# Optimizing OpenCL™ for Altera® FPGAs

David Neto

Principal Design Engineer, Altera Corporation

International Workshop on OpenCL, Bristol

2014-05-12

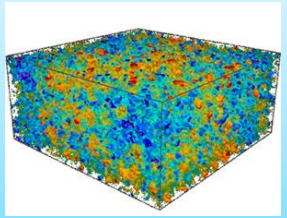
# Performance challenge

## Performance Wanted

Multimedia

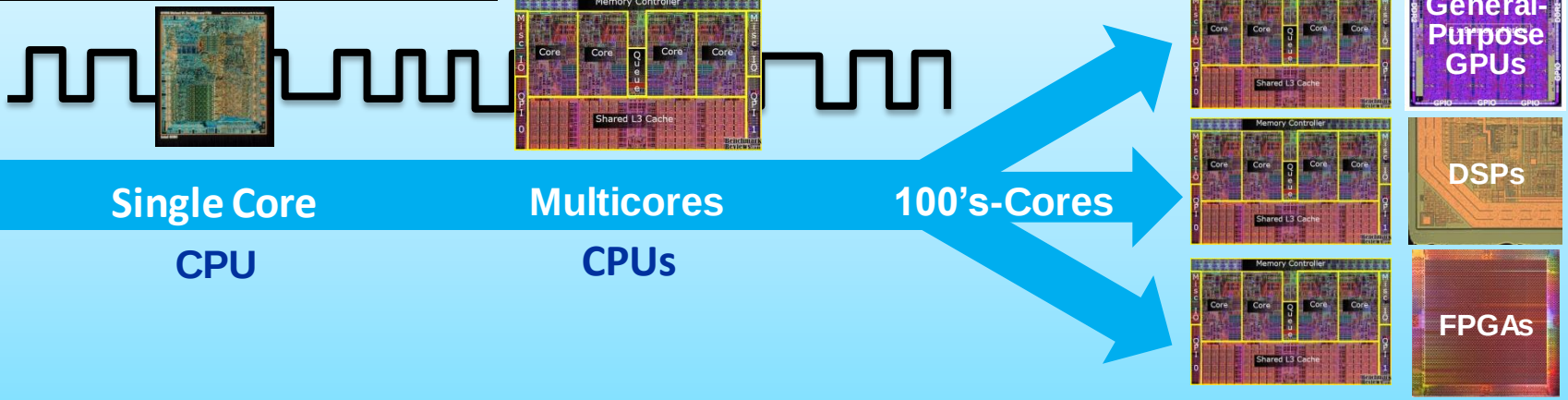
Medical

Radar

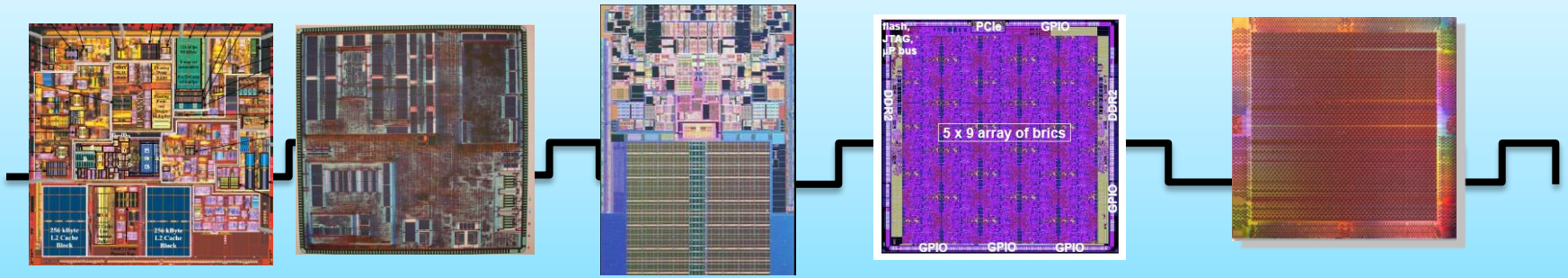


High-Performance Computing

## Architectural Strategies



# Spectrum of approaches to high performance




CPUs DSPs Multi-Cores Arrays FPGAs

Single Cores

Multi-Cores
Coarse-Grained
CPUs and DSPs

Coarse-Grained
Massively
Parallel
Processor
Arrays

Fine-Grained
Massively
Parallel
Arrays

ALTERA
MEASURABLE ADVANTAGE™

**FPGAs are radically different from CPUs and GPUs**

**What kind of OpenCL runs well on an FPGA?**

ALTERA
MEASURABLE ADVANTAGE™

# Outline

- FPGA architecture
- Altera's mapping of OpenCL to FPGAs
- What's expensive, what's cheap
- Design and coding strategies
- Q&A

