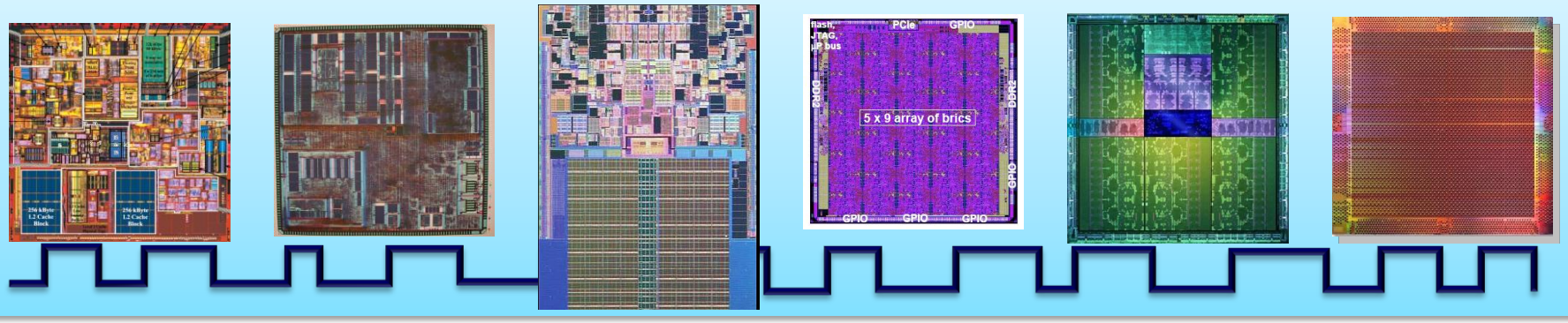
Tutorial: Harnessing the Power of FPGAs using Altera's OpenCL Compiler

Desh Singh, Tom Czajkowski, Andrew Ling

OPENCL INTRODUCTION

Programmable Solutions

- Technology scaling favors **programmability** and **parallelism**



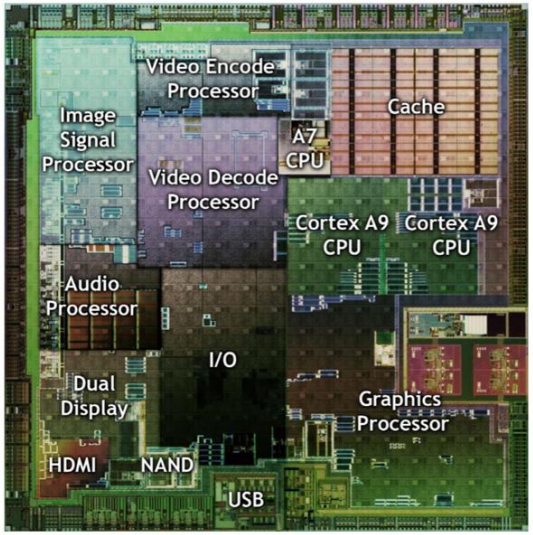
CPUs → DSPs → Multi-Cores → Array → GPGPUs → FPGAs

Single Cores	Multi-Cores Coarse-Grained CPUs and DSPs	Coarse-Grained Massively Parallel Processor Arrays	Fine-Grained Massively Parallel Arrays

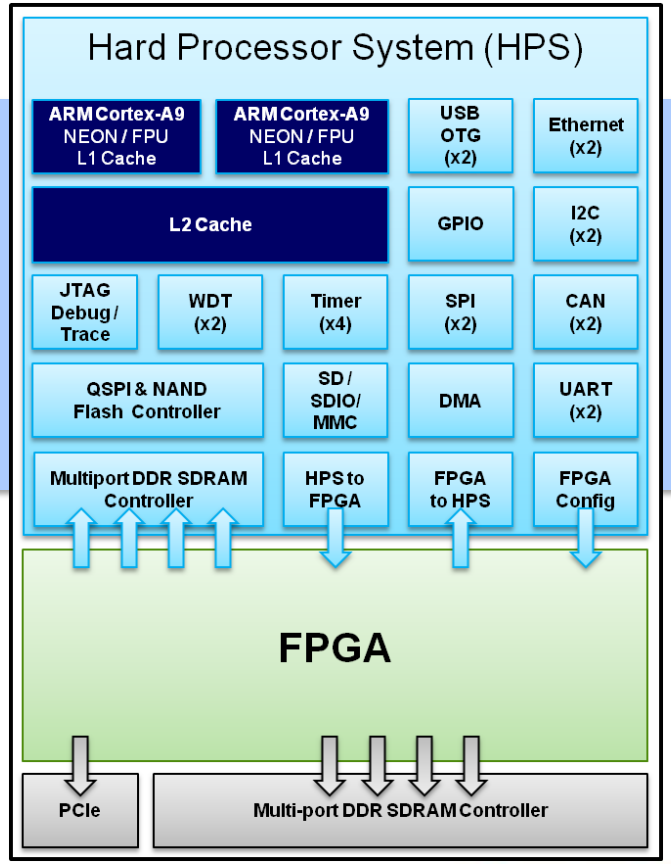
ALTERA
MEASURABLE ADVANTAGE™

CPU + Hardware Accelerator

- ARM®-based system-on-a-chip (SoC) devices by Intel, Apple, NVIDIA, Qualcomm, TI, and others



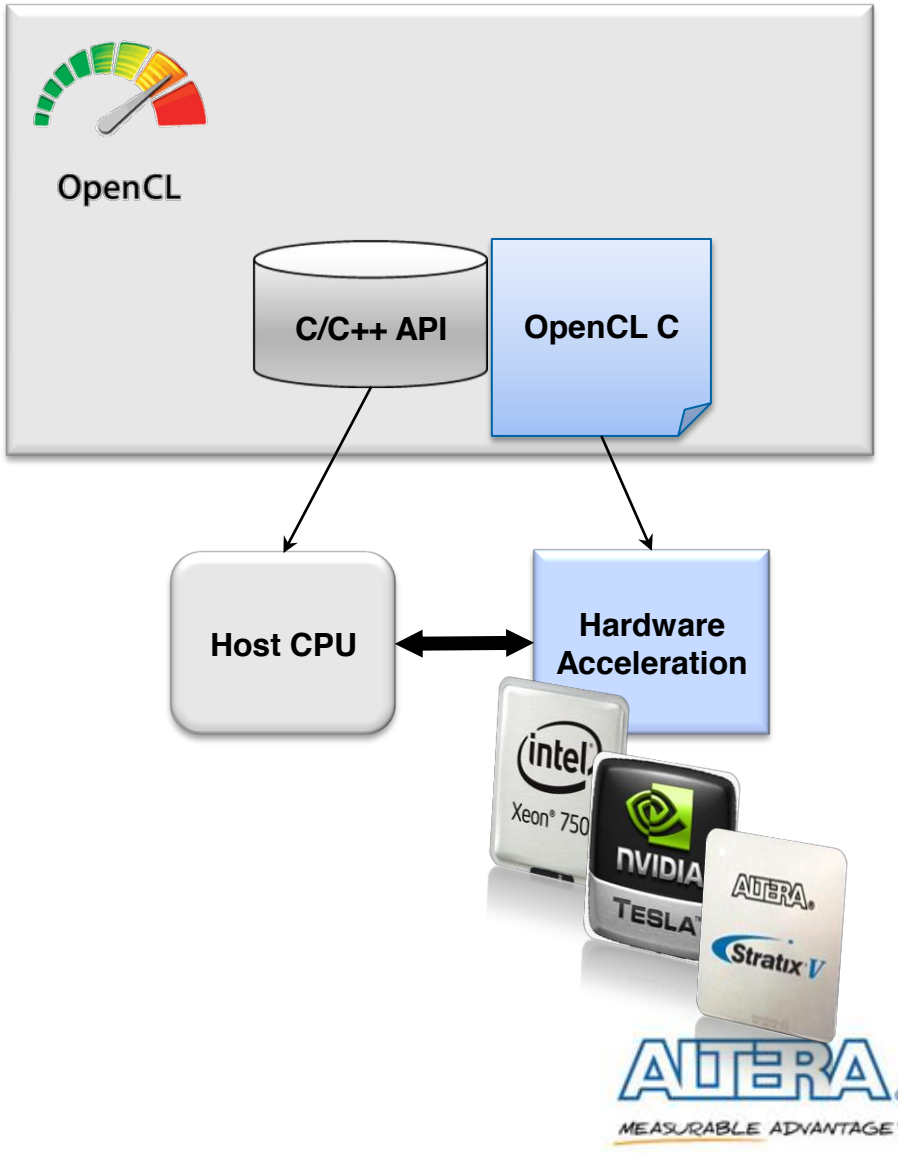
- SoC FPGAs by Altera



ALTERA
MEASURABLE ADVANTAGE™

OpenCL Overview

- **Open Computing Language**
  - Software-centric
    - C/C++ API for host program
    - OpenCL C (C99-based) for acceleration device
  - Unified design methodology
    - CPU offload
      - Memory Access
      - Parallelism
      - Vectorizaton



OpenCL Driving Force

- **Developed and published by consortium**
  - http://www.khronos.org
  - Royalty-free standard
- **Application-driven**

Consumer Electronics

- Image Processing & Video Encoding
- Augmented reality & Computational Photography

Military

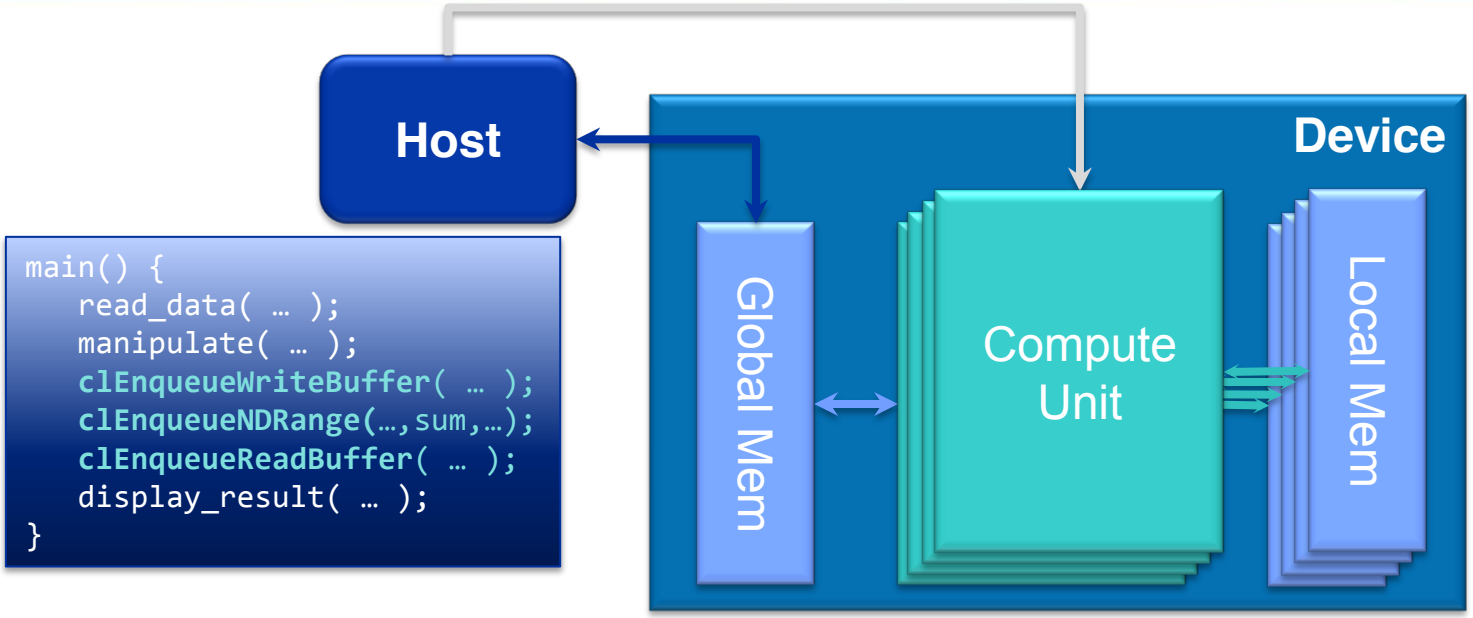
- Video and Radar Analytics

Scientific / High Performance Computing

- Financial, Molecular Dynamics, Bioinformatics, etc.
- Medical rendering algorithms

ALTERA
MEASURABLE ADVANTAGE™

OpenCL Abstract Programming Model




```
main() {
   read_data( … );
   manipulate( … );
   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…,sum,…);
   clEnqueueReadBuffer( … );
   display_result( … );
}
```

```
__kernel void
sum(__global float *a,
    __global float *b,
    __global float *y)
{
  int gid = get_global_id(0);
  y[gid] = a[gid] + b[gid];
}
```

- **Explicit Data Storage**
  - Hierarchical Memory Model
- **Explicit Parallelism**
  - Vectorization
  - Multi-threading

OpenCL Host Program

- **Pure software written in standard 'C'**
- **Communicates with the Accelerator Device via a set of library routines which abstract the communication between the host processor and the kernels**

Copy data from Host to FPGA

Ask the FPGA to run a particular kernel

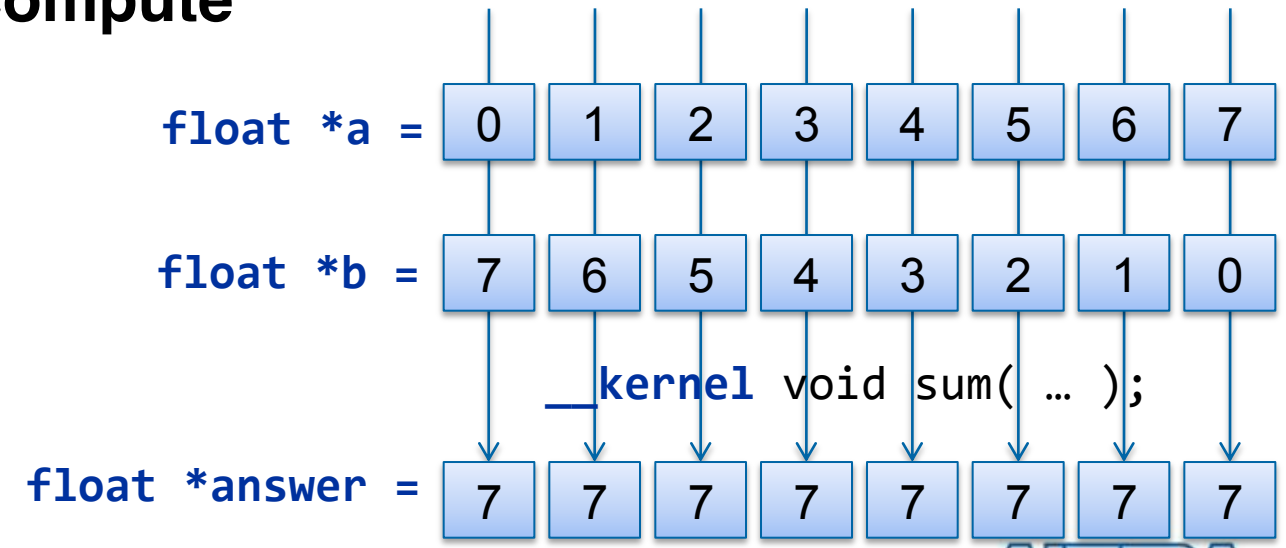
Copy data from FPGA to Host

```
main()
{
    read_data_from_file( … );
    maninpulate_data( … );

    clEnqueueWriteBuffer( … );
    clEnqueueTask(…, my_kernel, …);
    clEnqueueReadBuffer( … );

    display_result_to_user( … );
}
```

OpenCL Kernels

- **Data-parallel function**
  - Defines many parallel threads of execution
  - Each thread has an identifier specified by "**get_global_id**"
  - Contains keyword extensions to specify parallelism and memory hierarchy
- **Executed by compute object**
  - CPU
  - GPU
  - Accelerator

```
__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];
}
```

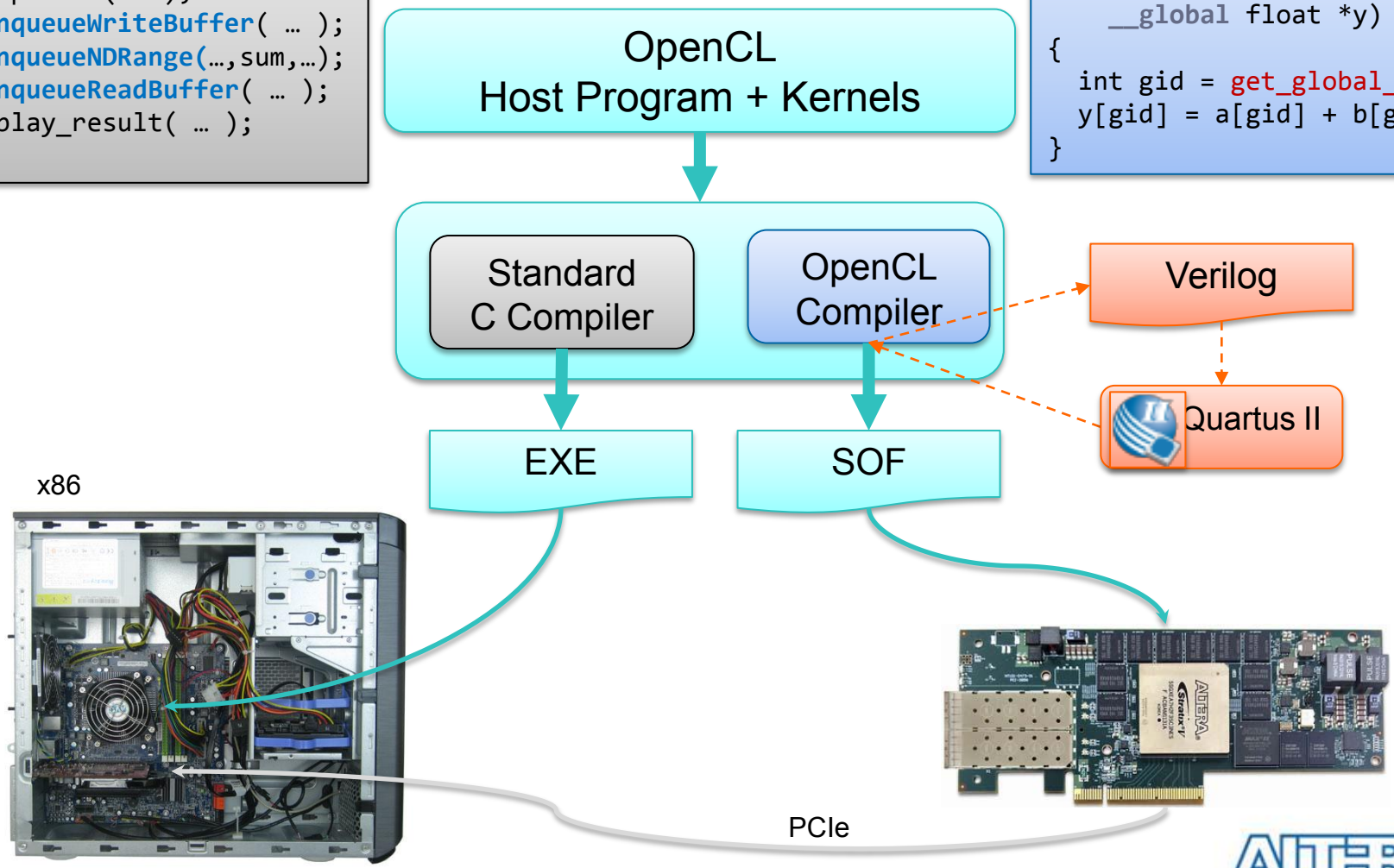


ALTERA
MEASURABLE ADVANTAGE™

Flow

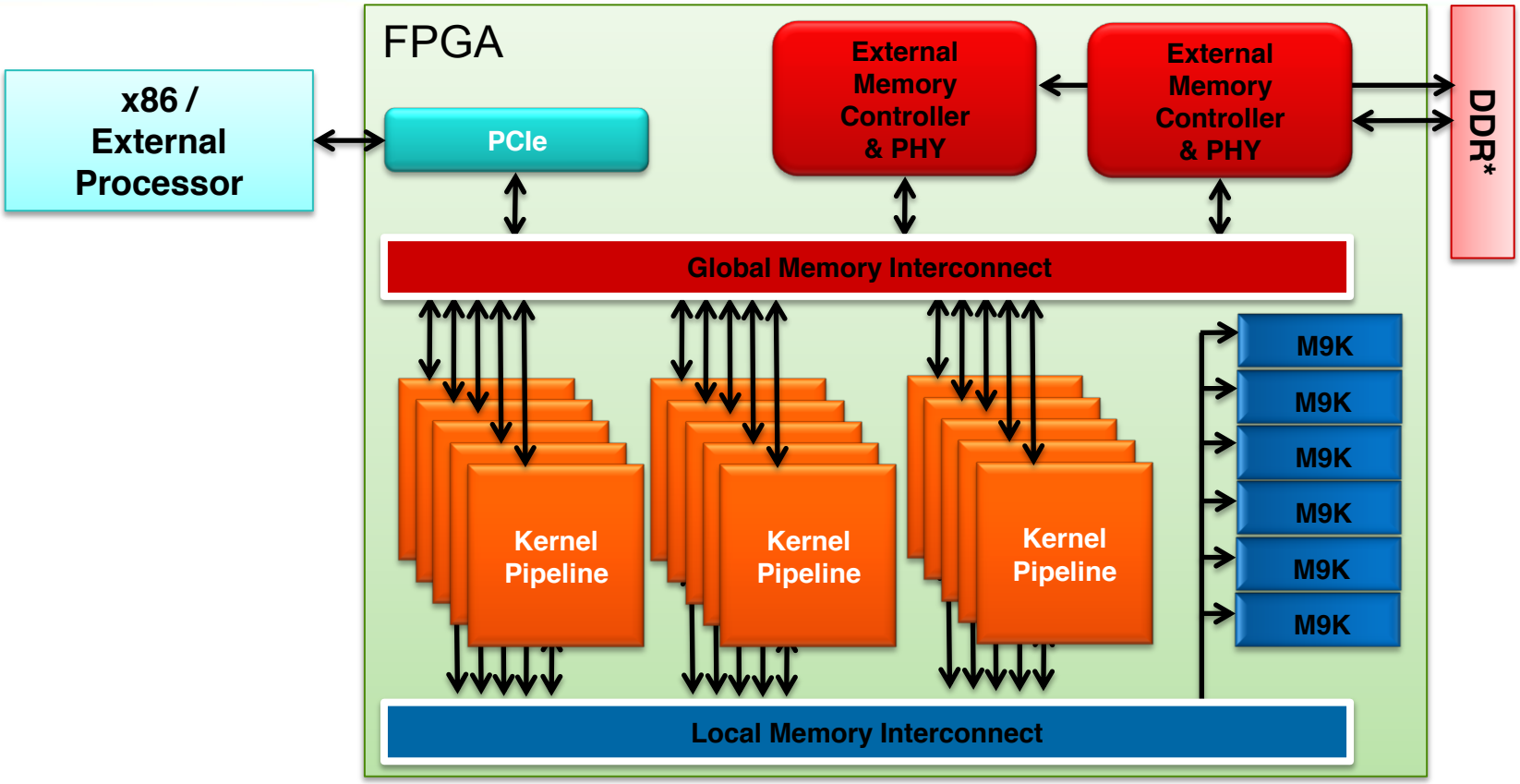
```
main() {
   read_data( … );
   manipulate( … );
   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…,sum,…);
   clEnqueueReadBuffer( … );
   display_result( … );
}
```

```
__kernel void
sum(__global float *a,
    __global float *b,
    __global float *y)
{
  int gid = get_global_id(0);
  y[gid] = a[gid] + b[gid];
}
```



ALTERA
MEASURABLE ADVANTAGE™

FPGA OpenCL Architecture



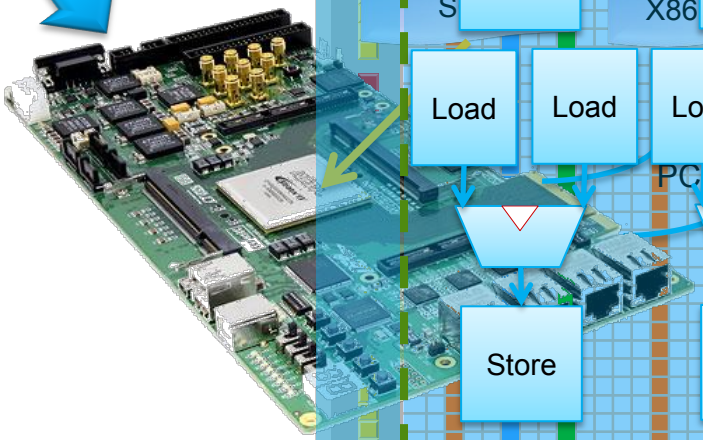
- Modest external memory bandwidth
- Extremely high internal memory bandwidth
- Highly customizable compute cores

