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.

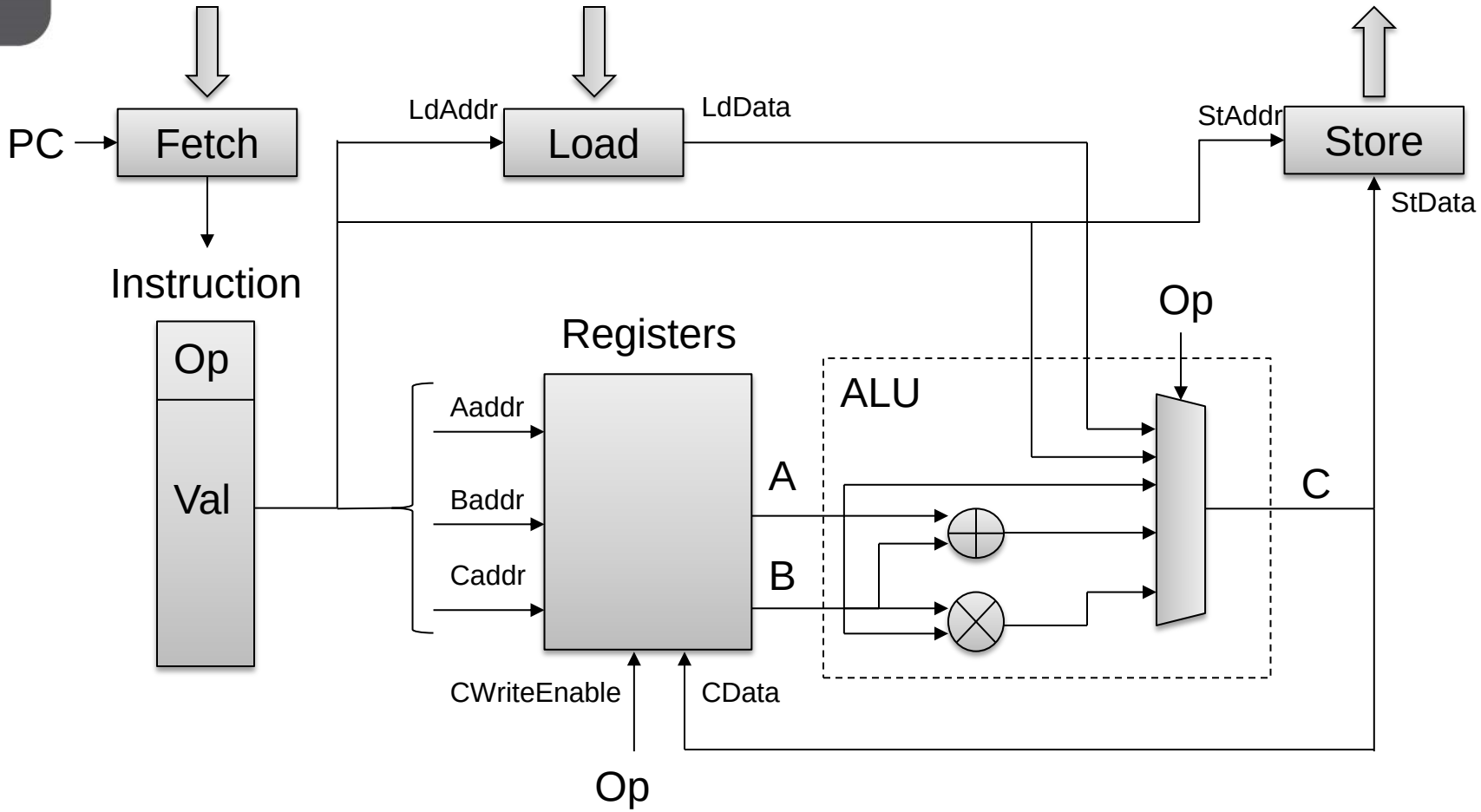
Talk Overview

- 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)

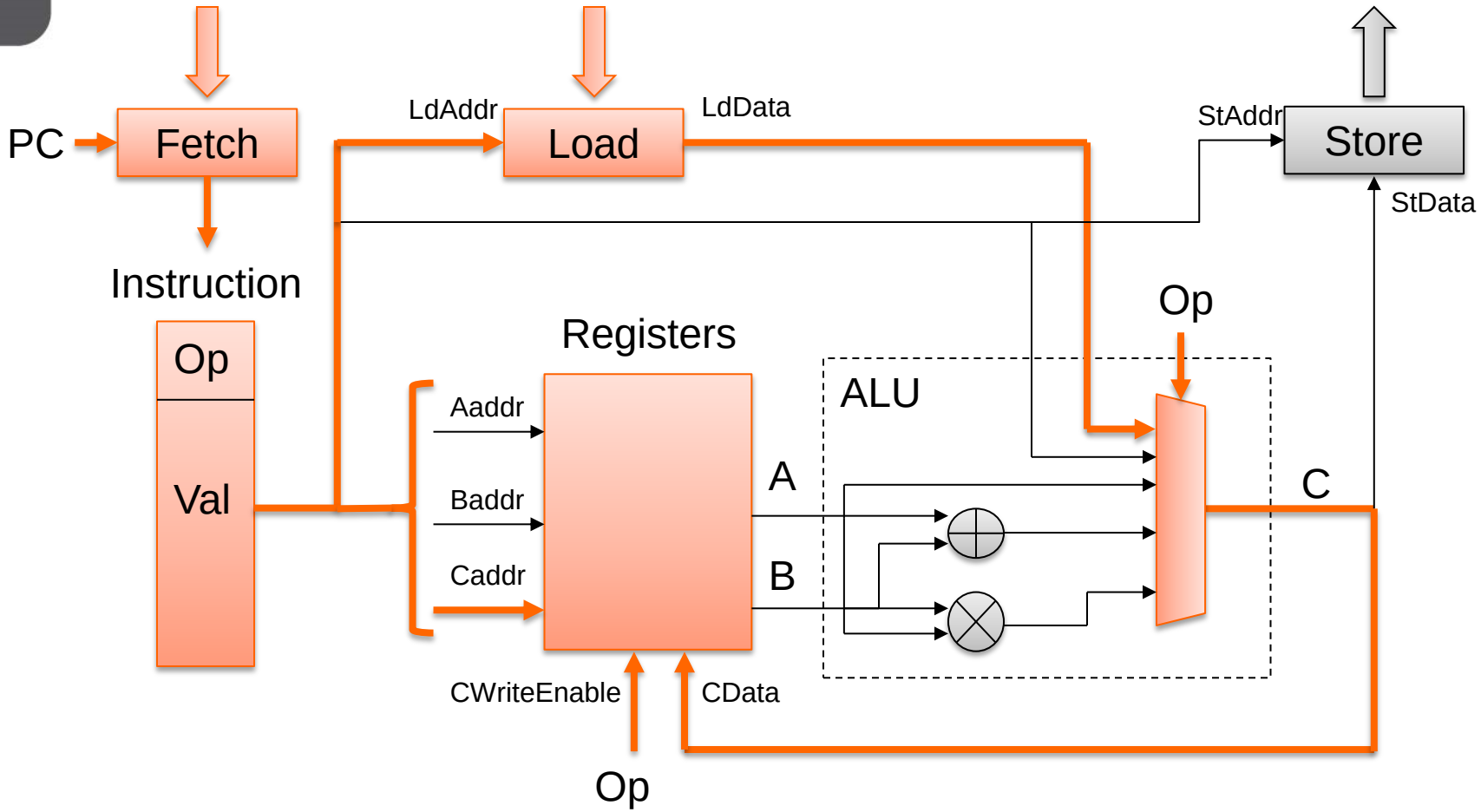
- Altera SDK for OpenCL Tools that deal with these concepts.

