OpenCL Compiler Tools for FPGAs

Dmitry Denisenko, Engineering
Programmable Solutions Group, Intel
April 21, 2016
Motivation

Great performance comes from deep understanding of hardware architecture, compiler, and the algorithm.

Compiler tools must educate the user about the underlying architecture and how user’s algorithm fits onto it.

How differences in FPGA architecture lead to differences in OpenCL FPGA compiler tools.
How are FPGAs different from other architectures?

1. Computation in Space versus Time
2. Importance of Area
3. Loop Pipelining
4. Local Memory Flexibility
5. (other ways we’re not going to cover here)

Altera SDK for OpenCL Tools that deal with these concepts.
1. Computation in Space
A simple 3-address CPU
Load memory value into register

Instruction

PC

Fetch

LdAddr

Load

LdData

Store

StAddr

StData

Op

Val

Op

Val

Aaddr

Baddr

Caddr

CWriteEnable

CData

Op

ALU

A

B

C
Add two registers, store result in register
A simple program

Mem[100] += 42 * Mem[101]

CPU instructions:

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
CPU activity, step by step

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
Unroll the CPU hardware…

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
... and specialize by position

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]

1. Instructions are fixed.
   Remove “Fetch”
... and specialize

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
... and specialize

1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
... and specialize

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
... and specialize

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.
5. Remove dead data.

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
Optimize the Datapath

1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.
5. Remove dead data.
6. Reschedule!

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
Data parallel kernel

```c
__kernel void sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

float *a = 0 1 2 3 4 5 6 7
float *b = 7 6 5 4 3 2 1 0
float *answer = 7 7 7 7 7 7 7 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

8 work items for vector add example

0 1 2 3 4 5 6 7

Work item IDs
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Work item IDs
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Work item IDs
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads:

- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored.

8 work items for vector add example

Work item IDs

3 4 5 6 7
On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored
How does my pipeline look like, how well is it performing, and are its bottlenecks?
2. Area
Area

FPGA area is multi-dimensional:

- Registers
- Look-Up Tables (LUTs)
- On-chip RAM blocks
- Dedicated Signal Processing (DSP) blocks

Each FPGA model provides a different mix of these four types of resources.

Each design demands a different mix of these four types.
Importance of Area

Area on an FPGA is major concern:
- Higher area $\rightarrow$ fewer kernels per chip
- Higher area $\rightarrow$ no-fit
- Higher area $\rightarrow$ more expensive chip
- Higher area $\rightarrow$ higher dynamic power

How much area does a kernel use and where does it go?
Area Report Detail

For area report to be actionable, it must be done on a sub-line level.

\[
\text{float
cache}[li] = \text{global_int_data}[gi+i];
\]

Operations that consume area from the line above:

\[
\text{float
cache}[li] = \text{(float)} \quad \text{// Store to local memory}
\]
\[
\text{global_int_data}[ ] \quad \text{// Implicit int-to-float conversion}
\]
\[
\text{gi+i} \quad \text{// Global load}
\]
\[
\text{gi+i} \quad \text{// Integer addition}
\]
3. Loop Pipelining
Data-Parallel Execution

On the FPGA, we use pipeline parallelism to achieve acceleration.

```c
kernel void sum(
global const float *a,
global const float *b,
global float *c)
{
    int xid = get_global_id(0);
    c[xid] = a[xid] + b[xid];
}
```

Threads execute in an embarrassingly parallel manner.
Ideally, all parts of the pipeline are active at the same time.
Data-Parallel Execution - drawbacks

Difficult to express programs which have partial dependencies during execution

```
kernel void
sum(global const float *a, 
    global const float *b, 
    global float *c)
{
    int xid = get_global_id(0);
    c[xid] = c[xid-1] + b[xid];
}
```

Would require complicated hardware and new language semantics to describe the desired behavior
Solution: Tasks and Loop-pipelining

- Allow users to express programs as a single-thread

```c
for (int i=1; i < n; i++) {
    c[i] = c[i-1] + b[i];
}
```

- Pipeline parallelism still leveraged to efficiently execute loops in Altera’s OpenCL

  - Parallel execution inferred by compiler
  - Loop Pipelining
Loop Pipelining Example

- **No Loop Pipelining**
  - No Overlap of Iterations!

- **With Loop Pipelining**
  - Finishes Faster because Iterations Are Overlapped
  - Looks almost like multi-threaded execution!

Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
Parallel Threads vs. Loop Pipelining

So what’s the difference NDRange and loop pipelining?

Parallel threads launch 1 thread per clock cycle in pipelined fashion

Loop Pipelining

Sometimes loop iterations cannot be started every cycle.
Loop-Carried Dependencies

Loop-carried dependencies are dependencies where one iteration of the loop depends upon the results of another iteration of the loop.