ALTERA
MEASURABLE ADVANTAGE™

Compiling OpenCL to FPGAs

```
__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];
}
```

```
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```



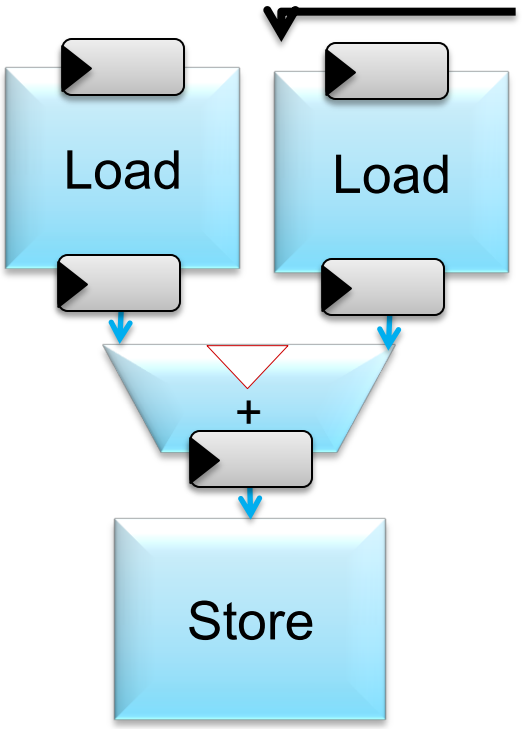
Host Program

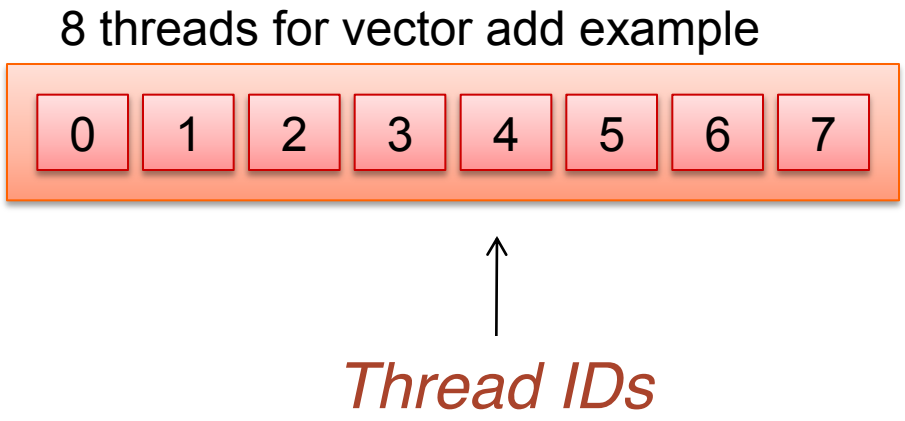
```
main()
{
  read_data_from_file( … );
```

Mapping Multithreaded Kernels to FPGAs

- **The most simple way of mapping kernel functions to FPGAs is to replicate hardware for each thread**
  - Inefficient and wasteful
- **Better method involves taking advantage of *pipeline parallelism***
  - Attempt to create a deeply pipelined representation of a kernel
  - On each clock cycle, we attempt to send in input data for a new thread
  - Method of mapping coarse grained thread parallelism to fine-grained FPGA parallelism

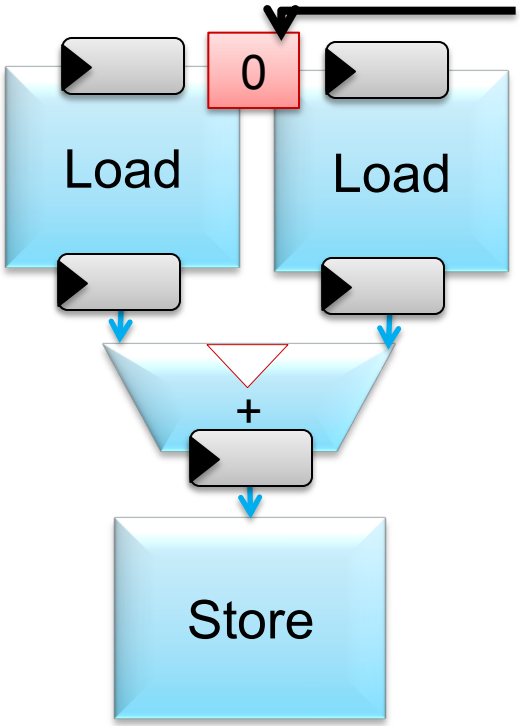
Example Pipeline for Vector Add

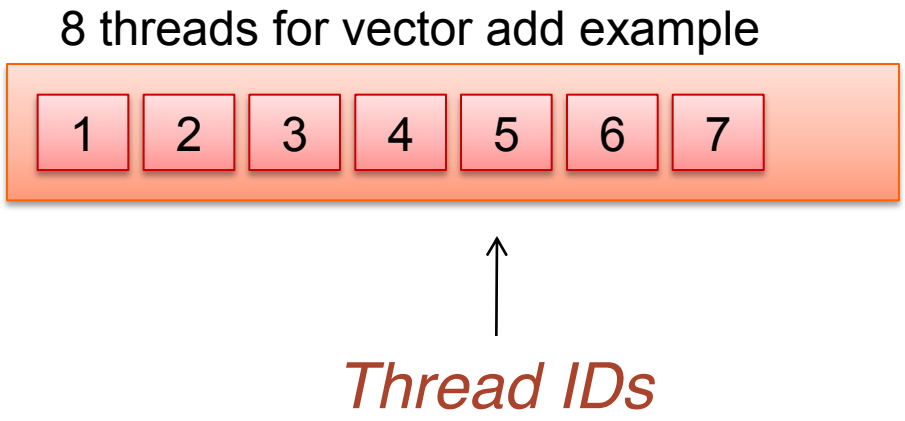




- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

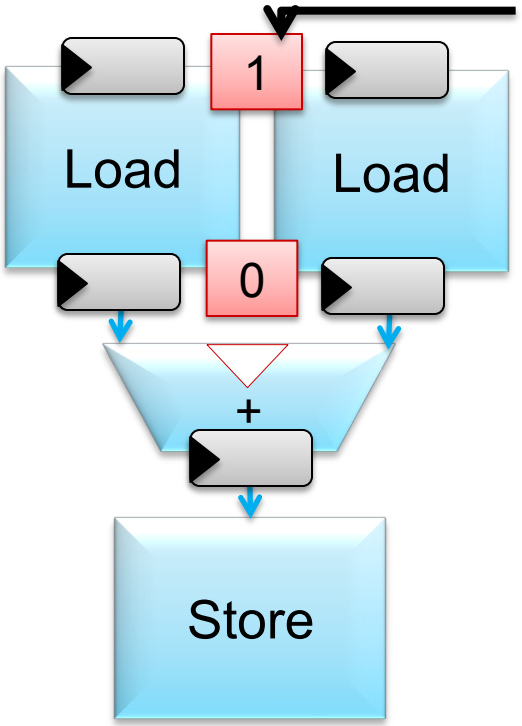
Example Pipeline for Vector Add

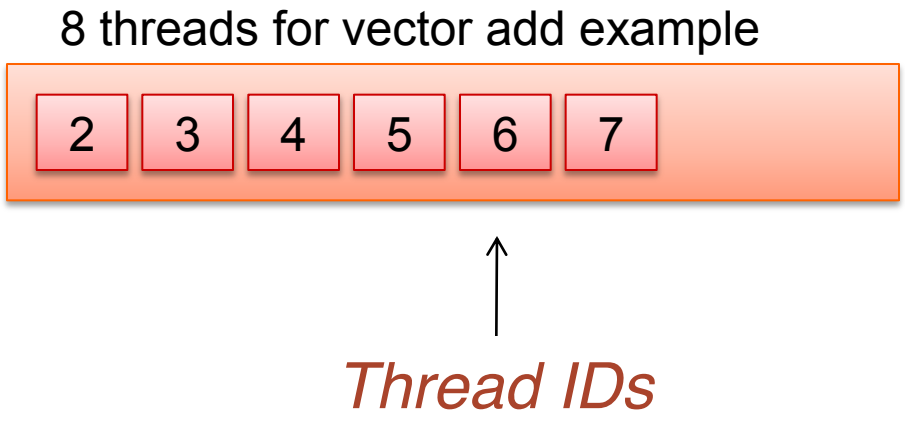




- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

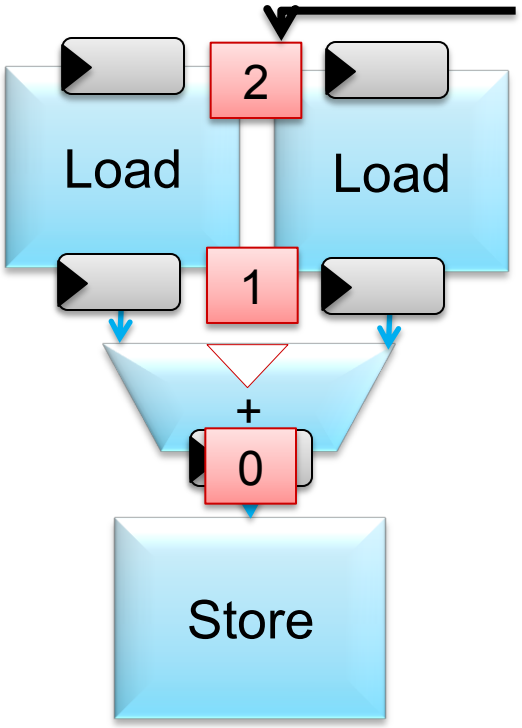
Example Pipeline for Vector Add

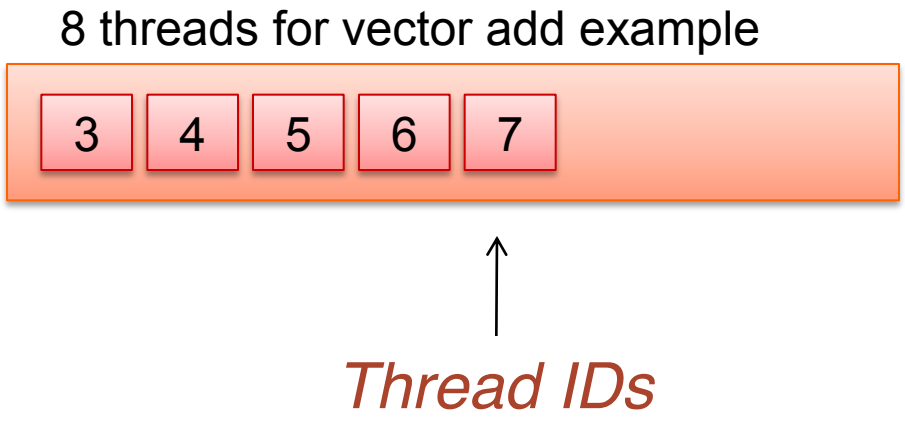




- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

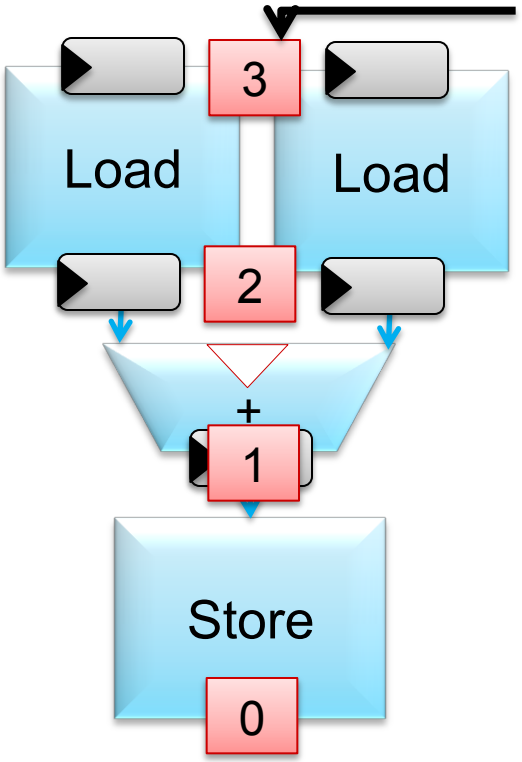
Example Pipeline for Vector Add

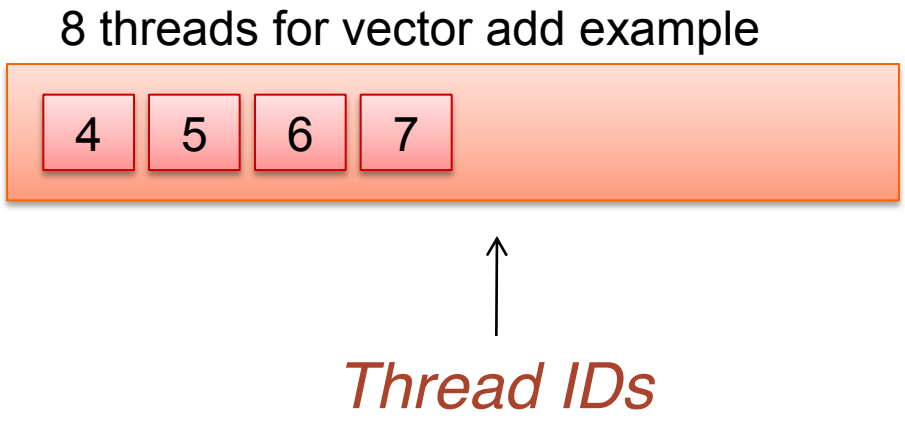




- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

Example Pipeline for Vector Add

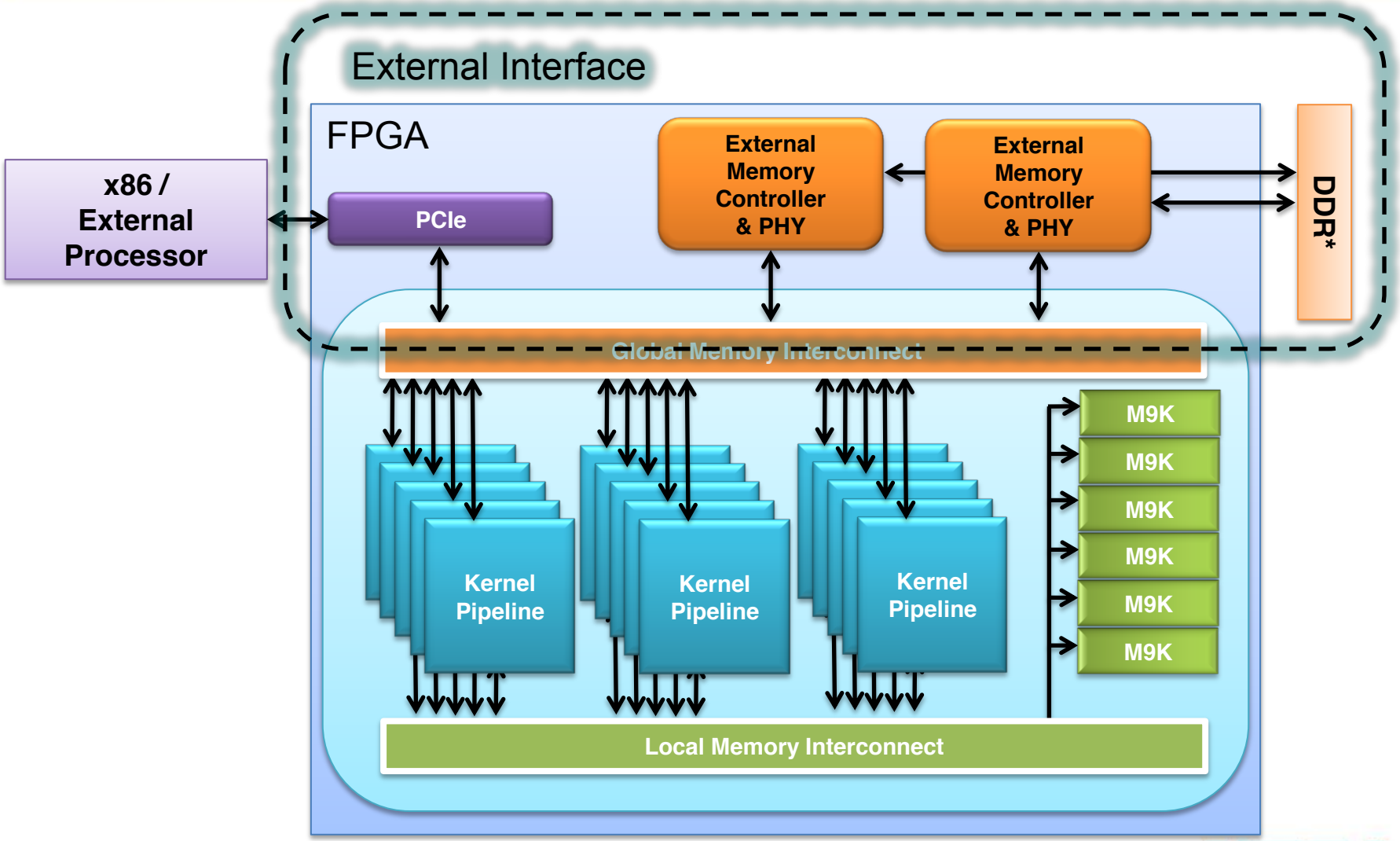




- **On each cycle the portions of the pipeline are processing different threads**
- **While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored**

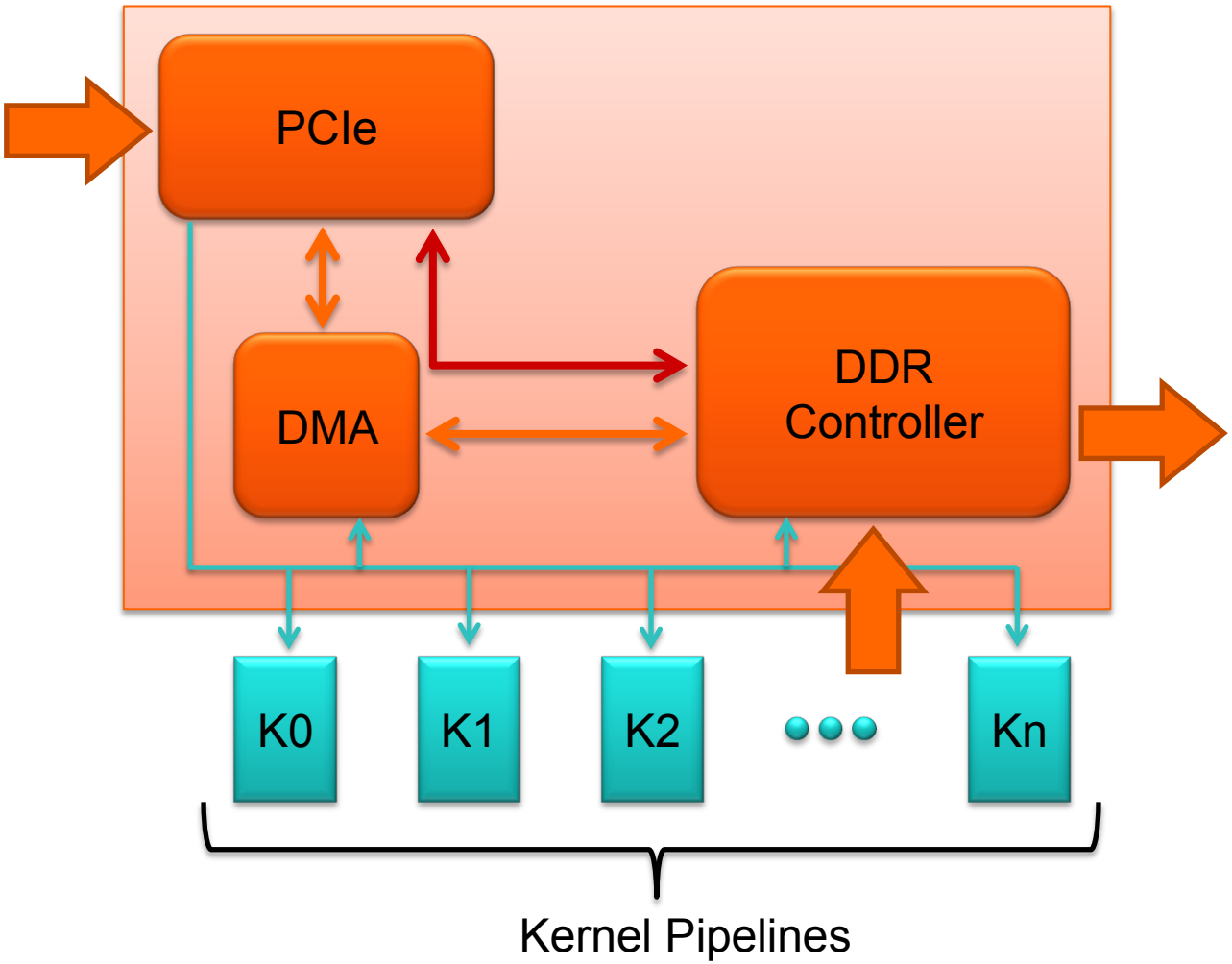
ALTERA OPENCL SYSTEM ARCHITECTURE

OpenCL System Architecture

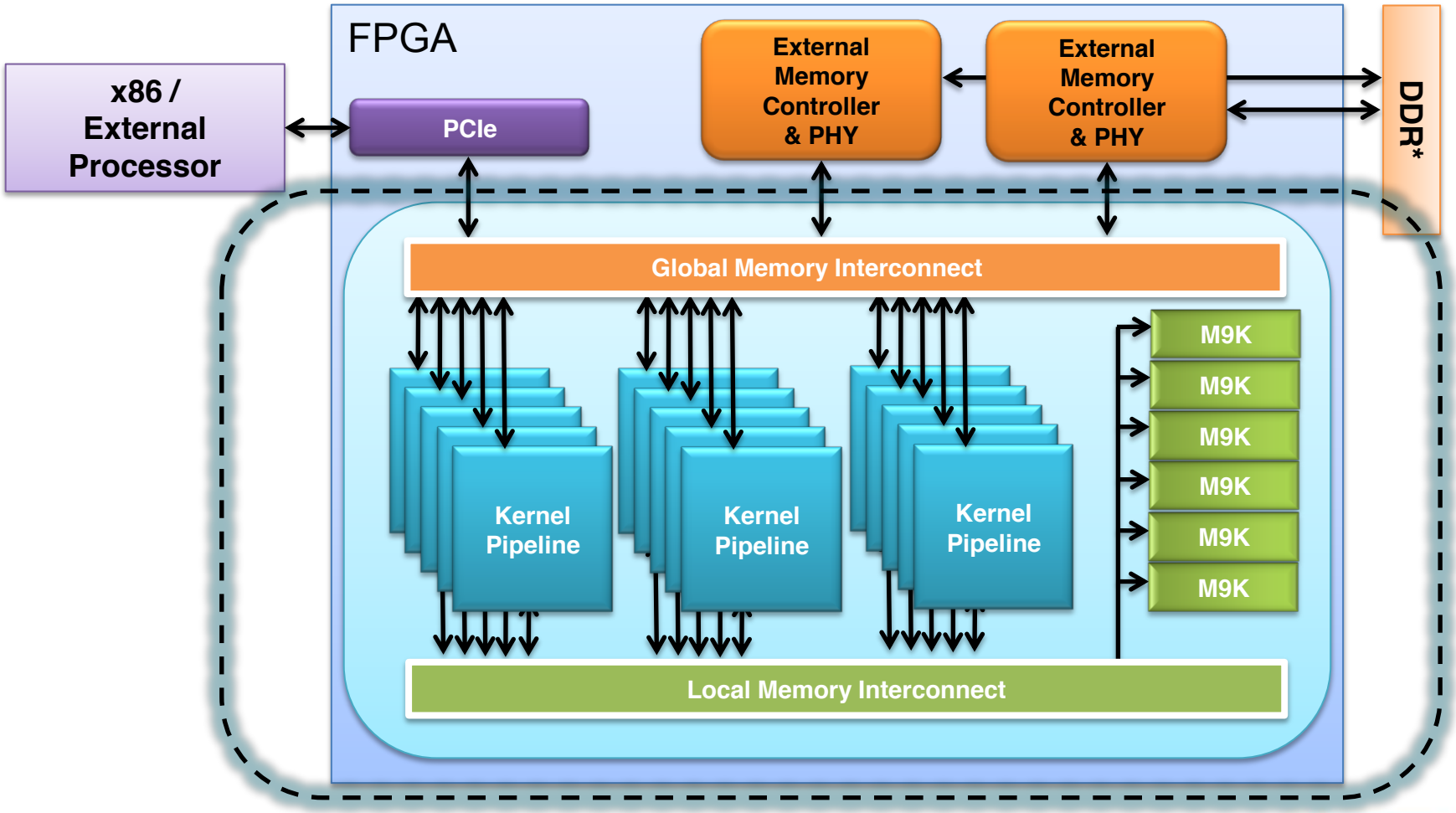


ALTERA
MEASURABLE ADVANTAGE™

External Interface : High Level

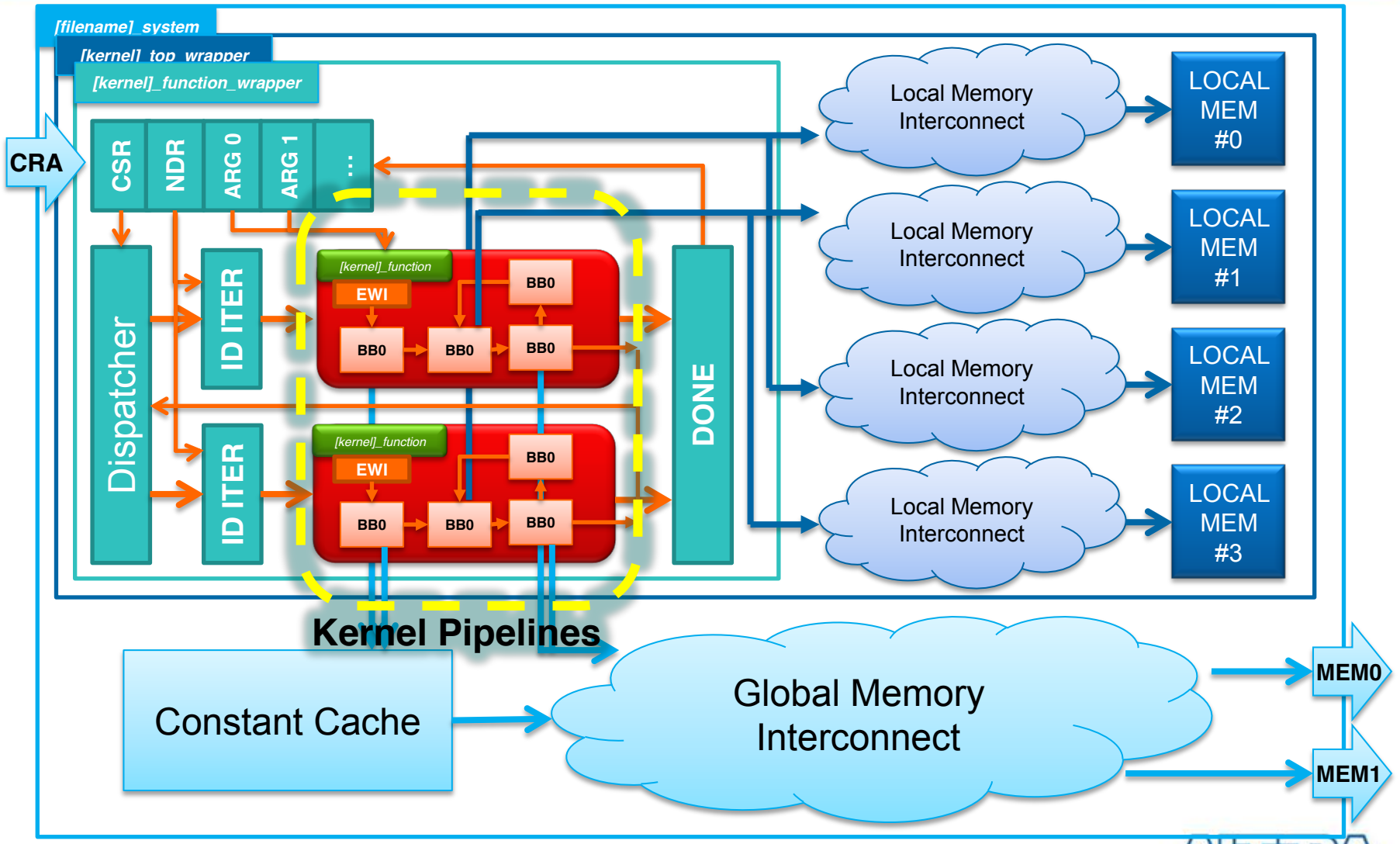


OpenCL System Architecture



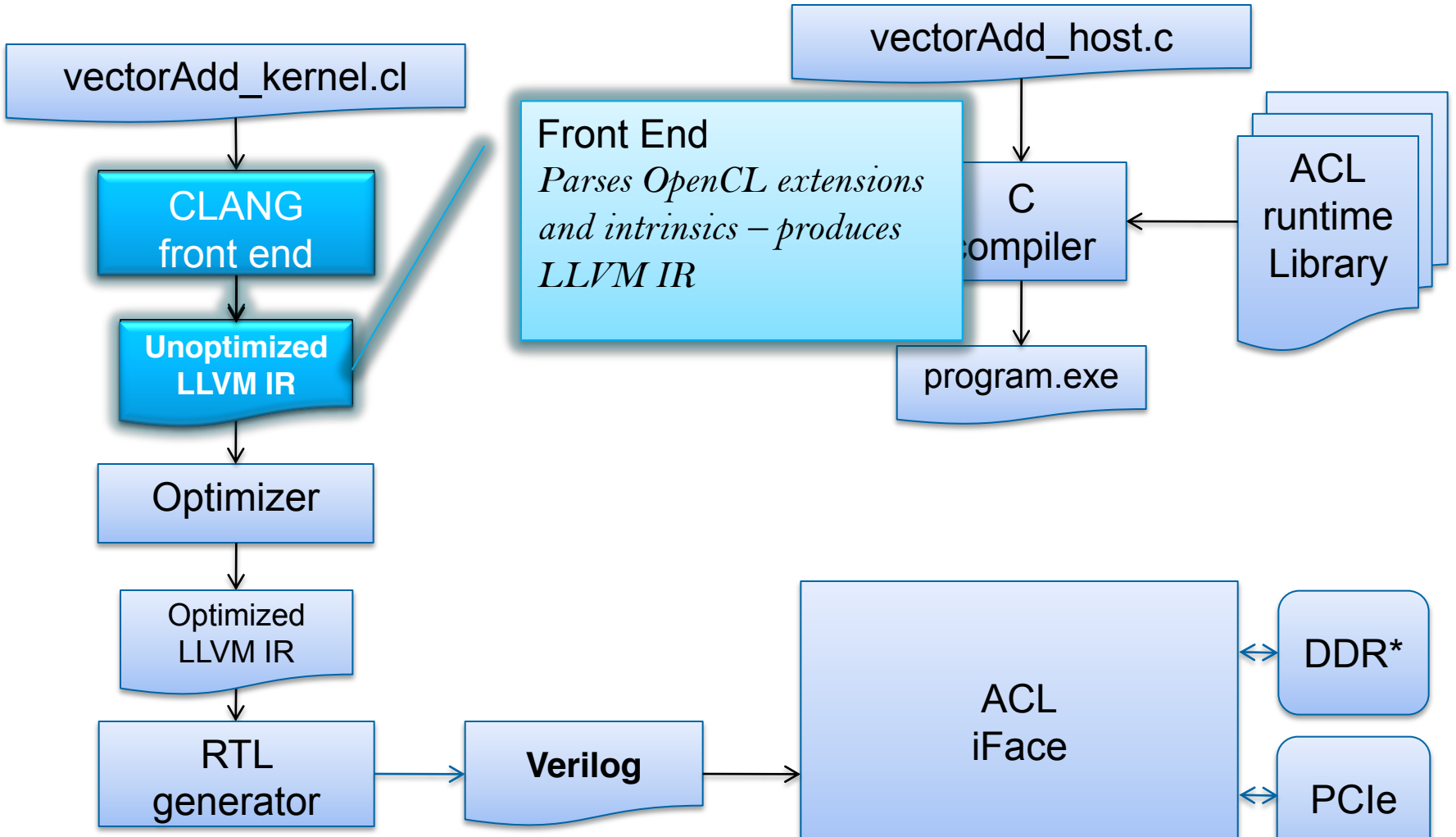
ALTERA
MEASURABLE ADVANTAGE™

Altera OpenCL Kernel Architecture



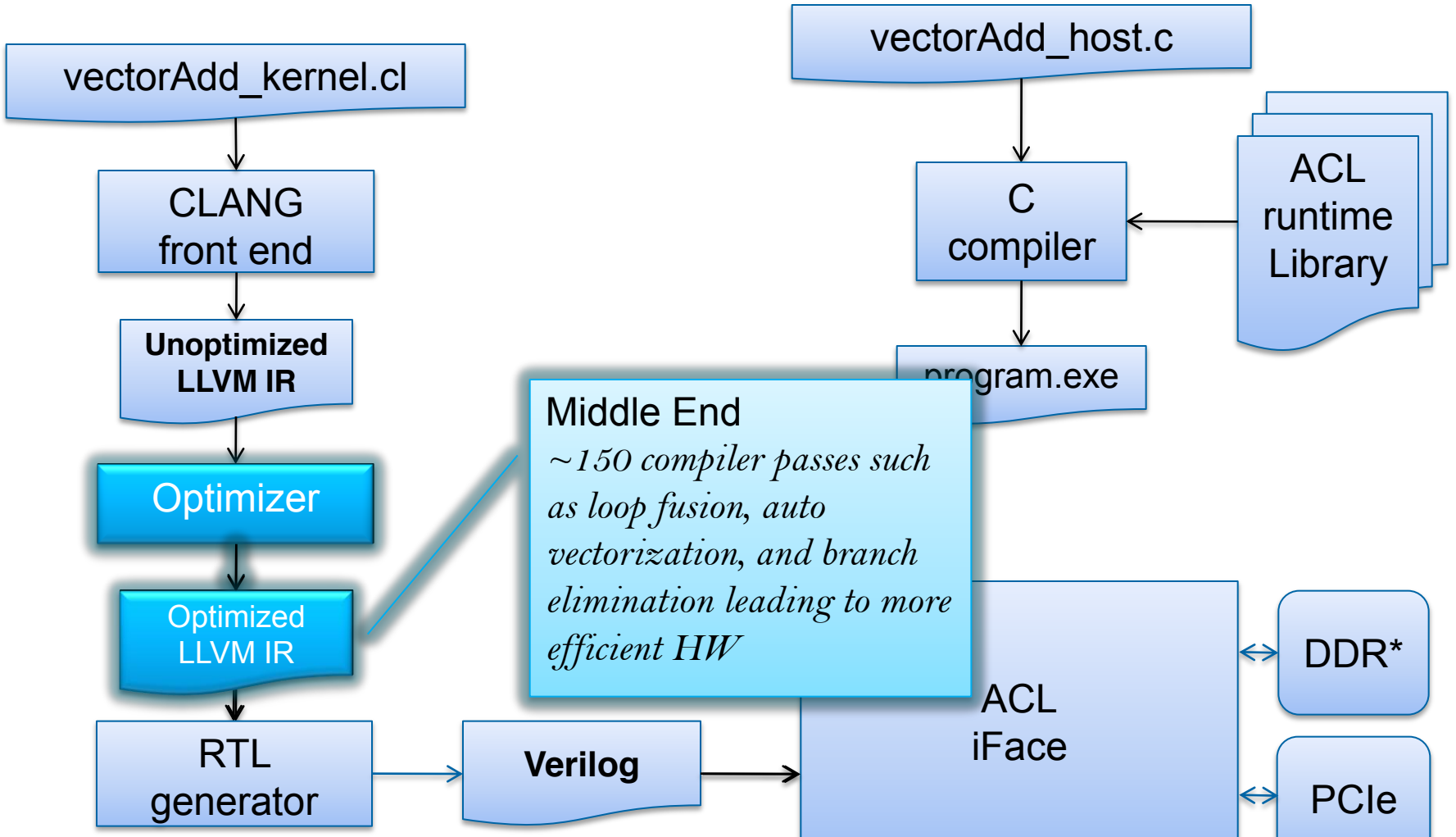
ALTERA
MEASURABLE ADVANTAGE™

OpenCL CAD Flow



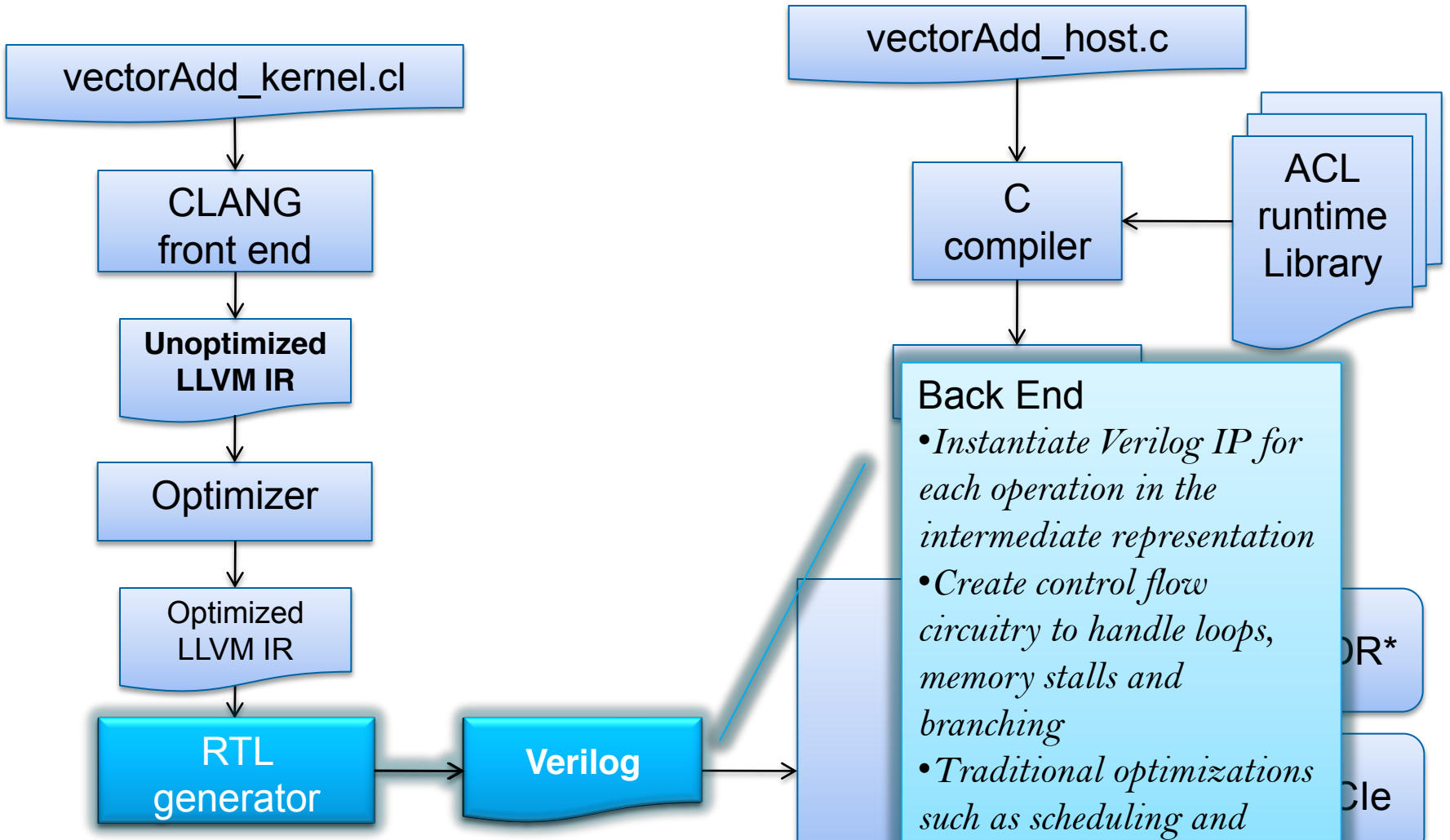
ALTERA
MEASURABLE ADVANTAGE™

OpenCL CAD Flow



ALTERA
MEASURABLE ADVANTAGE™

OpenCL CAD Flow

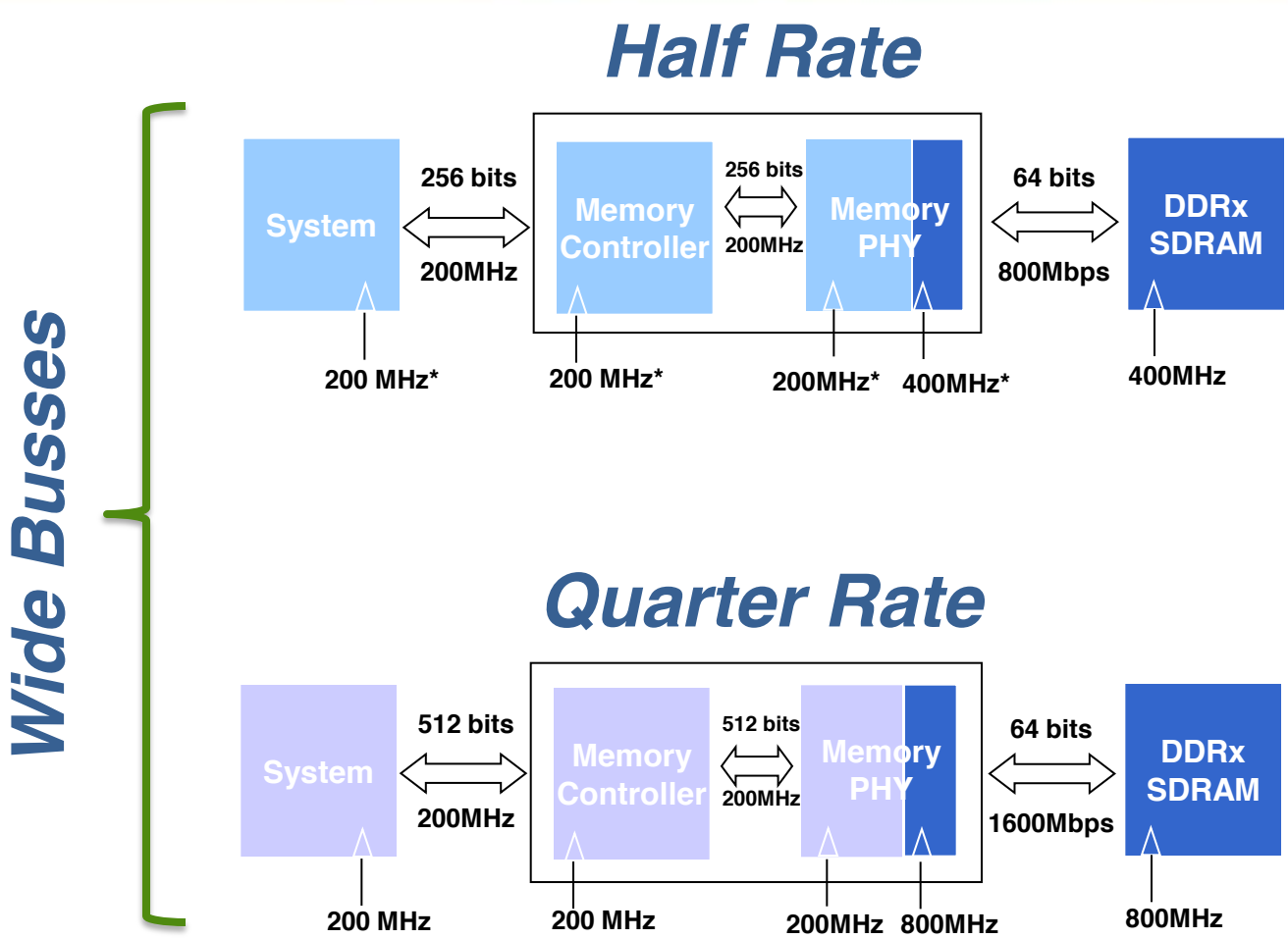


MEMORY ACCESS OPTIMIZATIONS

Memory Access Optimizations

- **At the high level, present OpenCL benchmarks are memory-to-memory transformations**
  - Some data is read from global memory, processed, and results stored in global memory
- **Global Memory**
  - Is implemented using DDR3-1600 memory
    - Two separate DIMMs