1. Computation in Space

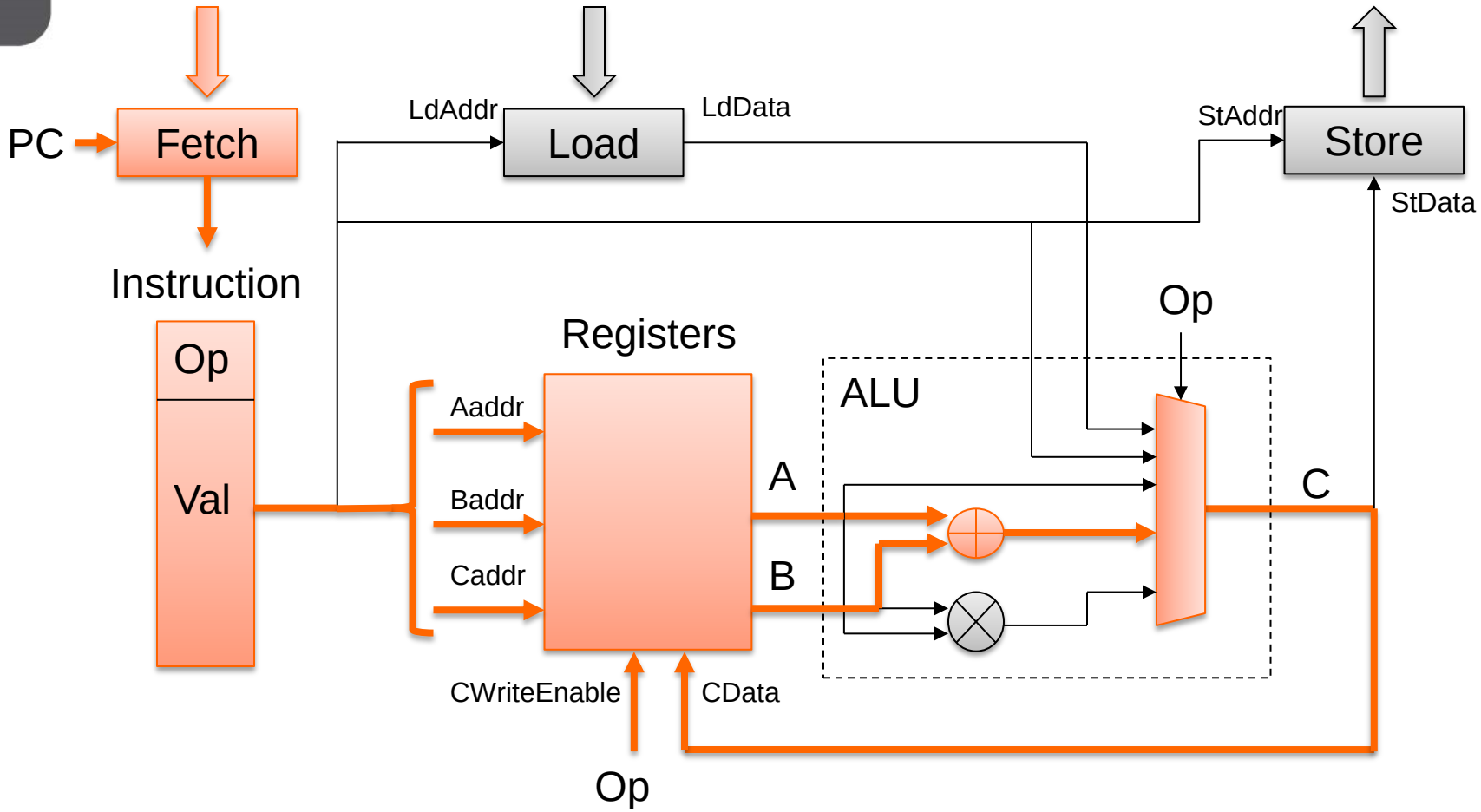
A simple 3-address CPU



Load memory value into register



Add two registers, store result in register



A simple program

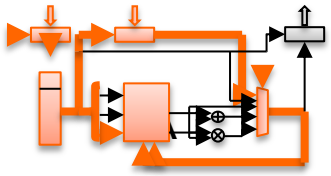
- Mem[100] += 42 * Mem[101]

- CPU instructions:

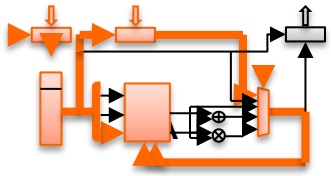
R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]

CPU activity, step by step

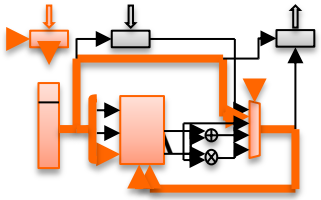
R0 ← Load Mem[100]



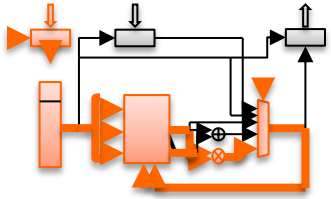
R1 ← Load Mem[101]



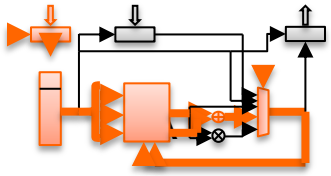
R2 ← Load #42



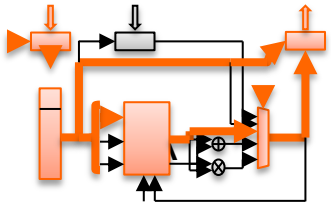
R2 ← Mul R1, R2



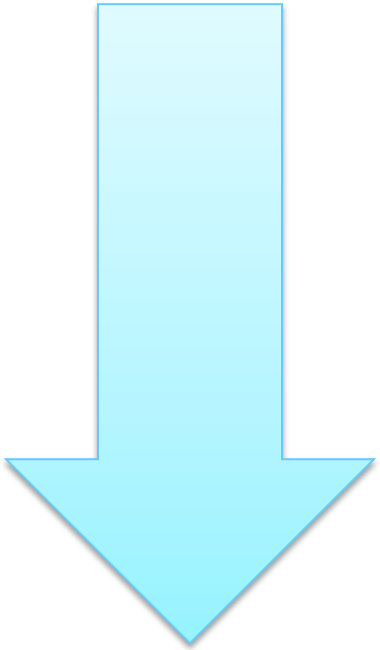
R0 ← Add R2, R0



Store R0 → Mem[100]



Time



Unroll the CPU hardware…

R0 ← Load Mem[100]

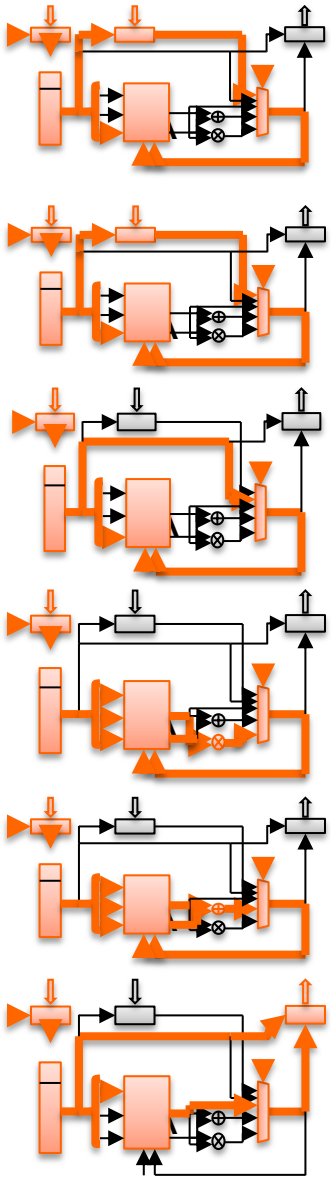
R1 ← Load Mem[101]

R2 ← Load #42

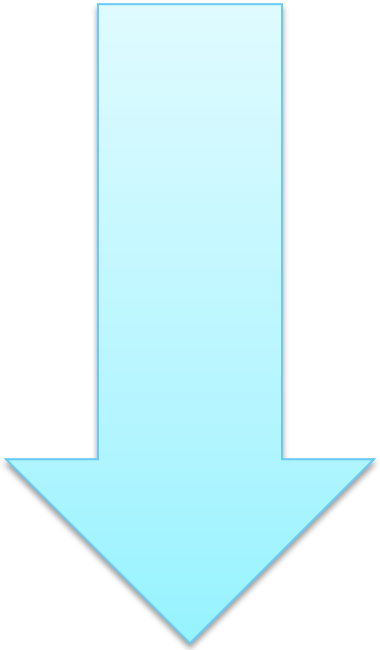
R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



Space



… and specialize by position

R0 ← Load Mem[100]