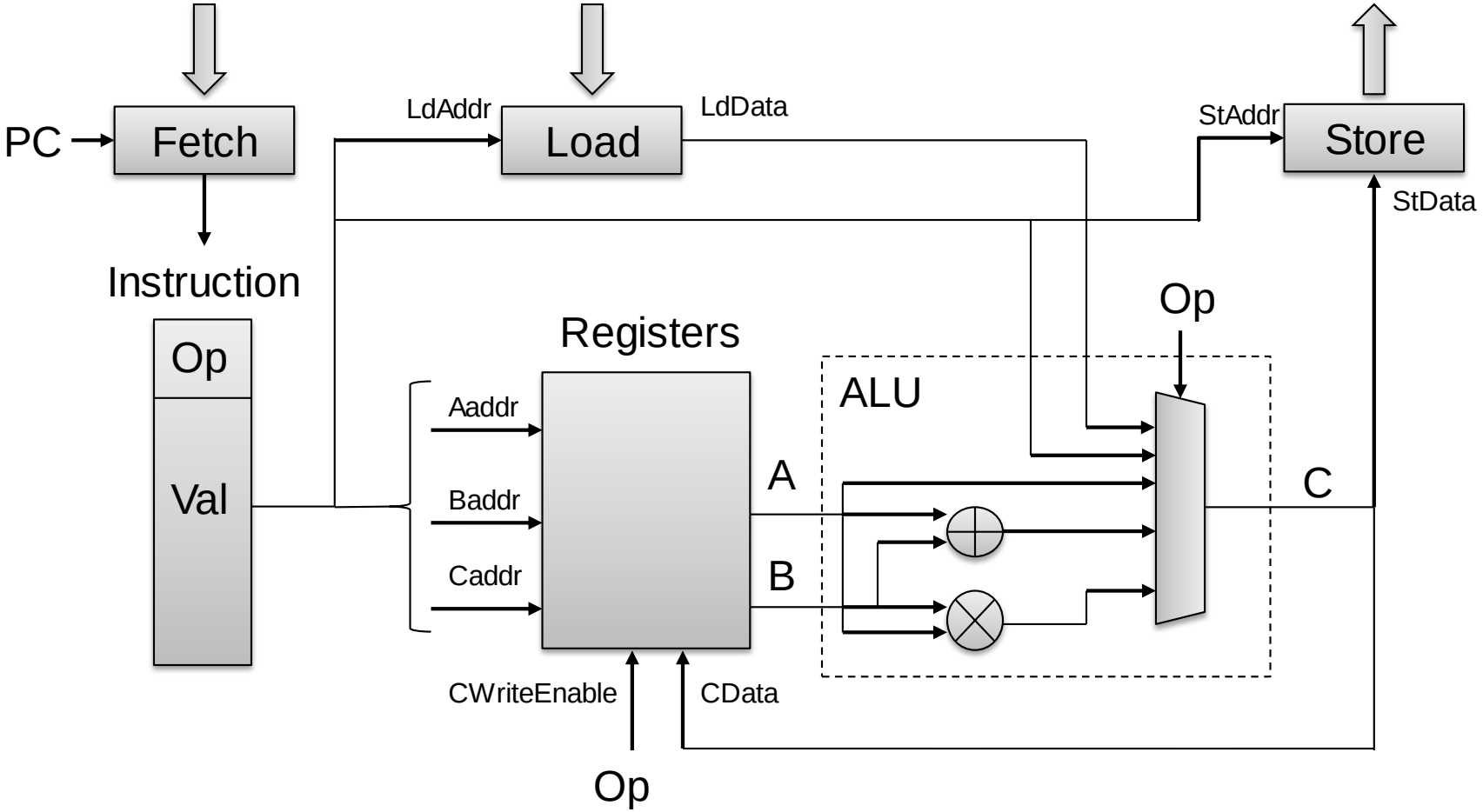
ALTERA
MEASURABLE ADVANTAGE™

# FPGA Architecture

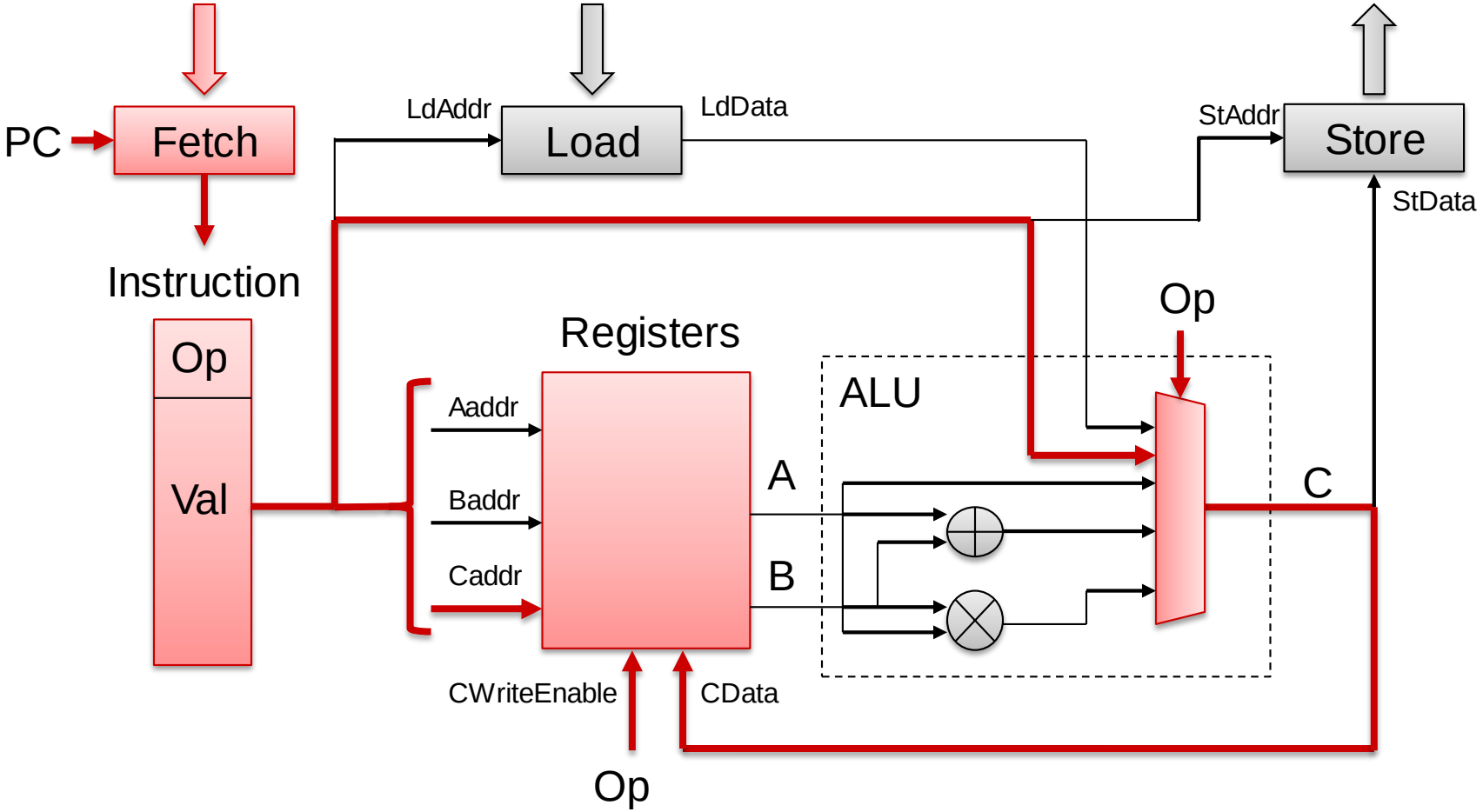
Part 1: FPGAs for software engineers

FPGA datapath ~ Unrolled CPU hardware

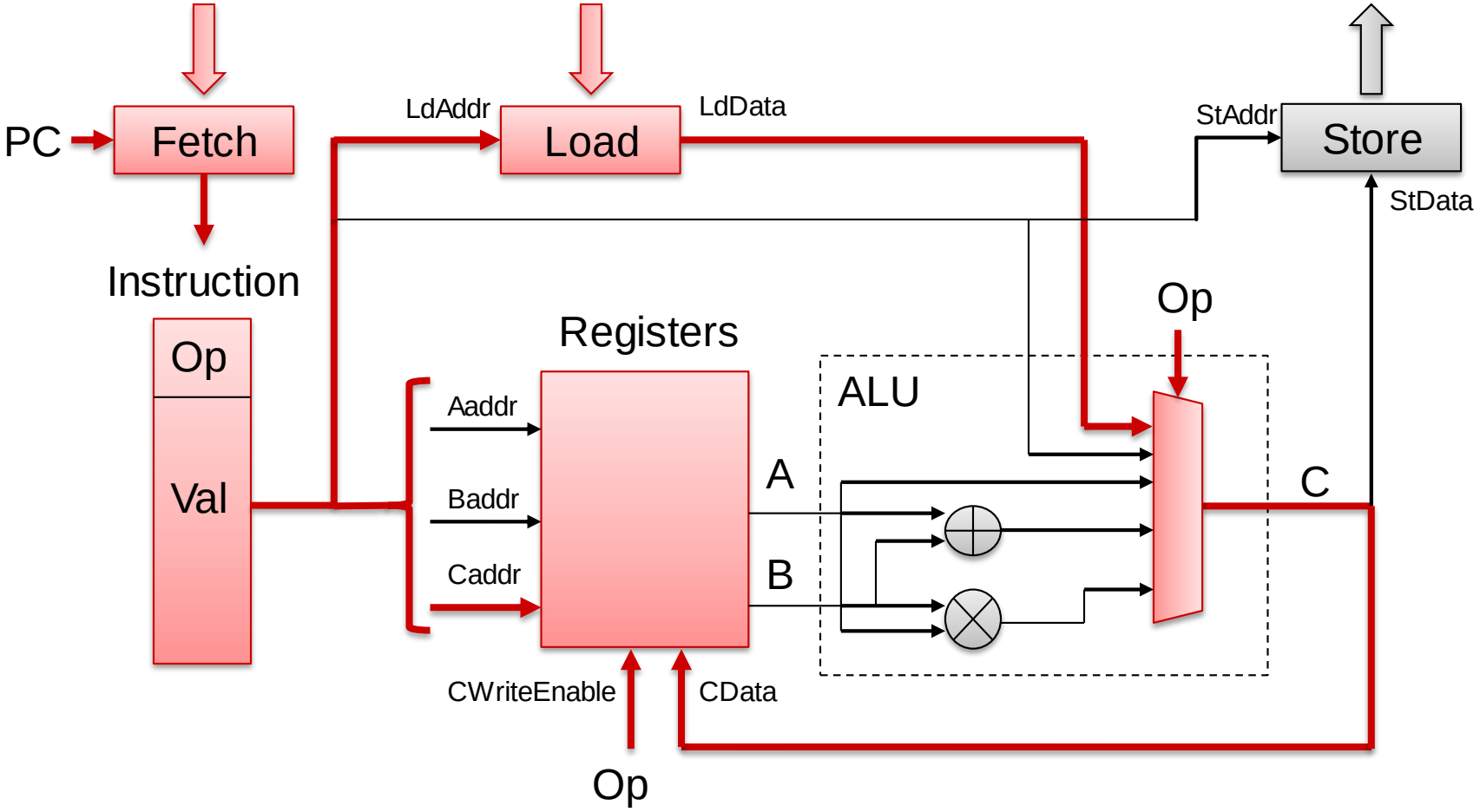
# A simple 3-address CPU



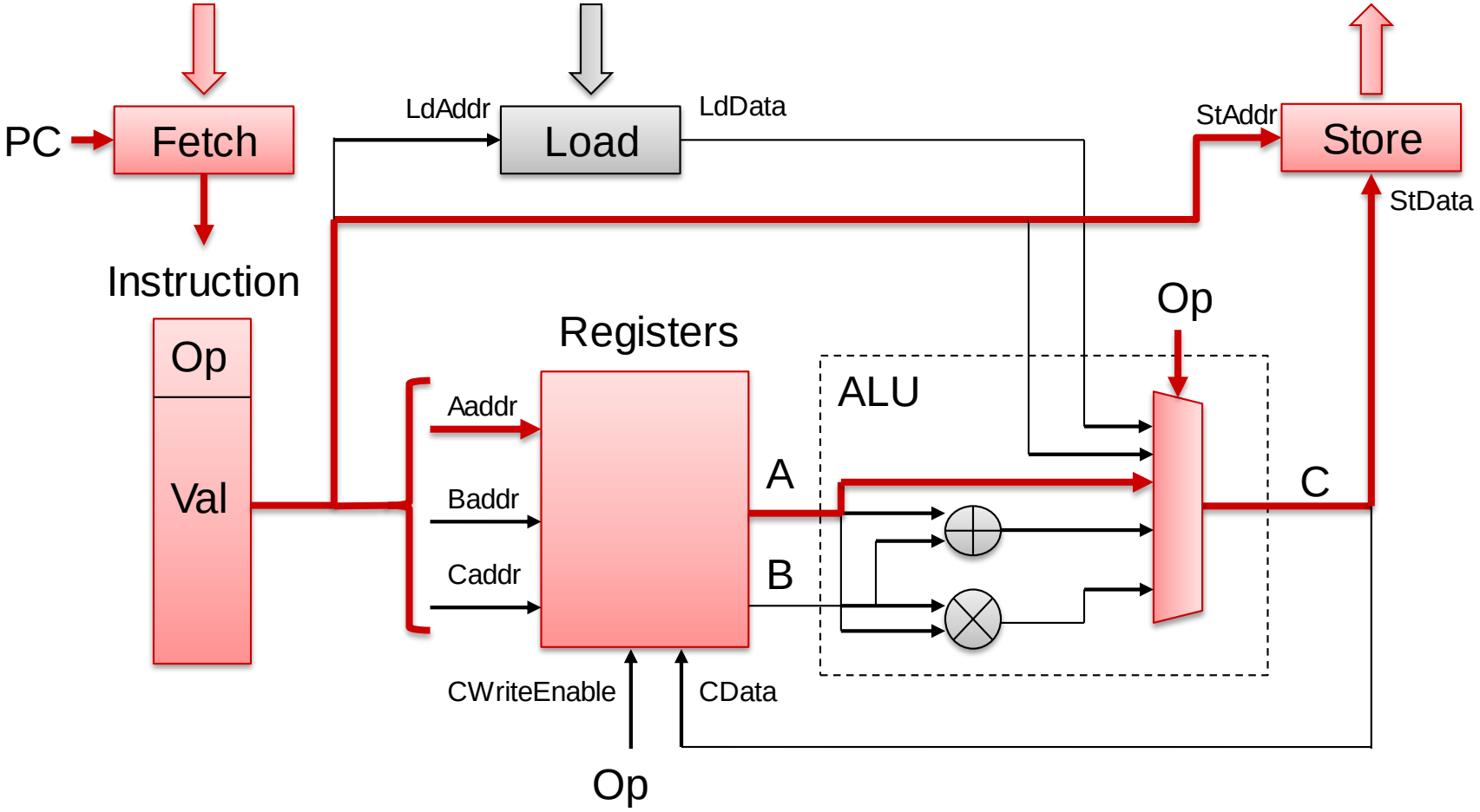
# Load immediate value into register



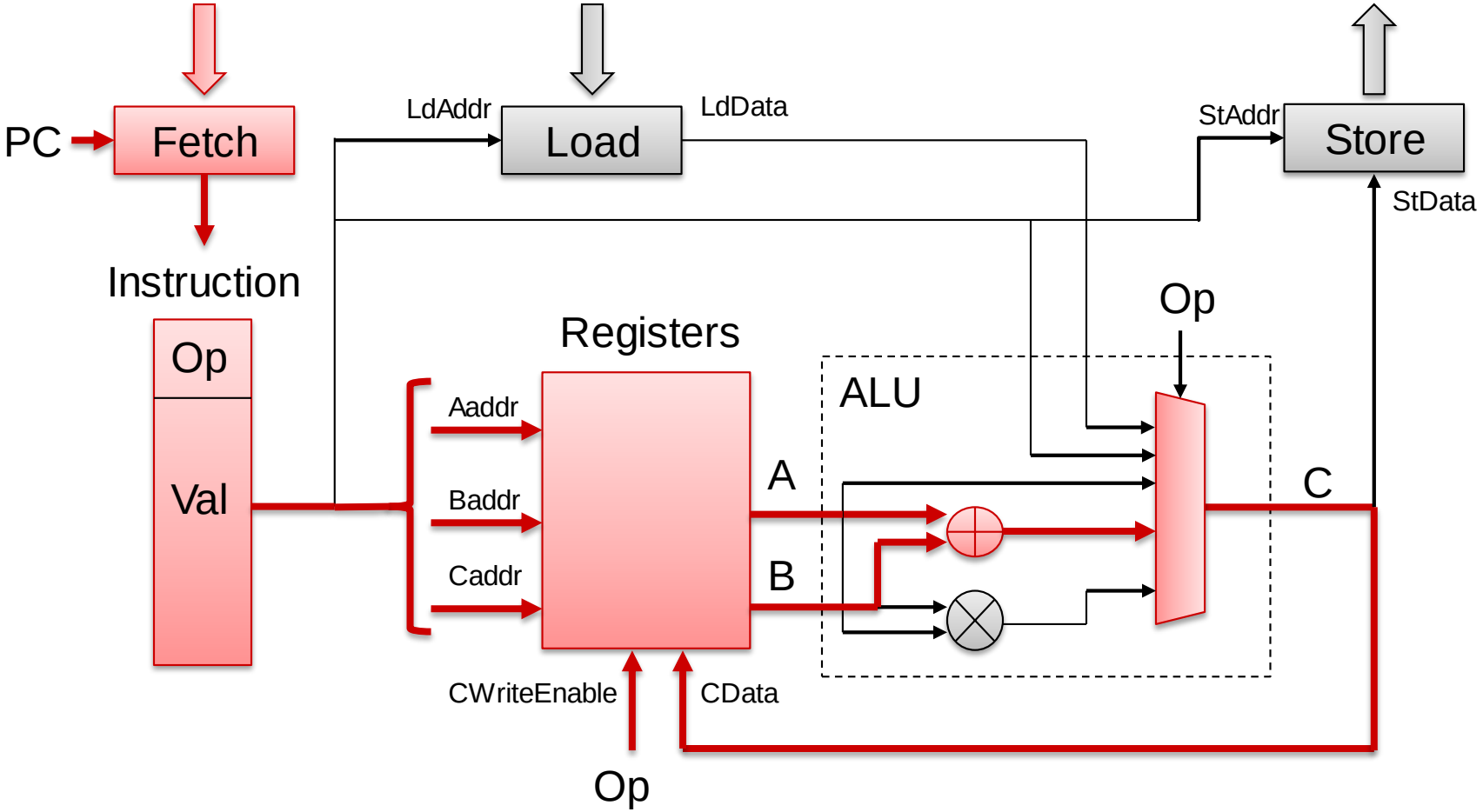
# Load memory value into register



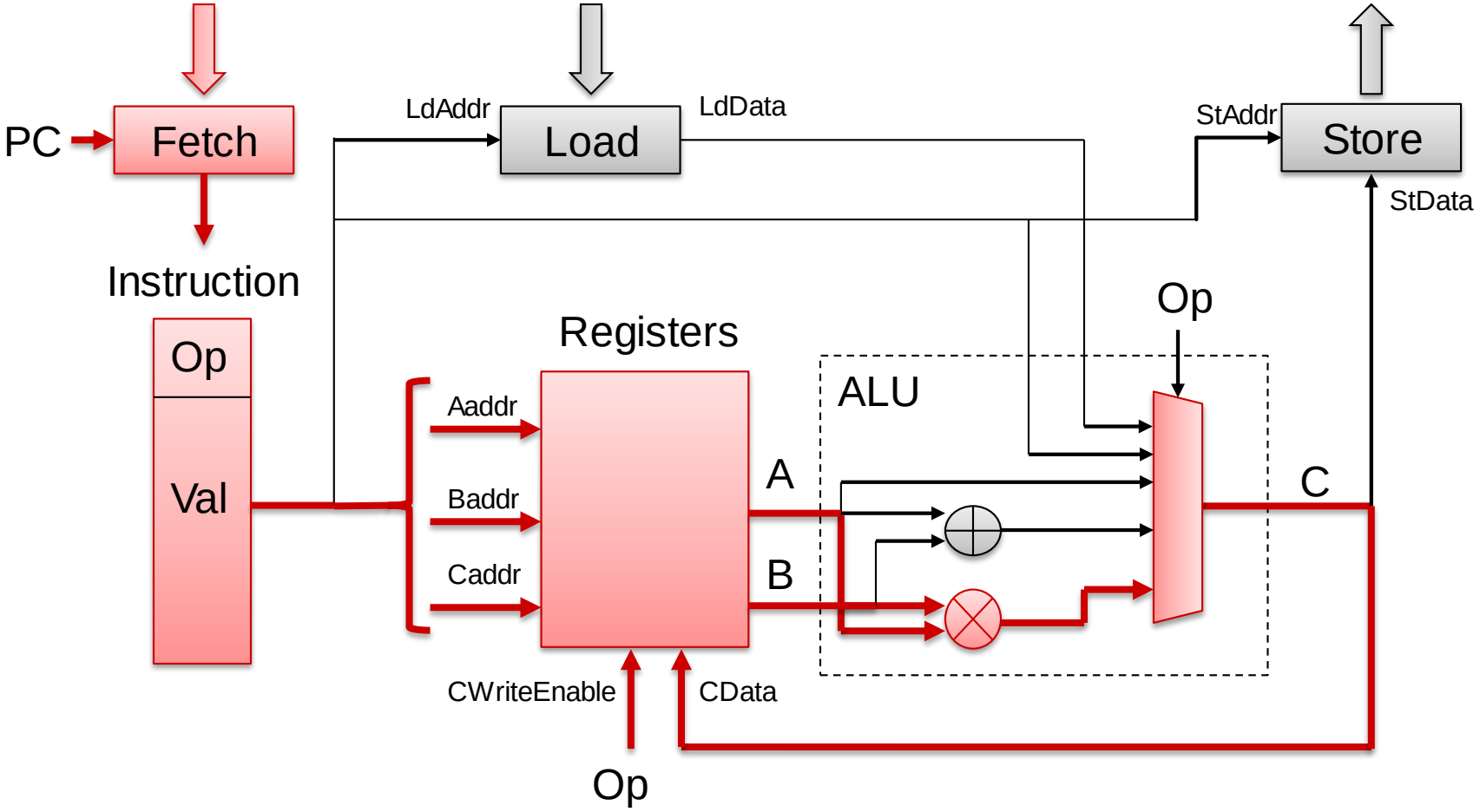
# Store register value into memory



# Add two registers, store result in register



# Multiply two registers, store result in register



# A simple program

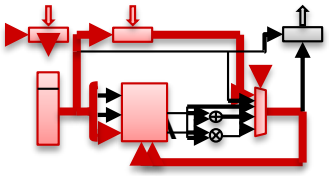
- **Mem[100] += 42 * Mem[101]**

- **CPU instructions:**

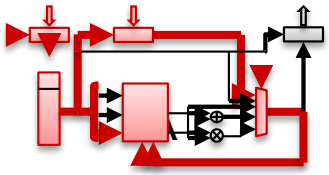
R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]

# CPU activity, step by step

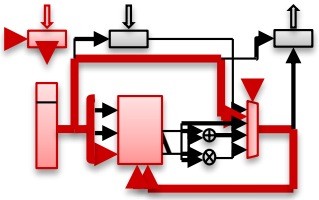
R0 ← Load Mem[100]



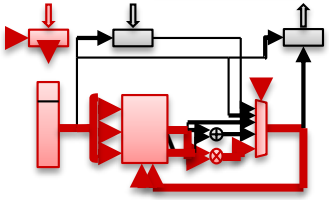
R1 ← Load Mem[101]



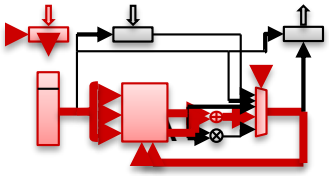
R2 ← Load #42



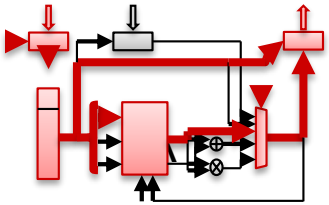
R2 ← Mul R1, R2



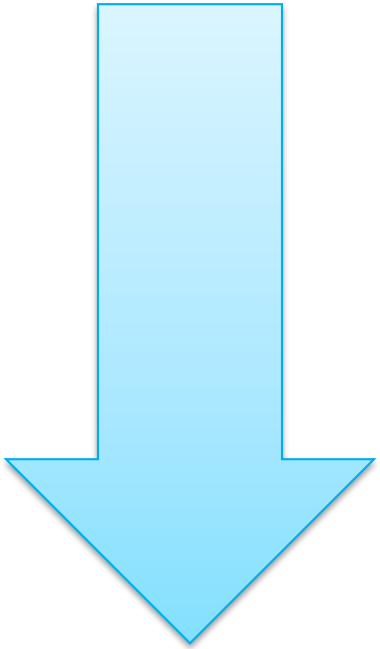
R0 ← Add R2, R0



Store R0 → Mem[100]

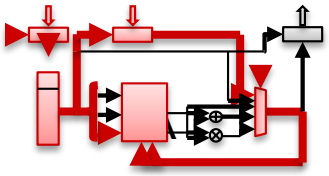


Time

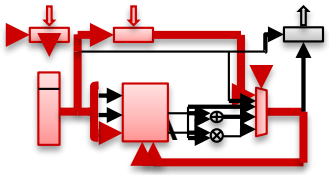


# Unroll the CPU hardware…

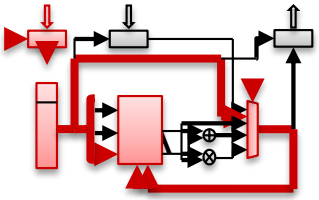
R0 ← Load Mem[100]



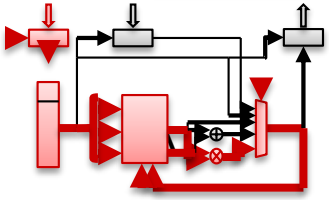
R1 ← Load Mem[101]



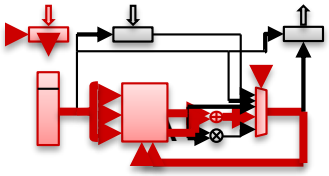
R2 ← Load #42



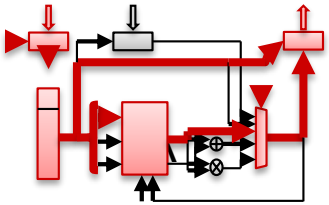
R2 ← Mul R1, R2



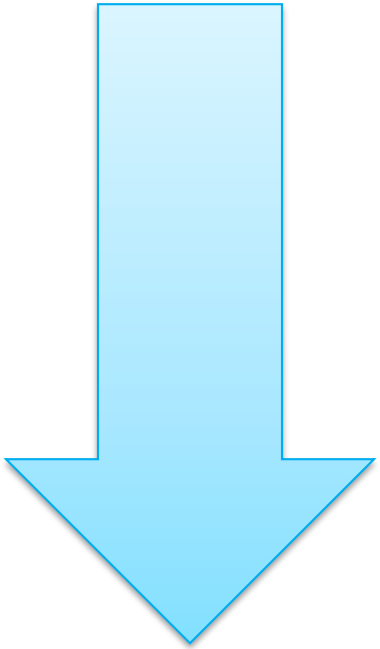
R0 ← Add R2, R0



Store R0 → Mem[100]

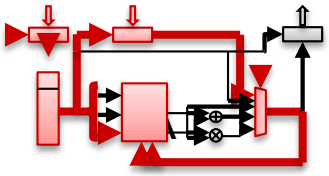


Space

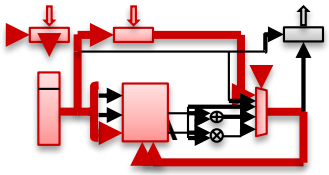


# … and specialize by position

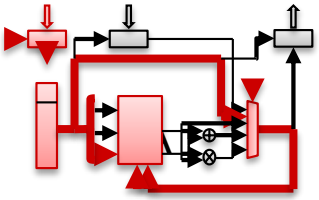
R0 ← Load Mem[100]



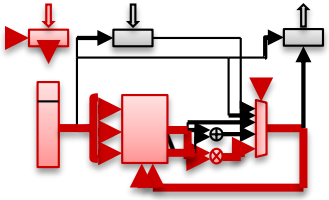
R1 ← Load Mem[101]



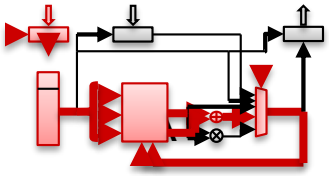
R2 ← Load #42



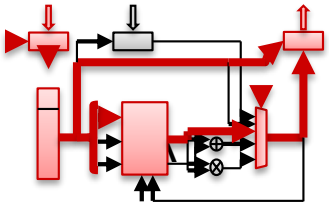
R2 ← Mul R1, R2



R0 ← Add R2, R0



Store R0 → Mem[100]



1. Instructions are fixed. Remove “Fetch”

# … and specialize

R0 ← Load Mem[100]

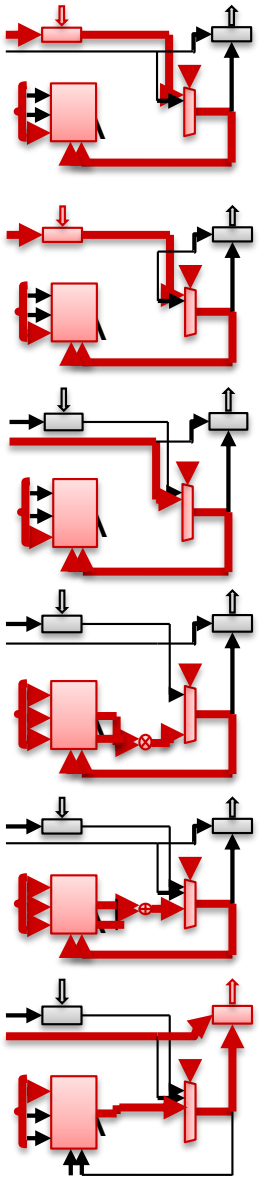
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

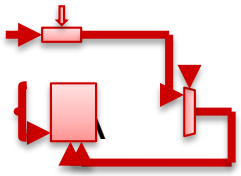
Store R0 → Mem[100]



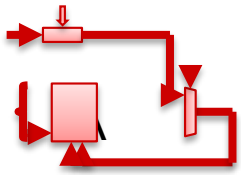
1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops

# … and specialize

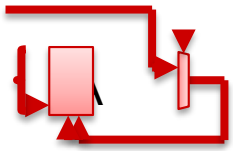
R0 ← Load Mem[100]



R1 ← Load Mem[101]

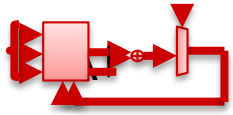


R2 ← Load #42

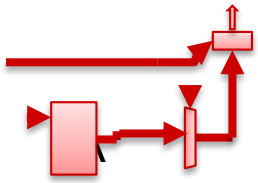


R2 ← Mul R1, R2

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]

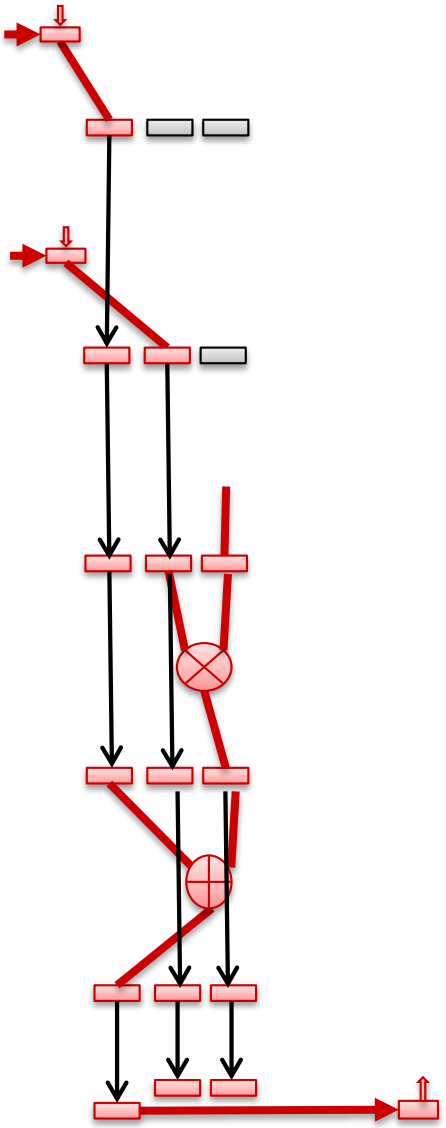
R1 ← Load Mem[101]

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]

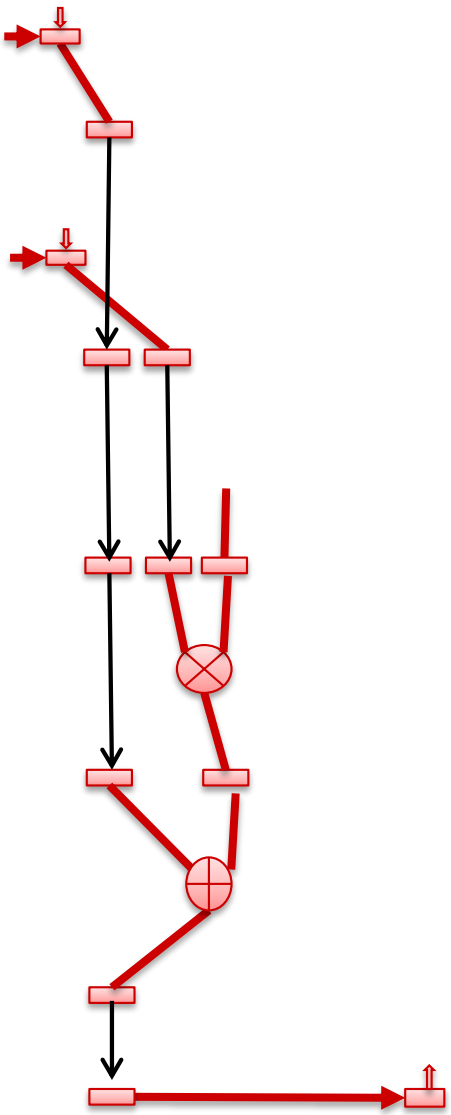
R1 ← Load Mem[101]

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.

# … and specialize

R0 ← Load Mem[100]

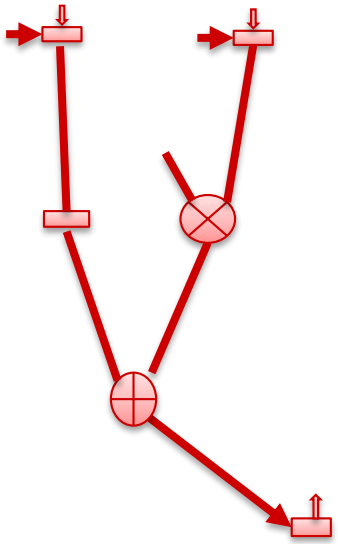
R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

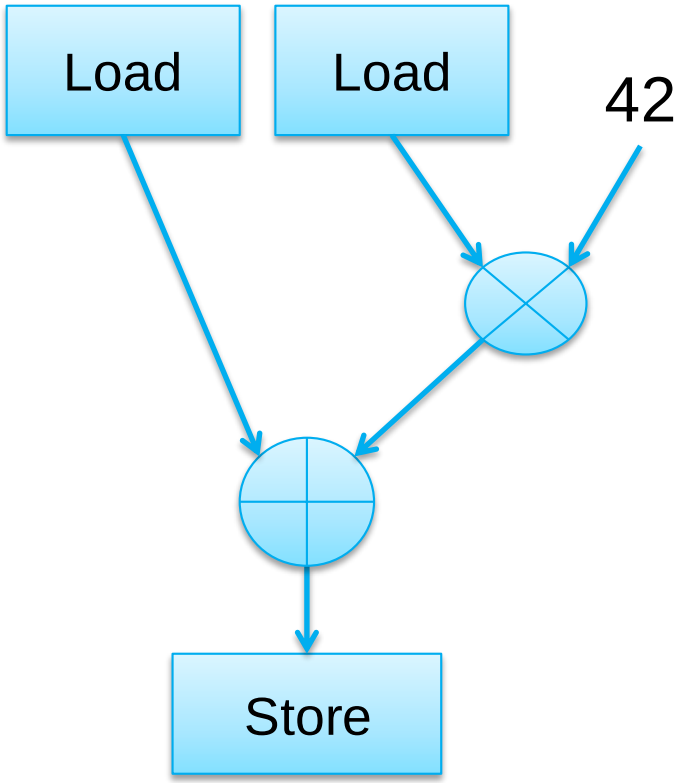
Store R0 → Mem[100]



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.
6. Reschedule!