  - Has long latency associated with it
  - Access pattern to this memory has a significant impact on application performance
    - In many cases it can dominate application performance
- **Local Memory**
  - Is implemented on-chip
  - Is fast, but very limited in size (54M bits on Stratix V A7 and 41M bits on Stratix V D5)
- **The first and most significant challenge is to architect your application to balance the use of memories available to meet design requirements**

Global Memory - DDRx Configuration

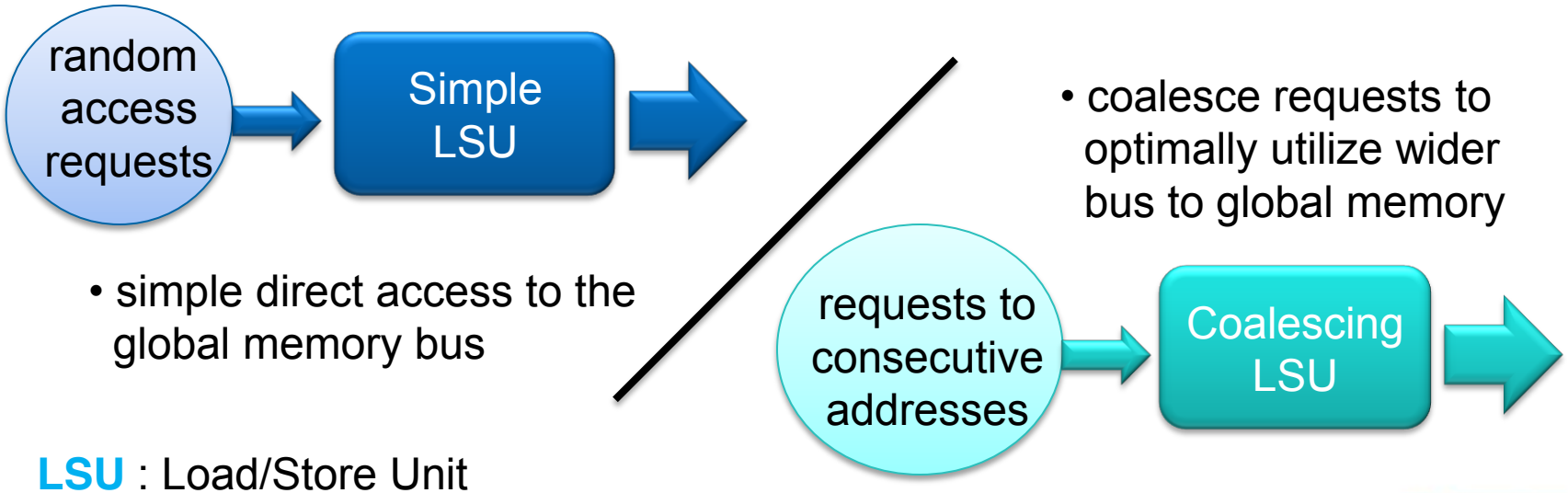


* This frequency target is an example only, it does not reflect the actual frequency configuration that is supported by DDRx controller

ALTERA
MEASURABLE ADVANTAGE™

DDRx Memory Efficiency

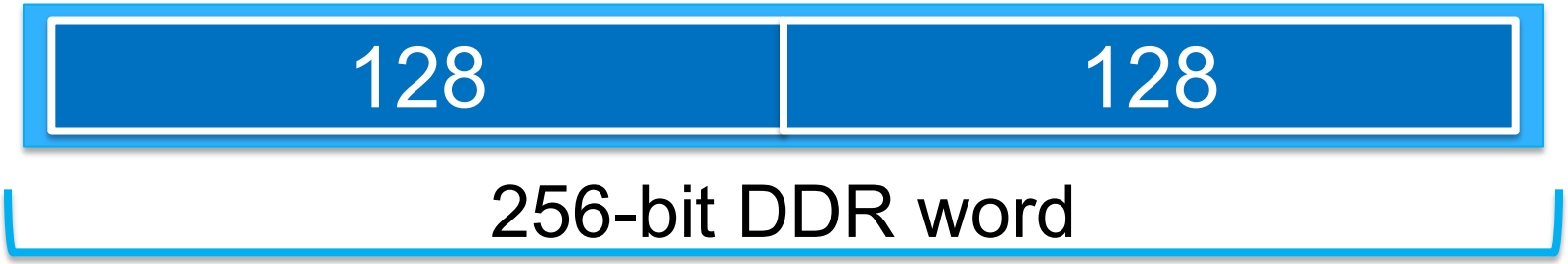
- **Access pattern is critical to DDR efficiency**
  - Favors large burst transactions
- **Tradeoff between hardware complexity, and efficiency of the memory access pattern**
  - Certain access patterns do not warrant complex hardware
  - Complex hardware can be simplified with knowledge about the pattern or surrounding hazards
- **OpenCL challenge: Make memory access as efficient as possible, for a variety of common access patterns, without placing a burden on device resources**
  - Convert inefficient patterns (e.g. a series of short single word reads) into efficient patterns (e.g. a single large burst read)



LSU : Load/Store Unit

Load/Store Memory Coalescing

- External memory has wide words (256/512 bits)
- Loads/stores typically access narrower words (32 or 128 bits)

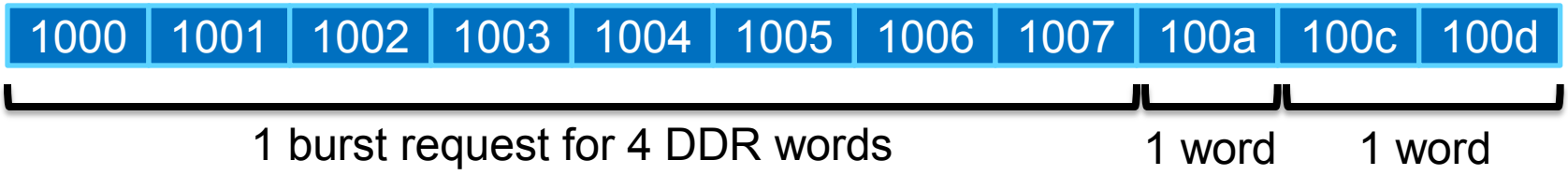


- Given a sequence of loads/stores, we want to make as few external memory read/write requests as possible

Load/Store Memory Coalescing

- **Coalescing is important for good performance**
  - Combine loads/stores that access the same DDR word **or the one ahead of the previously-accessed DDR word**
  - Make one big multi-word burst request to external memory whenever possible
  - ➢ Fewer requests → less contention to global memory
  - ➢ Contiguous bursts → less external memory overhead

Load/Store Addresses (128-bit words):



→ 3 requests in total

Static Versus Dynamic Coalescing

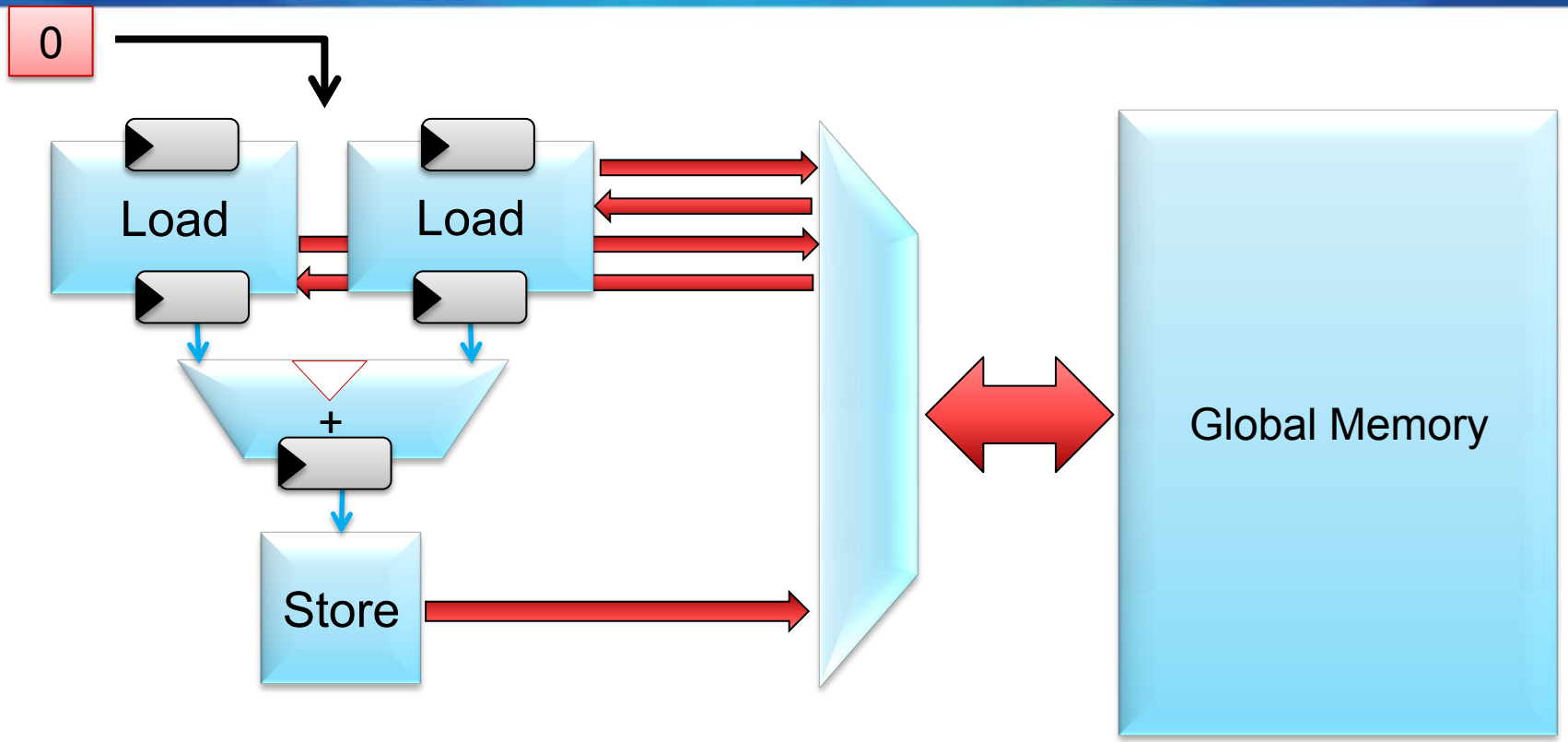
- **In Altera OpenCL SDK, there are two types of coalescing available**
  - Static, that can be inferred by examining the kernel source code
  - Dynamic, performed by the hardware whenever possible
    - But the hardware to do this is not insignificant

- **Making it easy for the compiler to determine the access pattern in your application will drive it towards a better implementation**

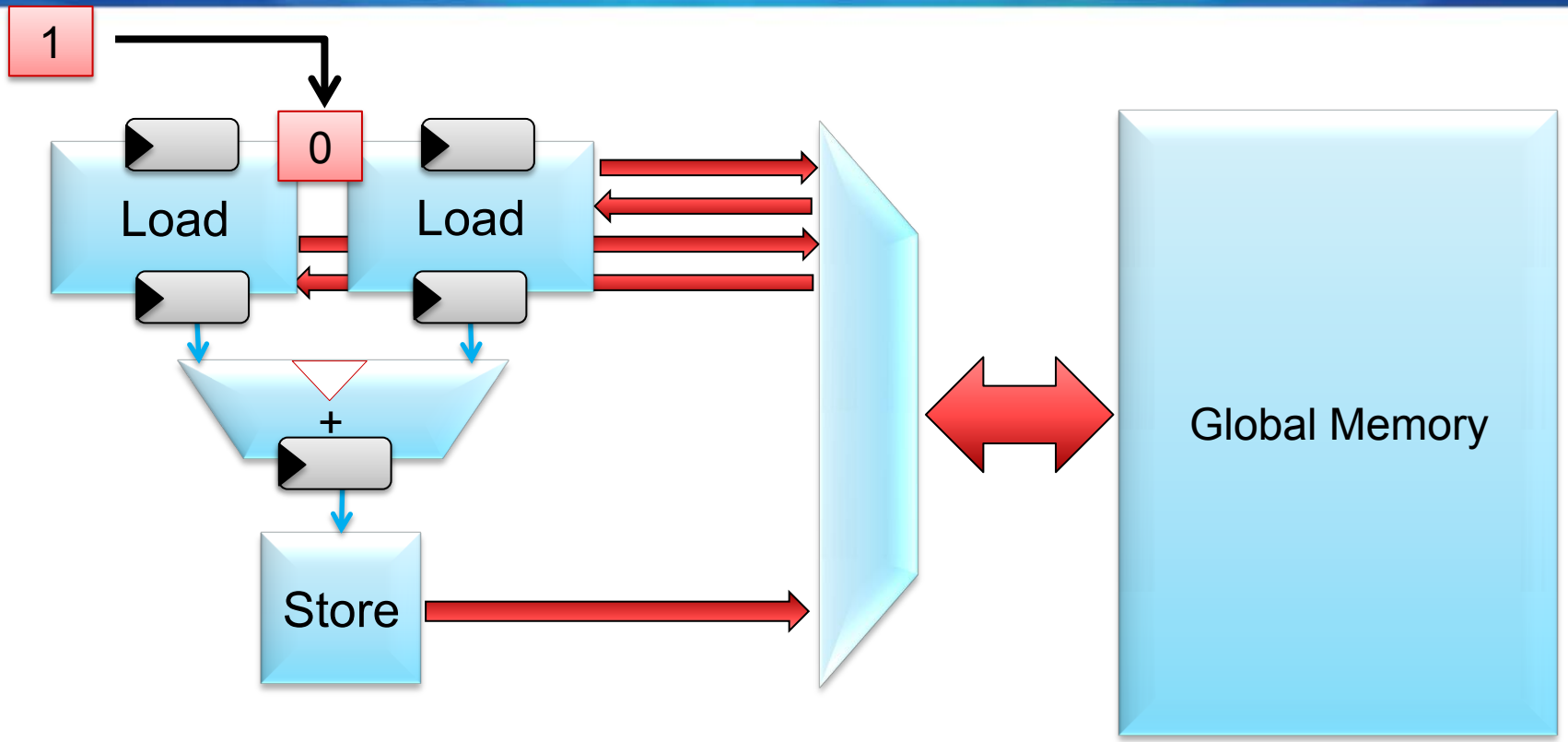
Example 1 – RANDOM access (with Modelsim)