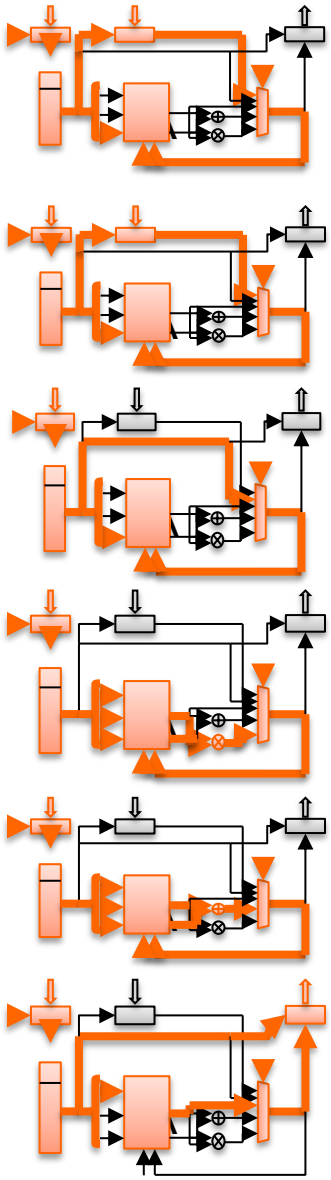
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



1. Instructions are fixed. Remove “Fetch”

… and specialize

R0 ← Load Mem[100]

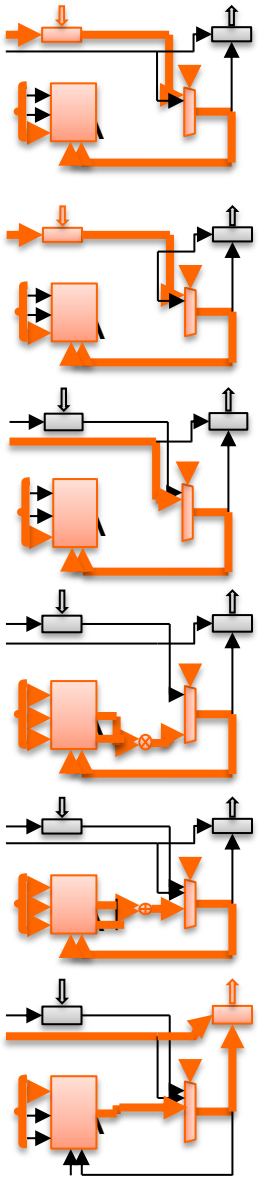
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

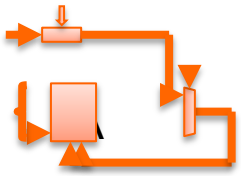
Store R0 → Mem[100]



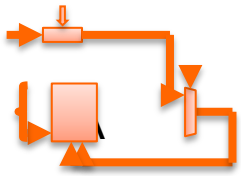
1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops

… and specialize

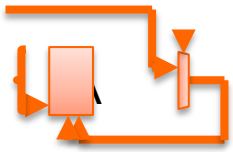
R0 ← Load Mem[100]



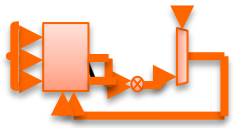
R1 ← Load Mem[101]



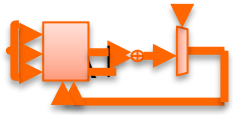
R2 ← Load #42



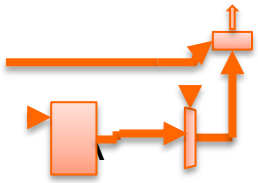
R2 ← Mul R1, R2



R0 ← Add R2, R0



Store R0 → Mem[100]



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

… and specialize

R0 ← Load Mem[100]

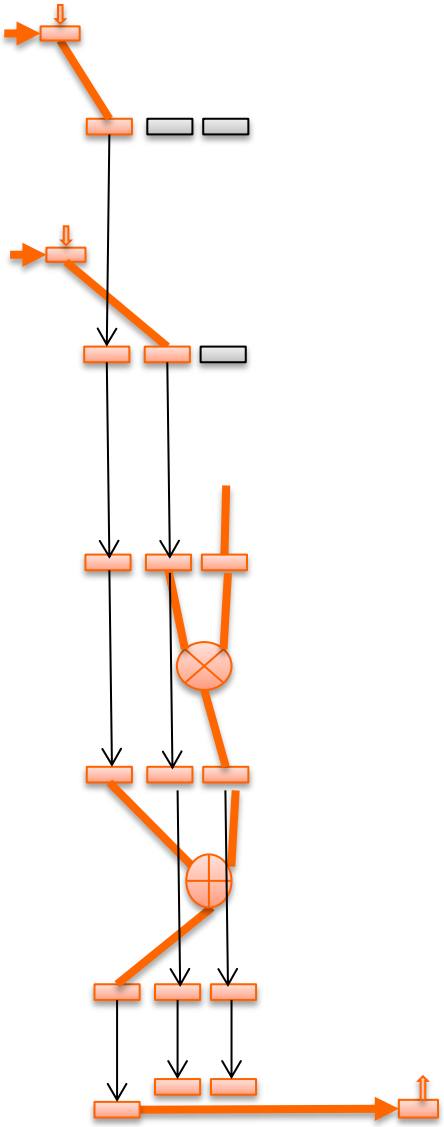
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



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

… and specialize

R0 ← Load Mem[100]

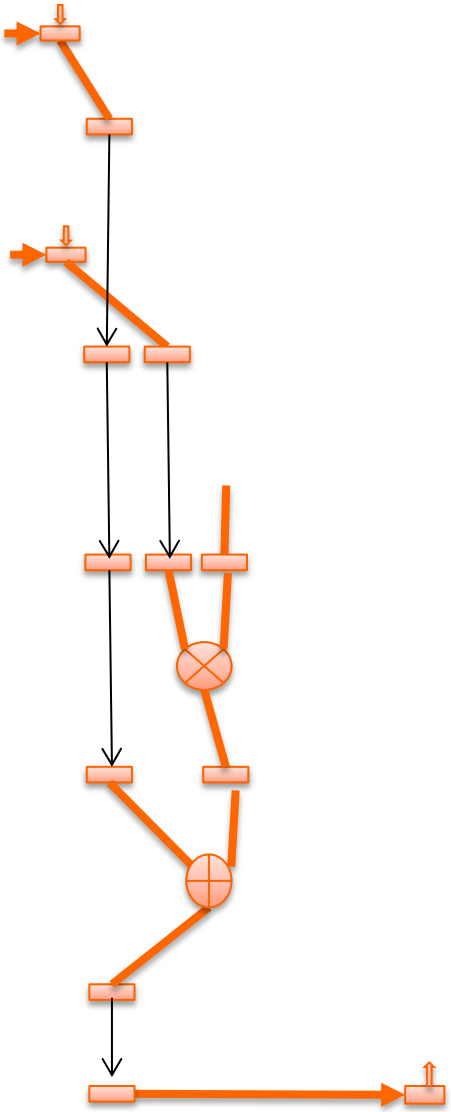
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]



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

Optimize the Datapath

R0 ← Load Mem[100]

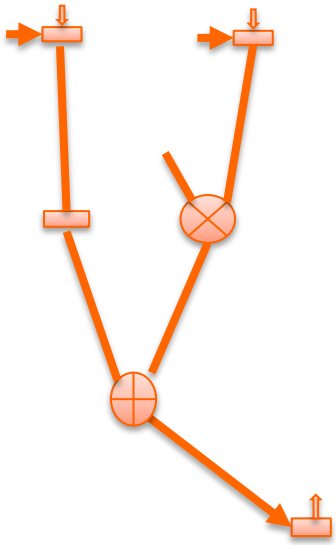
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