ALTERA
MEASURABLE ADVANTAGE™

# FPGA datapath = Your algorithm, in silicon



# So what?

## FPGA datapath = Your algorithm, in silicon

**Build exactly what you need:**

**Operations**

**Data widths**

**Memory size, configuration**

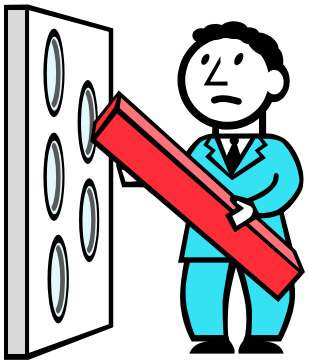
**Efficiency:**

**Throughput / Latency / Power**

ALTERA
MEASURABLE ADVANTAGE™

# Deep thought #1

**OpenCL code is portable**
**Not always performance portable**



**Would you rather contort your code,**
**Or contort your machine?**

**Altera gives you a program-specific machine**

ALTERA
MEASURABLE ADVANTAGE™

# FPGA Architecture

Part 2: Business influences

Why FPGAs are the way they are

# Wide range of applications

## Consumer Automotive

### Entertainment
Broadband
Audio/video
Video display

### Automotive
Navigation
Entertainment

## Test, Measurement & Medical

### Instrumentation
Medical
Test equipment
Manufacturing

## Communications Broadcast

### Wireless
Cellular
Basestations
Wireless LAN

### Networking
Switches
Routers

### Wireline
Optical
Metro
Access

### Broadcast
Studio
Satellite
Broadcasting

## Military & Industrial

### Military
Secure comm.
Radar
Guidance and control

### Security & Energy Management
Card readers
Control systems
ATM

## Computer & Storage

### Computers
Servers
Mainframe

### Storage
RAID
SAN

### Office Automation
Copiers
Printers
MFP

ALTERA
MEASURABLE ADVANTAGE™

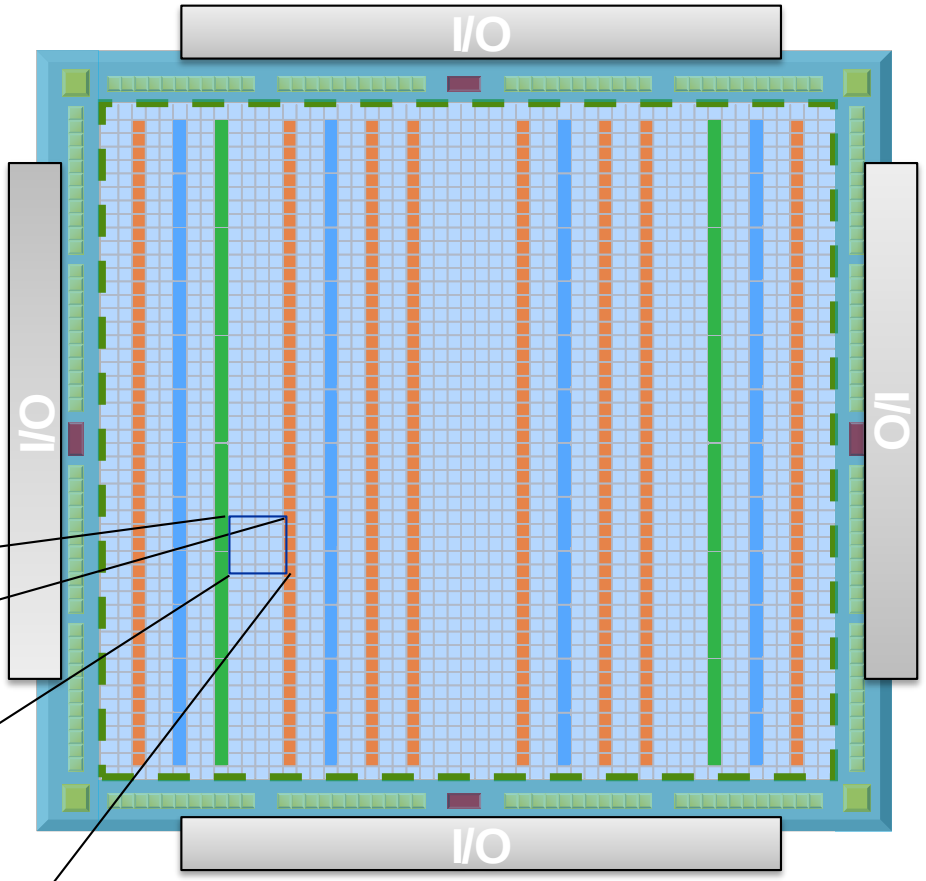
# Typical FPGA use cases (up to now)

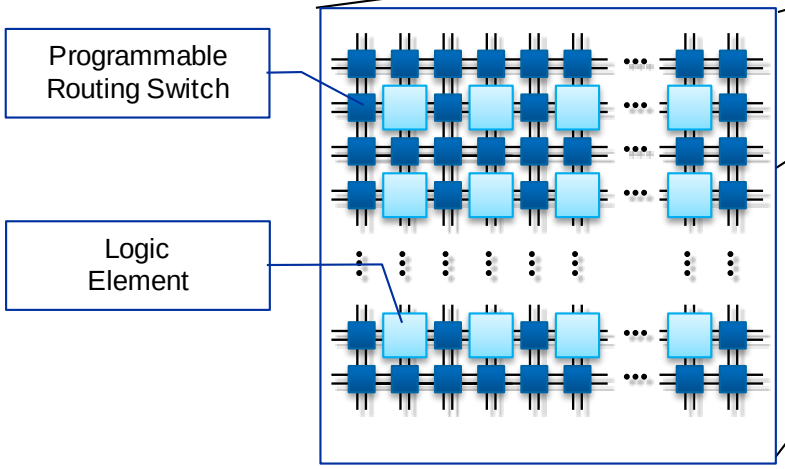
- **Technical demands**
  - CPU / GPU too slow or power hungry
  - Exotic high speed IO
  - Hard real-time
  - Can't afford 100M$ and 2 year design cycle for an ASIC
- **Deployment scenario**
  - Usually single long-lived application
- **Consequences**
  - At the edge of silicon capability
  - Extreme flexibility and control
  - Custom embedded system
  - Higher initial design investment (than software)

ALTERA
MEASURABLE ADVANTAGE™

# Altera FPGA: fine grain massively parallel array

- **Massive Parallelism**
  - Millions of **bit level** logic elements
  - Thousands of 20Kb memory blocks
  - Thousands of DSP blocks
  - Dozens of High-speed transceivers
    - E.g. 28Gb/s each
  - Millions of programmable wires
- ***Traditionally* hardware-centric design flow**



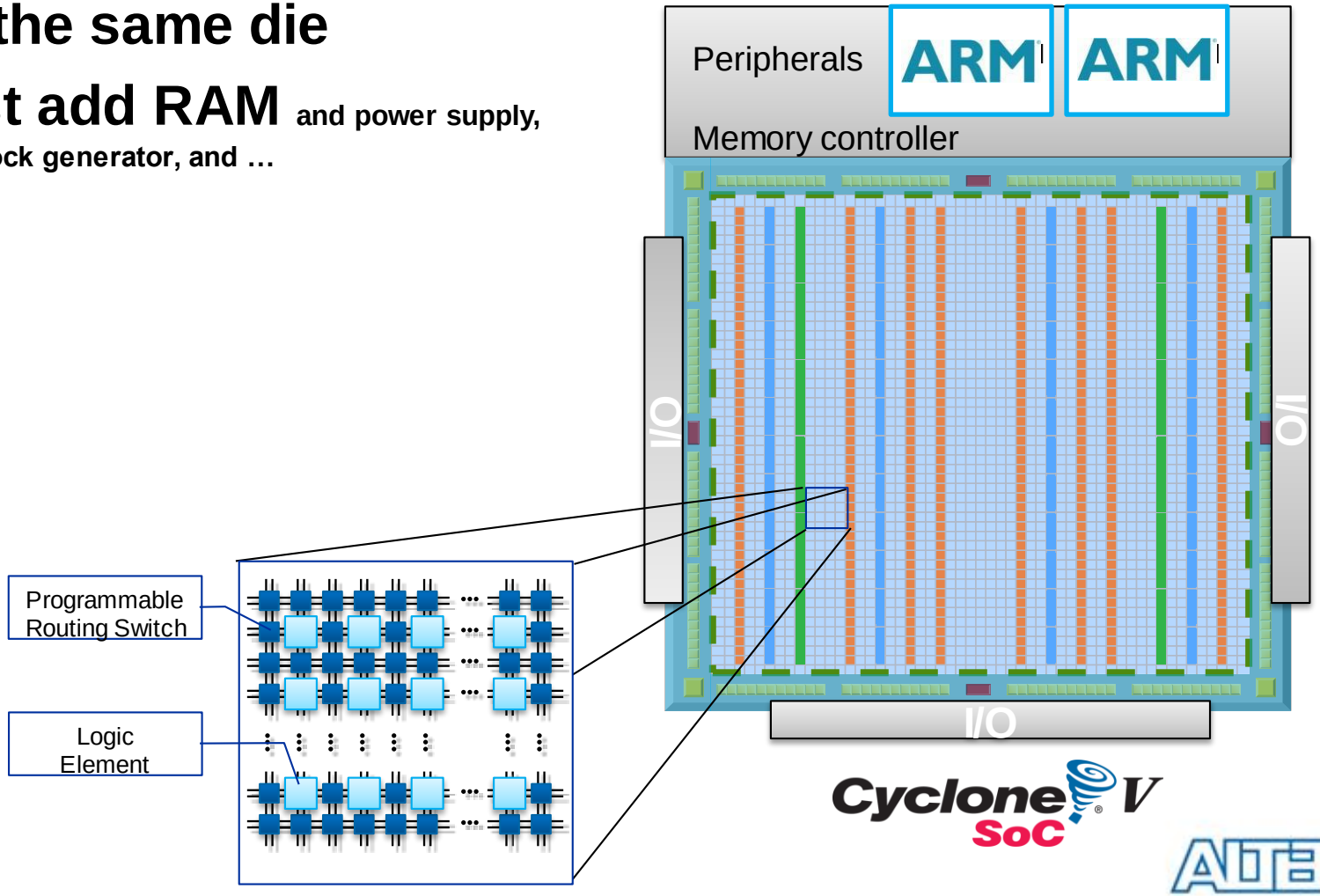




Stratix V

ALTERA
MEASURABLE ADVANTAGE™

# Altera SoC FPGA: ARM® processors on the die

- **OpenCL Host and Device on the same die**
- **Just add RAM** and power supply, and clock generator, and …




ALTERA
MEASURABLE ADVANTAGE™

# Mapping OpenCL to Altera FPGAs

Altera's SDK for OpenCL:

Software design flow for Altera FPGAs

Exploit FPGA strengths

# Altera SDK for OpenCL

- Two major releases a year
- May 2013: v13.0: OpenCL conformance


- Nov 2013: v13.1: Board partner program
- Coming soon: v14.0

ALTERA
MEASURABLE ADVANTAGE™

# Compiling OpenCL to FPGAs

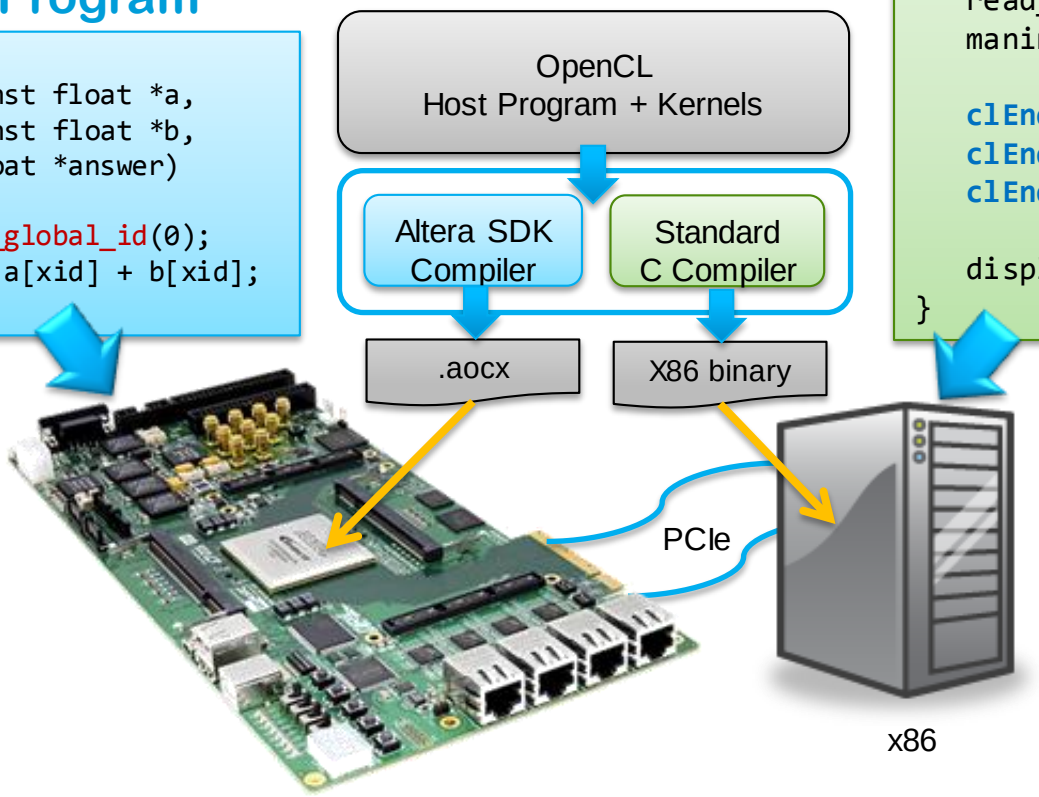
## Host Program

```
main()
{
   cl_program prog
   = clCreateProgramWithBinary(… );

   read_data_from_file( … );
   maninpulate_data( … );

   clEnqueueWriteBuffer( … );
   clEnqueueNDRangeKernel(…,sum, …);
   clEnqueueReadBuffer( … );

   display_result_to_user( … );
}
```

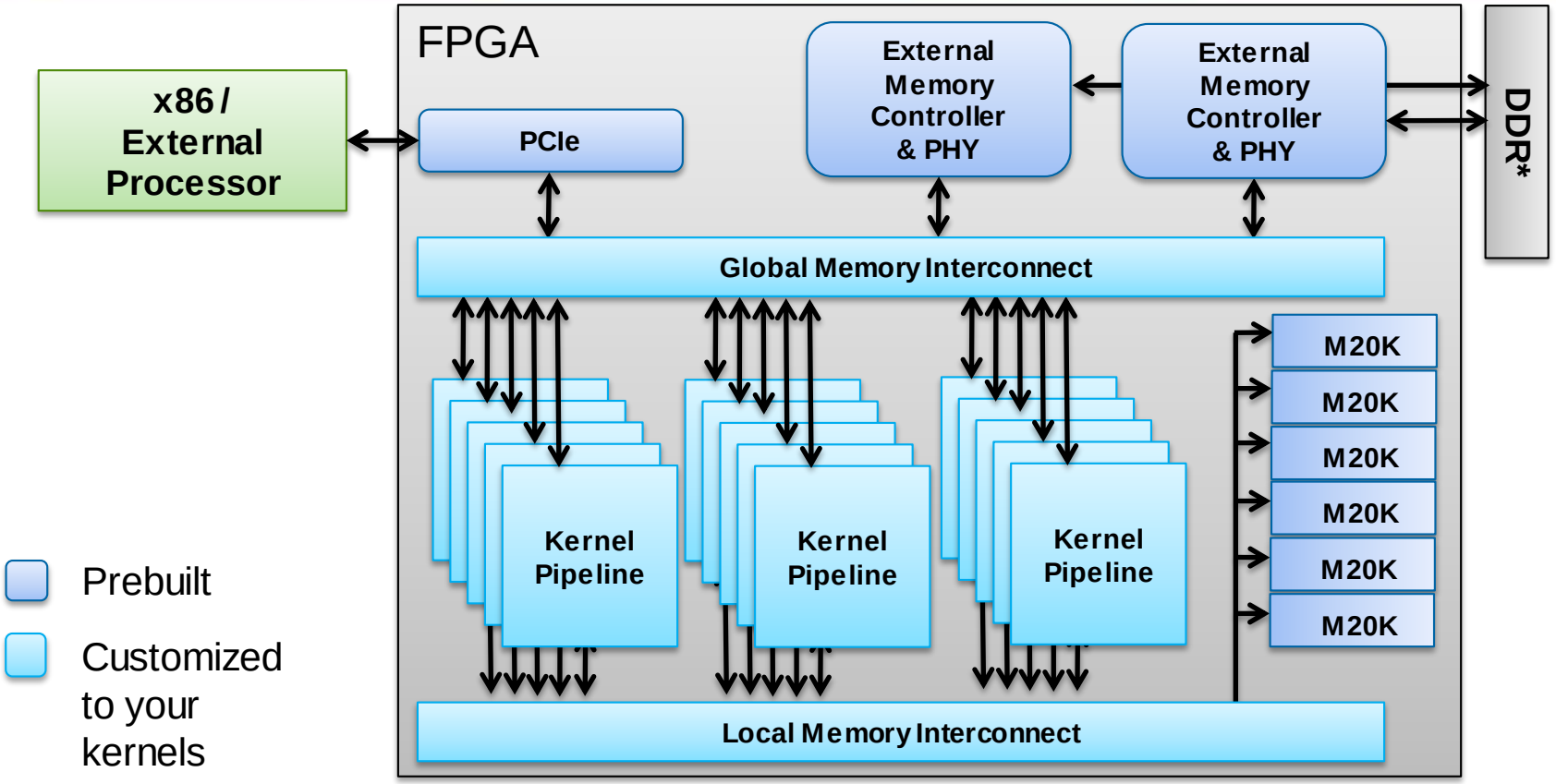
Production flow:
Offline compilation only

## Kernel Program

```
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];
}
```



# FPGA OpenCL Architecture




***Modest external memory bandwidth***

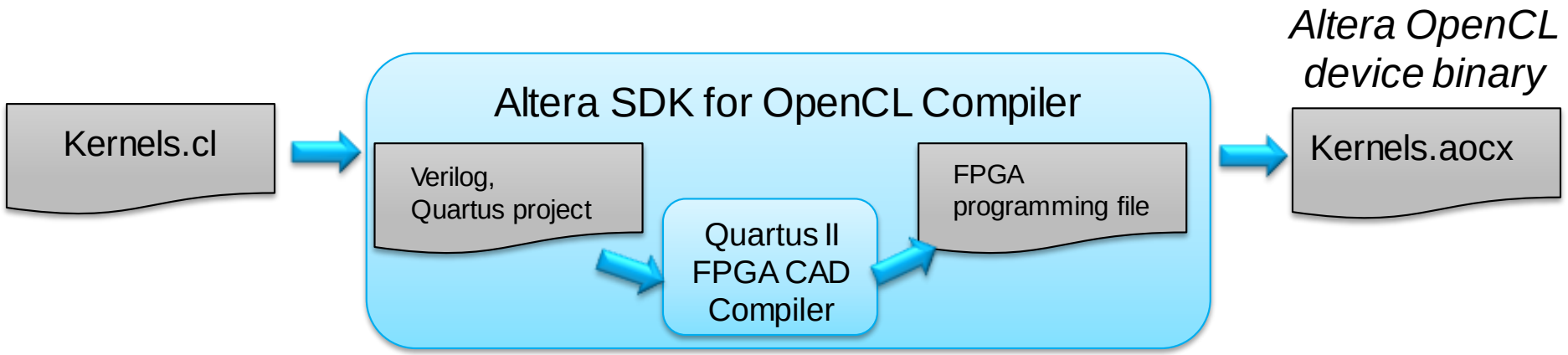
***Extremely high internal memory bandwidth***

***Highly customizable compute cores***

ALTERA
MEASURABLE ADVANTAGE™

# Relevant questions differ by architecture

| GPU | FPGA |
|---|---|
| How many private registers? | How much **area** does this kernel occupy? |
| How much local memory? | What's the **initiation interval**? |
| How many compute units? | *How do I get a license?* |
| How many processing elements? | *I need Quartus® II?* |
| What is the memory banking scheme? | *What's Quartus II?* |