```
__kernel void
    __attribute__(((max_work_group_size(256)))
    example1(__global int *a, __global int *b)
{
   int lid = get_local_id(0);
   int gid = get_global_id(0);
   int size = get_local_size(0);
   int group = get_group_id(0);
   int offset = (group+1)*size;
   b[gid] = a[gid] + a[offset-lid-1];
}
```

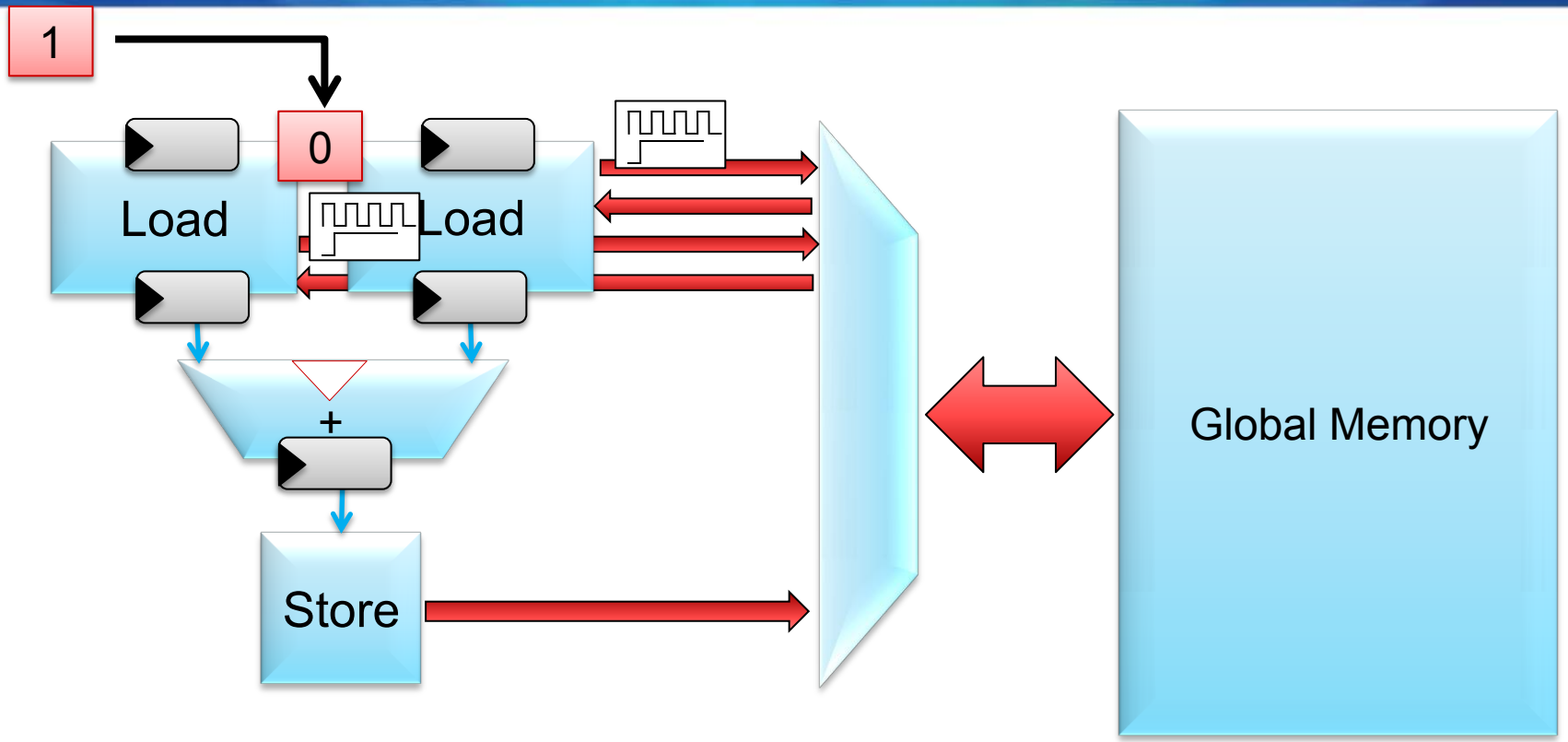
Example 1 – RANDOM access



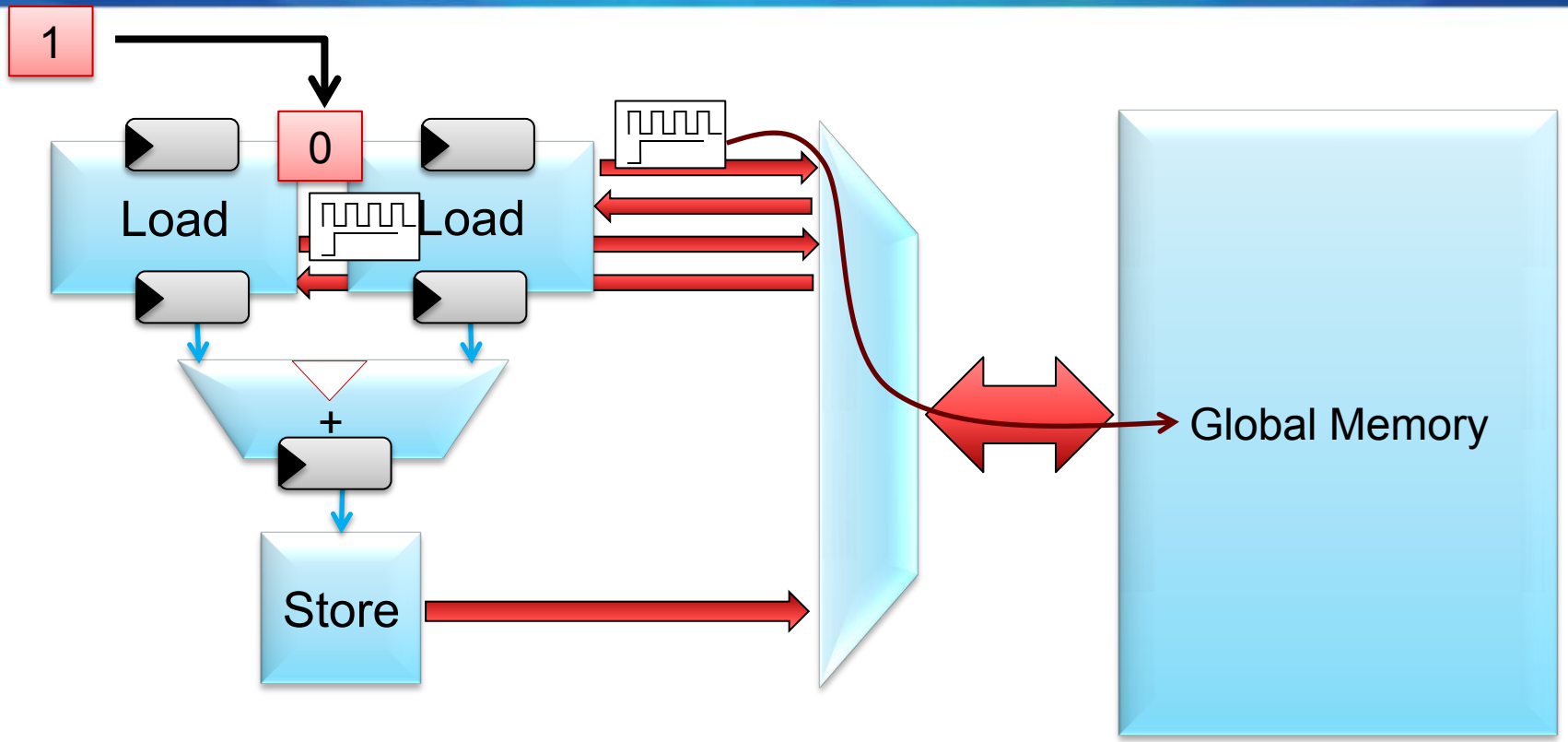
Example 1 – RANDOM access



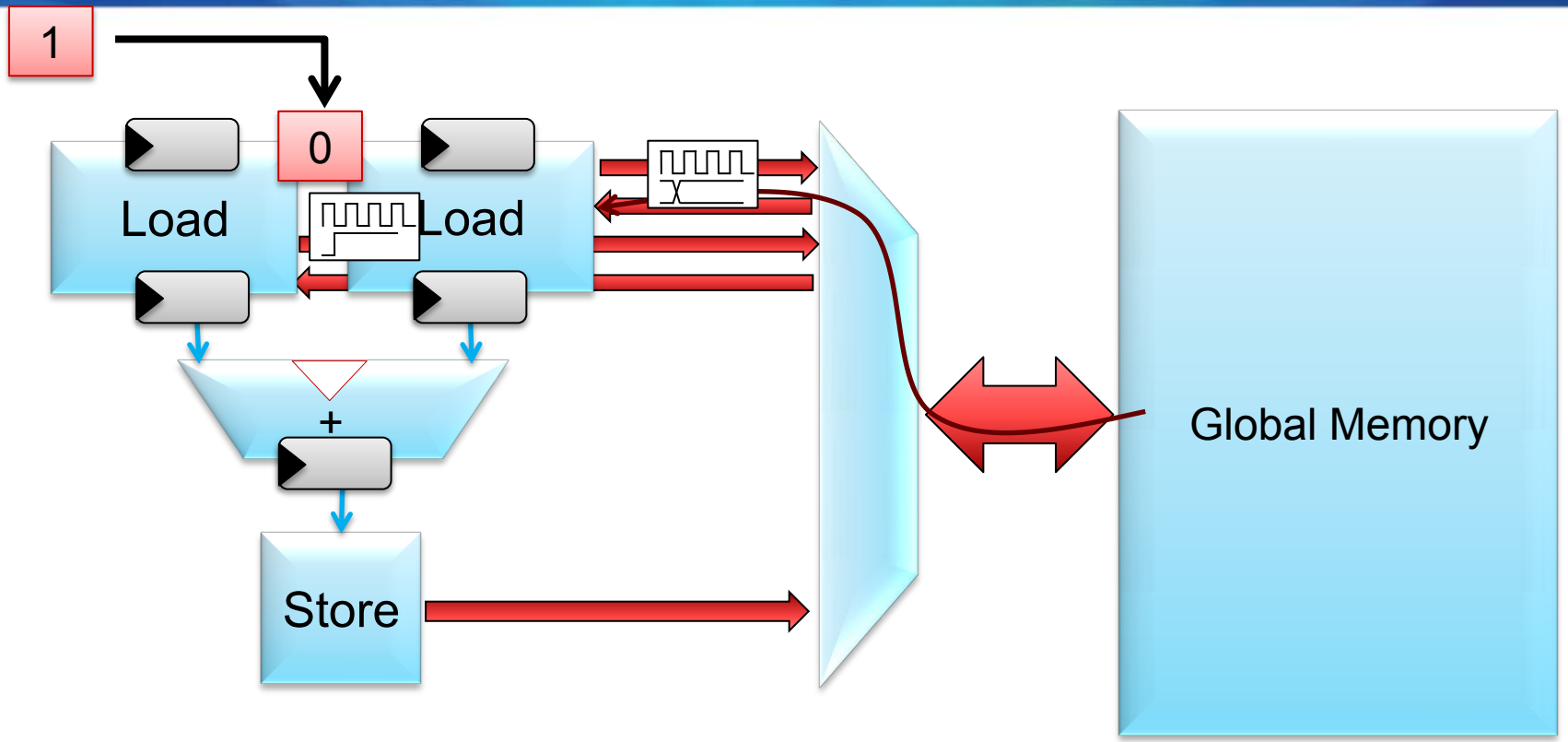
Example 1 – RANDOM access



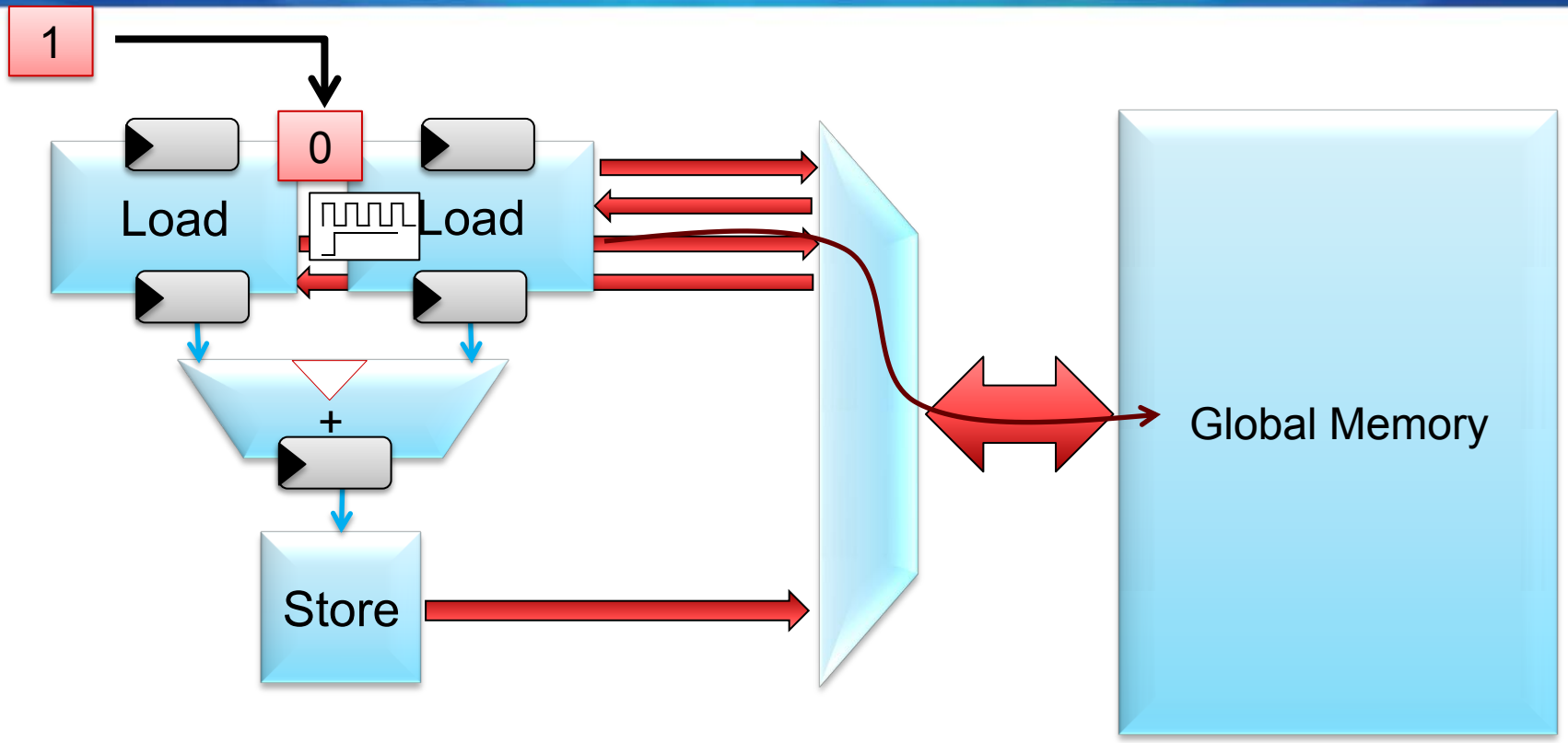
Example 1 – RANDOM access



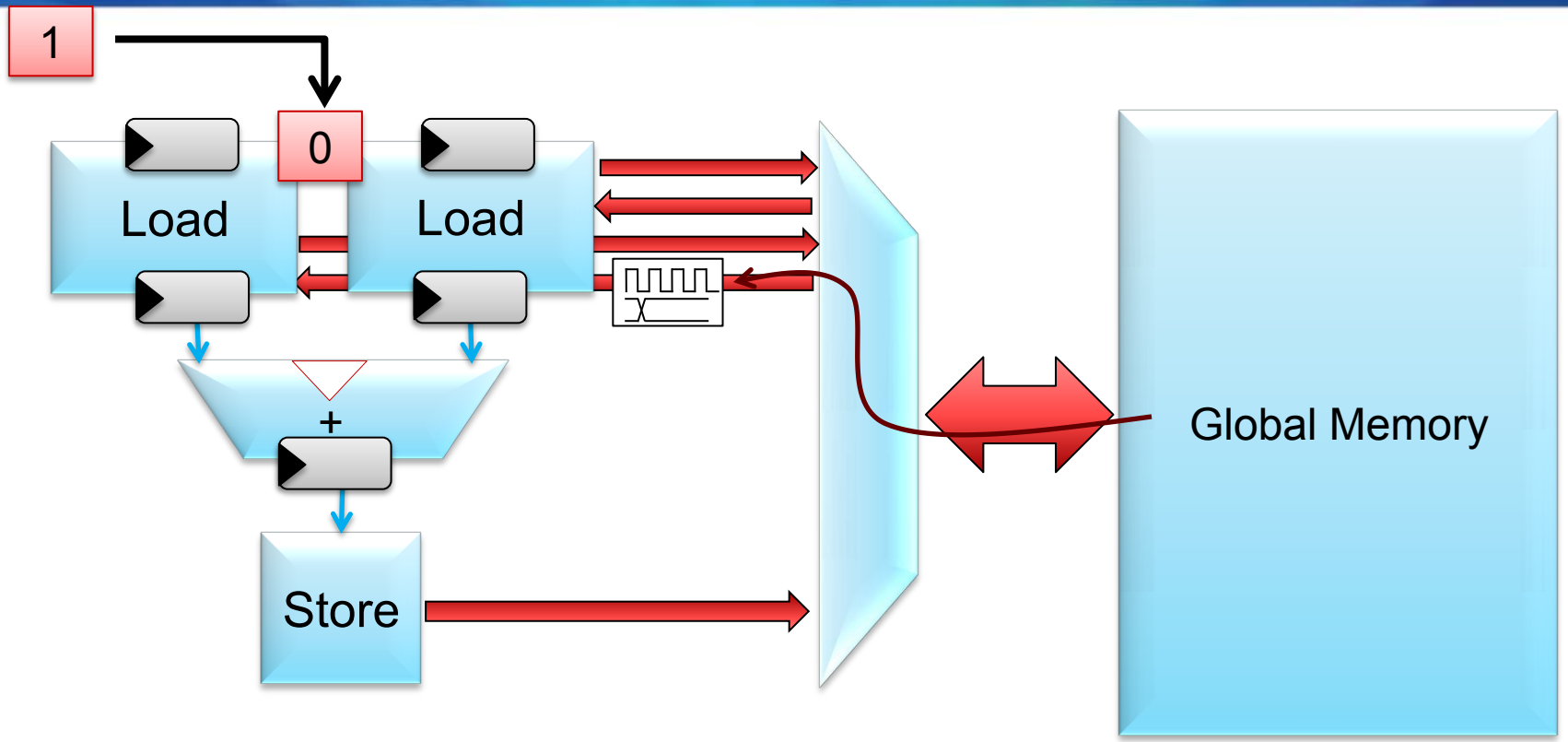
Example 1 – RANDOM access



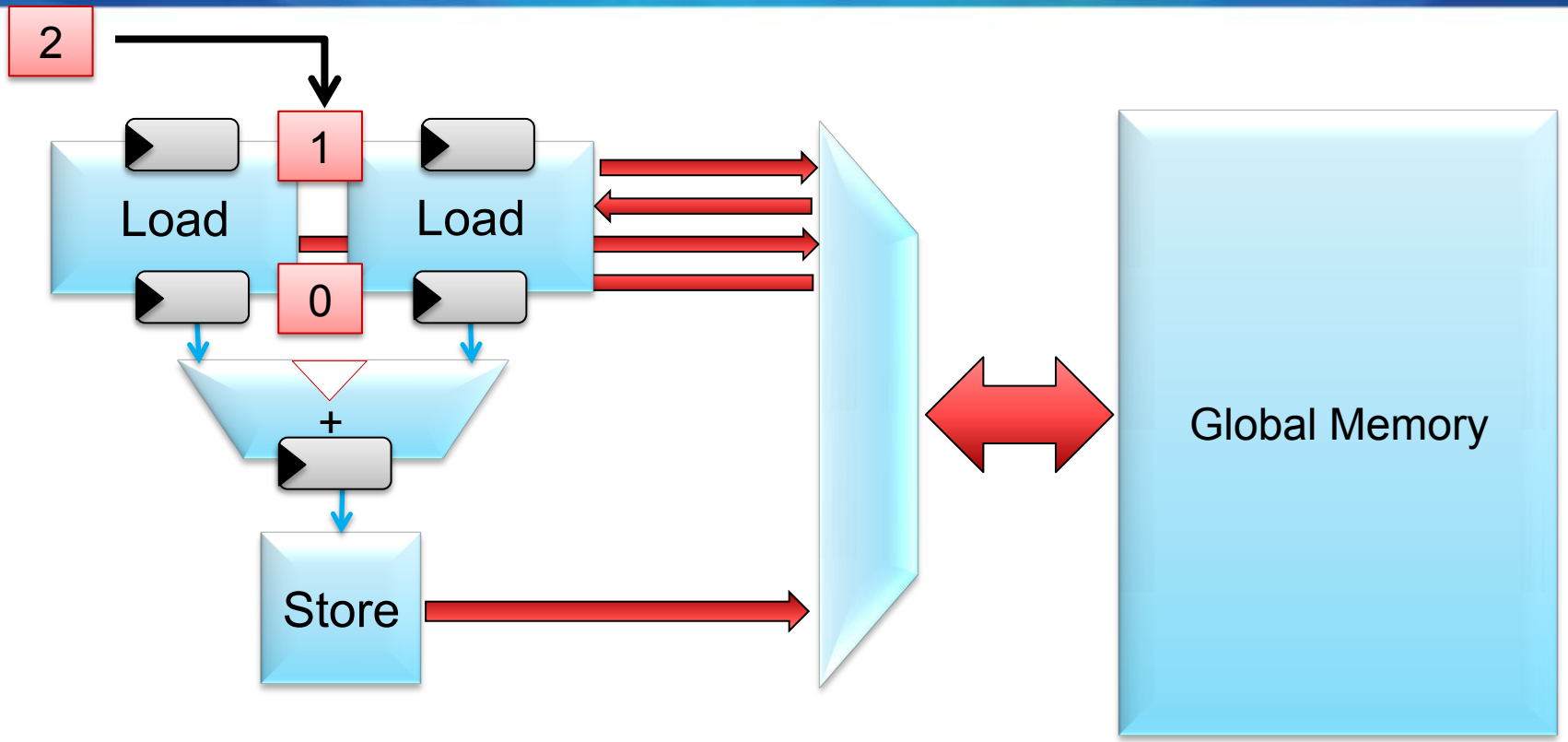
Example 1 – RANDOM access



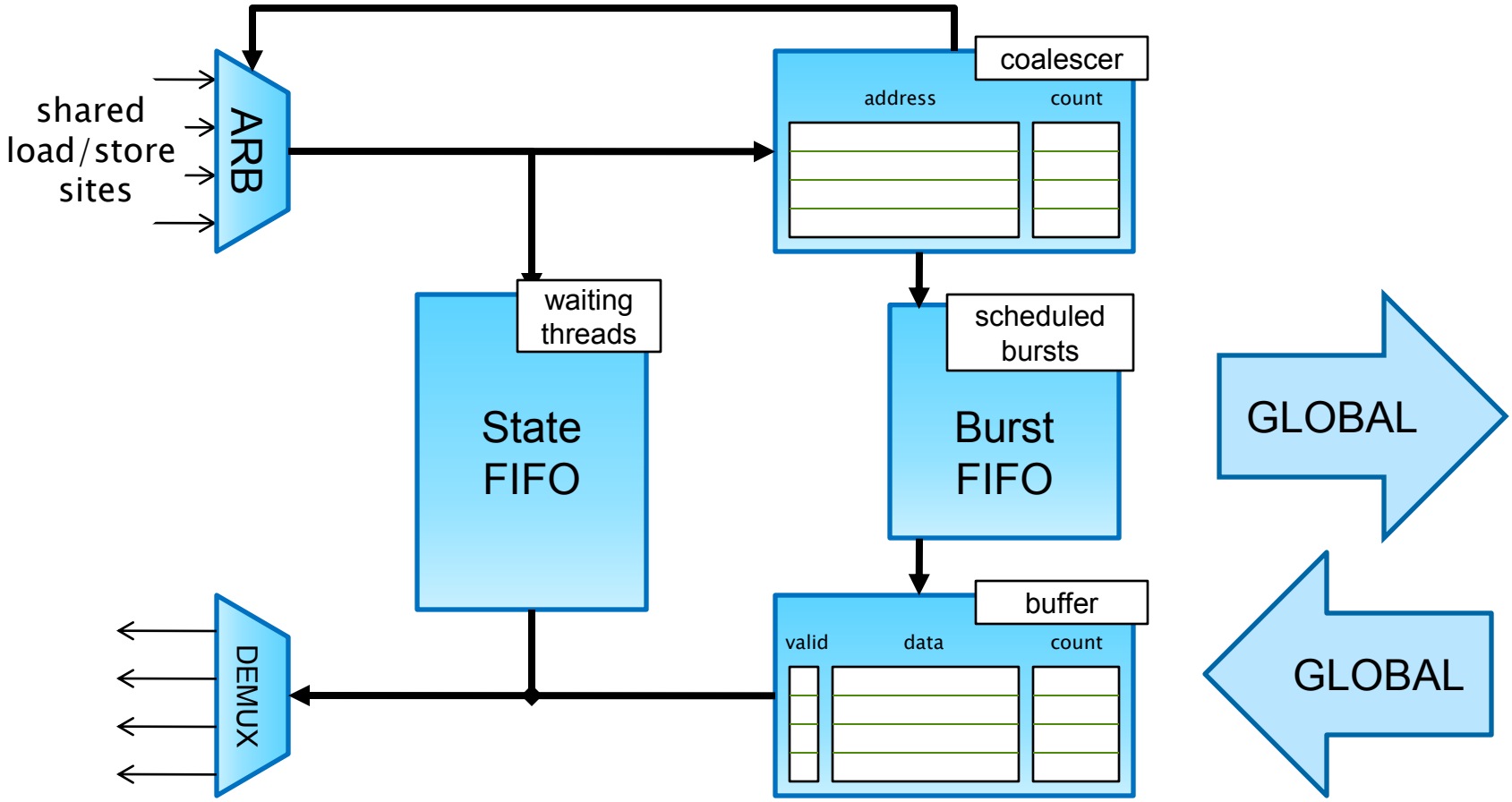
Example 1 – RANDOM access



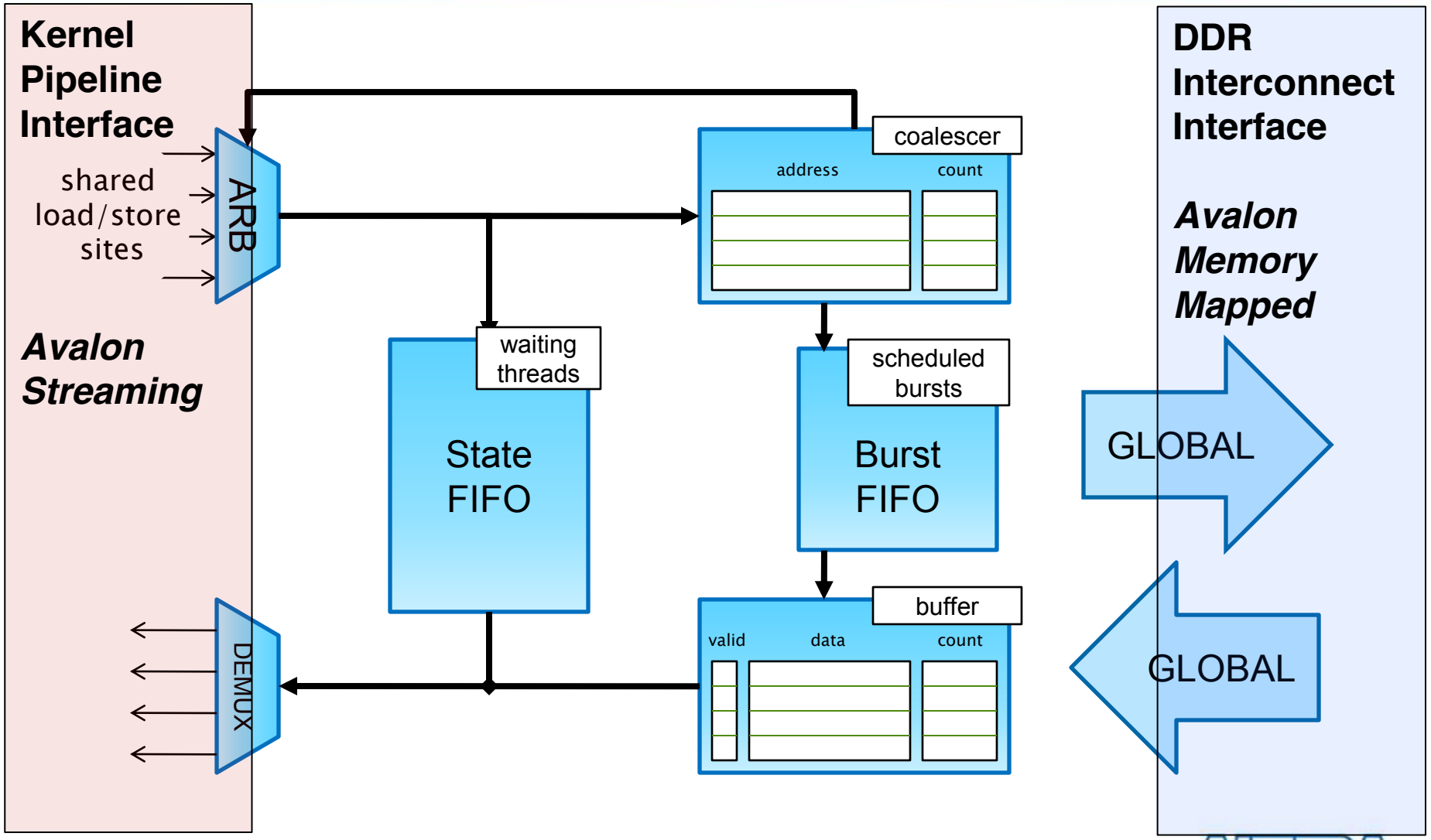
Example 1 – RANDOM access



Load-Store Unit - High-Level Block Diagram

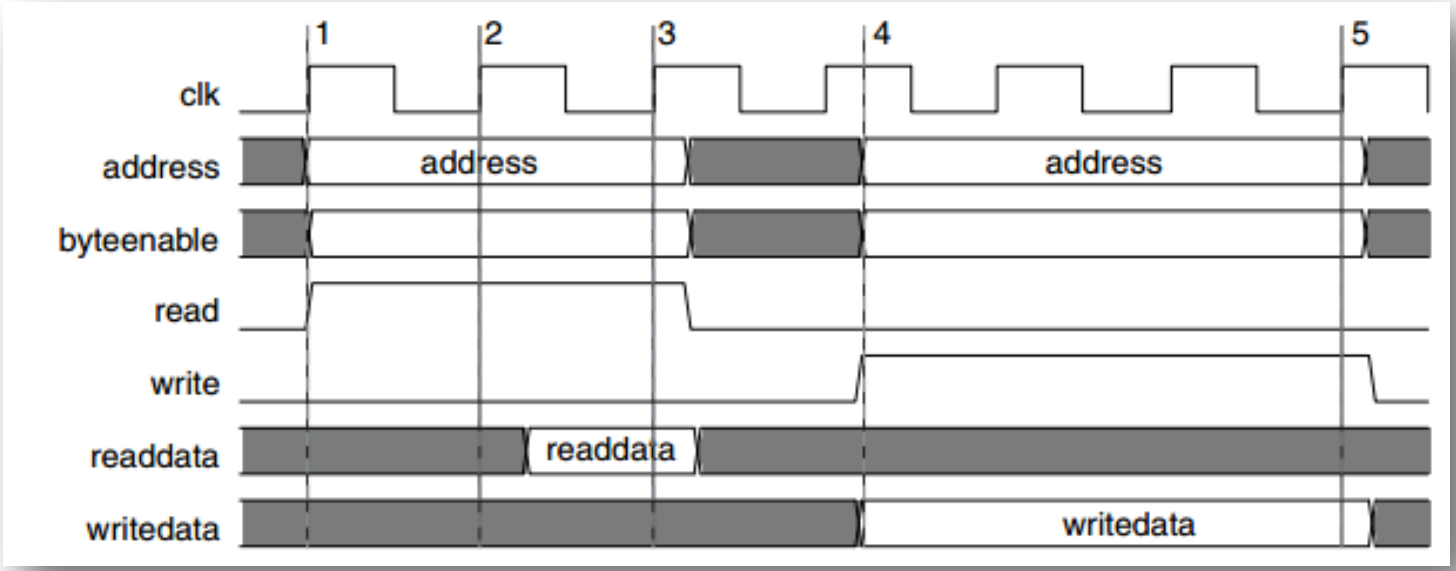


Load-Store Unit - High-Level Block Diagram



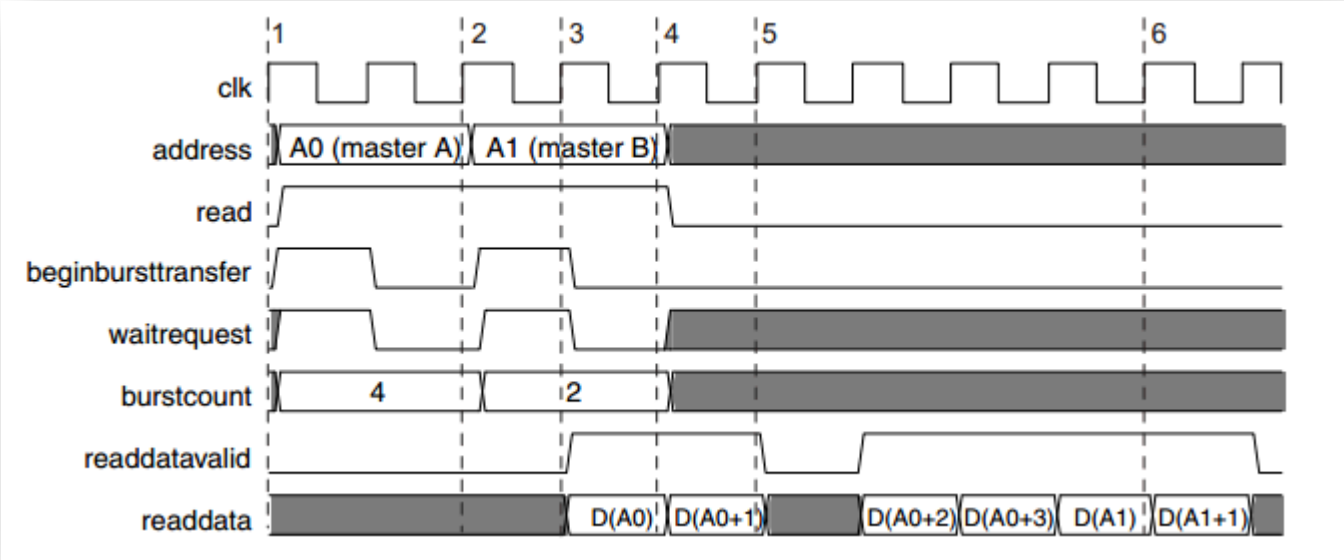
ALTERA
MEASURABLE ADVANTAGE™

Avalon Memory Mapped – Single Request (Read/Write)



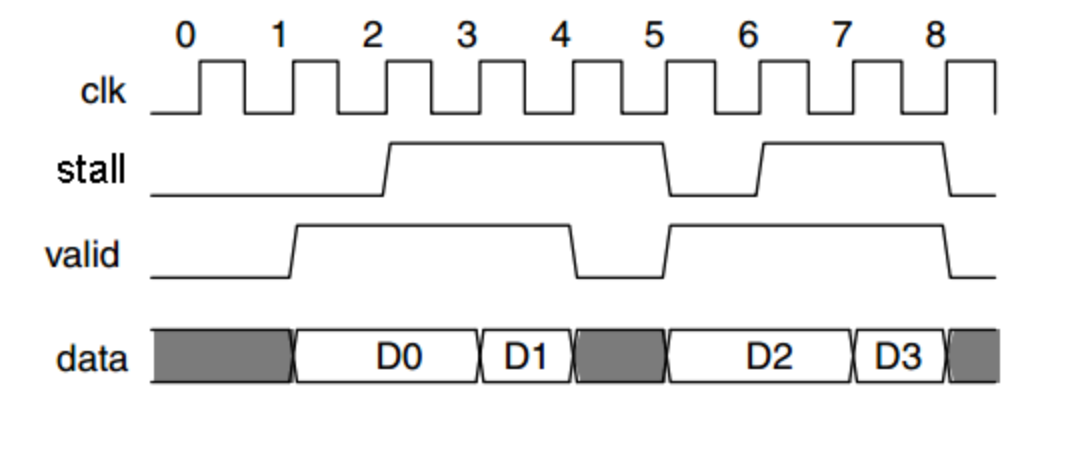
ALTERA
MEASURABLE ADVANTAGE™

Avalon Memory Mapped – Burst Request (Read)

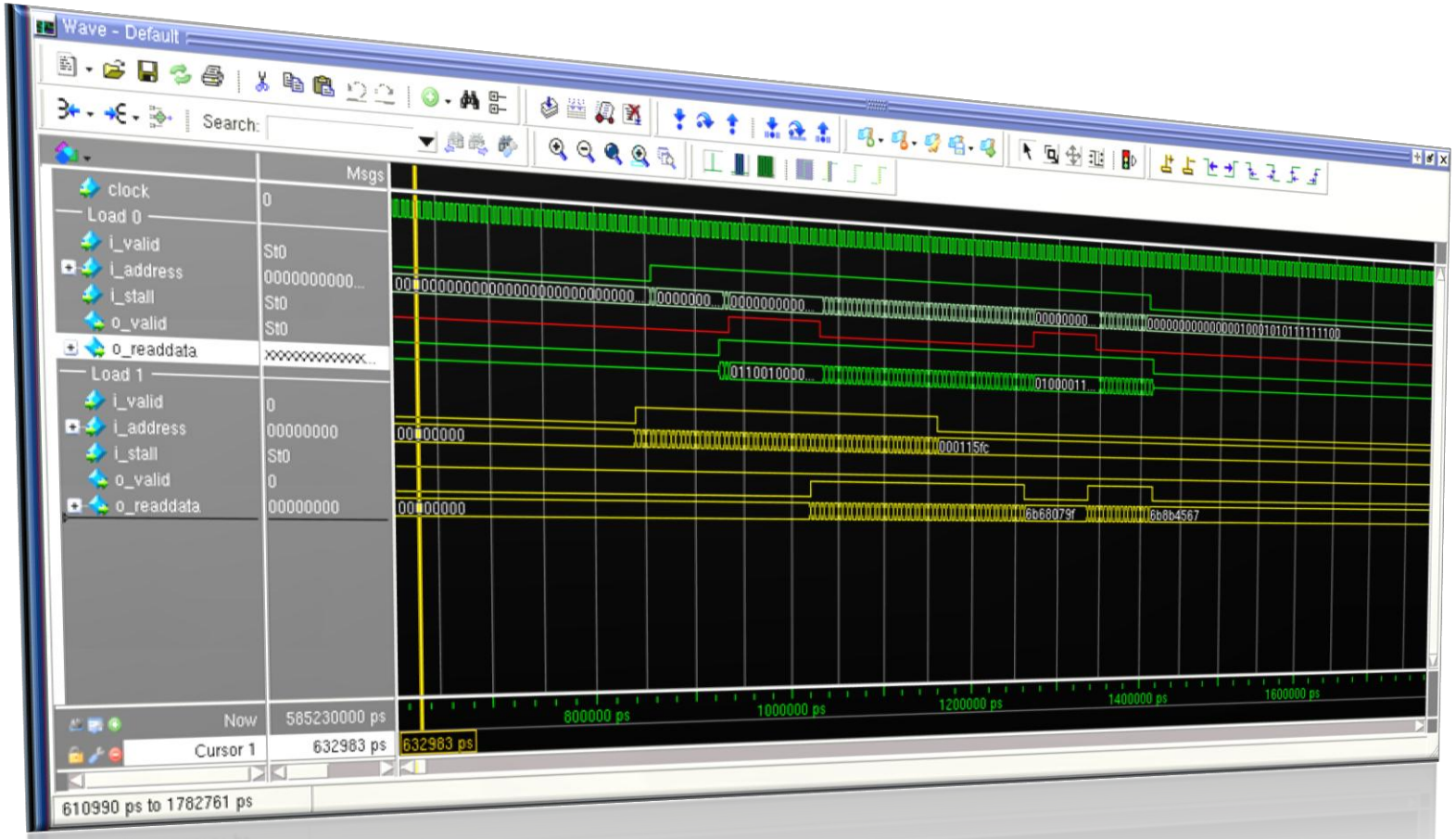


ALTERA
MEASURABLE ADVANTAGE™

Avalon Streaming



ModelSim Example – Load Contention

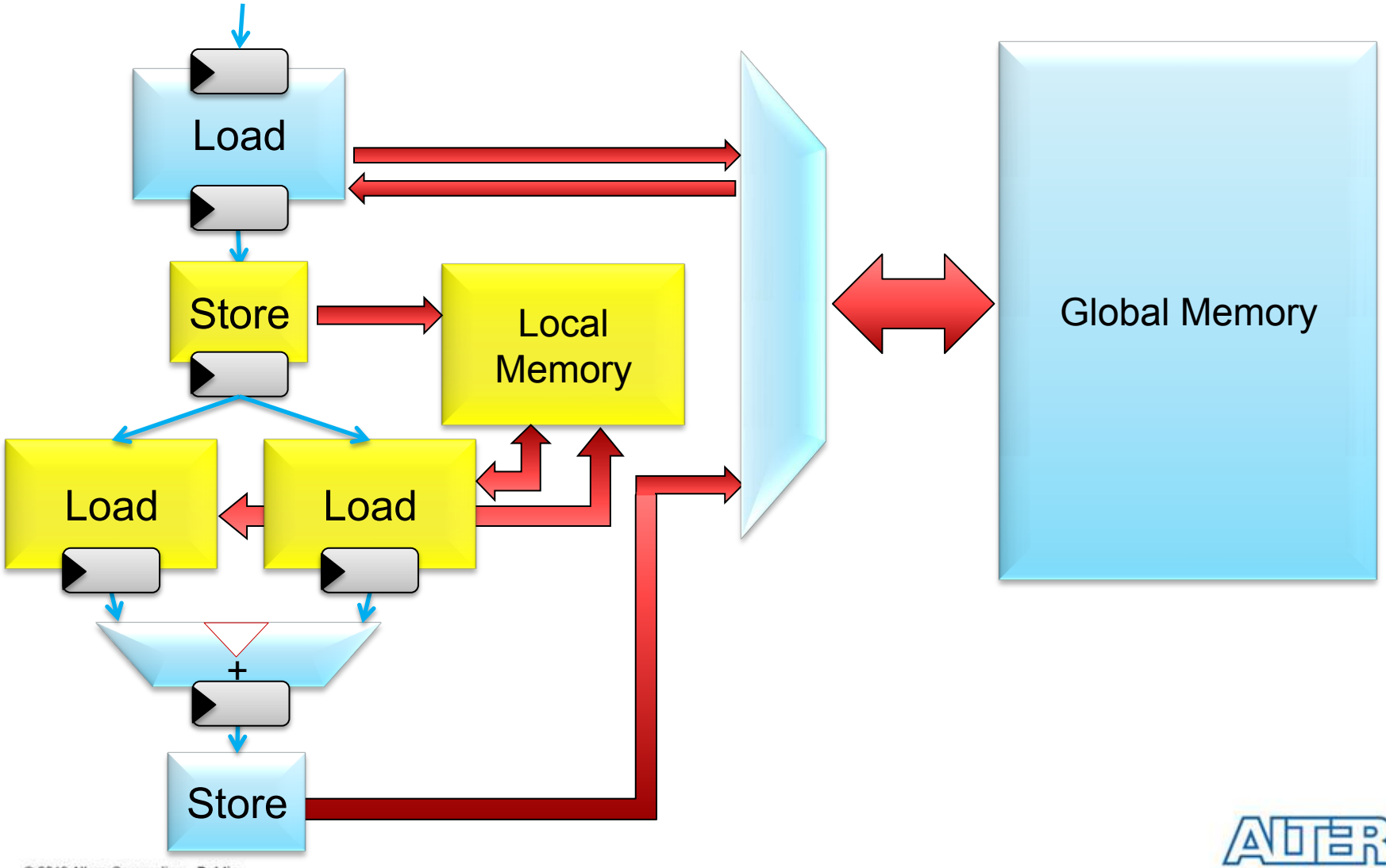


ALTERA
MEASURABLE ADVANTAGE™

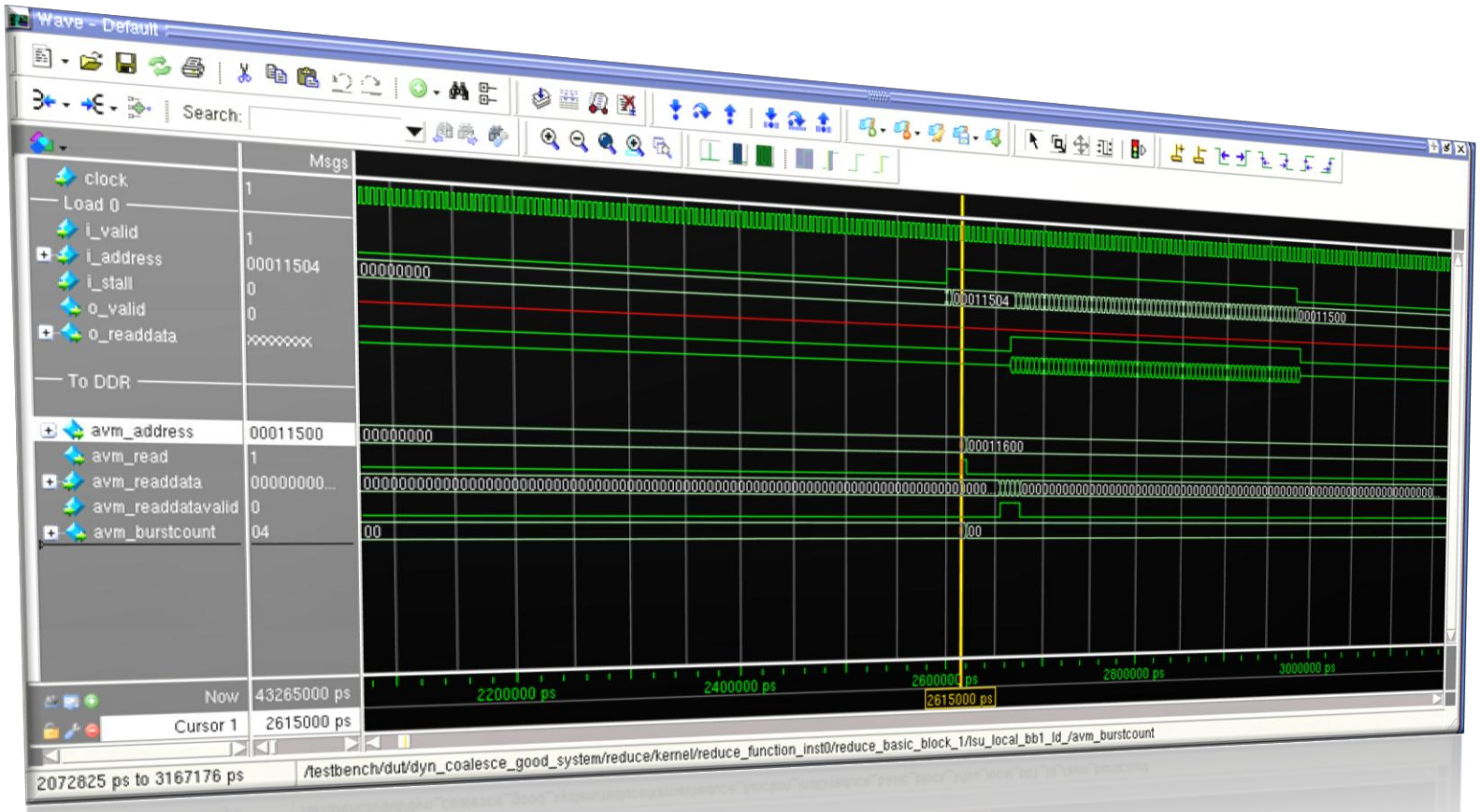
Example 1 – STREAMING access