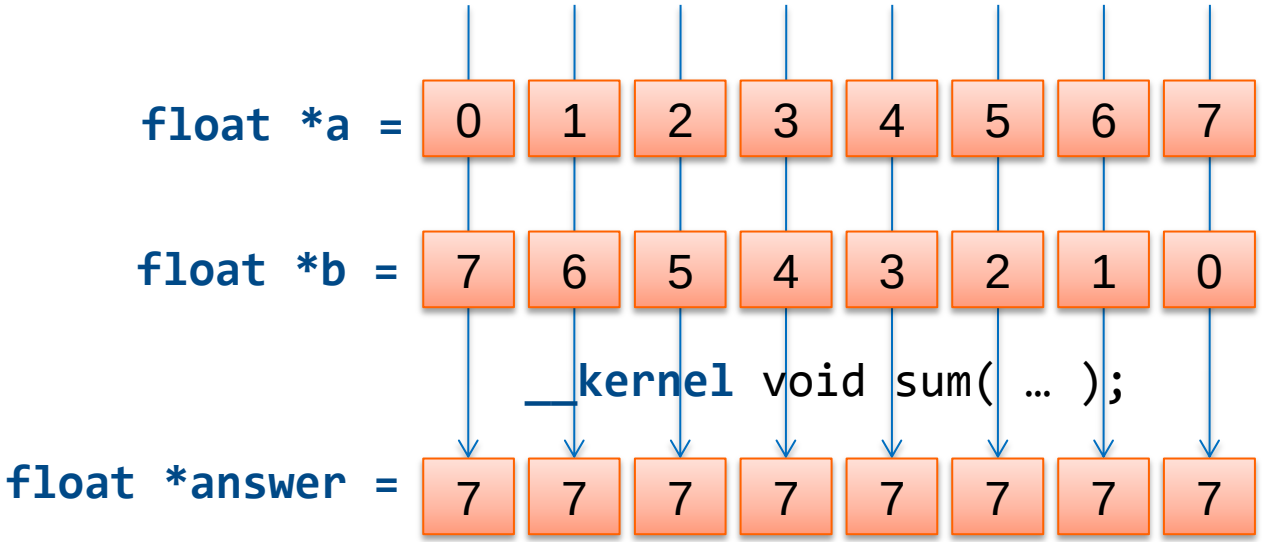
Store R0 → Mem[100]



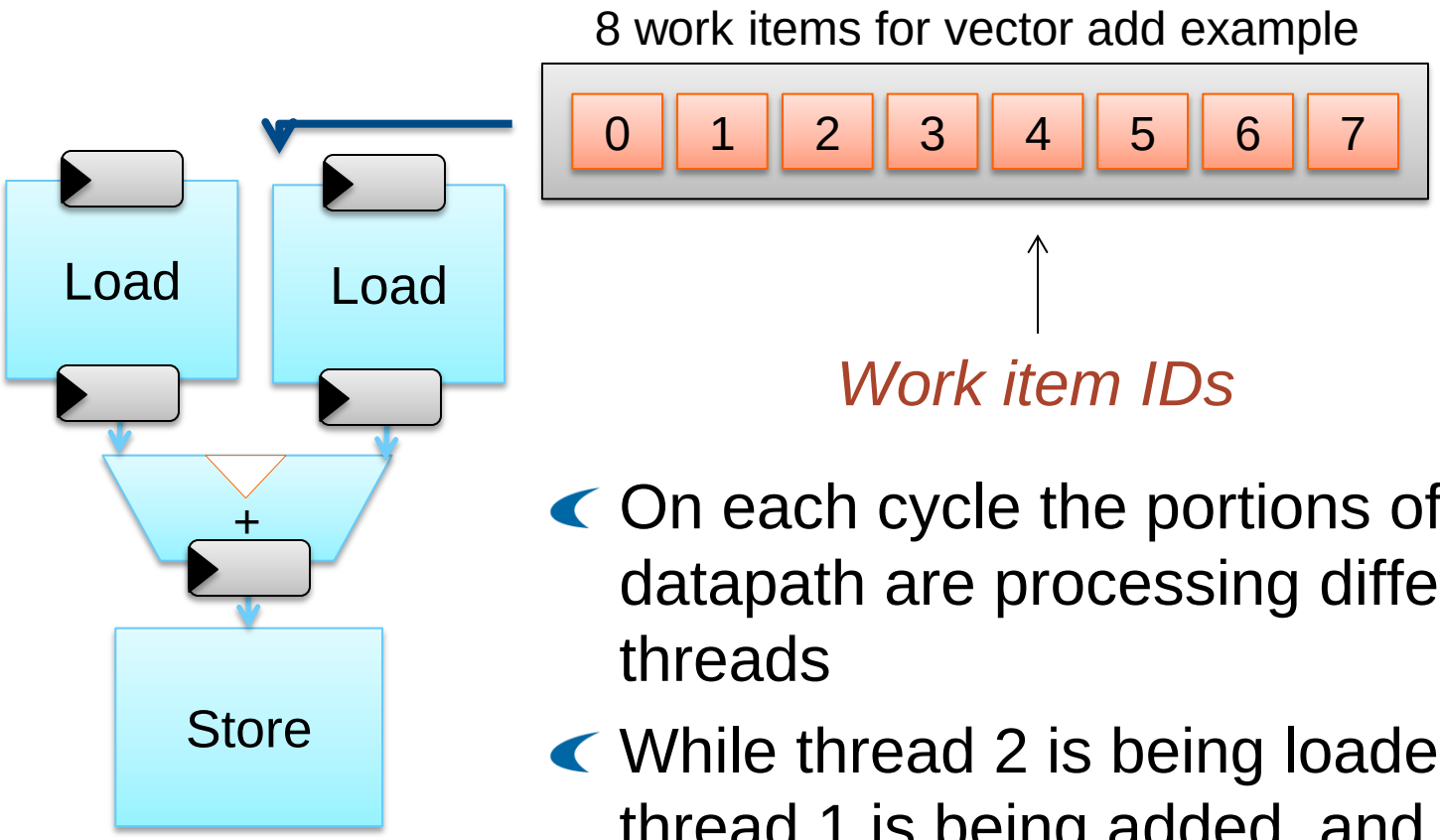
1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.
5. Remove dead data.
6. Reschedule!

Data parallel kernel

```
__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];
}
```

