) {
        for (int ry = 0; ry < 3; ++ry) {
          for (int rx = 0; rx < 3; ++rx) {
            compute[((((ff * 13) + yy) * 13) + xx)]
                += A_local[(yy + ry) * 15 + (xx + rx)]
                * W[ff * 9216 + rc * 3 * 3 + ry * 3 + rx];
          }
        }
      }
    }
  }
}

The number of read is decreased from 1521 to 225, but the cache size can be optimized further.
Optimize with shift register

```c
char A_local[15*15];
for (int yy = 0; yy < 15; ++yy) {
    for (int xx = 0; xx < 15; ++xx) {
        A_local[yy * 15 + xx] = A[rc * 225 + yy * 15 + xx];
    }
}

char sr[15*2+3];
for (int yy = 0; yy < 15; ++yy) {
    for (int xx = 0; xx < 15; ++xx) {
        shift(sr, A[rc * 225 + yy * 15 + xx]);
        if (xx >= 2 && yy >= 2) {
            data = {
                sr[ 0], sr[ 1], sr[ 2],
                sr[15], sr[15+1], sr[15+2],
                sr[30], sr[30+1], sr[30+2]
            };
            write_channel_intel(ch_A, data);
        }
    }
}
```
OpenCL kernel design

• Split kernel into memory reader/writer and core function
  • To feed conv2d with high throughput data streams
  • To optimize conv2d apart from memory access
Current generated code

__kernel void A_reader(__global char* A) {
    for (int ff = 0; ff < 1024; ++ff) {
        for (int rc = 0; rc < 1024; ++rc) {
            char sr[15*2+3];
            for (int yy = 0; yy < 15; ++yy) {
                for (int xx = 0; xx < 15; ++xx) {
                    shift(sr, A[rc*225+yy*15+xx]);
                    if (xx >= 2 && yy >= 2) {
                        data = {
                            sr[ 0], sr[ 1], sr[ 2],
                            sr[15], sr[15+1], sr[15+2],
                            sr[30], sr[30+1], sr[30+2]
                        };
                        write_channel_intel(A_ch, data);
                    }
                }
            }
        }
    }
}

__kernel void conv2d_kernel0(__global char* W,
                             __global char* compute) {
    for (int ff = 0; ff < 1024; ++ff) {
        for (int rc = 0; rc < 1024; ++rc) {
            for (int yy = 0; yy < 13; ++yy) {
                for (int xx = 0; xx < 13; ++xx) {
                    A_local = read_channel_intel(A_ch);
                    for (int ry = 0; ry < 3; ++ry) {
                        for (int rx = 0; rx < 3; ++rx) {
                            compute[(((ff * 13) + yy) * 13) + xx]]
                                += A_local[ry * 3 + rx] * W[ff * 9216 + rc * 9 + ry * 3 + rx];
                    }
                }
            }
        }
    }
}

Performance results

• CPU (LLVM): 3.8 sec
• FPGA (AOCL): 7.9 → 0.2 sec
What are required for TVM?

• **Pipeline-friendly operations**
  • How should we implement systolic array in TVM?
  • Scheduler operations for FIFO
    • E.g. pipe_read, pipe_write
    • Looks necessary to support channels or shift registers

• **Heterogeneous execution**
  • Necessary to offload functions to FPGA
  • Relay will support it in future

```c
// pseudo code, no boundary checks
for (int k = 0; k < a_col; k++) {
    for (int i = 0; i < MAX_SIZE; i++) {
        for (int j = 0; j < MAX_SIZE; j++) {
            // Get previous sum
            last = C[i][j];

            // Update current sum
            result = last + A[i][k] * B[k][j];

            // Write back results
            C[i][j] = result;
        }
    }
}
```
For further improvement (1)

• Area optimization
  • The result of offloading whole Resnet-18 network (int8)

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic utilization</td>
<td>1412%</td>
</tr>
<tr>
<td>ALUTs</td>
<td>912%</td>
</tr>
<tr>
<td>Dedicated logic registers</td>
<td>585%</td>
</tr>
<tr>
<td>Memory blocks</td>
<td>690%</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>2%</td>
</tr>
</tbody>
</table>

• Need to generalize similar kernels to remove duplicated codes
• Support customized bits like int3_t, int7_t, …
For further improvement (2)

• **Performance optimization**
  • Use pipelines between kernels

<table>
<thead>
<tr>
<th>Device Id</th>
<th>Kernel</th>
<th>0.00ms</th>
<th>206.56ms</th>
<th>413.11ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device 0</td>
<td>fuse_conv2d_kernel0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device 0</td>
<td>Memory Transfers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• **Auto-tuning**
  • How many times we should unroll loops?
  • Should we coalesce nested loops?