```
__kernel void
   __attribute__(((max_work_group_size(256)))
   example1(__global int *a, __global int *b)
{
  __local int buffer[256];
  int lid = get_local_id(0);
  int gid = get_global_id(0);
  int size = get_local_size(0);
  buffer[lid] = a[gid];
  barrier();
  b[gid] = buffer[lid] + buffer[size-lid-1];
}
```

FPGA Translation – simplified view with memory, optimized



ModelSim Example – No Load Contention



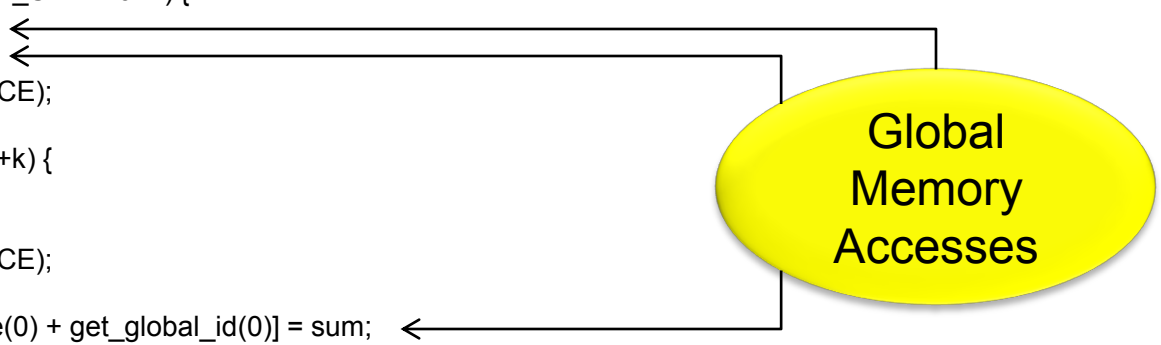
ALTERA
MEASURABLE ADVANTAGE™

Memory Partitioning

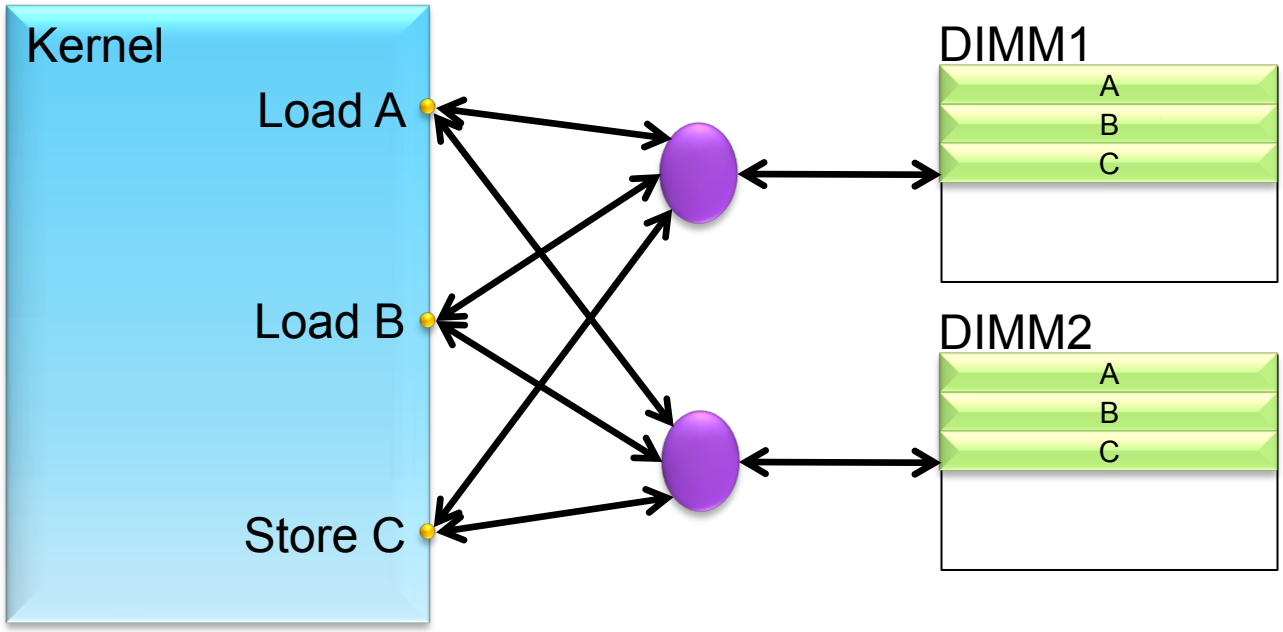
- **In some applications, it is important to place data in appropriate memory locations to fully utilize DDR3 bandwidth for both DIMMs**
  - A good example is a single-precision general-element matrix multiplication

Example 2 – SGEMM (1024x1024)

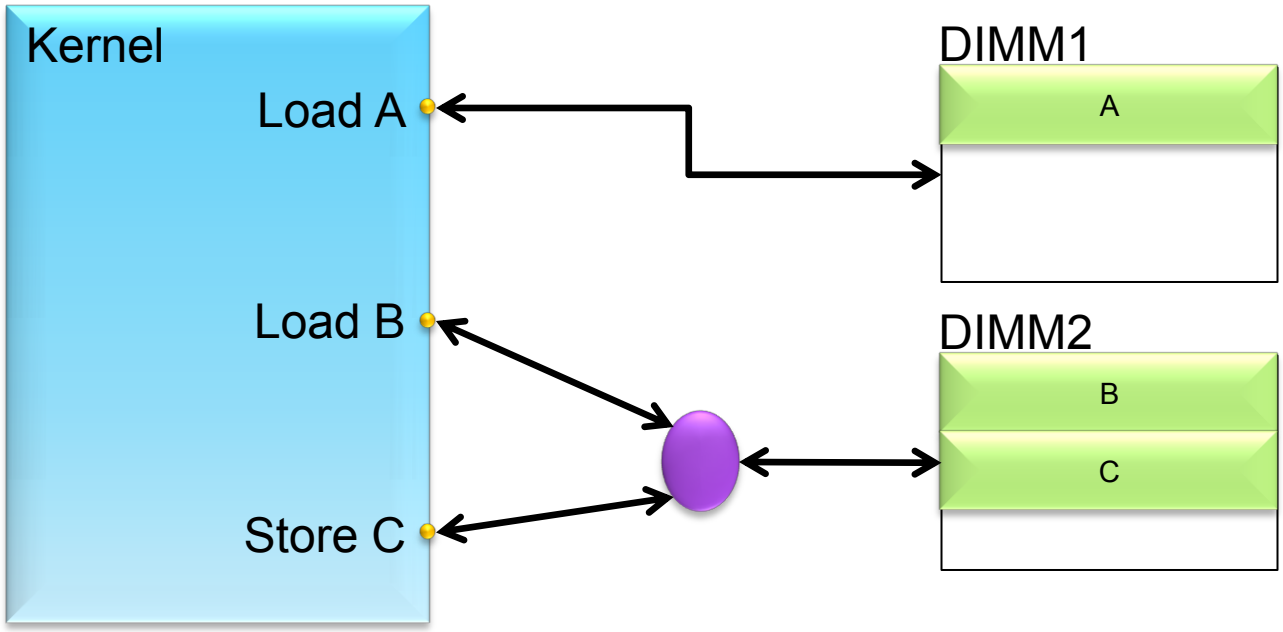
```
#define BLOCK_SIZE 32
#define AS(i, j) A_local [j + i * BLOCK_SIZE ]
#define BS(i, j) B_local [j + i * BLOCK_SIZE ]