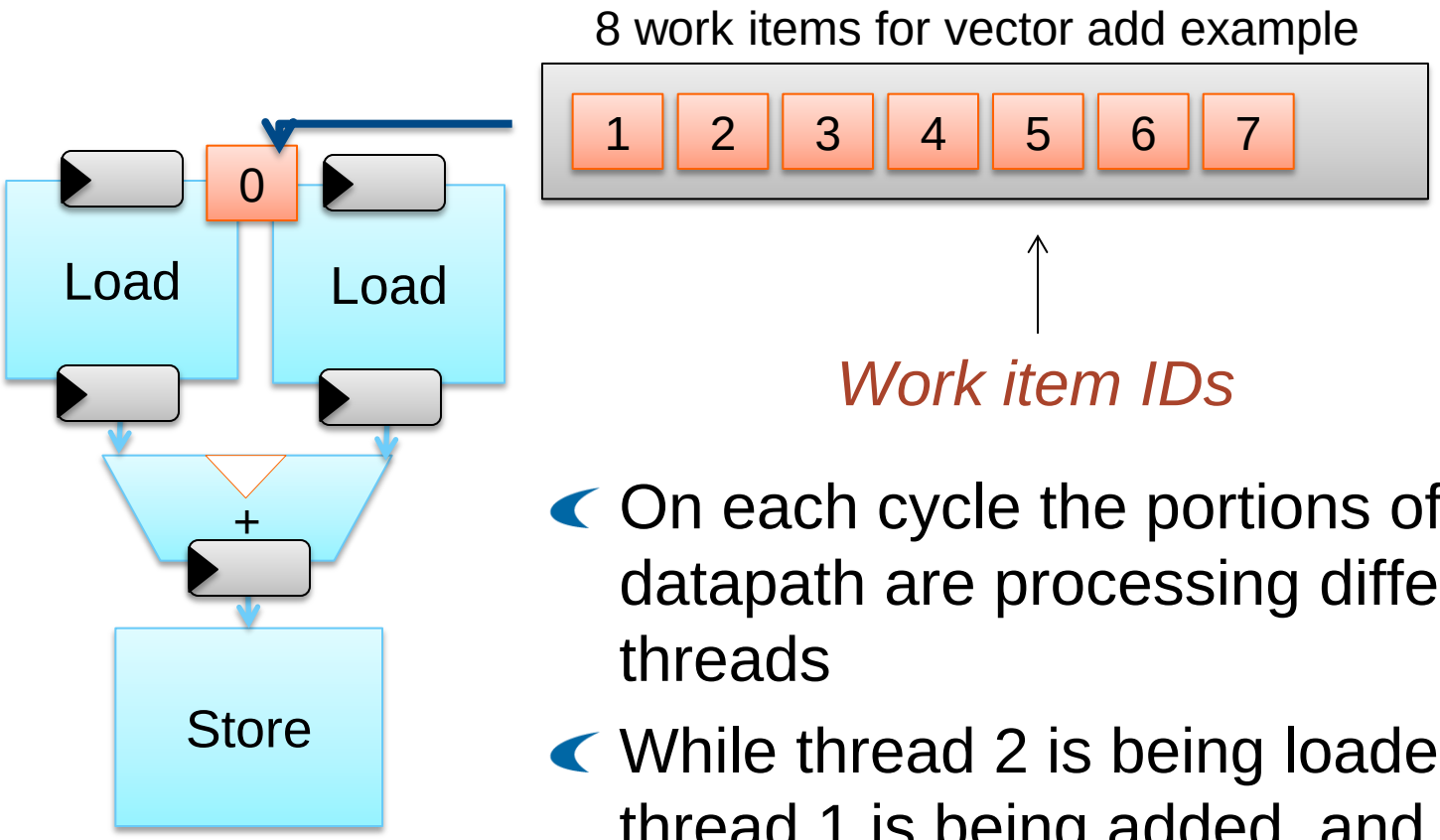


Example Datapath for Vector Add



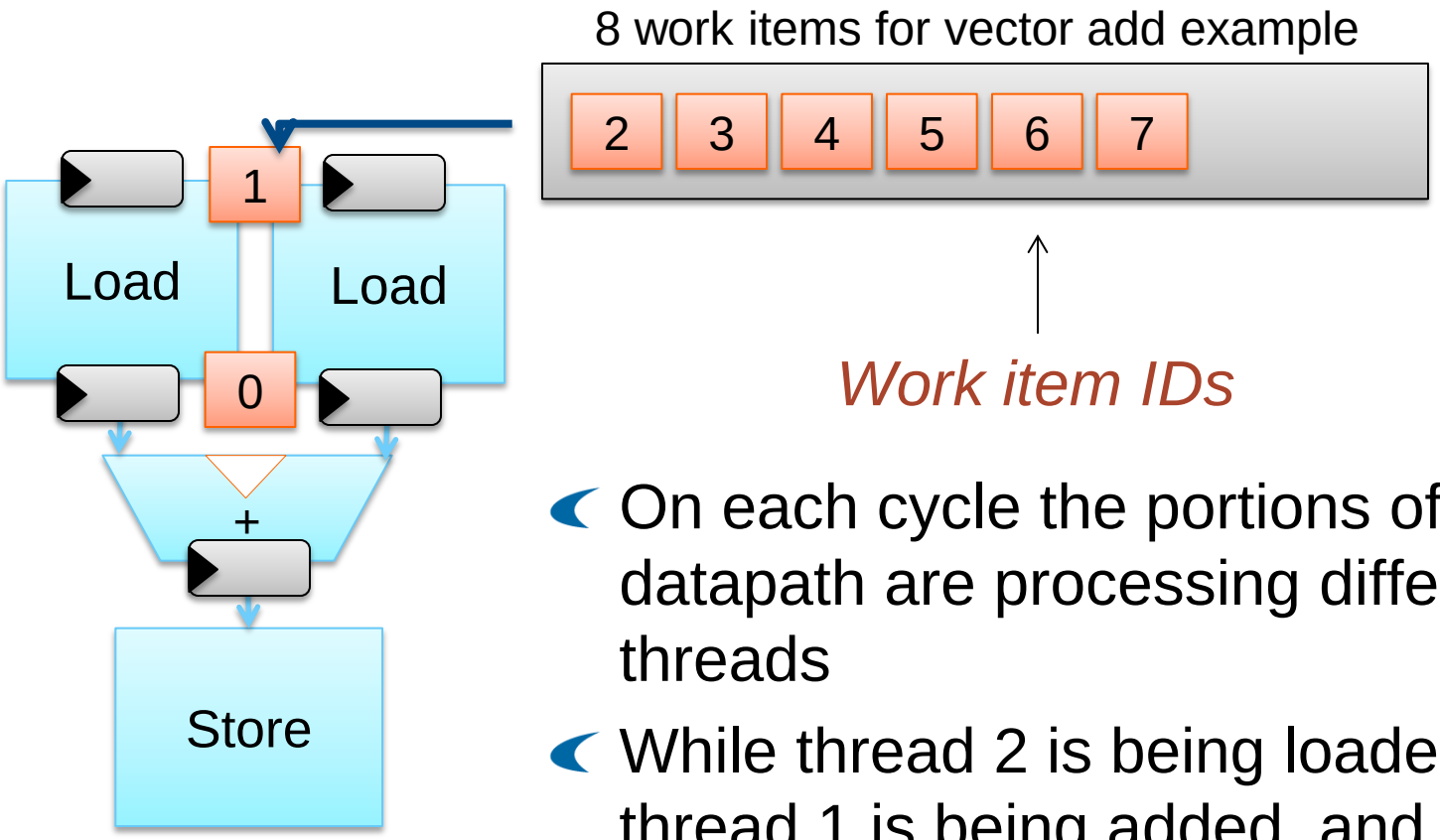
- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Example Datapath for Vector Add



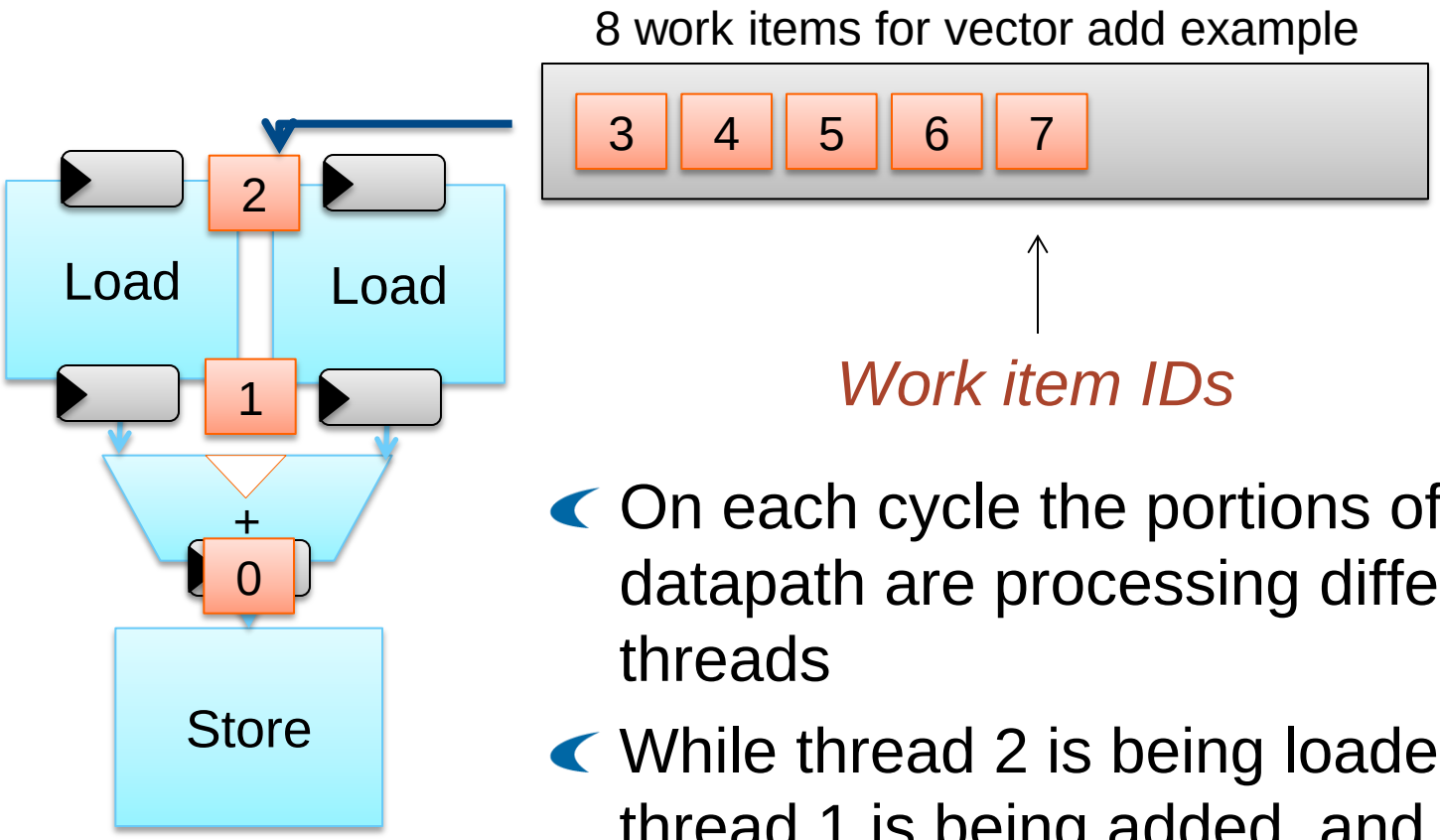
- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Example Datapath for Vector Add