```c
kernel void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_output(y);
    }
}
```

The variable state in iteration 1 depends on the value from iteration 0. Similarly, iteration 2 depends on the value from iteration 1, etc.
Loop-Carried Dependencies

To achieve acceleration, we pipeline each iteration of a loop with loop-carried dependencies
- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

```c
kernel void state_machine(ulong n) {
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        write_output(y);
    }
}
```

At this point, we can launch the next iteration.
Trouble with Loop-Carried Dependencies

Many things can go wrong with loop pipelining:
- Loop-carried dependency takes too long to compute.
- Loop with externally-visible events has iterations that get out of order.
- Loop may have sub-loops with iterations that get out of order.

How well is each loop pipelined, are there any loop-carried dependency issues, and how to fix them?
Local Memory Flexibility
FPGA On-chip memory systems

“Local” and some “private” memories use on-chip block RAM resources
- Very high bandwidth, true random access.

All memory system parameters are customized to your application to eliminate or minimize access contention:
- Width, depth, number of banks, port-to-bank assignment, etc.

Caveat: Compiler has to understand access patterns to properly configure a local memory system.
Example: Conflict-free for 1 store, 7 loads.

Up to Four ports

Block RAM

1 write

7 read

never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall
never-stall

Memory 2x clock

store

load

load

load

load

load

load

load

load

Memory 2x clock

Port0

Port1

Port2

Port3

Port0

Port1

Port2

Port3
Local Memory Feedback

Is my local memory efficient, how and why the compiler configured it, and what can I do to fix any inefficiencies?
Altera SDK for OpenCL Tools
Dynamic Profiler
for measuring pipeline efficiency

Pipeline Performance Stats
(called with hardware counters)

<table>
<thead>
<tr>
<th>Line</th>
<th>Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>a &lt;= aEnd;</td>
</tr>
<tr>
<td>65</td>
<td>a = aStep, b += bStep;</td>
</tr>
<tr>
<td>66</td>
<td>// Load the matrices from device memory</td>
</tr>
<tr>
<td>67</td>
<td>// to shared memory, each thread loads</td>
</tr>
<tr>
<td>68</td>
<td>// one element of each matrix</td>
</tr>
<tr>
<td>69</td>
<td>AS(y, x) = A[a + uIW * y + b];</td>
</tr>
<tr>
<td>70</td>
<td>BS(y, x) = B[b + uWB * y + b];</td>
</tr>
<tr>
<td>71</td>
<td>// Synchronize to make sure the matrices are loaded</td>
</tr>
<tr>
<td>72</td>
<td>barrier(CLK_LOCAL_MEM_FENCE);</td>
</tr>
<tr>
<td>73</td>
<td>#pragma unroll</td>
</tr>
<tr>
<td>74</td>
<td>for (int k = 0; k &lt; BLOCK_SIZE; ++k) {</td>
</tr>
<tr>
<td>75</td>
<td>Csub += AS(y, x) * BS(k, x);</td>
</tr>
<tr>
<td>76</td>
<td>}</td>
</tr>
<tr>
<td>77</td>
<td>}</td>
</tr>
<tr>
<td>78</td>
<td>// Synchronize to make sure that the preceding</td>
</tr>
<tr>
<td>79</td>
<td>// computation is done before loading two new</td>
</tr>
<tr>
<td>80</td>
<td>// sub-matrices of A and B in the next iteration</td>
</tr>
<tr>
<td>81</td>
<td>barrier(CLK_LOCAL_MEM_FENCE);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Attributes</th>
<th>Stall%</th>
<th>Occupancy%</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>__global(MEMORY), read</td>
<td>0.23%</td>
<td>0.95%</td>
<td>0.139 MB/s, 100.00% Efficiency</td>
<td></td>
</tr>
<tr>
<td>__global(MEMORY), read</td>
<td>0.00%</td>
<td>0.95%</td>
<td>0.101 MB/s, 100.00% Efficiency</td>
<td></td>
</tr>
</tbody>
</table>

Memory bandwidth demand of a load unit.

How often this unit stalls the pipeline.

How often this unit does useful work.
[#define NUM_READS 8
#define NUM_WRITES 8

__attribute__((reqd_work_group_size(1024,1,1)))

kernel void big_lmem (global int* restrict in,
global int* restrict out) {

local int lmem[1024];
int gi = get_global_id(0);
int gs = get_global_size(0);
int li = get_local_id(0);
int res = in[gi];
#pragma unroll
for (int i=0; i<NUM_WRITES; i++) {
    lmem[li - i] = res;
    res >>= 1;
}
barrier(CLK_GLOBAL_MEM_FENCE);
res = 0;
#pragma unroll
for (int i=0; i < NUM_READS; i++) {
    res ^= lmem[li - i];
}
out[gi] = res;
}]
Optimization Report
for Loop Pipelining Feedback

Kernel: my_kernel

The kernel is compiled for single work-item execution.

Loop Report:

+ Loop "Block1" (file a.cl line 2)
  Pipelined with successive iterations launched every 324 cycles due to:

  Memory dependency on Load Operation from: (file a.cl line 3)
  Store Operation (file a.cl line 3)

Largest Critical Path Contributors:
  49%: Load Operation (file a.cl line 3)
  49%: Store Operation (file a.cl line 3)
Thank You