# Exploiting FPGA strengths

- Pipelined parallelism
- Agnostic to divergent control flow
- Optimized mix of operations, functions
- Customized local, global, constant memory

- (All traditional compiler optimizations too)

# The BIG Idea behind OpenCL

- **OpenCL execution model …**
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

**Traditional loops**

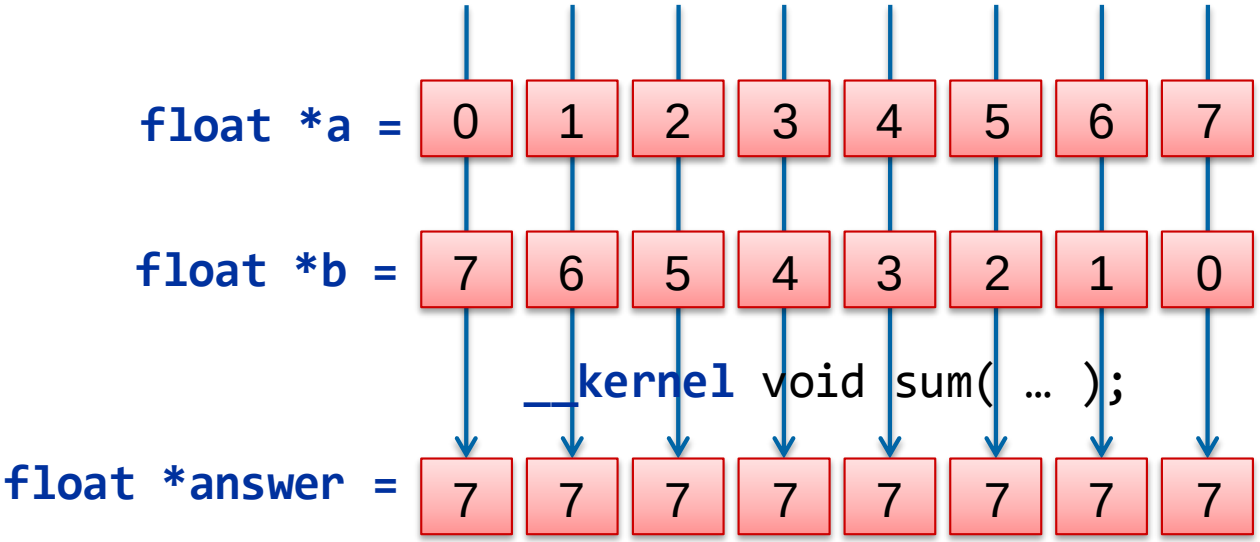
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

**Data Parallel OpenCL**

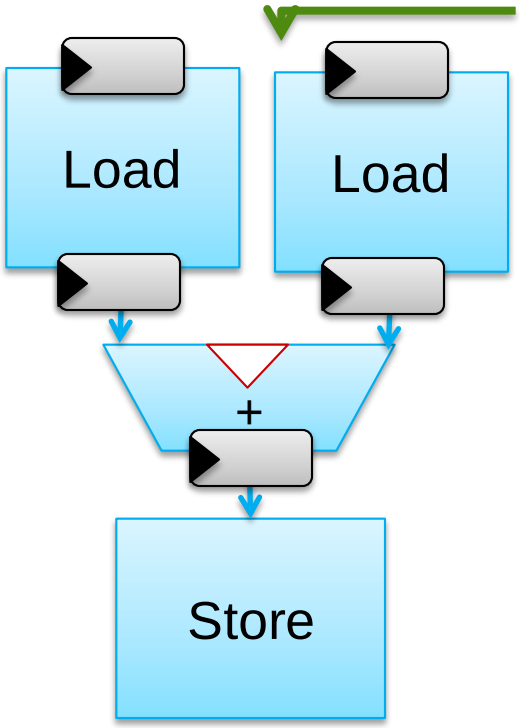
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

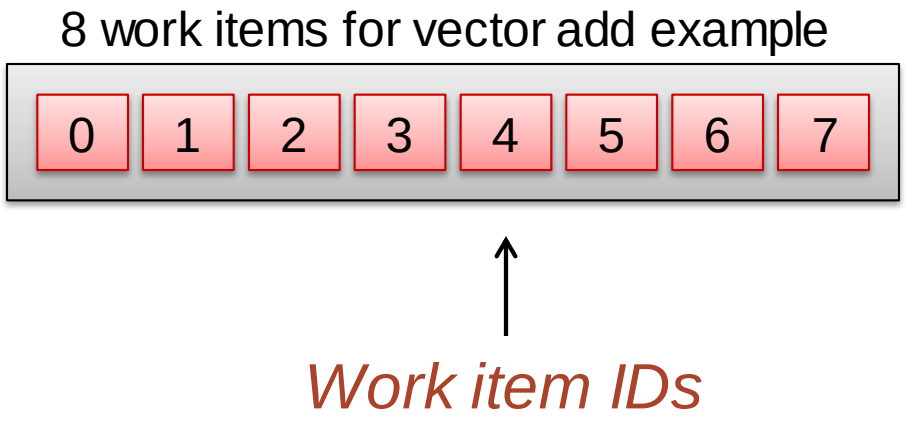
# Data parallel kernel

```
__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];
}
```



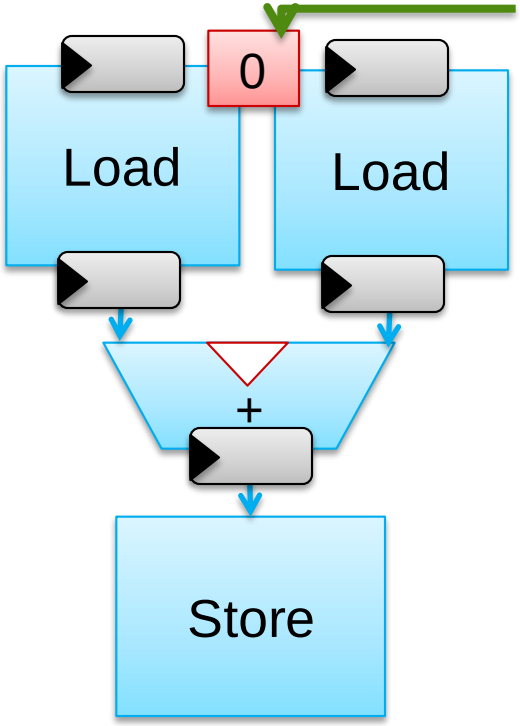
# Example Pipeline for Vector Add

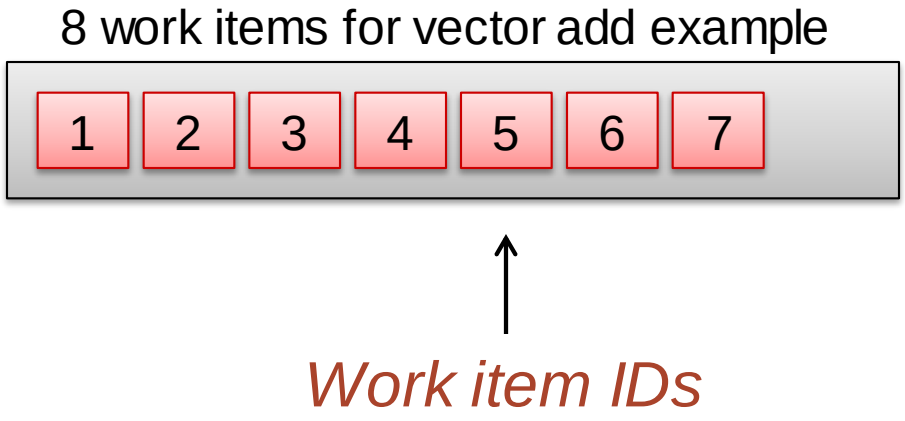



- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

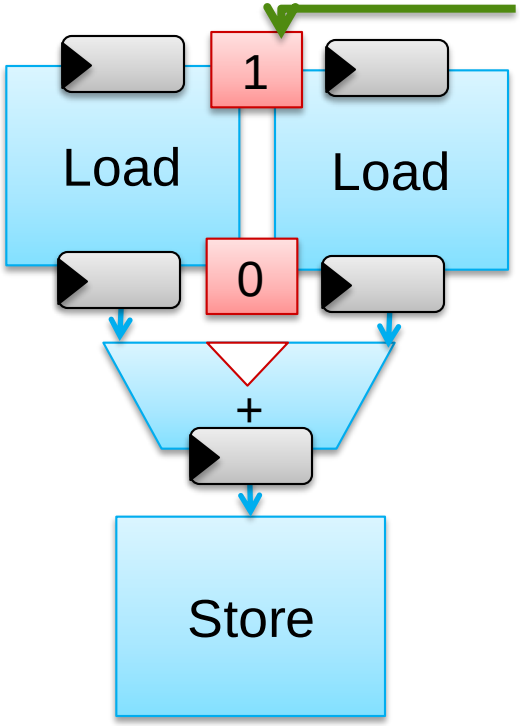
# Example Pipeline for Vector Add

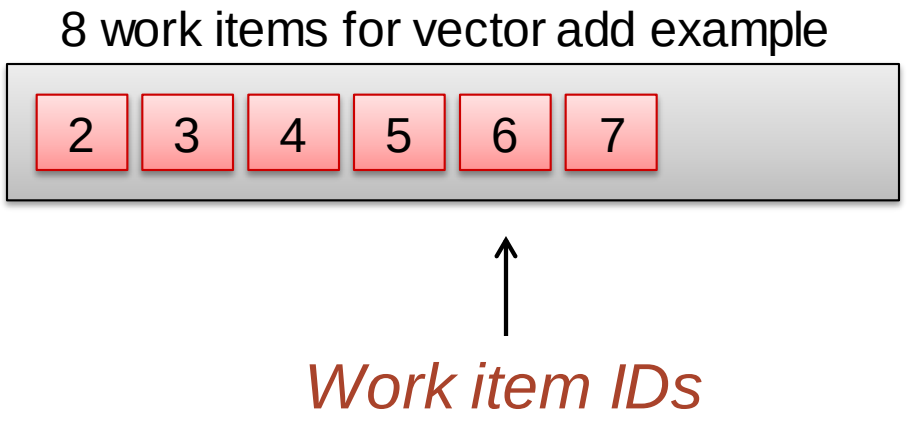



- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

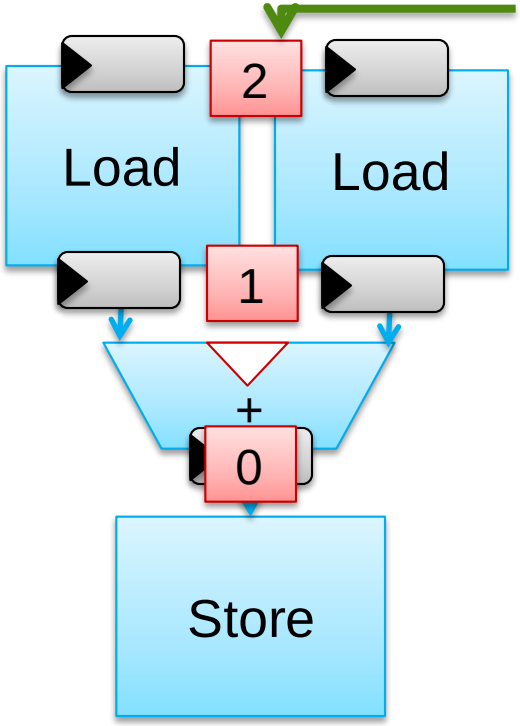
# Example Pipeline for Vector Add

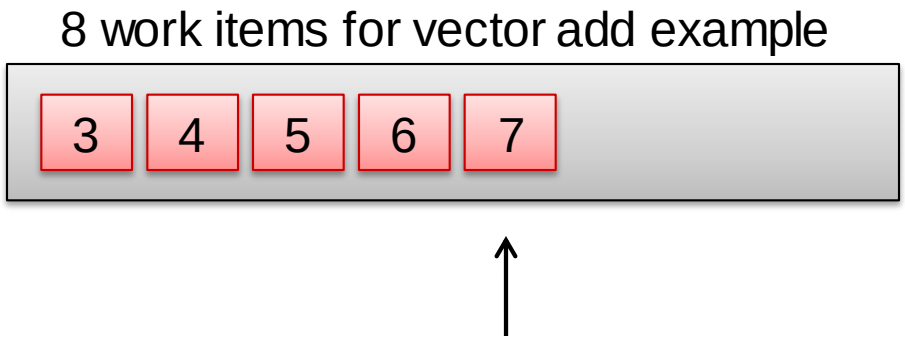



- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

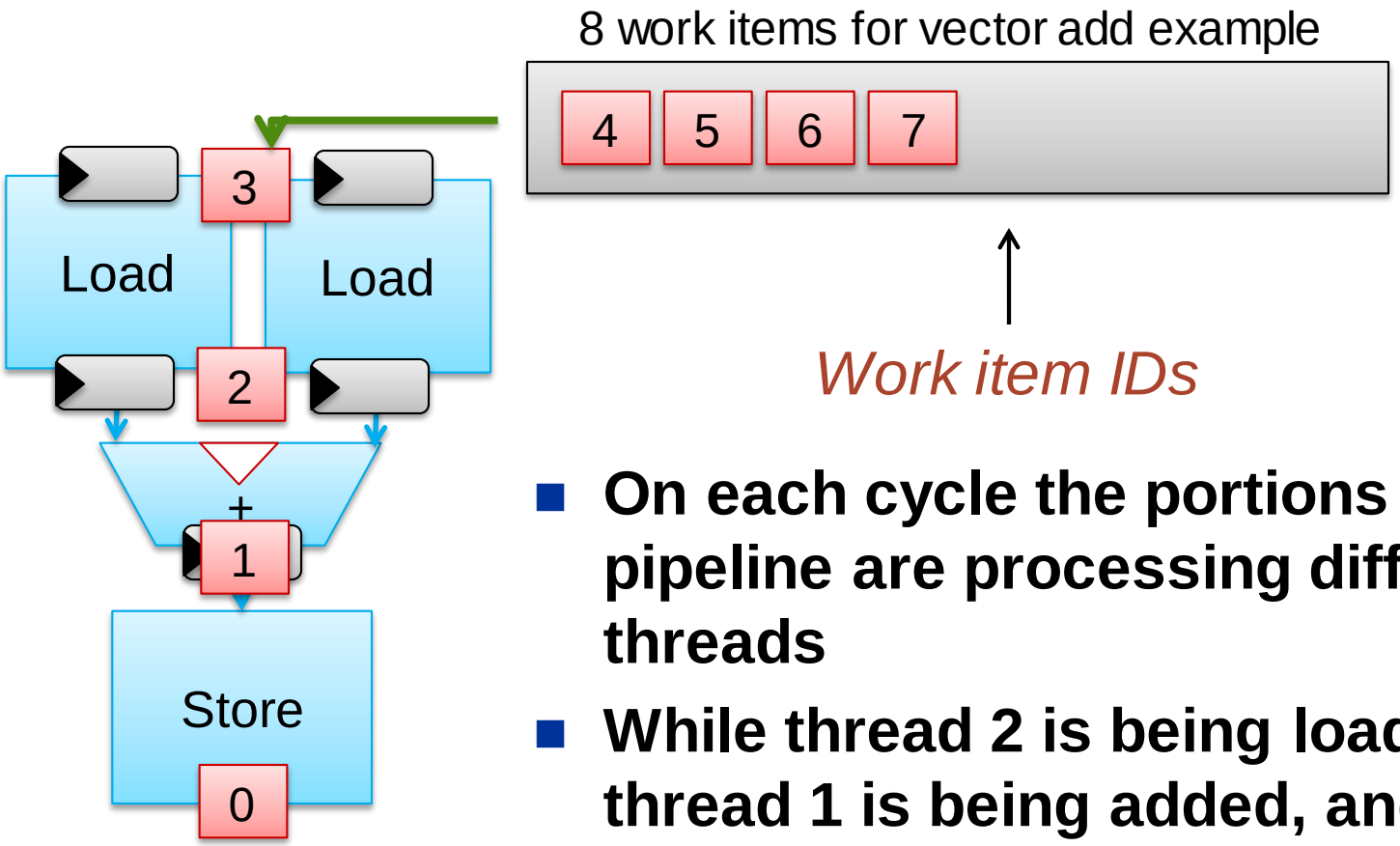
# Example Pipeline for Vector Add




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

# Example Pipeline for Vector Add




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

***All silicon used efficiently at steady-state***

# So what's expensive, and what's cheap?

# Cheap operations

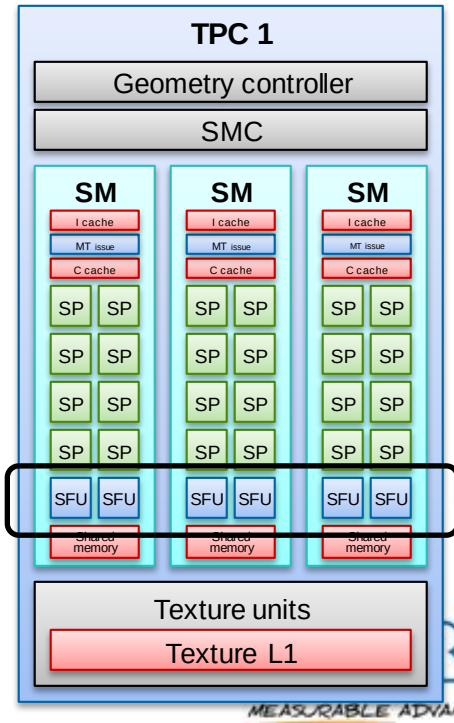
- **Bit manipulation**
  - Nearly free
- **Simple integer arithmetic**
  - add, sub, mul
  - The narrower the better, e.g. short vs. int vs. long
- **Use of private memory**
  - It's abundant, especially if structured as shift-registers.
- **Example applications:**
  - Encryption, hashing, fixed point signal processing

ALTERA
MEASURABLE ADVANTAGE™

# Optimal function mix: E.g. Inverse Normal CDF

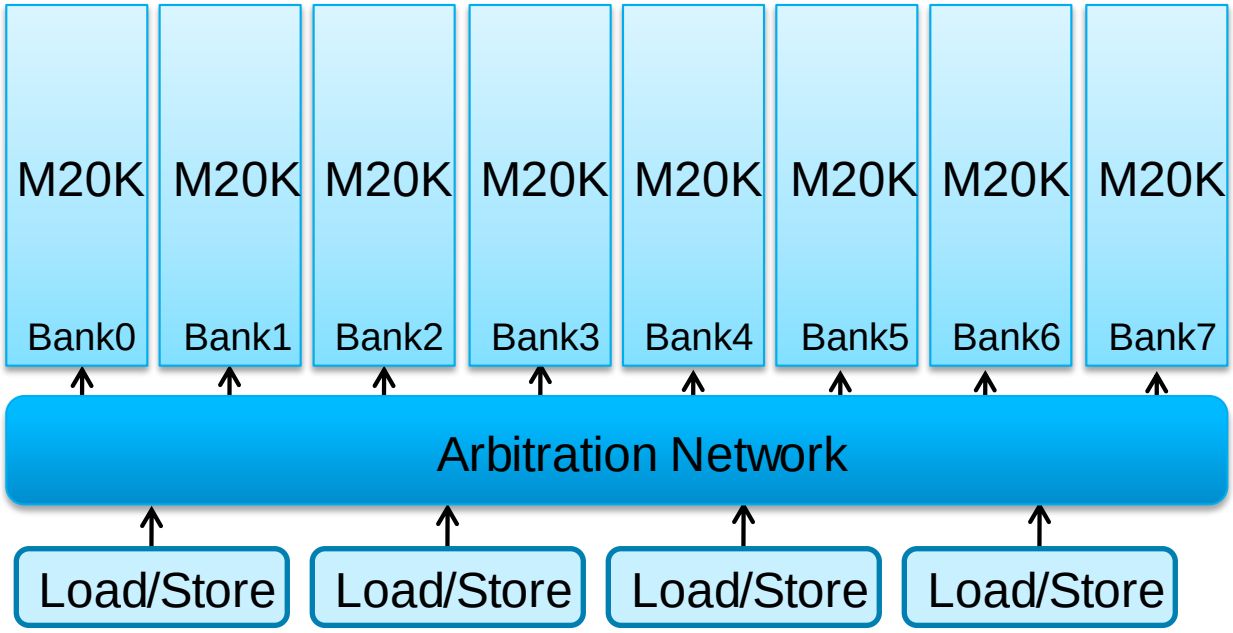
```
float ltqnorm(float p)
{
   float q, r;
   if (p < 0 || p > 1) { return 0.0; }
   else if (p == 0) { return -HUGE_VAL /* minus "infinity" */; }
   else if (p == 1) { return HUGE_VAL /* "infinity" */; }
   else if (p < LOW) {
      /* Rational approximation for lower region */
      q = sqrt(-2*log(p));
      return (((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
             ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
   } else if (p > HIGH) {
      /* Rational approximation for upper region */
      q  = sqrt(-2*log(1-p));
      return -(((((c[0]*q+c[1])*q+c[2])*q+c[3])*q+c[4])*q+c[5]) /
             ((((d[0]*q+d[1])*q+d[2])*q+d[3])*q+1);
   } else {
      /* Rational approximation for central region */
      q = p - 0.5;
      r = q*q;
      return (((((a[0]*r+a[1])*r+a[2])*r+a[3])*r+a[4])*r+a[5])*q /
             (((((b[0]*r+b[1])*r+b[2])*r+b[3])*r+b[4])*r+1);
   }
}
```