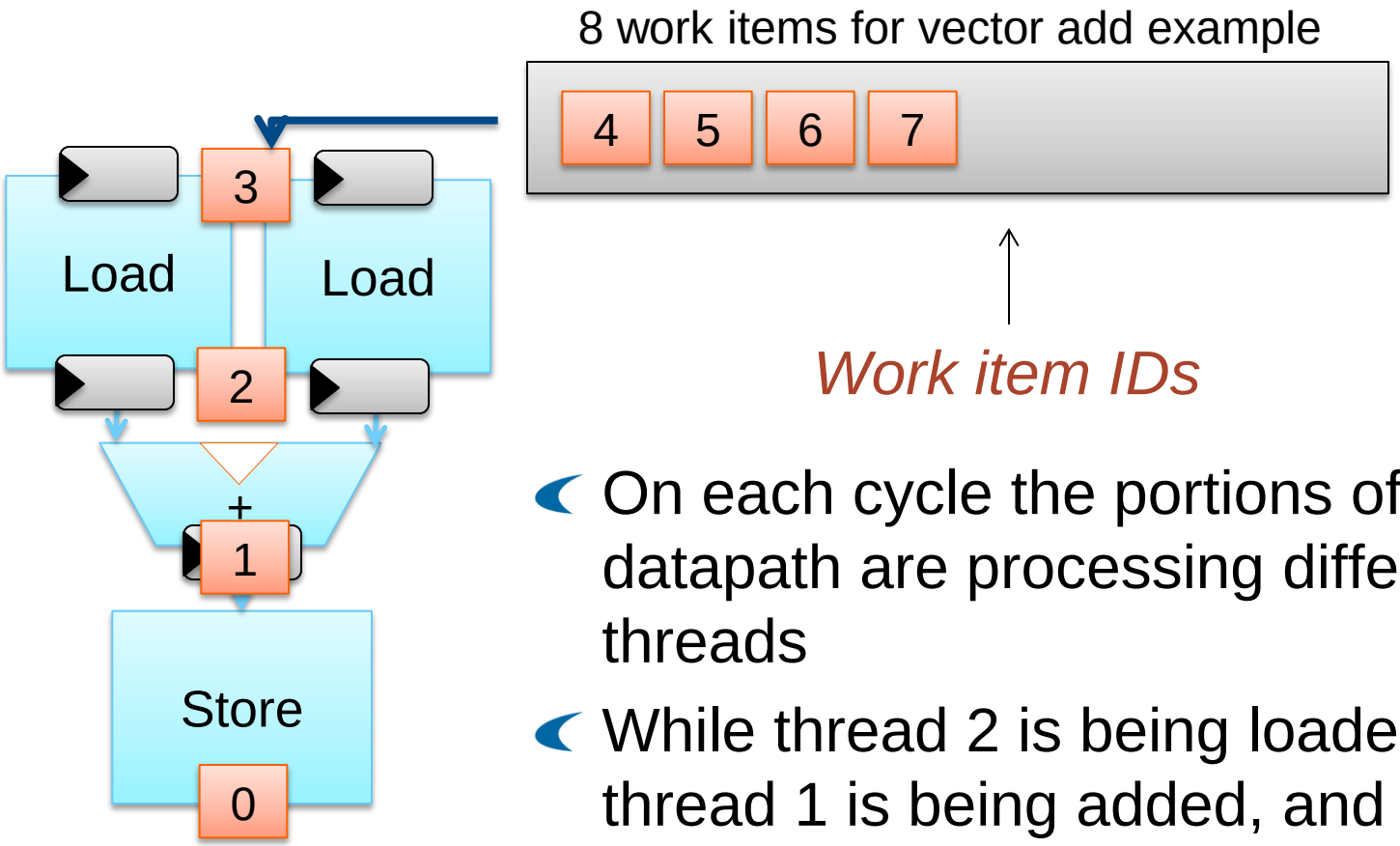
- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

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

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

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 ➔ fewer kernels per chip
  - Higher area ➔ no-fit
  - Higher area ➔ more expensive chip
  - Higher area ➔ 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.

```
float_cache[li] = global_int_data[gi+i];
```

- Operations that consume area from the line above:

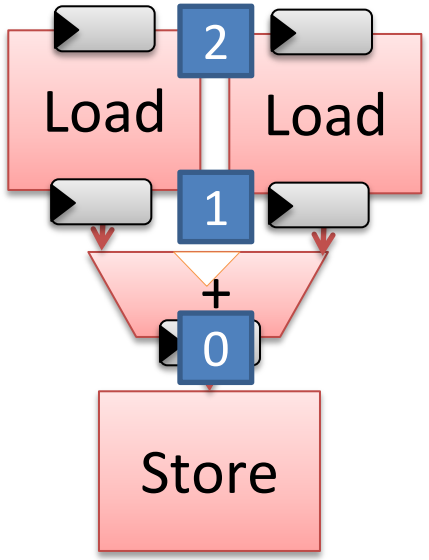
```
float_cache[li] =                    // Store to local memory
   (float)                           // Implicit int-to-float conversion
      global_int_data[  ]            // Global load
                    gi+i             // Integer addition
```

3. Loop Pipelining

Data-Parallel Execution

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

```
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];
}
```

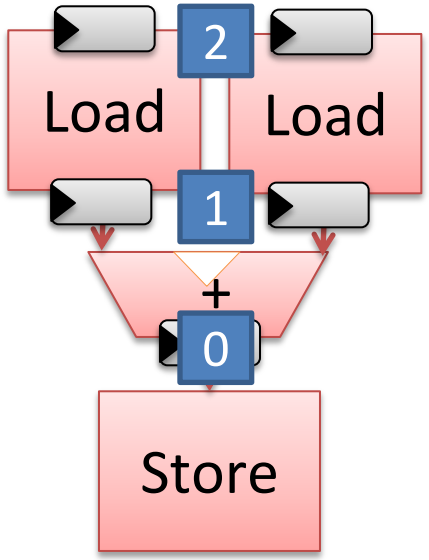


- Threads execute in an embarrassingly parallel manner.
- Ideally, all parts of the pipeline are active at the same time.