__kernel
    void sgemm( __global float *restrict C, __global float *A, __global float *B,
                __local float *restrict A_local, __local float *restrict B_local)
{
   // Block index
   int bx = get_group_id(0);
   int by = get_group_id(1);
   // Thread index
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   float sum = 0.0f;
   for (int i = 1024* BLOCK_SIZE * by, j = BLOCK_SIZE * bx;
         i <= 1024* BLOCK_SIZE * (1+by)  - 1;
         i += BLOCK_SIZE, j += BLOCK_SIZE*1024) {
      AS(ty, tx) = A[i + 1024* ty + tx];
      BS(ty, tx) = B[j + 1024* ty + tx];
      barrier(CLK_LOCAL_MEM_FENCE);

      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += AS(ty, k) * BS(k, tx);
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = sum;
}
```



Example 2 – SGEMM (default)



Example 2 – SGEMM (partitioned)

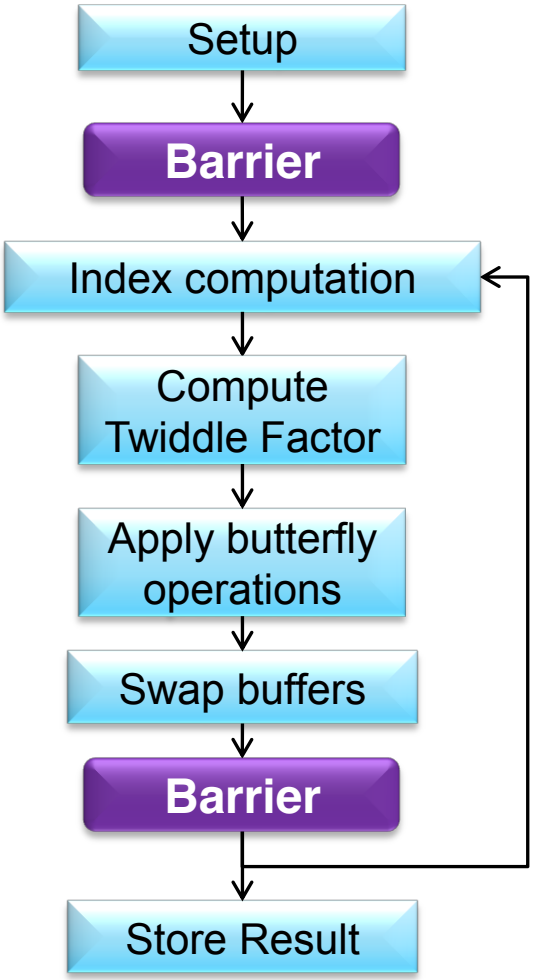


Kernel/Host work Balancing

- It can sometimes be the case that some work can be performed efficiently by the host, that could also be done as efficiently in the kernel (or a little more efficiently)
- However, the cost of implementing the minor improvement in hardware is simply not worth it
- It may be better to off-load some operations to the host, and simply store their result in memory

Example - FFT

```
__kernel void fft(  __global float2 * dataIn, __global float2 * dataOut, const unsigned N, const unsigned log2N)
{
  __local float2 array1[FFT_POINTS];
  __local float2 array2[FFT_POINTS];
  __local float2 *scratch1 = array1;
  __local float2 *scratch2 = array2;
  const unsigned tid = get_local_id(0);
  const unsigned group_id = get_group_id(0);
  const unsigned gid = group_id * FFT_POINTS + tid;

  // Load data - assuming N/2 threads
  scratch1[tid] = dataIn[gid];
  scratch1[tid + FFT_POINTS/2] = dataIn[gid + FFT_POINTS/2];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(unsigned t = 0 ; t < LOG2N; t++)
  {
    float2 c0 = scratch1[tid];
    float2 c1 = scratch1[tid + FFT_POINTS/2];
    unsigned outSubArrayID = tid >> t;
    unsigned outSubArraySize = 0x1 << (t+1);
    unsigned halfOutSubArraySize = outSubArraySize >> 1;
    unsigned posInHalfOutSubArray = tid & (halfOutSubArraySize-1);
    const float TWOPI = 2.0 * M_PI_F;
    theta = -1 * TWOPI * posInHalfOutSubArray / outSubArraySize;
    term2.x = ( c1.x*cos(theta) - c1.y*sin(theta) );
    term2.y = ( c1.y*cos(theta) + c1.x*sin(theta) );
    unsigned index = posInHalfOutSubArray * (FFT_POINTS >> (t+1));
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray] = c0 + term2;
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray + halfOutSubArraySize] = c0 - term2;
    __local float2 *f2tmp; f2tmp = scratch1; scratch1 = scratch2; scratch2 = f2tmp;
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  dataOut[gid] = scratch1[tid];
  dataOut[gid + FFT_POINTS/2] = scratch1[tid + FFT_POINTS/2];
}
```



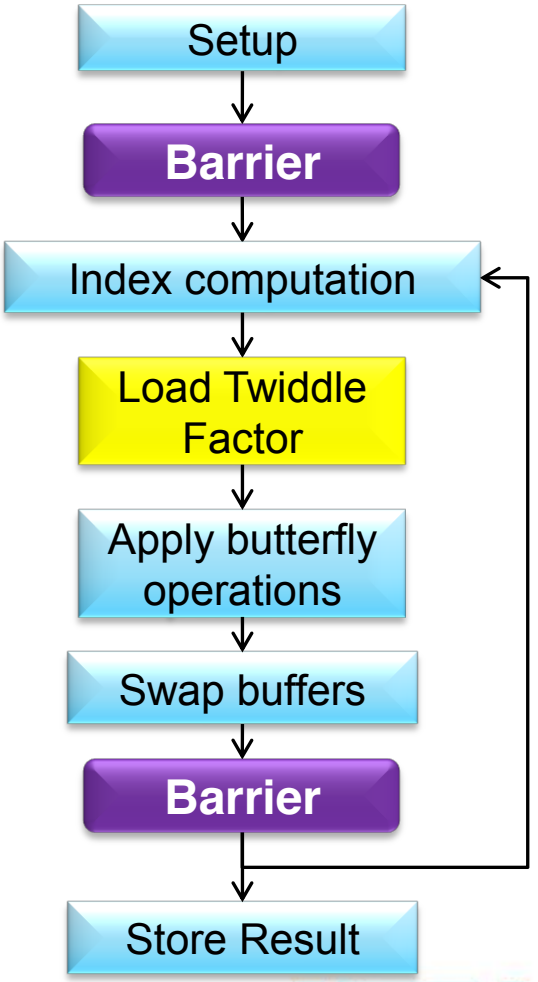
Example - FFT

```
__kernel void fft(  __global float2 * dataIn, __global float2 * dataOut, const unsigned N, const unsigned log2N)
{
  __local float2 array1[FFT_POINTS];
  __local float2 array2[FFT_POINTS];
  __local float2 *scratch1 = array1;
  __local float2 *scratch2 = array2;
  const unsigned tid = get_local_id(0);
  const unsigned group_id = get_group_id(0);
  const unsigned gid = group_id * FFT_POINTS + tid;

  // Load data - assuming N/2 threads
  scratch1[tid] = dataIn[gid];
  scratch1[tid + FFT_POINTS/2] = dataIn[gid + FFT_POINTS/2];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(unsigned t = 0 ; t < LOG2N; t++)
  {
    float2 c0 = scratch1[tid];
    float2 c1 = scratch1[tid + FFT_POINTS/2];
    unsigned outSubArrayID = tid >> t;
    unsigned outSubArraySize = 0x1 << (t+1);
    unsigned halfOutSubArraySize = outSubArraySize >> 1;
    unsigned posInHalfOutSubArray = tid & (halfOutSubArraySize-1);
    const float TWOPI = 2.0 * M_PI_F;
    theta = -1 * TWOPI * posInHalfOutSubArray / outSubArraySize;
    term2.x = ( c1.x*cos(theta) - c1.y*sin(theta) );
    term2.y = ( c1.y*cos(theta) + c1.x*sin(theta) );
    unsigned index = posInHalfOutSubArray * (FFT_POINTS >> (t+1));
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray] = c0 + term2;
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray + halfOutSubArraySize] = c0 - term2;
    __local float2 *f2tmp; f2tmp = scratch1; scratch1 = scratch2; scratch2 = f2tmp;
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  dataOut[gid] = scratch1[tid];
  dataOut[gid + FFT_POINTS/2] = scratch1[tid + FFT_POINTS/2];
}
```

Large computation, parts of which can be pre-computed

Example - FFT

```
__kernel void fft(  __global float2 * dataIn, __global float2 * dataOut, __global float2 * twiddle, const unsigned N, const unsigned log2N)
{
  __local float2 array1[FFT_POINTS];
  __local float2 array2[FFT_POINTS];
  __local float2 *scratch1 = array1;
  __local float2 *scratch2 = array2;
  const unsigned tid = get_local_id(0);
  const unsigned group_id = get_group_id(0);
  const unsigned gid = group_id * FFT_POINTS + tid;