- Complex functions sqrt, log
- Require (scarce) special function unit on typical GPU
- ***FPGA custom datapath: exactly the right balance of function units***



# Optimized Local Memory: Customized to your program

- **Abundant: Not just 32KB**
  - Stitch together small blocks as needed
- **Alias analysis enables parallel access**
- **Custom access widths, banking**

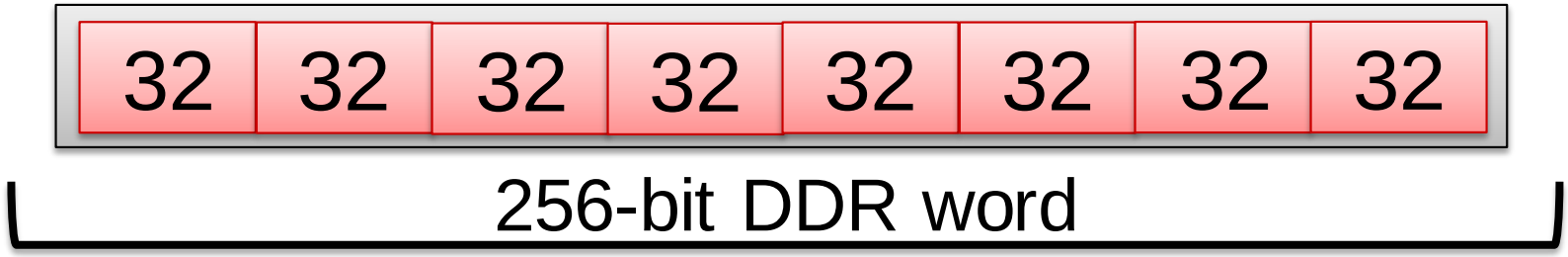



***Typically single cycle access***

***as wide as your code wants***

# Optimized Global Memory: Coalescing

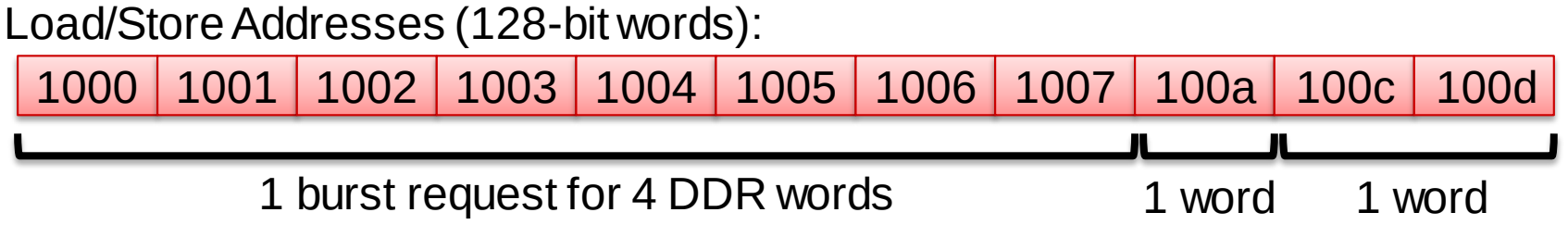
- **External memory has wide words (256 bits)**
- **Loads/stores typically access narrower words (32-128) bits)**




- **Combine requests to maximize DDR efficiency**
  - Reduce thrashing in DDR "protocol"

# Coalescing example

- **Dynamic: Hardware exploits runtime pattern**




*→ 3 requests in total*

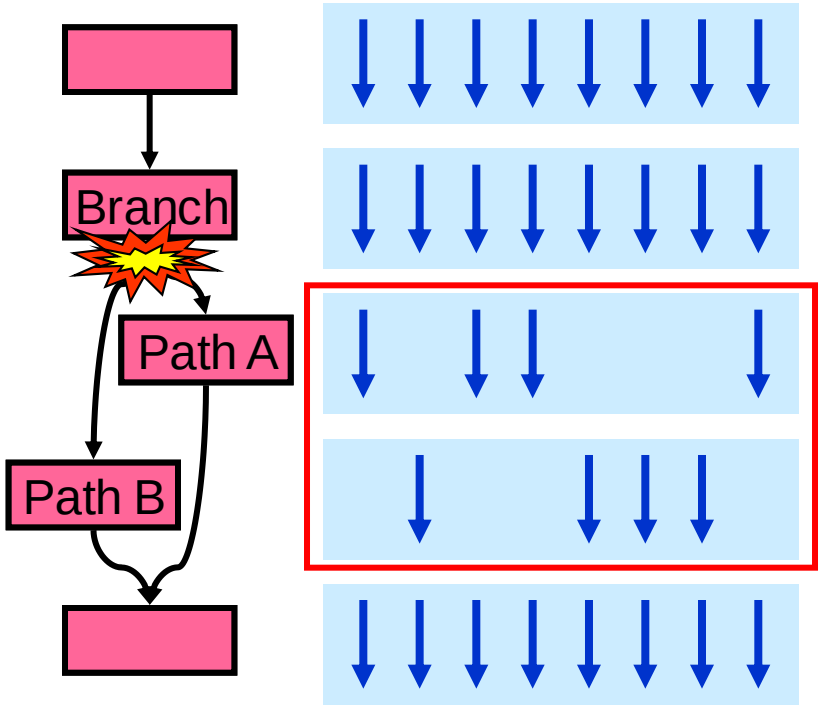
- **Static: Compiler analysis infers pattern**
  - Chooses best interface for **each load/store site in your code**
- ***Optimal when indexing by work-item ids***
  - *And simple linear combinations*

```
int id = get_global_id(0);

c[id] = a[id] * b[id];
```

*Caches only hurt you in this case: Burn power uselessly*

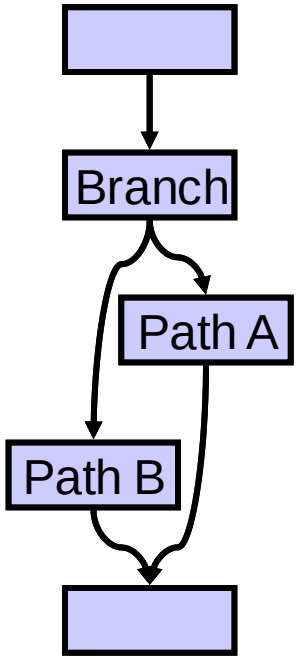
# Divergent control flow: Bad for GPU

- GPU uses SIMD pipeline to save area on control logic.
- Branches have a significant impact on GPU parallelism
- Parallel threads running through different branches cannot run concurrently



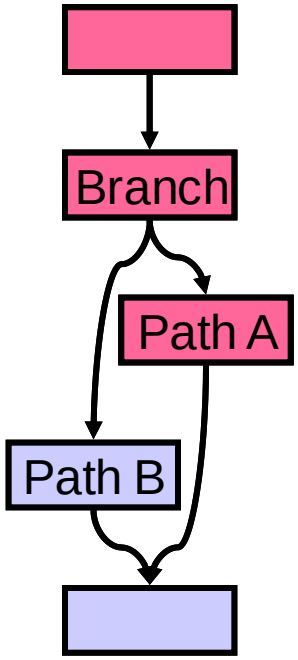
# Divergent control flow: Just fine for FPGA

- **FPGA datapath already has all operations in silicon**



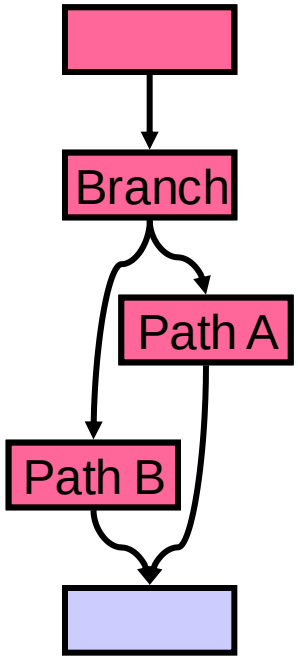
# Divergent control flow: Just fine for FPGA

- **FPGA datapath already has all operations in silicon**
- **Exploit pipelining**



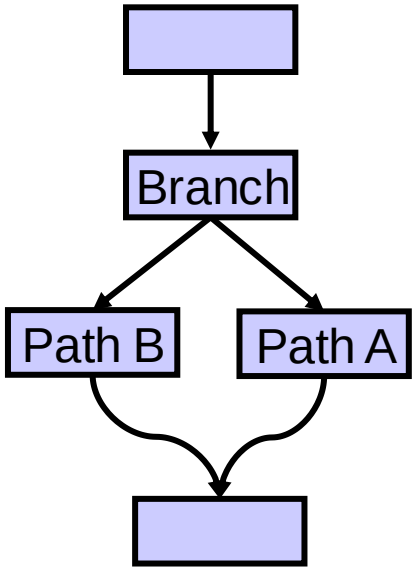
# Divergent control flow: Just fine for FPGA

- FPGA datapath already has all operations in silicon
- Exploit pipelining
- Speculatively execute



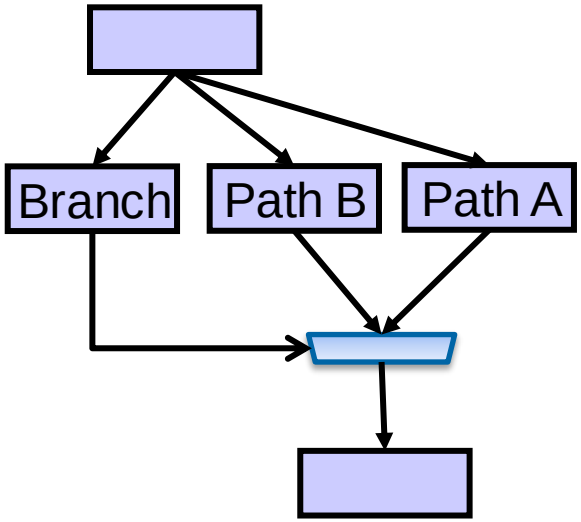
# Divergent control flow: Just fine for FPGA

- FPGA datapath already has all operations in silicon
- Exploit pipelining
- Speculatively execute
- Compress the schedule



# Divergent control flow: Just fine for FPGA

- FPGA datapath already has all operations in silicon
- Exploit pipelining
- Speculatively execute
- Compress the schedule
- Overlap branch computation too



# Divergent control flow: Just fine for FPGA

- FPGA datapath already has all operations in silicon
- Exploit pipelining
- Speculatively execute
- Compress the schedule
- Overlap branch computation too
- Absorb into one block

Branch. Path B. Path A

# Rules of thumb for great OpenCL code on FPGA

- Bit manipulation
- Integer arithmetic
- Large local storage requirements
- Complex control flow
- Unusual function mix
- Predictably unaliased memory access
- Predictable access patterns

***Architecture and compiler give these for free***

ALTERA
MEASURABLE ADVANTAGE™

# Low level compiler knobs

# Use `restrict` on kernel pointer arguments

- **Your promise that storage under this pointer doesn't alias with other `restrict` pointer arguments**
- **Standard C99**
- **Compiler can avoid conservatism in scheduling, conflicts**

```
kernel void test (   global const float * restrict a,
                     global const float * restrict b,
                     global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```

ALTERA
MEASURABLE ADVANTAGE™

# Use reqd_work_group_size

- **Enqueued work group size must match**
- **Standard OpenCL**

```
__attribute__((reqd_work_group_size(128,1,1)))
kernel void compute( … )
{
  …
}
```

***Any clues about the shape of the computation will help the compiler***

ALTERA
MEASURABLE ADVANTAGE™

# #pragma unroll

- **Compiler often automatically unrolls loops with fixed bounds**
- **Sometimes you should give a hint**
  - To control amount of hardware generated
  - You might know better than the compiler
- **Mandelbrot design example:**

```
// Perform up to the maximum number of iterations to solve
// the current work-item's position in the image
// The loop unrolling factor can be adjusted based on the amount of FPGA
// resources available.
 #pragma unroll 20
  while ( xSqr + ySqr < 4.0f && iterations < maxIterations)
  { …
```

- **OpenCL 2.0 has __attribute__ instead**

ALTERA
MEASURABLE ADVANTAGE™

# Custom sized local memory

- **Remember, local memory is abundant**

```
kernel void myLocalMemoryPointer(
   local float * A,
__attribute__((local_mem_size(1024))) local float * B,
__attribute__((local_mem_size(32768))) local float * C,
   global float * D )
{
 …
}
```

- **Defaults to 16KB per pointer-to-local argument**
- **In this example:**
  - A: 16KB, B: 1KB, C: 32KB
- **Controls area**
- **Enables wider range of algorithms**

ALTERA. MEASURABLE ADVANTAGE™

# Max work group size

- **Upper bound on allowable enqueued workgroup size**
  - Default is 256

```
__attribute__((max_work_group_size(128)))
kernel void test ( … )
{
   …
   barrier( CLK_LOCAL_MEM_FENCE );
   …
}
```

- **Controls area used by the kernel**
  - Only significant when using a barrier

# Force SIMD-like vectorization

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
kernel void test (  global const float * restrict a,
                    global const float * restrict b,
                    global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

- **Must use with reqd_work_group_size**
  - Must divide evenly
- **Reduces area: Less control logic (like for CPU/GPU)**
- **Increases throughput: More work done per clock cycle**
- **This example**
  - ¼ the cycles, each cycle computes float4

ALTERA. MEASURABLE ADVANTAGE™

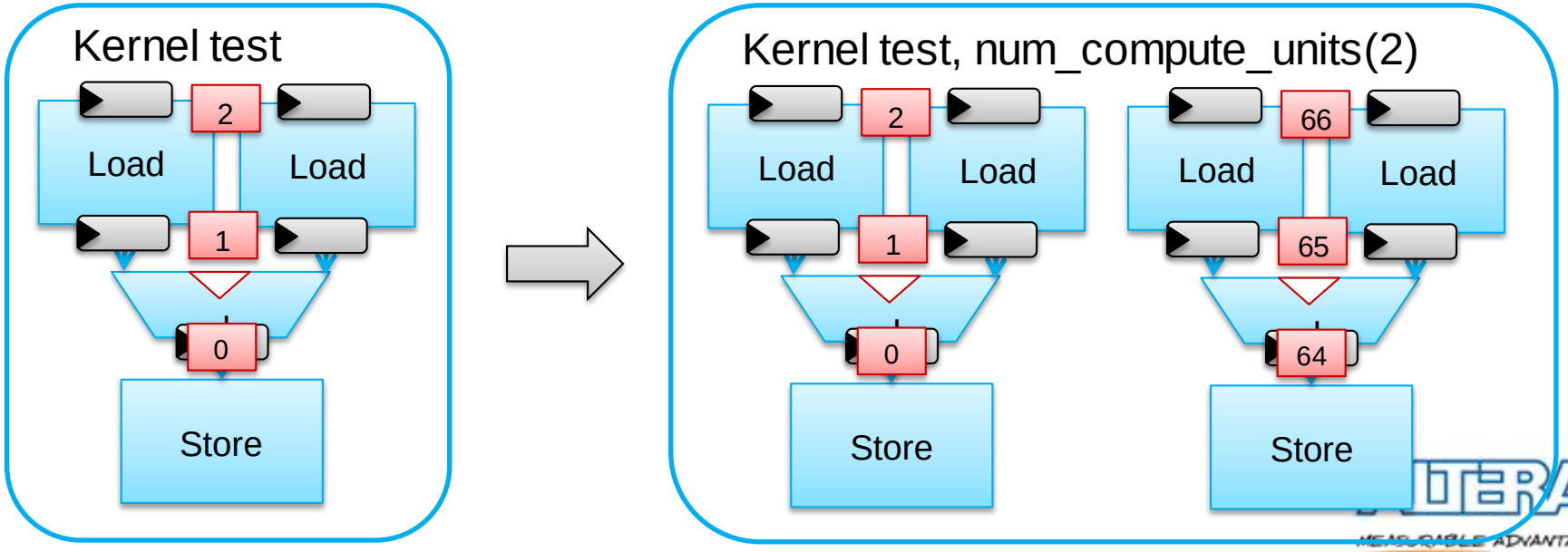
# Force replication of the kernel internal datapath

```
__attribute__(( num_compute_units(2) ))
__attribute__((reqd_work_group_size(64,1,1)))
kernel void test (  global const float * restrict a,
                    global const float * restrict b,
                    global float * restrict answer)
{
   size_t gid = get_global_id(0);
   answer[gid] = a[gid] + b[gid];
}
```

- **Increases number of concurrent workgroups**
- **Increases area**
- **Use num_simd_work_items instead if you can**

# Force replication of the kernel internal datapath

```
__attribute__(( num_compute_units(2) ))
__attribute__((reqd_work_group_size(64,1,1)))
kernel void test (   global const float * restrict a,
                     global const float * restrict b,
                     global float * restrict answer)
{
    size_t gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```



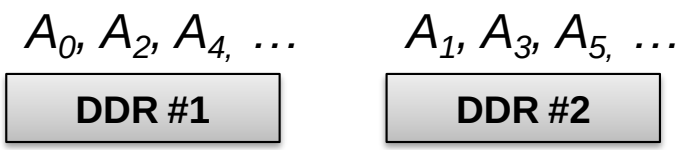
# Control __constant cache size

- Caches only built if __constant buffer kernel arguments
- Use the --const-cache-bytes <N> compiler argument to control its size

- Controls area
- Tune according to algorithm data access pattern

# Guided buffer placement for bandwidth control

- **Normally data is interleaved (striped) across both DDRx interfaces**
  - Assuming your board has multiple memory interfaces
- **Often get 2x bandwidth of a single DDRx interface**




- **But some algorithms work better with:**
  - No interleaving
  - Manually placed buffers

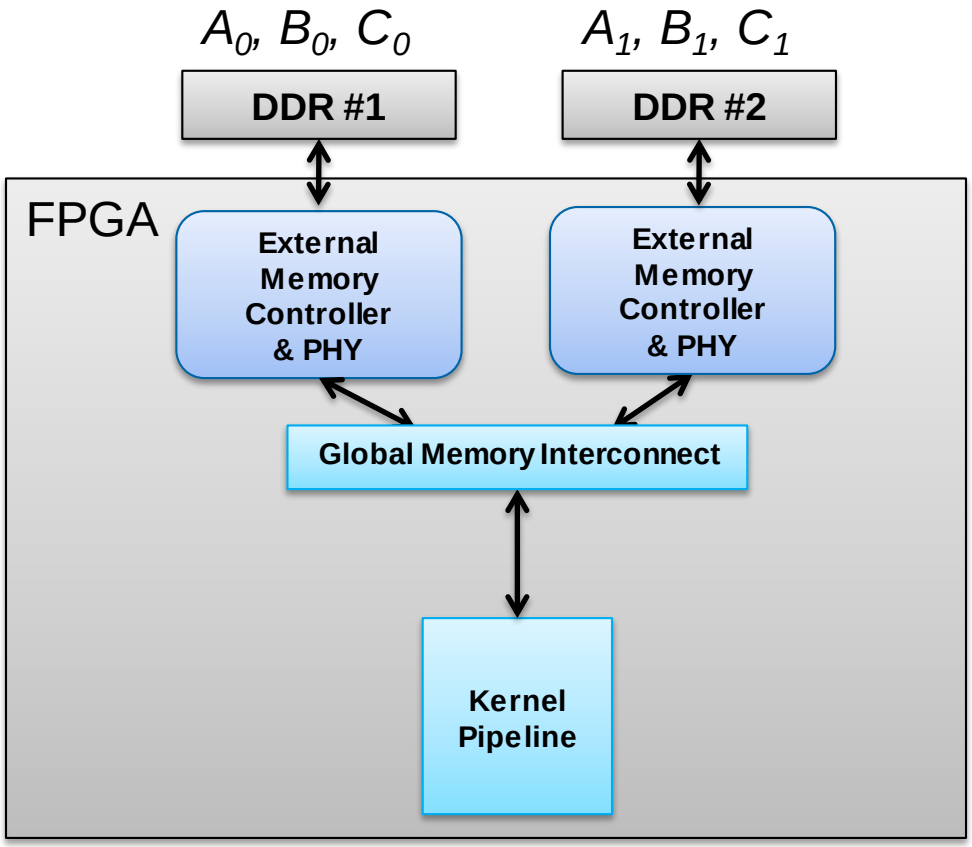
# Guided buffer placement for bandwidth control