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];
}
```



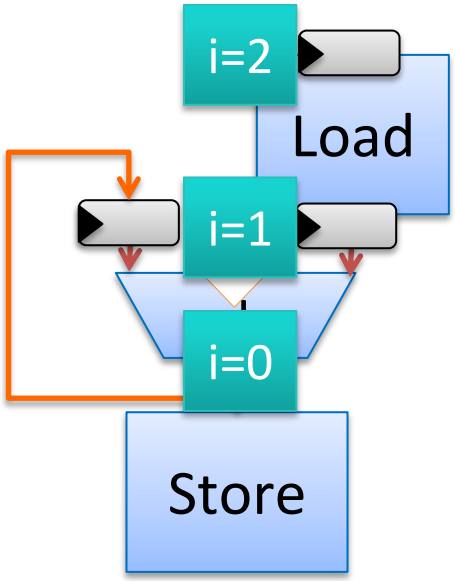
- Would require complicated hardware and new language semantics to describe the desired behavior

Solution: Tasks and Loop-pipelining

- Allow users to express programs as a single-thread

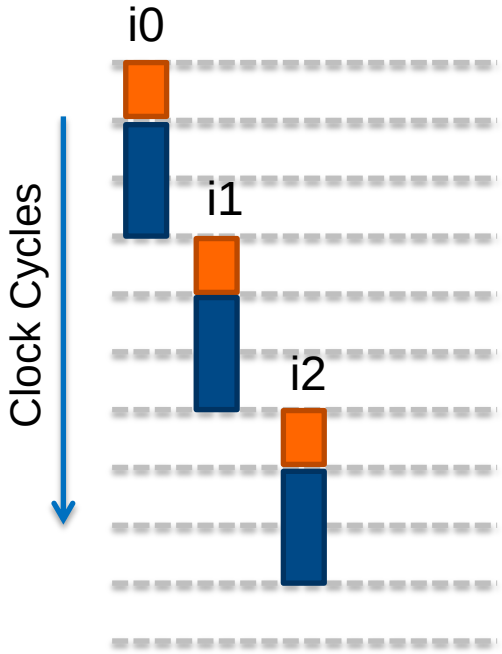
```
for (int i=1; i < n; i++) {
  c[i] = c[i-1] + b[i];
}
```

- Pipeline parallelism still leveraged to efficiently execute loops in Altera's OpenCL
  - **Parallel execution inferred by compiler**
  - ***Loop Pipelining***



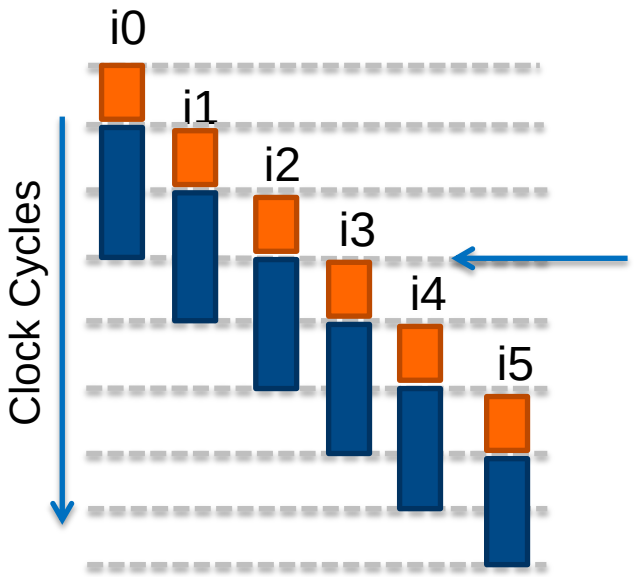
Loop Pipelining Example

- No Loop Pipelining



No Overlap of Iterations!

- With Loop Pipelining



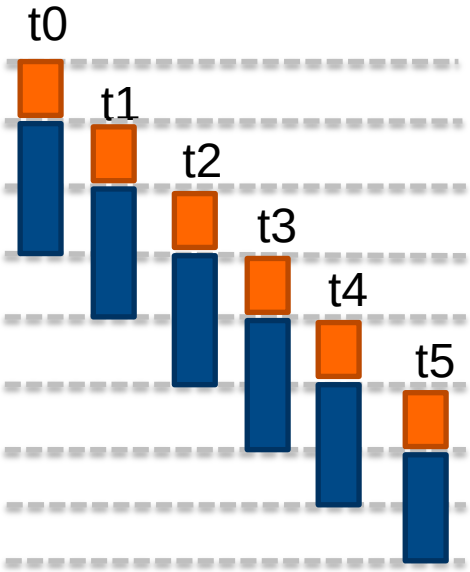
Looks almost like multi-threaded execution!

Finishes Faster because Iterations Are Overlapped

- Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.

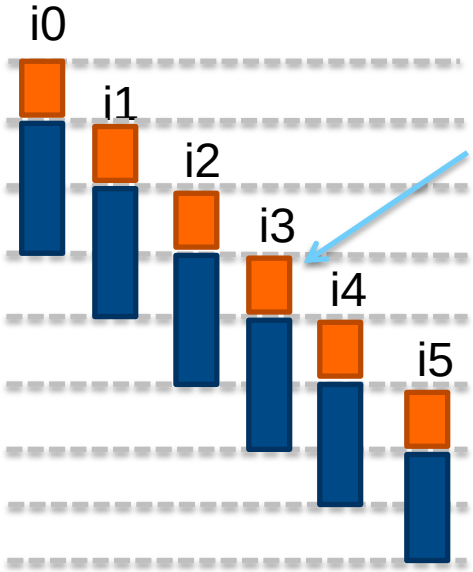
Parallel Threads vs. Loop Pipelining

- So what's the difference NDRange and loop pipelining?



Parallel Threads

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

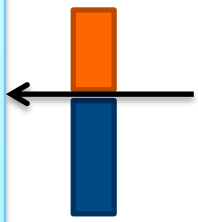
```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    write_output(y);
  }
}
```

- 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

```
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.

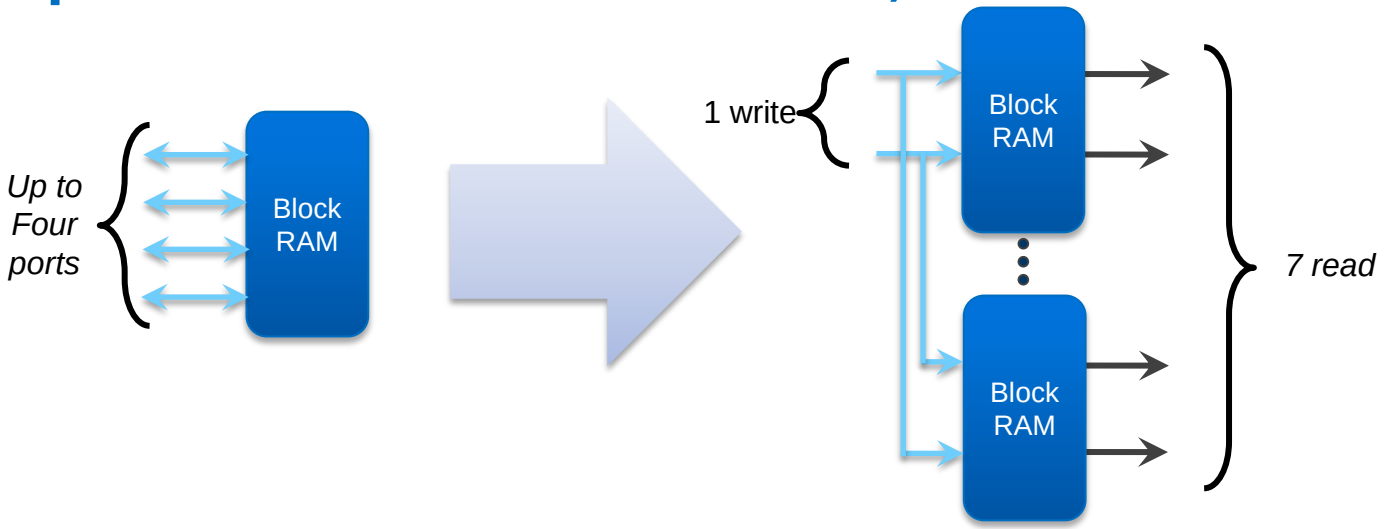
How well is each loop pipelined,
are there any loop-carried dependency issues,
and how to fix them?