  // Load data - assuming N/2 threads
  scratch1[tid] = dataIn[gid];
  scratch1[tid + FFT_POINTS/2] = dataIn[gid + FFT_POINTS/2];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(unsigned t = 0 ; t < LOG2N; t++)
  {
    float2 c0 = scratch1[tid];
    float2 c1 = scratch1[tid + FFT_POINTS/2];
    unsigned outSubArrayID = tid >> t;
    unsigned outSubArraySize = 0x1 << (t+1);
    unsigned halfOutSubArraySize = outSubArraySize >> 1;
    unsigned posInHalfOutSubArray = tid & (halfOutSubArraySize-1);
    float2 term2 = ((float2)(c1.x*twiddle[index].x - c1.y*twiddle[index].y,
                             c1.y*twiddle[index].x + c1.x*twiddle[index].y);
    unsigned index = posInHalfOutSubArray * (FFT_POINTS >> (t+1));
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray] = c0 + term2;
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray + halfOutSubArraySize] = c0 - term2;
    __local float2 *f2tmp; f2tmp = scratch1; scratch1 = scratch2; scratch2 = f2tmp;
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  dataOut[gid] = scratch1[tid];
  dataOut[gid + FFT_POINTS/2] = scratch1[tid + FFT_POINTS/2];
}
```



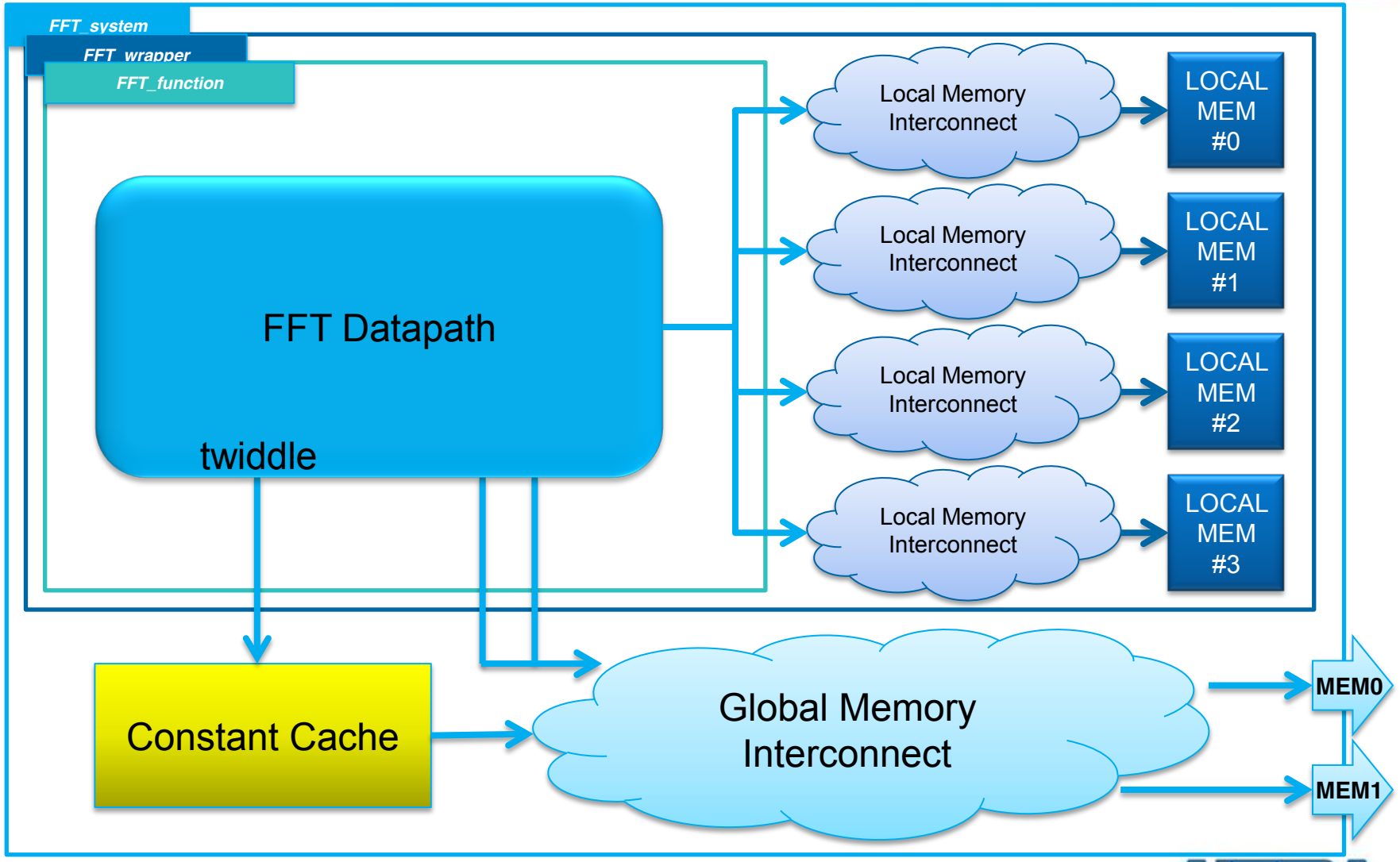
Constant Cache

- **In some applications, for example FFT, some data stored in memory is loaded once and then reused frequently**
  - This could pose a significant bottleneck on performance
- **What can we do?**
  - Copying data to local memory may not be enough, as each work group would have to perform the copy operation
- **Solution**
  - We can direct the OpenCL SDK to create a constant cache that only loads data when it is not present within it, regardless of which workgroup requires the data

Example - FFT

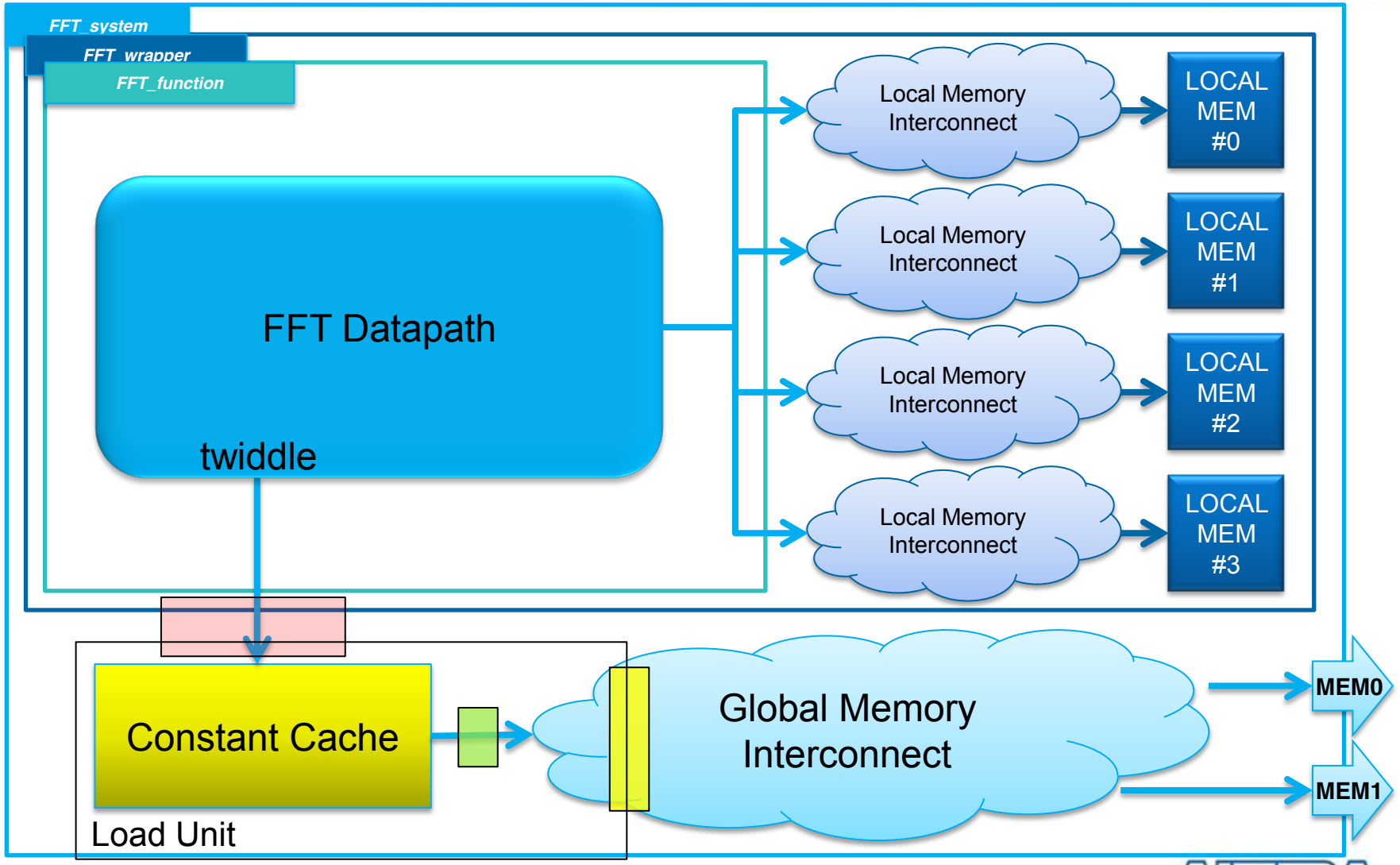
```
__kernel void fft(  __global float2 * dataIn, __global float2 * dataOut, __constant float2 * twiddle, const unsigned N, const unsigned log2N)
{
  __local float2 array1[FFT_POINTS];
  __local float2 array2[FFT_POINTS];

  __local float2 *scratch1 = array1;
  __local float2 *scratch2 = array2;

  const unsigned tid = get_local_id(0);
  const unsigned group_id = get_group_id(0);
  const unsigned gid = group_id * FFT_POINTS + tid;

  // Load data - assuming N/2 threads
  scratch1[tid] = dataIn[gid];
  scratch1[tid + FFT_POINTS/2] = dataIn[gid + FFT_POINTS/2];
  barrier(CLK_LOCAL_MEM_FENCE);

  for(unsigned t = 0 ; t < LOG2N; t++)
  {
    float2 c0 = scratch1[tid];
    float2 c1 = scratch1[tid + FFT_POINTS/2];
    unsigned outSubArrayID = tid >> t;
    unsigned outSubArraySize = 0x1 << (t+1);
    unsigned halfOutSubArraySize = outSubArraySize >> 1;
    unsigned posInHalfOutSubArray = tid & (halfOutSubArraySize-1);
    float2 term2 = ((float2)(c1.x*twiddle[index].x - c1.y*twiddle[index].y, c1.y*twiddle[index].x + c1.x*twiddle[index].y);
    unsigned index = posInHalfOutSubArray * (FFT_POINTS >> (t+1));
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray] = c0 + term2;
    scratch2[outSubArrayID * outSubArraySize + posInHalfOutSubArray + halfOutSubArraySize] = c0 - term2;
    // Swap the scratch arrays
    __local float2 *f2tmp; f2tmp = scratch1; scratch1 = scratch2; scratch2 = f2tmp;
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // Store data - assuming N/2 threads
  // The arrays were swapped, so scratch1 is the valid output data
  dataOut[gid] = scratch1[tid];
  dataOut[gid + FFT_POINTS/2] = scratch1[tid + FFT_POINTS/2];
}
```

FFT Datapath with a Constant Cache



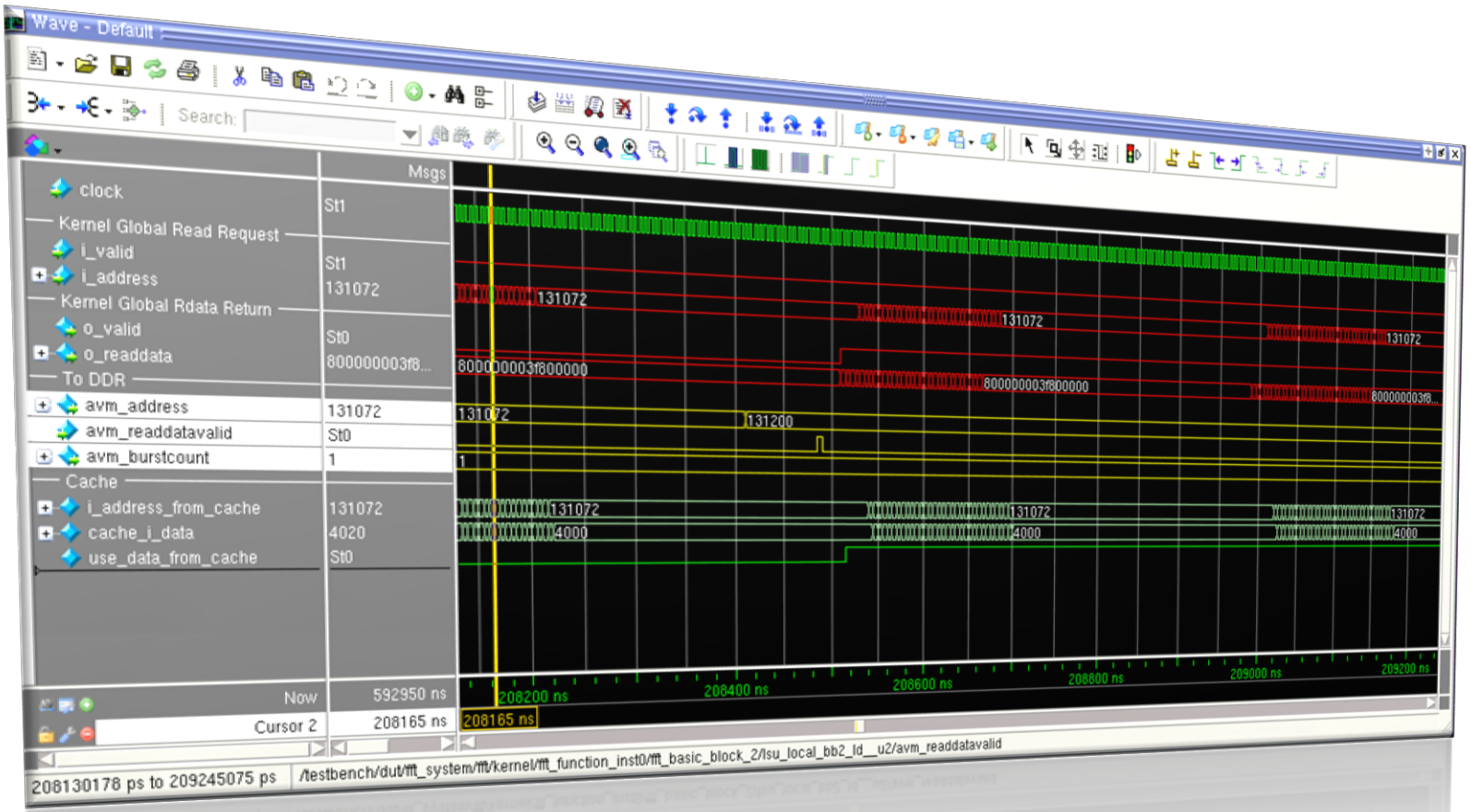
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – FFT Constant Cache



ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – FFT Constant Cache



ALTERA
MEASURABLE ADVANTAGE™

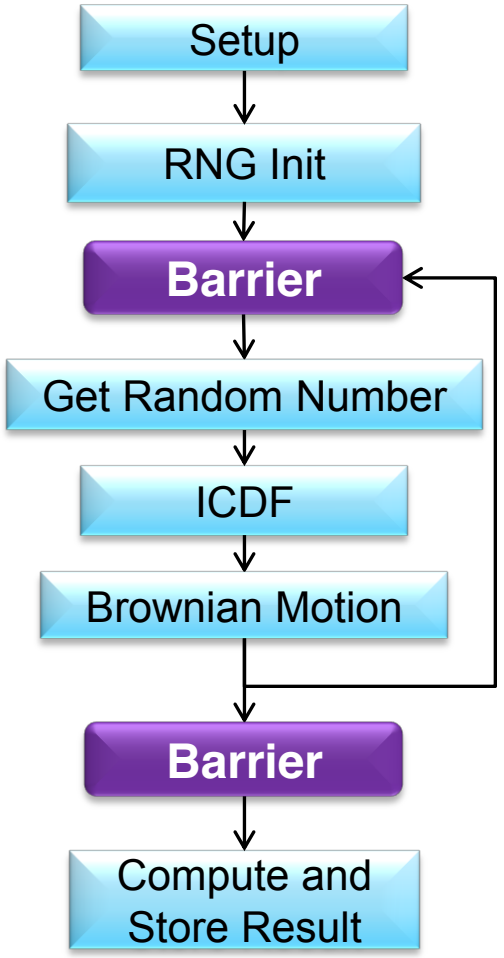
KERNEL OPTIMIZATIONS

Basic Optimizations

- **We can optimize kernels via attributes**
  - required work group size
  - maximum work group size
- **These attributes specify information to the compiler about the size of work groups**
  - How much hardware is used to implement barriers is directly related to work group sizes

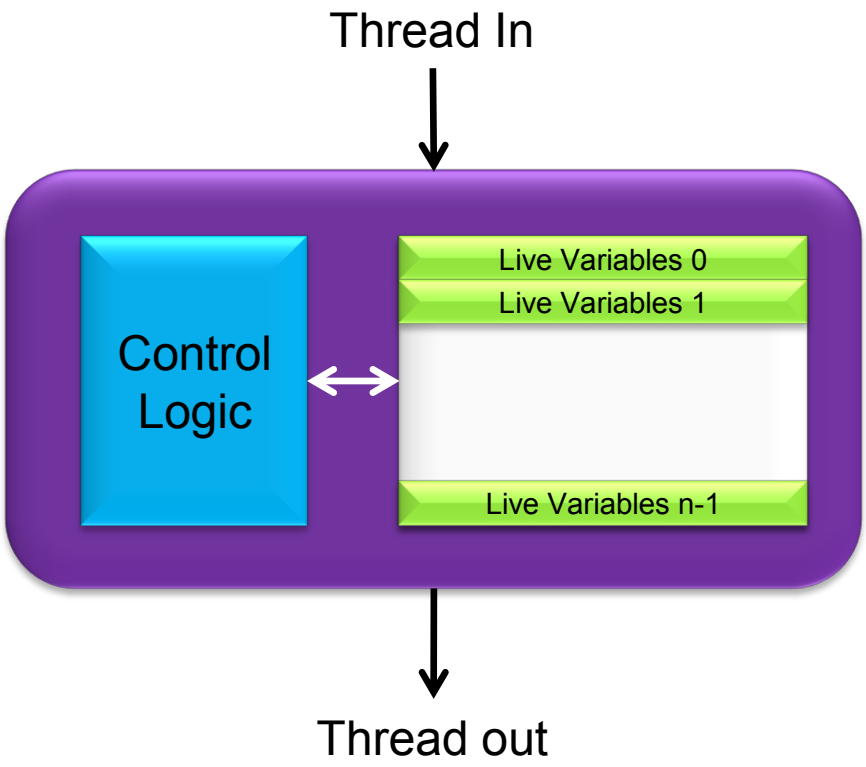
Example Monte-Carlo Black Scholes

```
__kernel void __attribute((reqd_work_group_size(128,1,1)))
        black_scholes( int m, int n, float K1, float K2, float S_0, float K, unsigned int seed, __global float *result)
{
  const float two_to_the_neg_32 = (float)(0.00000000023283064365386962 8f);
  int num_sim, num_pts, i, counter = 0;
  float value, z, diff, sum  = 0.0f, S = S_0;
  int local_id = get_local_id(0), running_offset = get_local_id(0);
  __local unsigned int mt[1024];
  __local float s_partial[128];
  if (local_id == 0)  RNG_init(seed, mt);
  for(num_sim=0;num_sim<m;num_sim++) {
    barrier(CLK_LOCAL_MEM_FENCE);
    unsigned int u = RNG_read_int32(running_offset, mt);
    if (u==0) u++;
    running_offset = (running_offset + get_local_size(0)) & 1023;
    value = ((float)u) * two_to_the_neg_32;
    if (value >= 1.0f) value = 0.999f;
    z = icdf(value);
    S *= exp(K1 + K2*z);
    counter = (counter + 1) % n;
    if (counter == 0) {
      diff = S - K;
      if(diff > ((float)0.0))  sum += diff;
      S = S_0;
    }
  }
  s_partial[local_id] = sum;
  barrier(CLK_LOCAL_MEM_FENCE);
  // Save the result when exiting.
  if (local_id == 0) {
    for (i = 1; i < get_local_size(0); i++) sum = sum + s_partial[i];
    result[get_group_id(0)] = sum;
  }
}
```

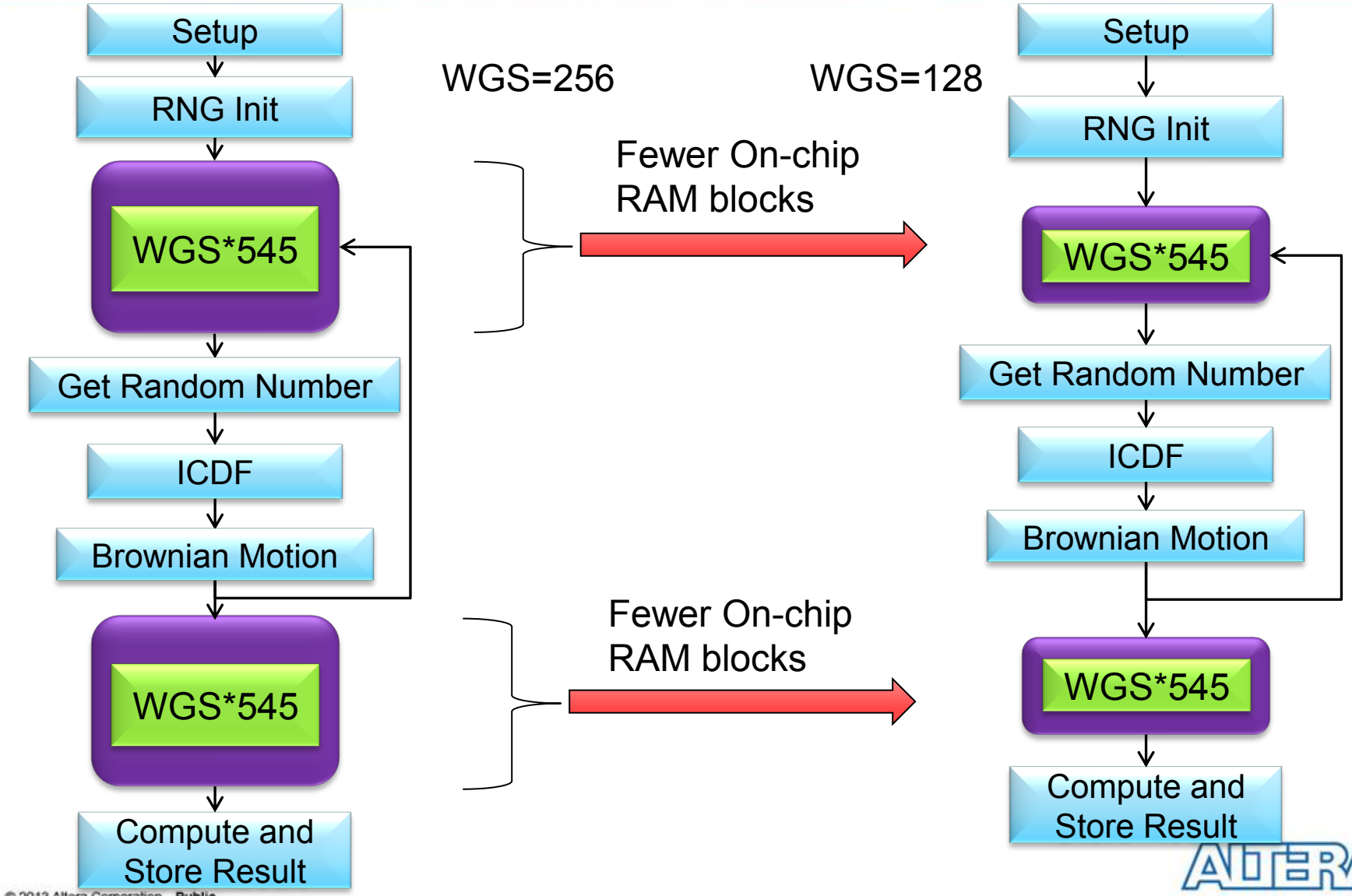


About Barriers

- Barriers synchronize memory operations within a workgroup to prevent hazards
- They ensure that each work item in a workgroup completed memory operations before it prior to allowing another work item in the same workgroup from executing memory operations after the barrier
- This requires that live variables for each thread be stored in a buffer



Reducing Barrier Area

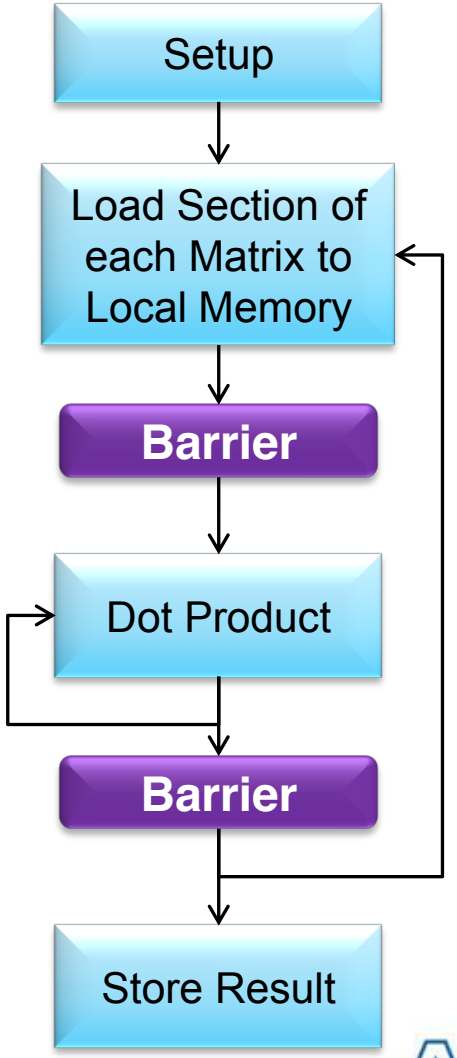


Loop Unrolling

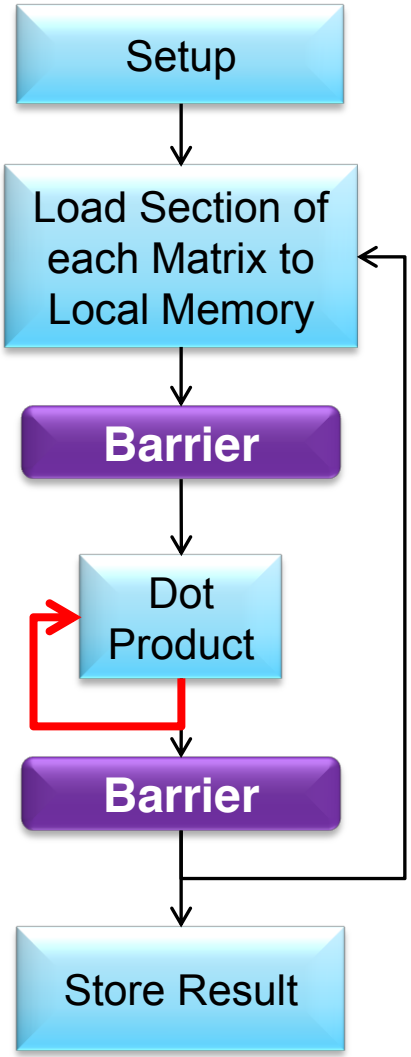
- Loop unrolling can help parallelize computation inside of a kernel
- This optimization is important for applications such as SGEMM, where more operations means more ability to optimize the circuit AND higher throughput

Example – SGEMM unrolling

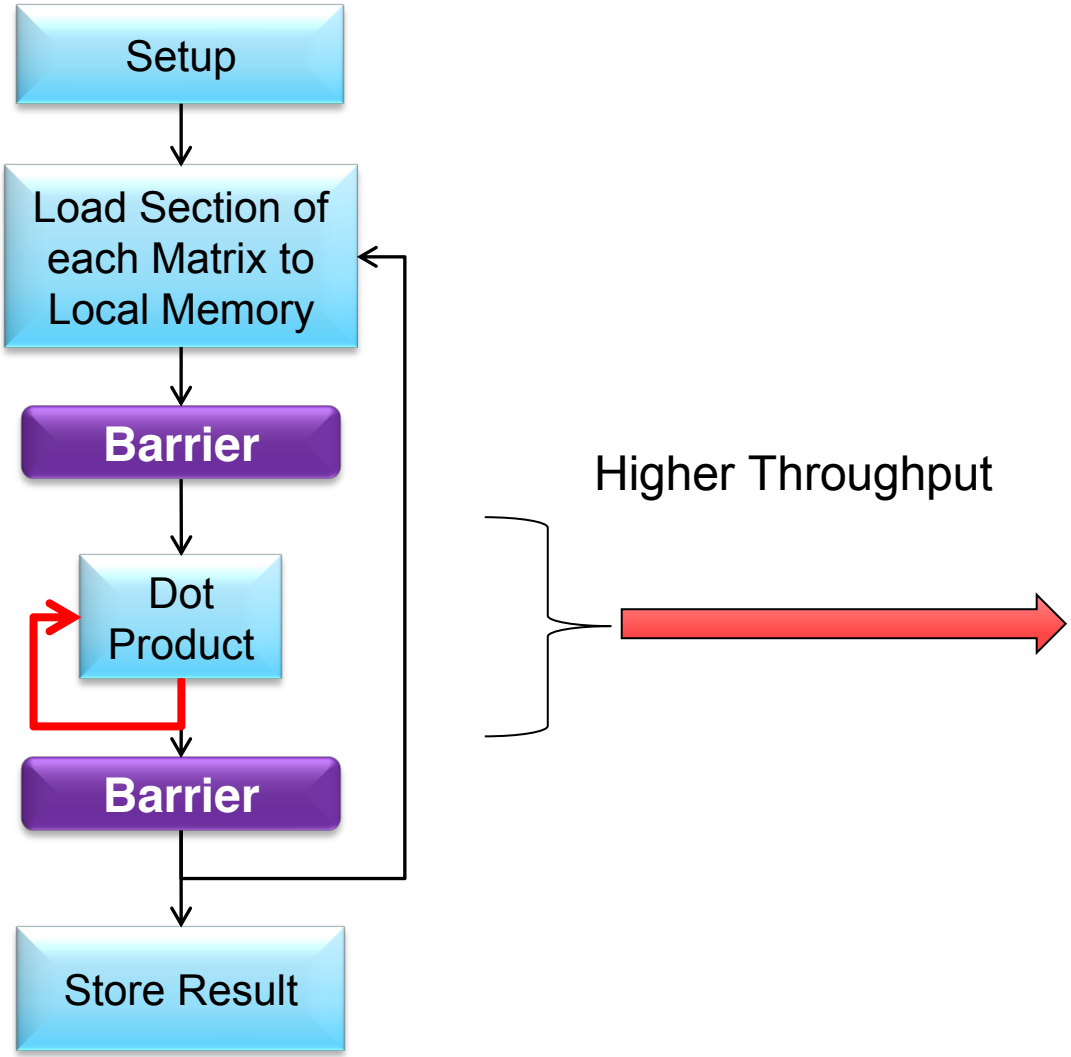
```
#define BLOCK_SIZE 4
#define AS(i, j) A_local [j + i * BLOCK_SIZE ]
#define BS(i, j) B_local [j + i * BLOCK_SIZE ]