- **Compiler: aoc -no-interleaving**
- **Host: Use special mem flags on buffer creation:**
  - CL_MEM_BANK_1_ALTERA
  - CL_MEM_BANK_2_ALTERA…
- **Design example: Matrix Multiply: *C ← A x B***
  - Uses matrix blocking (like usual)
  - And guided placement
    - A on DDR interface 1
    - B on DDR interface 2
    - C on DDR interface 1

ALTERA
MEASURABLE ADVANTAGE™

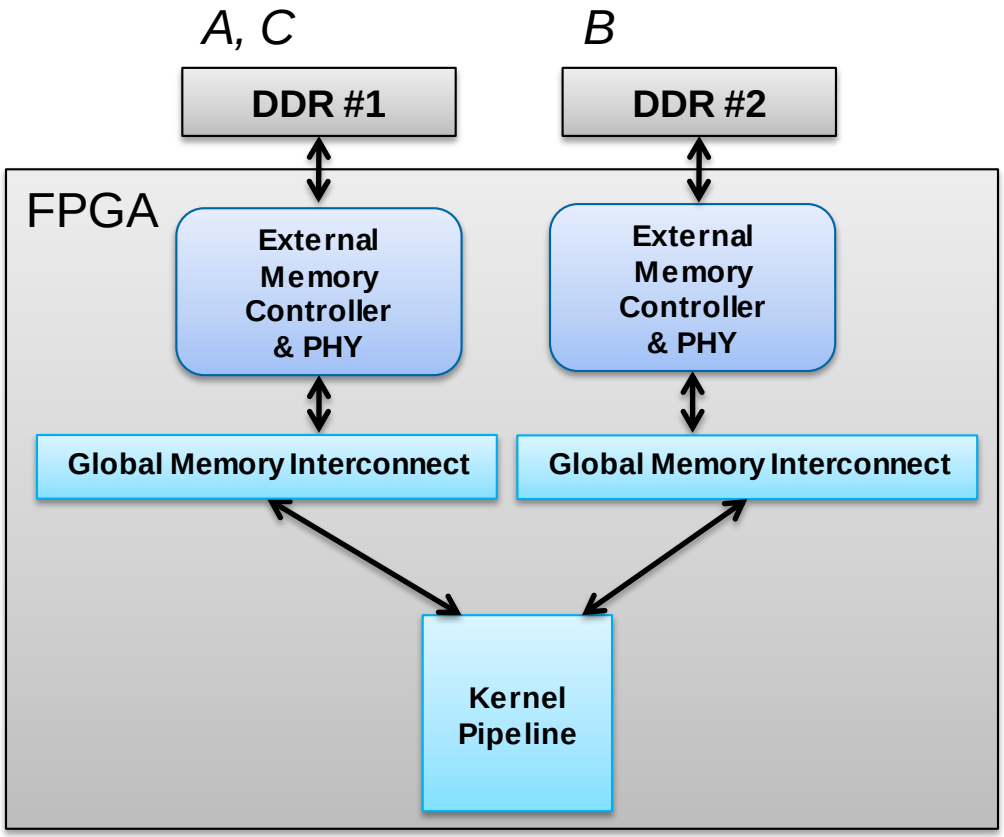
# E.g. Matrix Multiply: $C \leftarrow A \times B$

- Normal (with interleaving)
- Reading A and B, writing C: contend on both interfaces
- "Thrashes" DDR → Poor efficiency



# E.g. Matrix Multiply: *C ← A x B*

- **No interleaving; Use guided buffer placement**
- **Balanced non-thrashing access → High DDR efficiency**
  - (We finish reading block of A before we start writing to block of C)



# Design strategies: Streaming applications

# Some applications don't map well

- **Two assumptions made in previous OpenCL examples**
  1. Host initiates all data transfers
  2. Data level parallelism exists in the kernel program

- **Some applications don't match these assumptions**
  - Host initiated data transfers too expensive for some applications
  - Some applications do not map well to data-parallel paradigms

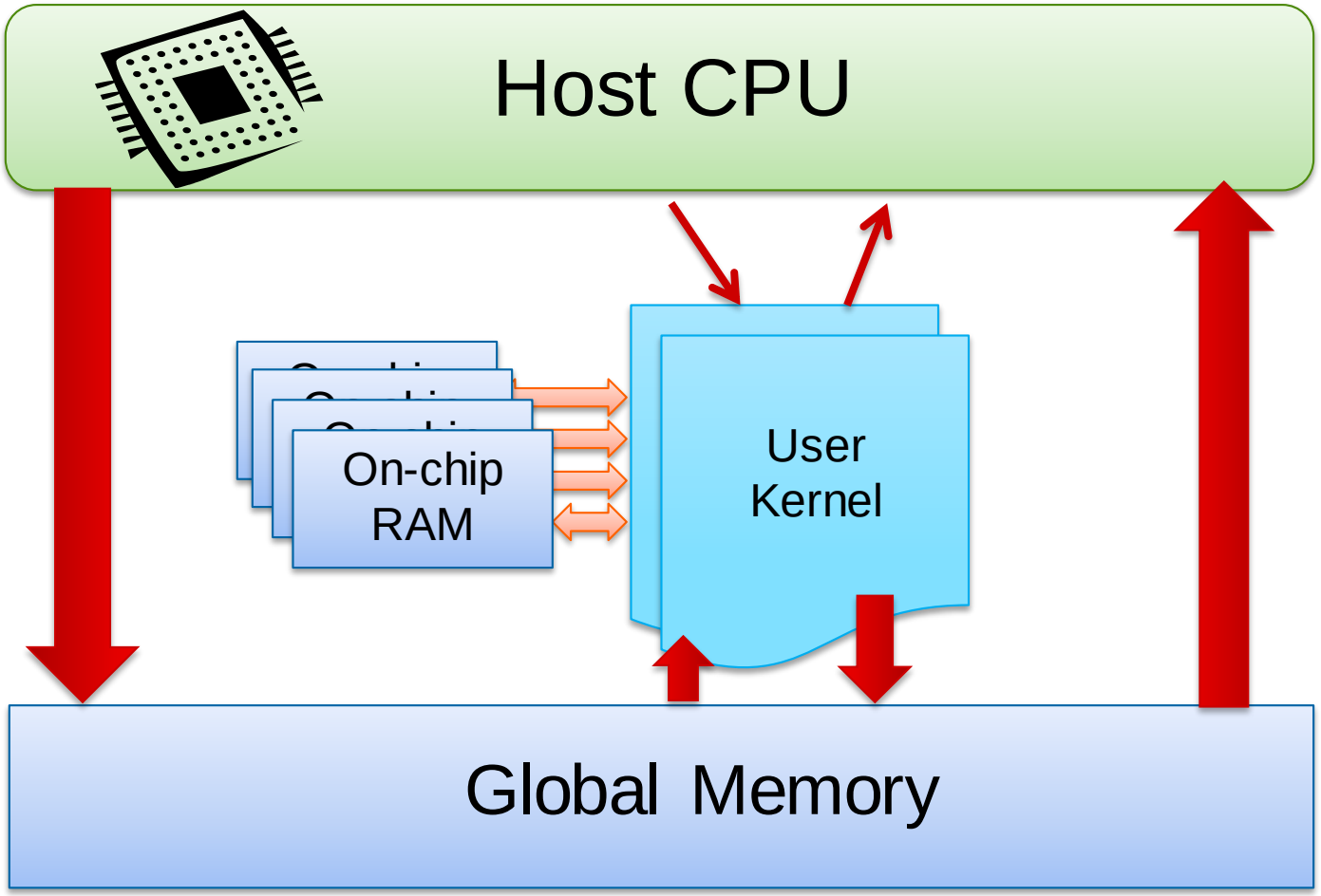
- **Can we avoid these problems, while still reaping the benefits of OpenCL on Altera's FPGAs?**

What about OpenCL 2.0 device-side enqueue?

- Complex code
- Data passed via global memory
- Data visible only after kernel termination:

  ***Chunky execution***

# Issue 1) Host Centric OpenCL Architecture




***Host Co-ordinates Kernel Invocations and Data Transfers***

# Issue 1) Host Centric OpenCL Architecture - *drawbacks*

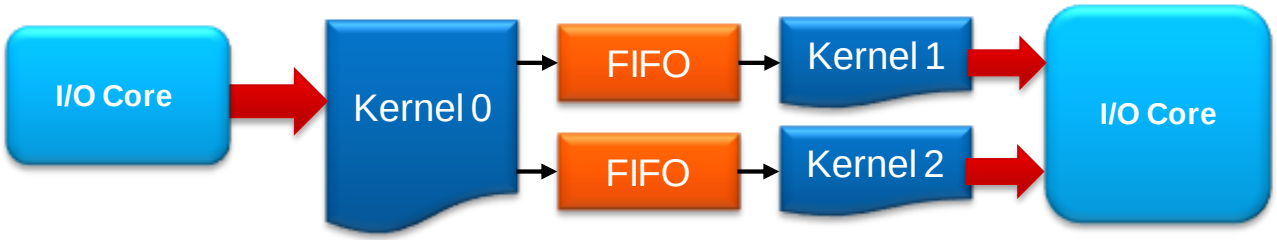
- **Intermediate data communicated between kernels must be transferred through global memory**
  - High performance requires high bandwidth and high power !
  - Limited buffer sizes when problem sizes scale to 100s of billions of points

- **Having multiple kernels operating in parallel and communicating requires the host to synchronize and coordinate activities**
  - Slow, power hungry

ALTERA
MEASURABLE ADVANTAGE™

# *Solution*: Altera Channels Vendor Extension

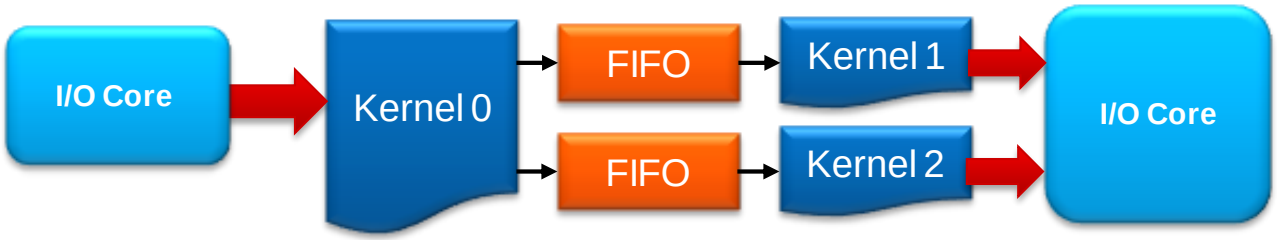
- **Low-Latency, High Bandwidth Channels**
  - Enables **IO → Kernel** and **Kernel → Kernel** Communication
  - **Everything inlined in the FPGA fabric: ~ one clock cycle transfer**




ALTERA
MEASURABLE ADVANTAGE™

# *Solution*: Altera Channels Vendor Extension

- **Low-Latency, High Bandwidth Channels**
  - Enables **IO → Kernel** and **Kernel → Kernel** Communication
  - **Everything inlined in the FPGA fabric: ~ one clock cycle transfer**

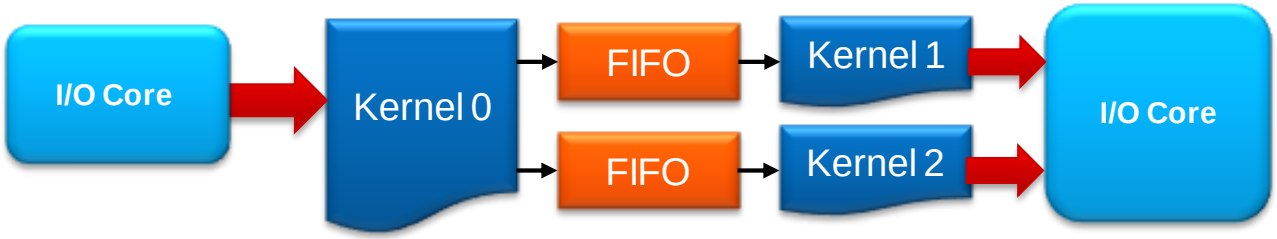



- **Communication: simple and intuitive API**
  - Channels are program scope variables that define the communication links
    - Ex: channel float4 FLOATING_POINT_CHANNEL;
  - read_channel_altera
    - Read data from channel endpoint
    - Ex: float4 xvec = read_channel_altera( FLOATING_POINT_CHANNEL );
  - write_channel_altera
    - Write data to channel endpoint
    - Ex: write_channel_altera( FLOATING_POINT_CHANNEL, zvec );

ALTERA
MEASURABLE ADVANTAGE™

# *Solution*: Altera Channels Vendor Extension

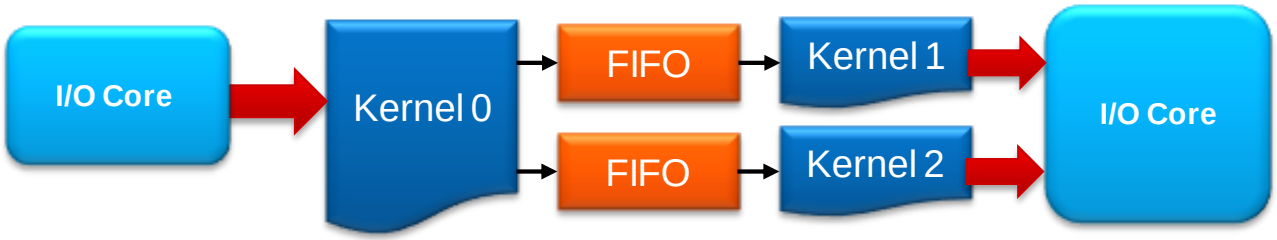
- **Low-Latency, High Bandwidth Channels**
  - Enables **IO → Kernel** and **Kernel → Kernel** Communication
  - **Everything inlined in the FPGA fabric: ~ one clock cycle transfer**




- **(I/O communication requires special board support)**

# *Solution*: Altera Channels Vendor Extension

- **Low-Latency, High Bandwidth Channels**
  - Enables IO → Kernel and Kernel → Kernel Communication
  - **Everything inlined in the FPGA fabric: ~ one clock cycle transfer**

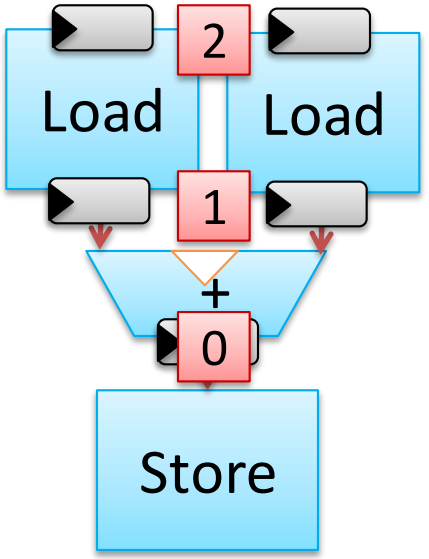



- **Launch kernels in parallel (same cl_program)**
  - Use one command queue per kernel
  - clFlush all of them
  - Then clFinish all of them
- **They're already all in the FPGA fabric, just waiting to go**

# Issue 2) Data-Parallel Execution

- **On the FPGA, we use pipeline parallelism to accelerate**

```
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];
}
```

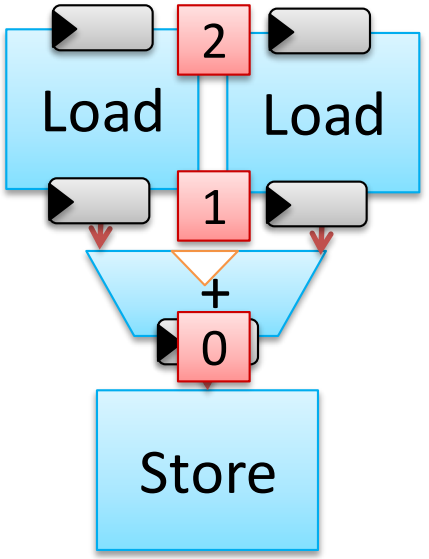



- **Threads can execute in an embarrassingly parallel manner**

# Issue 2) Data-Parallel Execution - *drawbacks*

- **Hard to express programs having partial data 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 need complex (expensive, error prone) constructs to express *correctly***

# Deep thought #2

*A designer knows he has achieved perfection not when there is nothing left to add, but when there is **nothing left to take away**.*

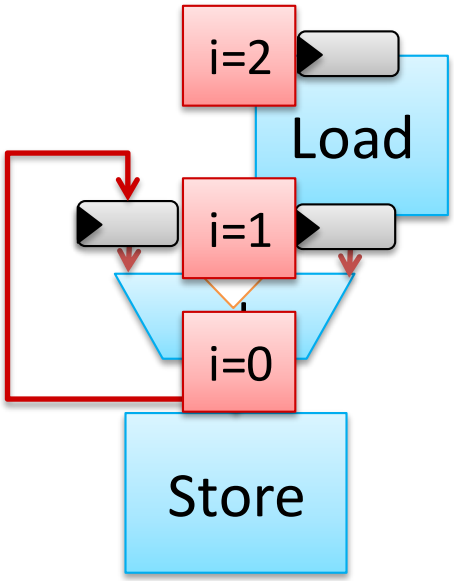
***— ANTOINE DE SAINT EXUPÉRY***

# *Solution*: Tasks and Loop-pipelining

- **Allow users to express programs in a single-threaded manner (OpenCL Task)**

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

- **Pipeline parallelism still used to efficiently execute loops in Altera's OpenCL**
  - ***Loop Pipelining***



# Deep thought #3

**OpenCL does not require NDRange or SIMD execution**

# Loop Carried Dependencies

- **Loop-carried dependency: one iteration of the loop depends upon the results of another iteration of the loop**

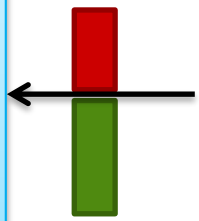
```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    write_channel_altera(OUTPUT, y);
  }
}
```

- **The value of `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 can pipeline each iteration of a loop containing loop carried dependencies**
  - Analyze any dependencies between iterations
  - Schedule these operations
  - Launch the next iteration as soon as possible

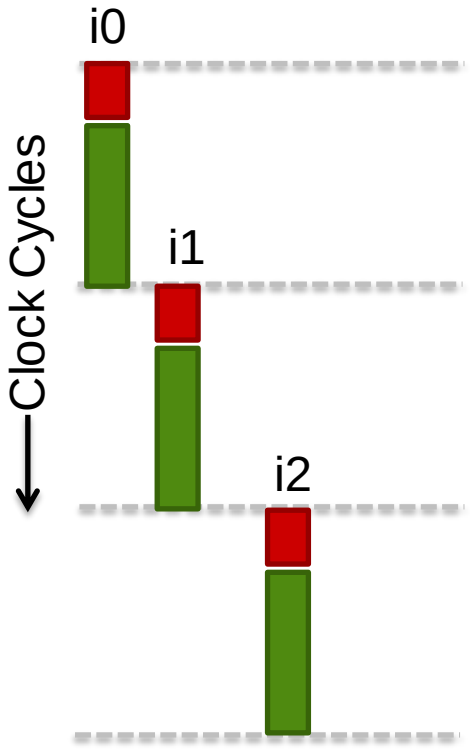
```
kernel void state_machine(ulong n)
{
  t_state_vector state = initial_state();
  for (ulong i=0; i<n; i++) {
    state = next_state( state );
    unit y = process( state );
    write_channel_altera(OUTPUT, y);
  }
}
```



At this point, we can launch the next iteration

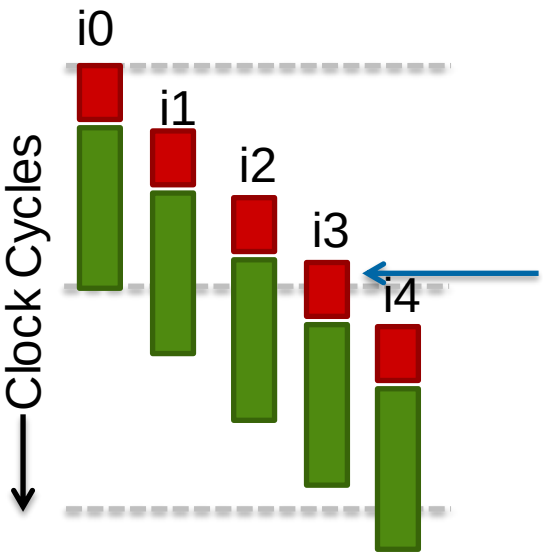
# Loop Pipelining Example

- **No Loop Pipelining**




*No Overlap of Iterations!*

- **With Loop Pipelining**

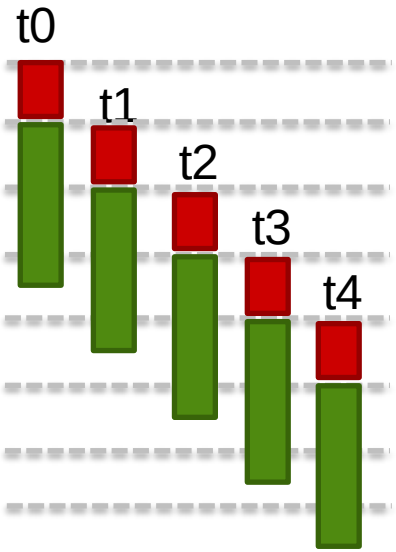



Looks almost like ND-range thread execution!