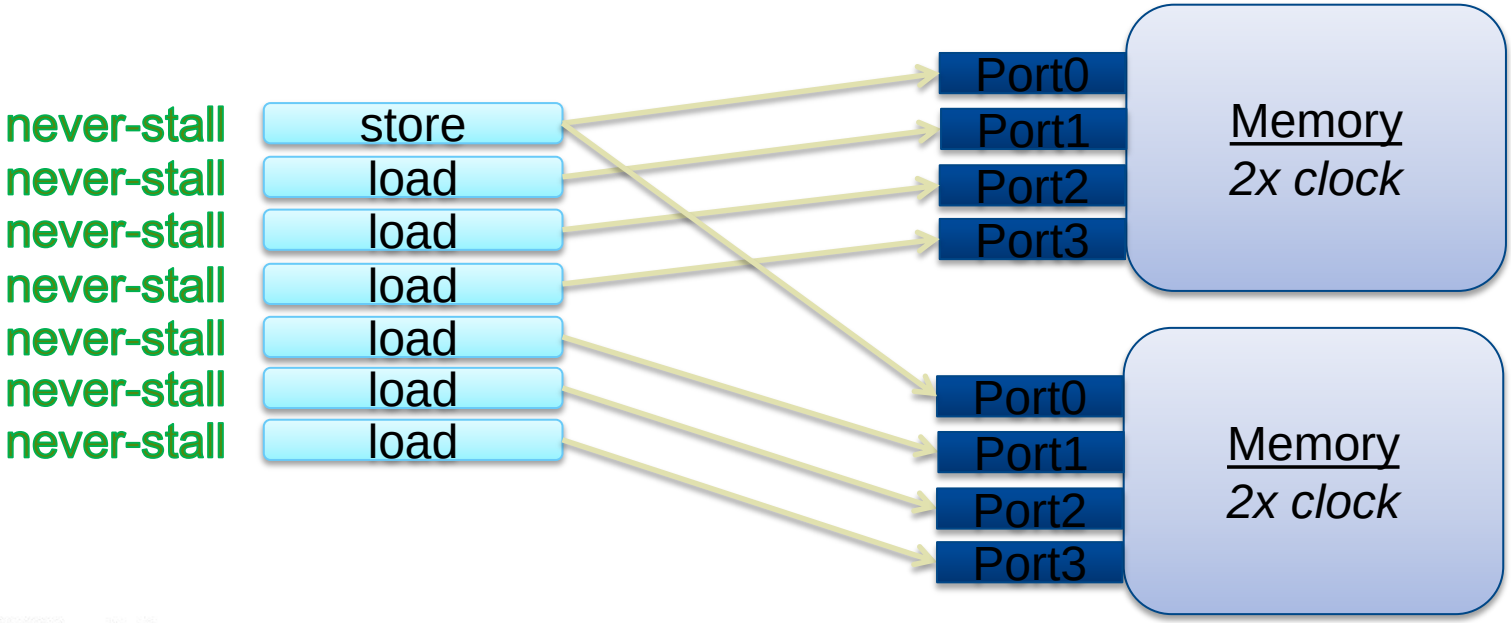
Local Memory Flexibility

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.



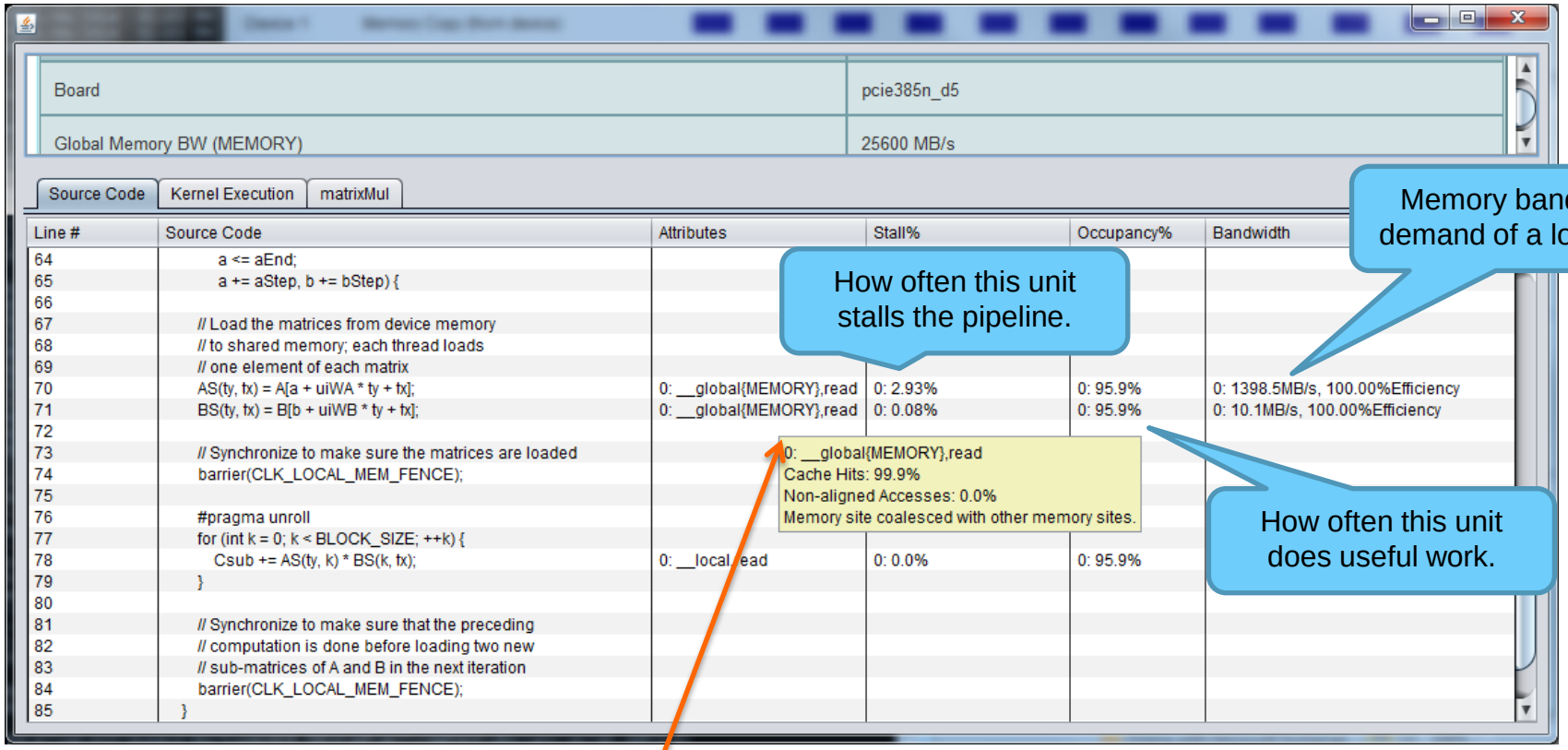


Local Memory Feedback

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
(collected with hardware counters)

Area Report

```
1 #define NUM_READS    8
2 #define NUM_WRITES   8
3
4
5 __attribute((reqd_work_group_size(1024,1,1)))
6 kernel void big_lmem (global int* restrict in,
7                       global int* restrict out) {
8
9   local int lmem[1024];
10   int gi = get_global_id(0);
11   int gs = get_global_size(0);
12   int li = get_local_id(0);
13   int res = in[gi];
14   #pragma unroll
15   for (int i=0; i<NUM_WRITES; i++) {
16     lmem[li - i] = res;
17     res >>= 1;
18   }
19   barrier(CLK_GLOBAL_MEM_FENCE);
20   res = 0;
21   #pragma unroll
22   for (int i=0; i < NUM_READS; i++) {
23     res ^= lmem[li - i];
24   }
25   out[gi] = res;
26 }
```

Break down Area utilization into BSP, global interconnect, kernels, and line numbers.

Detailed description of local memory with actionable suggestions for improvements.

Total global interconnect configuration.

Area Report
(area utilization values are estimated)

	LEs	FFs	RAMs	DSPs	Details
System Total (Logic: 15%)	**49508 (9%)**	**64399 (6%)**	**387 (15%)**	**0 (0%)**	
Board interface	38262	44528	257	0	• Platform interface logic.
Global interconnect	5034	9568	52	0	• Global interconnect for 1 global load and 1 global store.
big_lmem (Logic: 2%)	**6212 (1%)**	**10303 (1%)**	**78 (3%)**	**0 (0%)**	
Function overhead	1628	1799	0	0	• Kernel dispatch logic.
b.cl:9 (lmem)	132	1024	8	0	• Local memory: Potentially inefficient configuration. Requested size 4096 bytes (rounded up to nearest power of 2), implemented size 8192 bytes, replicated 2 times total, **stallable**, 8 reads and 8 writes. Additional information: - Reduce the number of write accesses or fix banking to make this memory system stall-free. - Replicated 2 times to efficiently support multiple simultaneous workgroups. This replication resulted in no increase in actual block RAM usage. - Banked on lowest dimension into 4 separate banks (this is a good thing).
Block0 (Logic: 1%)	4452 (1%)	7480 (1%)	70 (3%)	0 (0%)	
State	64	64	0	0	
b.cl:13	358	497	13	0	
b.cl:16	779	2591	8	0	• Stallable write to memory declared on b.cl:9.
b.cl:17	25	25	0	0	

All accesses to local memory are described, including their stall status.

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