*Finishes Faster because Iterations Are Overlapped*

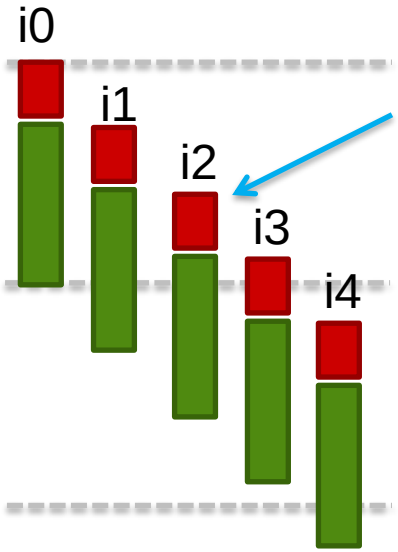
# Pipelined Work Items vs. Loop Pipelining

- **So what's the difference?**




Pipelined work items launch 1 item per clock cycle in pipelined fashion

Pipelined Threads




Loop dependencies may not be resolved in 1 clock cycle

Loop Pipelining

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

# Altera's compiler does a lot for you

- **Generating a loop-specific machine**
- **Tells you how many clock cycles between iterations**
  - "Initiation Interval"
- **Static optimization report tells you which data dependencies are slowing down the loop**
  - Use –g to get better line number and variable info

ALTERA
MEASURABLE ADVANTAGE™

# User response to improve loop pipelining

- **Remove dependencies**
  - E.g. use simpler access pattern to remove memory dependencies
- **Relax dependence**
  - Increase ***dependence distance***: Number of iterations between generation and use of a value
    - Often by using a shift register
- **Simplify dependence complexity**
  - Avoid expensive operations when computing loop-carried values

***Often requires application knowledge to restructure code***

ALTERA
MEASURABLE ADVANTAGE™

# Example: Load to Store dependency

```
1  kernel void prefixsum( global int* restrict A, unsigned N ) {
2    for ( unsigned i = 1 ; i < N ; i++ ) {
3      int a = A[i-1];
4      A[i] += a;
5    }
6  }
```

```
================================================================================
|                        *** Optimization Report ***                           |
================================================================================
| Kernel: prefixsum                                                | Ln.Col    |
================================================================================
| Loop for.body                                                    | 2.25      |
|     Pipelined execution inferred.                                |           |
|     Successive iterations launched every 321 cycles due to:      |           |
|                                                                  |           |
|         Memory dependency on Load Operation from:                | 3.21      |
|           Store Operation                                        | 4.7       |
|         Largest Critical Path Contribut                          |           |
|             49%: Load Operation                                 | 3.21      |
|             49%: Store Operation                                | 4.7       |
================================================================================
```

Relative cost of global memory to local computation

True fix requires restructuring the code

# Example: Accumulating a value

```
1  kernel void test(   global float* restrict input,
2                      global float* restrict output, unsigned N )
3  {
4     float mul = 1.0f;
5     for ( unsigned i = 0; i < N; i++ ) {
6        mul *= input[ i ];
7     }
8     *output = mul;
9  }
```

```
================================================================================
|                          *** Optimization Report ***                         |
================================================================================
| Kernel: test                                                        | Ln.Col   |
================================================================================
| Loop for.body                                                       | 5.24     |
|     Pipelined execution inferred.                                   |          |
|     Successive iterations launched every 3 cycles due to:           |          |
|                                                                     |          |
|         Data dependency on variable mul                             | 4.10     |
|         Largest Critical Path Contributor:                          |          |
|            100%: Fmul Operation                                     | 6.7      |
================================================================================
```

# Example: Accumulating a value, quickly

```
#define DEP 6
kernel void test( global float* restrict input,
                  global float* restrict output, unsigned N )
{
  float mul = 1.0f;
  float mul_copies[ DEP ]; // Shift register in private memory

  for ( unsigned i = 0; i < DEP; i++ ) // Initialize copies
    mul_copies[ i ] = 1.0f;

  for ( unsigned i = 0; i < N; i++ ) {
    // Use one copy.  Needs data from DEP iterations ago
    float cur = mul_copies[ DEP-1 ] * input[ i ];

    // Shift!
    for ( unsigned j = DEP-1; j > 0 ; j-- ) {
      mul_copies[ j ] = mul_copies[ j - 1 ];
      mul_copies[ 0 ] = cur;
    }
  }

  // Accumulate result with leftovers
  for ( unsigned i = 0; i < DEP; i++ )
    mul *= mul_copies[ i ];

  *output = mul;
}
```

Relax dependence across more iterations.

Compiler infers a **shift register,** becomes a FIFO in hardware

**Extremely efficient**

# Example: Accumulating a value, quickly

```
#define DEP 6
kernel void test( global float* restrict input,
                  global float* restrict output, unsigned N )
{
  float mul = 1.0f;
  float mul_copies[ DEP ]; // Shift register in private memory

  for ( unsigned i = 0; i < DEP; i++ ) // Initialize copies
    mul_copies[ i ] = 1.0f;

  for ( unsigned i = 0; i < N; i++ ) {
    // Use one copy.  Needs data from DEP iterations ago
    float cur = mul_copies[ DEP-1 ] * input[ i ];

    // Shift!
    for ( unsigned i = DEP-1; i > 0; i-- ) {

    *output = mul;
  }
```

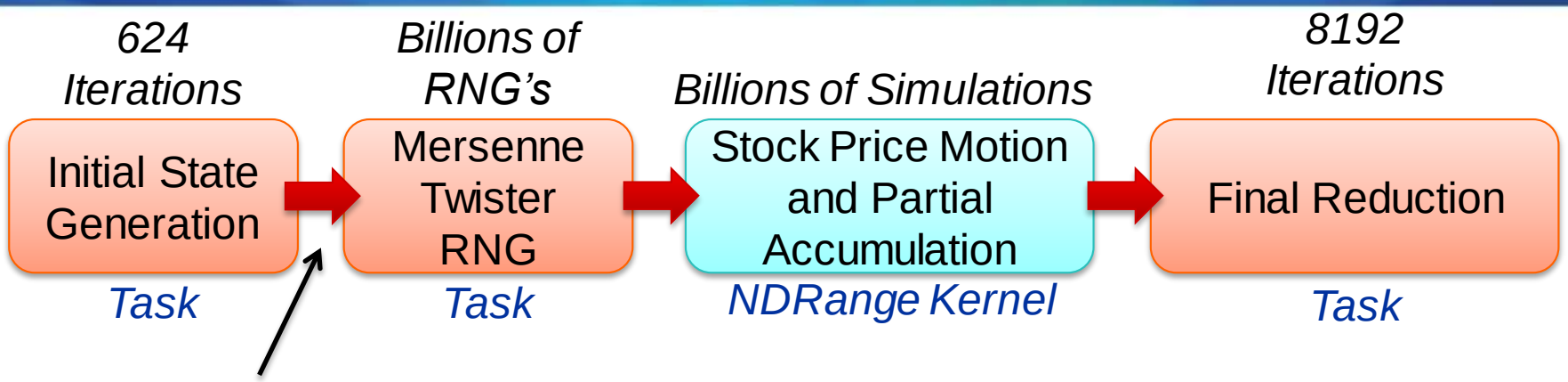
Relax dependence across more iterations.

Compiler infers a **shift register,** becomes a FIFO in hardware

**Extremely efficient**

```
================================================================================
|                        *** Optimization Report ***                           |
================================================================================
| Kernel: test                                                    | Ln.Col   |
================================================================================
| Loop for.body4                                                  | 11.24    |
|     Pipelined execution inferred.                               |          |
================================================================================
```

# Monte Carlo Asian Option Simulation




- **Channels used for direct kernel-to-kernel communication without requiring intermediate global memory buffers**
- **Uses both Tasks (single work-item) and ND-range kernels.**
- **Results:**
  - Altera Stratix® V D8: **12.0 Billion Simulations / Second @ 45 Watts**

ALTERA
MEASURABLE ADVANTAGE™

# Stock Price Motion and Partial Accumulation Kernel

```
kernel void black_scholes( int m, int n, float drift, float vol, float S_0, float K) {
   // running statistics -- use double precision for the accumulator
   double sum = 0.0;
   for(int path=0;path<m;path++) {
      float S = S_0;
      float arithmetic_average = 0.0f; // We're not including the initial price in the average
      for (int t_i=0; t_i<n/16; t_i++) {
         barrier(CLK_GLOBAL_MEM_FENCE);
         float Z[16];
         float16 U = read_channel_altera(RANDOM_STREAM);
         ...
         #pragma unroll 8
         for (int i=0; i<8; i++) {
            // Convert uniform distribution to normal
            float2 z = box_muller(U[2*i], U[2*i+1]);
            Z[2*i] = z.x; Z[2*i+1] = z.y;
         }
         #pragma unroll 16
         for (int i=0; i<16; i++) {
            // Simulate the path movement using geometric brownian motion
            S *= drift * exp(vol * Z[i]);
            arithmetic_average += S;
         }
      }
      arithmetic_average /= (float)(n);
      // Check if the average value exceeds the strike price
      float call_value = arithmetic_average - K;
      if (call_value > 0.0f) {
         sum += call_value;
      }
   }
   // send a final result to the accumulate kernel after each thread has
   write_channel_altera(ACCUMULATE_STREAM, sum);
}
```

*Simplified: Showing 16 parallel sims/cycle instead of 64*

Each work item reads a sequence of random numbers from a channel

Key computation loop is a fully unrolled floating point datapath

Write result to reduction kernel

# More elaborate task/channel examples

- **On Altera's website:**
  - Channelizer
  - Time-Domain FIR filter
  - Sobel filter
  - OPRA FAST parser
  - Finite Difference Computation (3D)

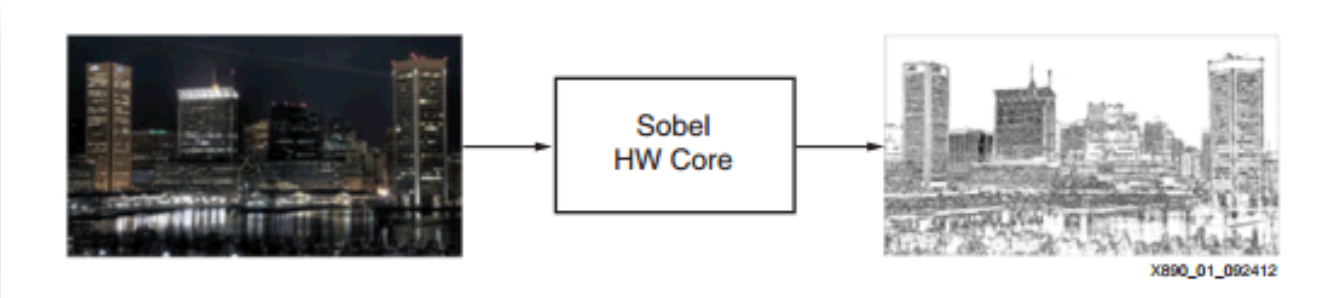
- ***Tomorrow at IWOCL:***
  - "OpenCL Implementation of Gzip on Field-Programmable Gate-Arrays"
    **Mohamed S. Abdelfattah**

ALTERA
MEASURABLE ADVANTAGE™

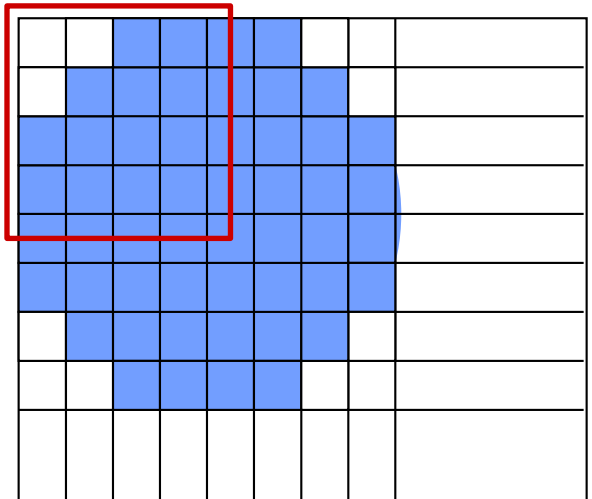
# Design strategy:
# Sliding window vs. stenciling

# Sobel Filter

- **Fundamental image processing algorithm**
  - Used commonly in industrial and automotive applications



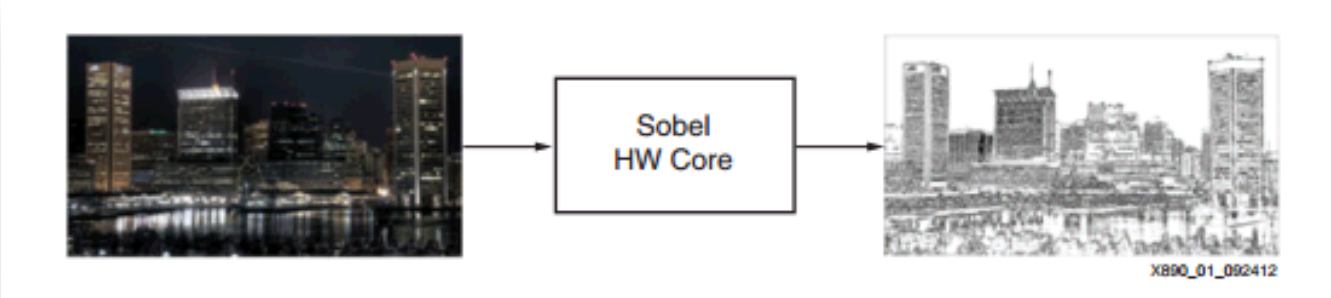

- **GPU codes would use architecture specific memory access blocking, banking**
  - Goal: Automatically coalesce memory accesses across work items



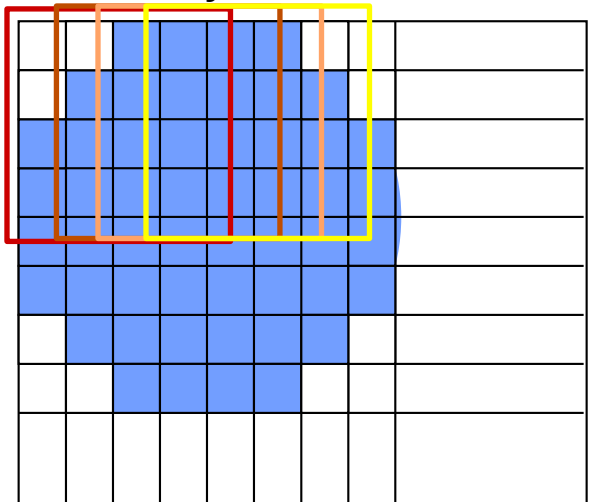
***Creates many loads from memory. Can be relatively expensive on FPGA: area and time***

# Sobel Filter

- **Fundamental image processing algorithm**
  - Used commonly in industrial and automotive applications



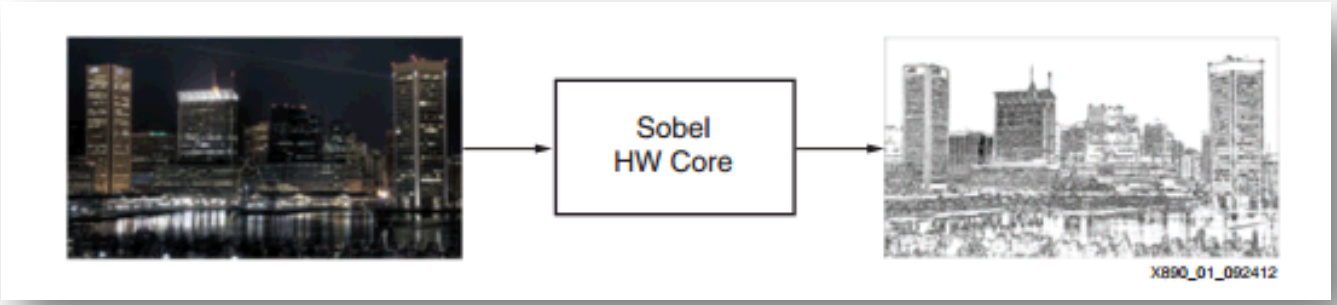

- **GPU codes would use architecture specific memory access blocking, banking**
  - Goal: Automatically coalesce memory accesses across work items



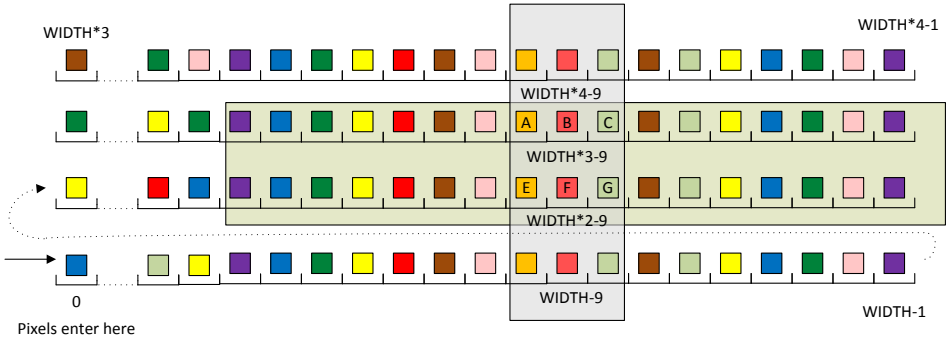
***Creates many loads from memory. Can be relatively expensive on FPGA: area and time***

# Sobel Filter

- **Fundamental image processing algorithm**
  - Used commonly in industrial and automotive applications



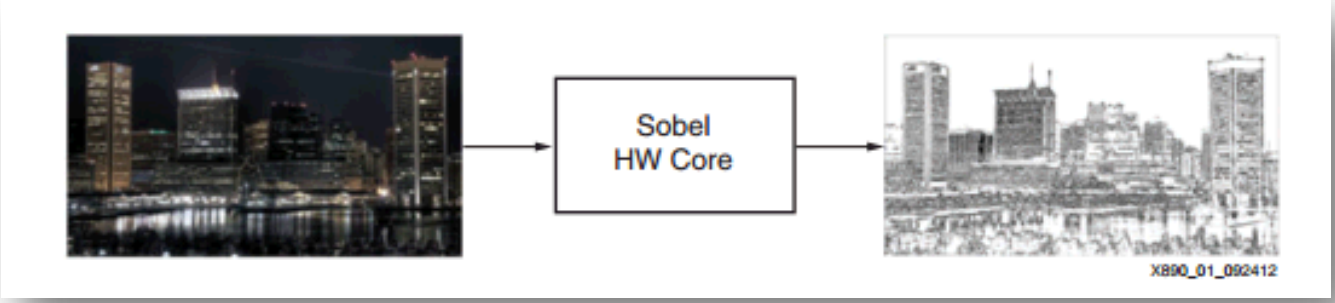

- **Altera FPGA: Use *Sliding Window* design pattern**
  - Shift register structure, but now in two dimensions
  - Need enough storage for a few image lines, depending on stencil size



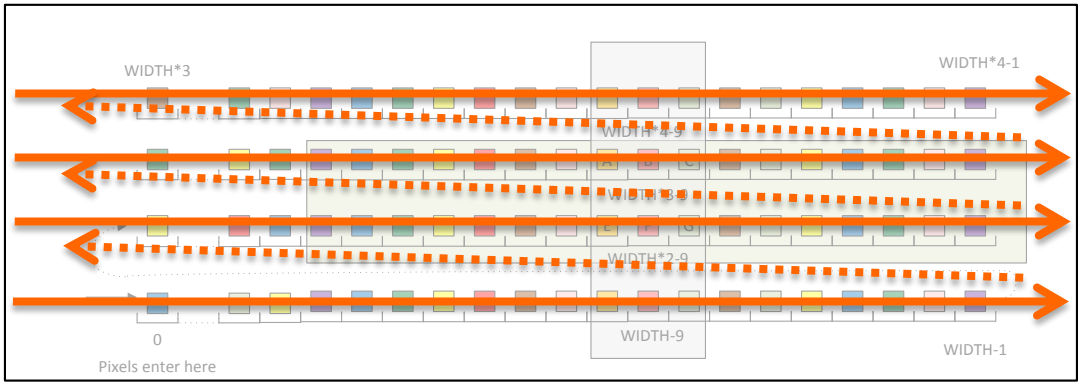

***Use cheap shift registers in private memory. Everything is local to kernel datapath.***

# Sobel Filter

- **Fundamental image processing algorithm**
  - Used commonly in industrial and automotive applications




- **Altera FPGA: Use *Sliding Window* design pattern**
  - Shift register structure, but now in two dimensions
  - Need enough storage for a few image lines, depending on stencil size




***Use cheap shift registers in private memory. Everything is local to kernel datapath.***

ALTERA
MEASURABLE ADVANTAGE™

# Sobel Filter on HD Video

```
for (ipixel=0; ipixel<HD_SIZE;ipixel++) {
    #pragma unroll
    for (iwidth=0;iwidth<1920*NROWS-1;iwidth++) {
          line_buf[iwidth] = line_buf[iwidth+1];
    }
    line_buf[iwidth] = input[ipixel];
    sobel = transform(line_buf[0], line_buf[1],
                      line_buf[1920], line_buf[1921]);
}
```

| Device | Resolution | Frames per second |
|---|---|---|
| Stratix V | 1920 x 1080p | **135** |

ALTERA
MEASURABLE ADVANTAGE™

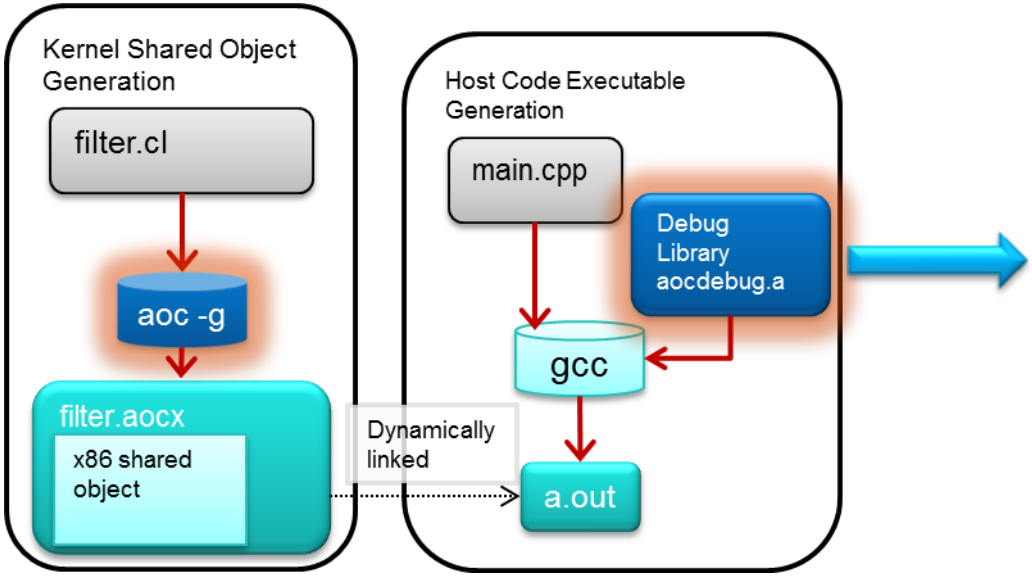
# Deep thought #4

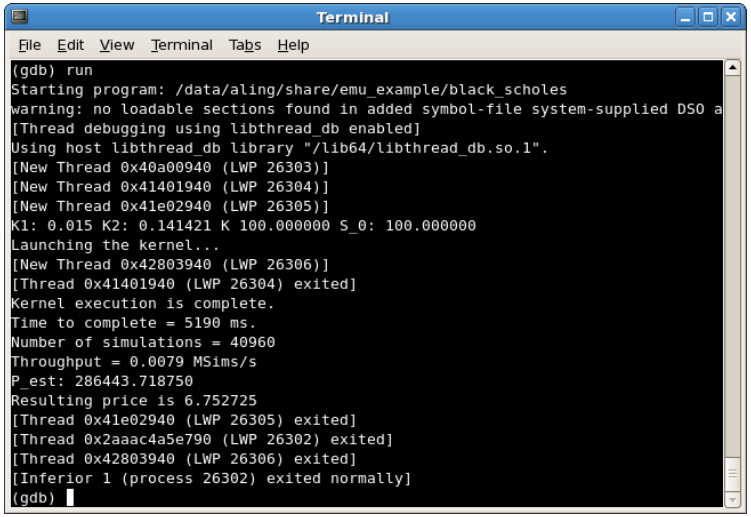
**Sometimes it is more efficient to move the data than to use pointer arithmetic to access a different set of data**

# Dynamic tools to help you

# Emulation on x86: New in v14.0

- **Functionally debug your Altera OpenCL code on your Linux, Windows**
  - Important since we've extended OpenCL in several ways
  - Especially Channels
- **Used extensively inside Altera**
- **aoc –c –march=emulator –g**
  - Then use your favourite debugger, e.g. GDB

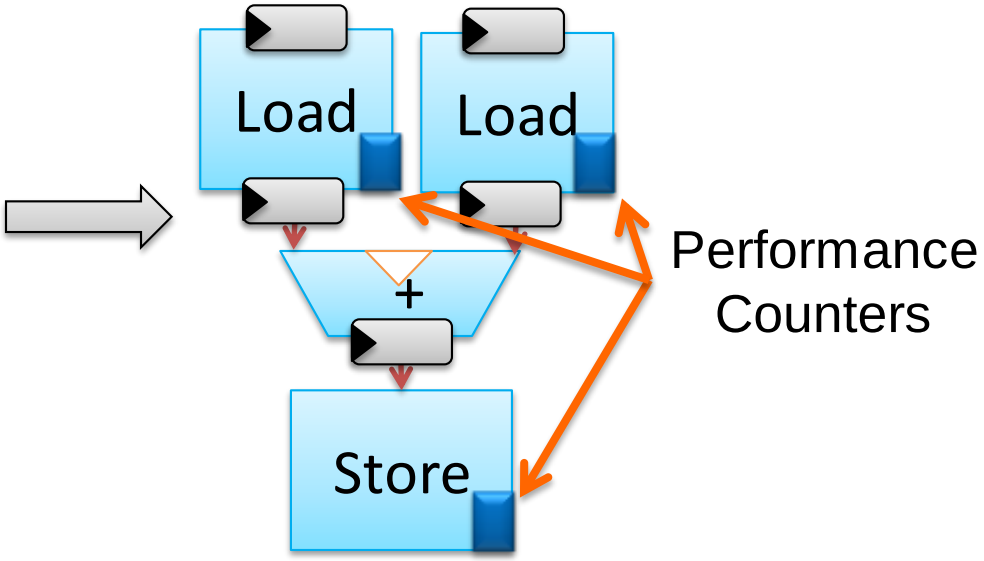


# Profiler support in v14.0

- **Runtime profiled information**
  - *Because the compiler can't know everything, and neither do you*
- **Instrument hardware pipeline with performance counters**
  - Read back at kernel termination

```
kernel void add(
    global int * a,
    global int * b,
    global int * c ) {
      int gid = get_global_id(0);
      c[gid] = a[gid]+b[gid];
}
```



# Profiler GUI: Actionable feedback mapped to your code

- **Bottlenecks, bandwidth, saturation, pipeline occupancy**

| | |
|---|---|
| Board | pcie385n_d5 |
| Global Memory BW (MEMORY) | 25600 MB/s |

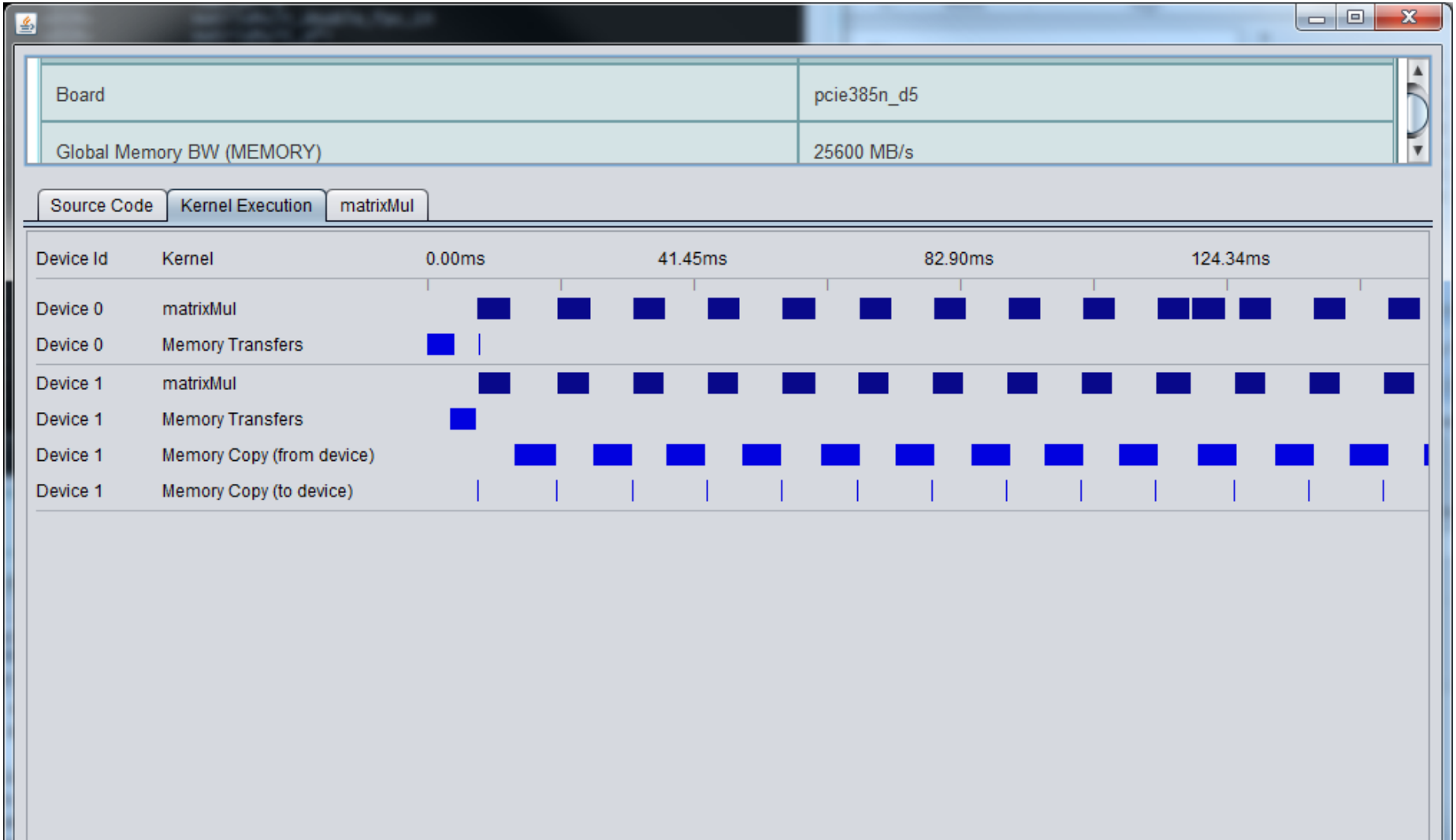
Source Code | Kernel Execution | matrixMul

| Line # | Source Code | Attributes | Stall% | Occupancy% | Bandwidth |
|---|---|---|---|---|---|
| 64 | a <= aEnd; | | | | |
| 65 | a += aStep, b += bStep) { | | | | |
| 66 | | | | | |
| 67 | // Load the matrices from device memory | | | | |
| 68 | // to shared memory; each thread loads | | | | |
| 69 | // one element of each matrix | | | | |
| 70 | AS(ty, tx) = A[a + uiWA * ty + tx]; | 0: __global{MEMORY},read | 0: 2.93% | 0: 95.9% | 0: 1398.5MB/s, 100.00%Efficiency |
| 71 | BS(ty, tx) = B[b + uiWB * ty + tx]; | 0: __global{MEMORY},read | 0: 0.08% | 0: 95.9% | 0: 10.1MB/s, 100.00%Efficiency |
| 72 | | | | | |
| 73 | // Synchronize to make sure the matrices are loaded | | | | |
| 74 | barrier(CLK_LOCAL_MEM_FENCE); | | | | |
| 75 | | | | | |
| 76 | #pragma unroll | | | | |
| 77 | for (int k = 0; k < BLOCK_SIZE; ++k) { | | | | |
| 78 | Csub += AS(ty, k) * BS(k, tx); | 0: __local,read | 0: 0.0% | 0: 95.9% | 0: -- |
| 79 | } | | | | |
| 80 | | | | | |
| 81 | // Synchronize to make sure that the preceding | | | | |
| 82 | // computation is done before loading two new | | | | |
| 83 | // sub-matrices of A and B in the next iteration | | | | |
| 84 | barrier(CLK_LOCAL_MEM_FENCE); | | | | |
| 85 | } | | | | |

0: __global{MEMORY},read
Cache Hits: 99.9%
Non-aligned Accesses: 0.0%
Memory site coalesced with other memory sites.

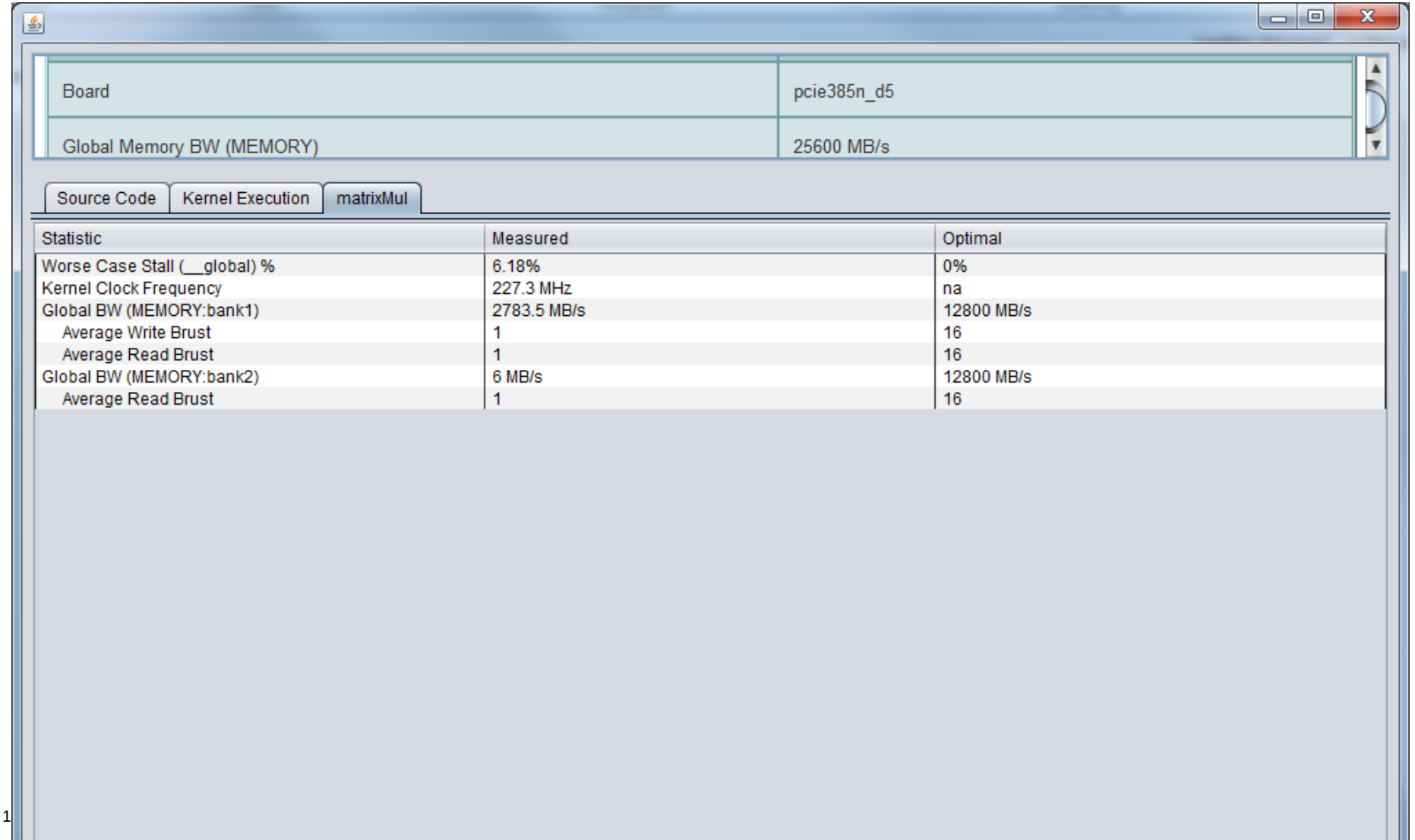
# Profiler GUI: Application trace

- **Kernel execution in context, Buffer transfer traffic**



# Profiler GUI: Overall bandwidth analysis

- **Relative to board device capabilities**



| | |
|---|---|
| Board | pcie385n_d5 |
| Global Memory BW (MEMORY) | 25600 MB/s |

Source Code | Kernel Execution | matrixMul

| Statistic | Measured | Optimal |
|---|---|---|
| Worse Case Stall (__global) % | 6.18% | 0% |
| Kernel Clock Frequency | 227.3 MHz | na |
| Global BW (MEMORY:bank1) | 2783.5 MB/s | 12800 MB/s |
| Average Write Brust | 1 | 16 |
| Average Read Brust | 1 | 16 |
| Global BW (MEMORY:bank2) | 6 MB/s | 12800 MB/s |
| Average Read Brust | 1 | 16 |

# Just one more thing…

# Breakthrough in FPGA floating point


World's first FPGA with hardened Single Precision Floating Point Multiplier-Adder

***(This is going to be amazing)***

ALTERA
MEASURABLE ADVANTAGE™

# Wrapup

**FPGAs are radically different from CPUs and GPUs**

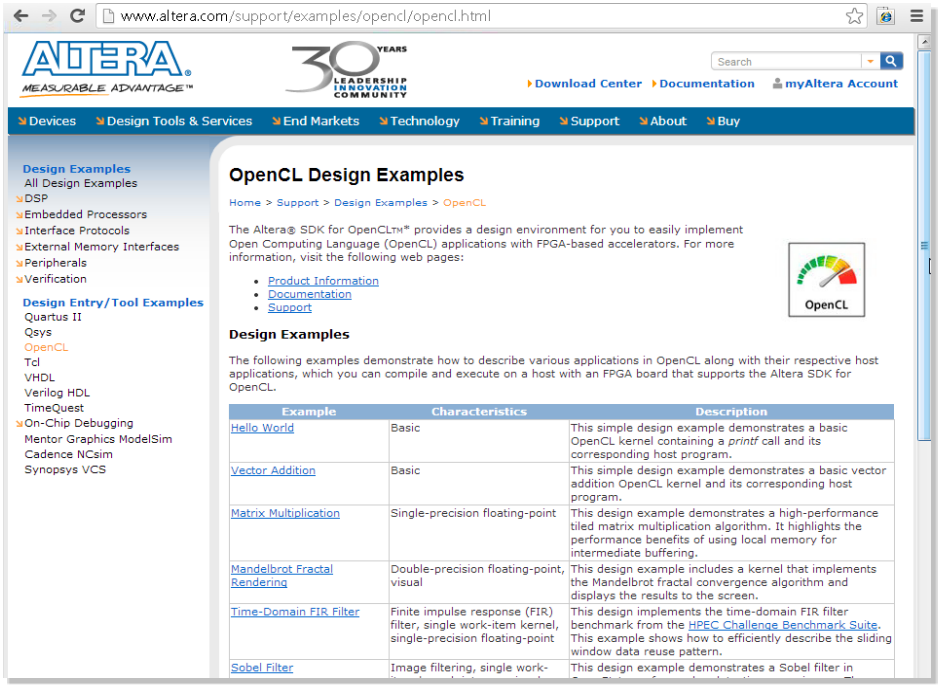
**OpenCL can be awesome on Altera FPGAs**

**But need a little bit of knowledge for best results**

ALTERA
MEASURABLE ADVANTAGE™

# References

- **Everything Altera and OpenCL**
  **http://www.altera.com/opencl**
- **Design examples, covering basics and showcasing optimized applications**
  - Please visit!
  - http://www.altera.com/support/examples/opencl/opencl.html



# Thank You



ALTERA®

MEASURABLE ADVANTAGE™

# Q & A