__kernel  void sgemm( __global float *restrict C, __global float *A, __global float *B,
                      __local float *restrict A_local, __local float *restrict B_local)
{
   int bx = get_group_id(0);
   int by = get_group_id(1);
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   float sum = 0.0f;
   for (int i = 1024* BLOCK_SIZE * by, j = BLOCK_SIZE * bx;
         i <= 1024* BLOCK_SIZE * (1+by)  - 1;
         i += BLOCK_SIZE, j += BLOCK_SIZE*1024) {
      AS(ty, tx) = A[i + 1024* ty + tx];
      BS(ty, tx) = B[j + 1024* ty + tx];
      barrier(CLK_LOCAL_MEM_FENCE);
      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += AS(ty, k) * BS(k, tx);
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = sum;
}
```



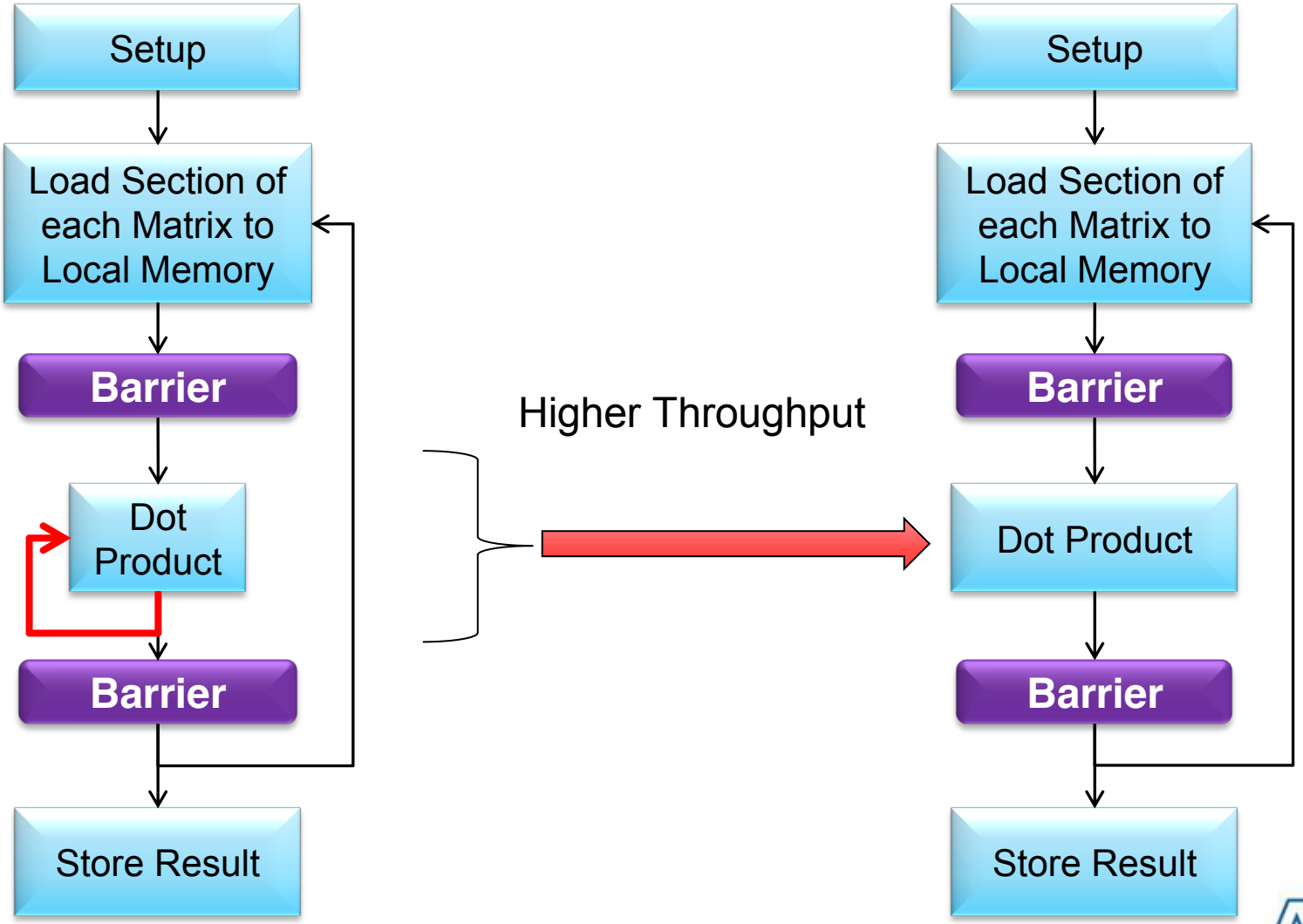
Unrolling SGEMM – Block Diagram



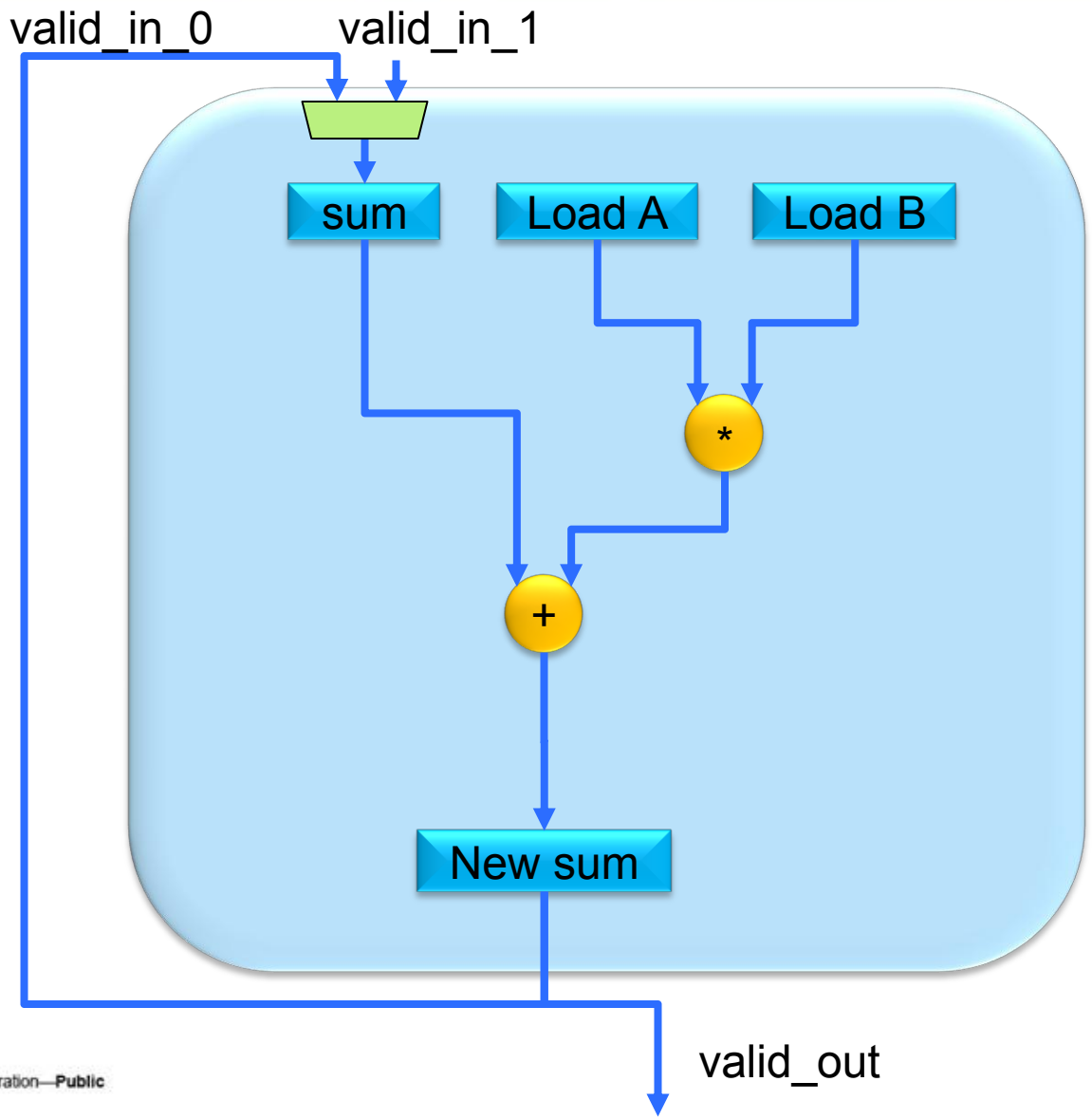
Unrolling SGEMM – Block Diagram



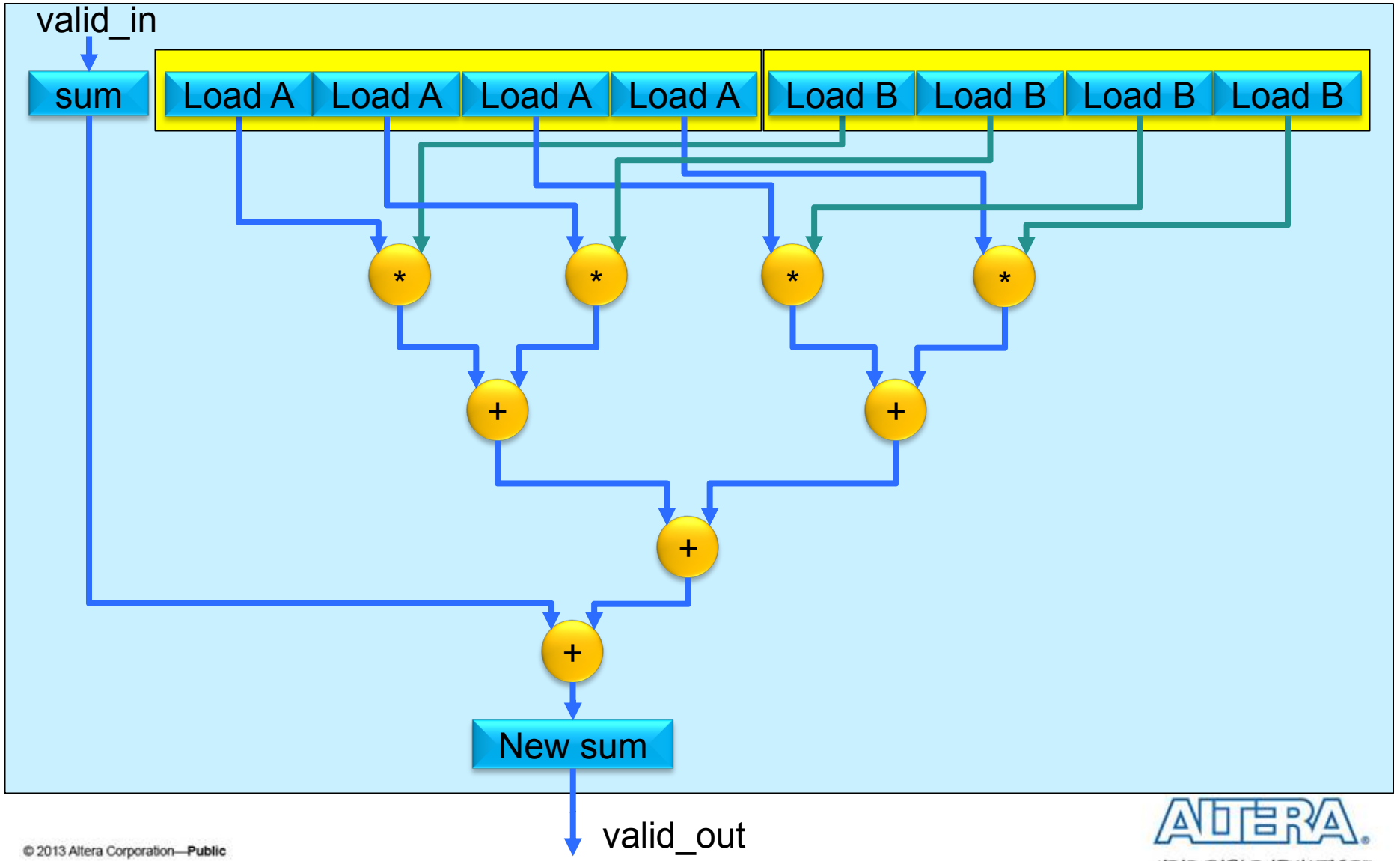
Unrolling SGEMM – Block Diagram



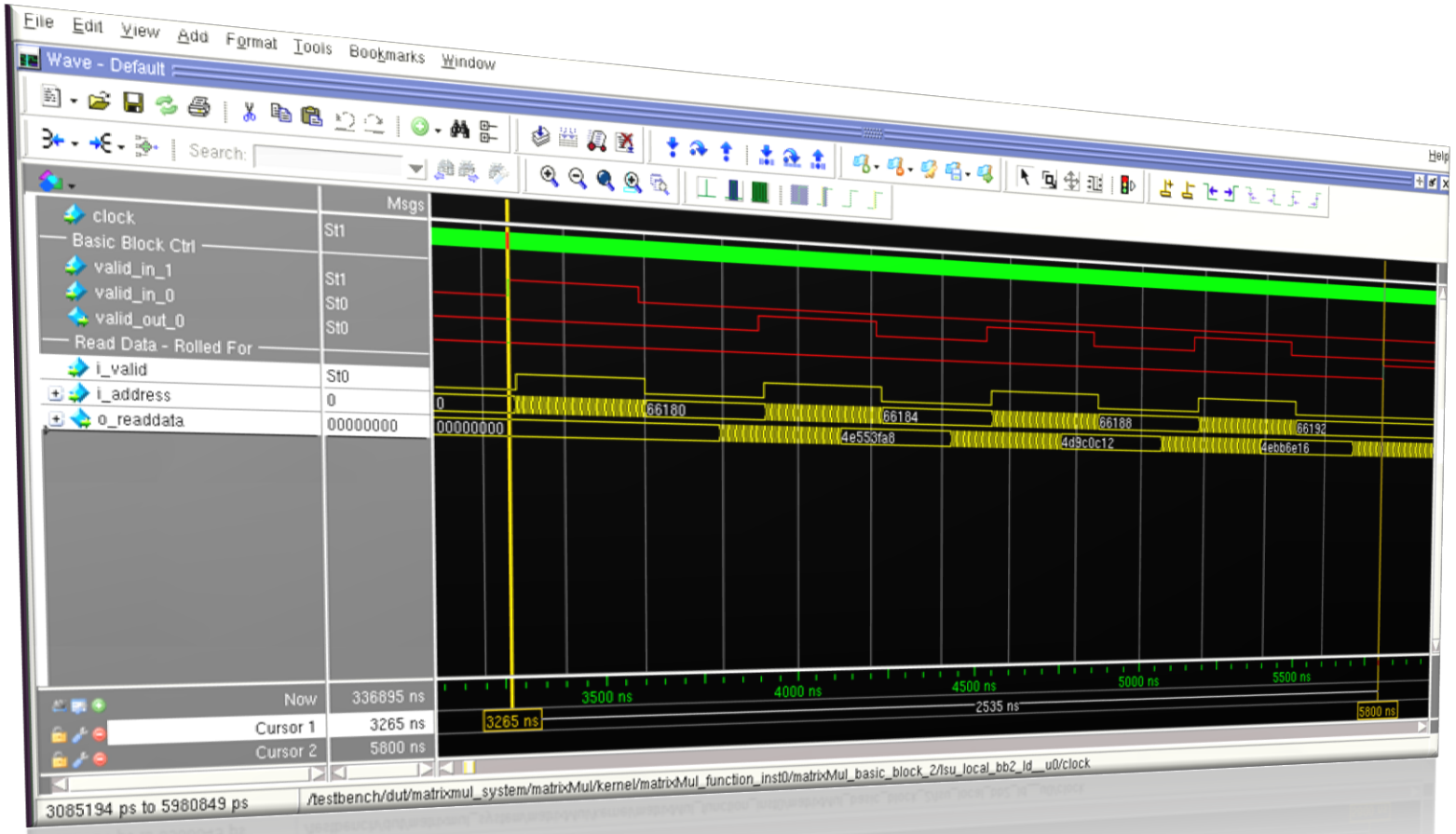
Example – Initial Implementation



Example – Improved Implementation



ModelSim Example – Loop Rolled vs Unrolled



ALTERA
MEASURABLE ADVANTAGE™

Vectorization and Replication

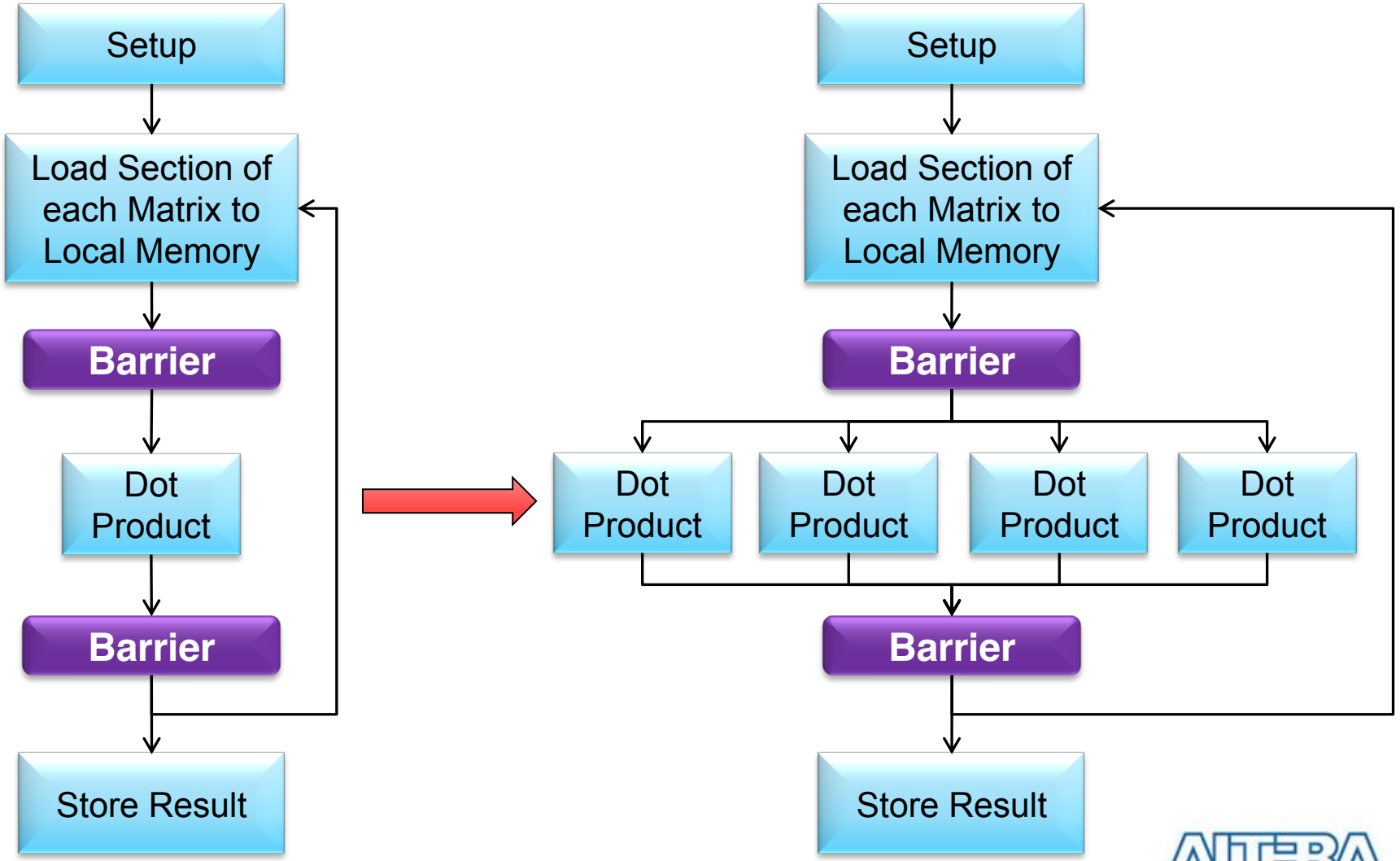
- **In many cases, we may wish to have the kernel replicate its operations several times**
- **There are two ways to do this:**
  - Vectorization
  - Replication
- **Vectorization extends your datapath to use wider operands**
  - Ex. Changes **float** to **float2**
- **Replication creates a copy of a kernel pipeline on an FPGA**

Vectorization Example - SGEMM

```
#define BLOCK_SIZE 32
#define AS(i, j) A_local [j + i * BLOCK_SIZE ]
#define BS(i, j) B_local [j + i * BLOCK_SIZE ]

__kernel  void
    __attribute__((num_vector_lanes(4)))
    __attribute__((reqd_work_group_size(BLOCK_SIZE , BLOCK_SIZE ,1)))
    sgemm( __global float *restrict C, __global float *A, __global float *B,
                    __local float *restrict A_local, __local float *restrict B_local)
{
   // Block index
   int bx = get_group_id(0);
   int by = get_group_id(1);
   // Thread index
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   float sum = 0.0f;
   for (int i = 1024* BLOCK_SIZE * by, j = BLOCK_SIZE * bx;
          i <= 1024* BLOCK_SIZE * (1+by)  - 1;
          i += BLOCK_SIZE, j += BLOCK_SIZE*1024) {
      AS(ty, tx) = A[i + 1024* ty + tx];
      BS(ty, tx) = B[j + 1024* ty + tx];
      barrier(CLK_LOCAL_MEM_FENCE);
      #pragma unroll
      for (int k = 0; k < BLOCK_SIZE; ++k) {
         sum += AS(ty, k) * BS(k, tx);
      }
      barrier(CLK_LOCAL_MEM_FENCE);
   }
   C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = sum;
}
```

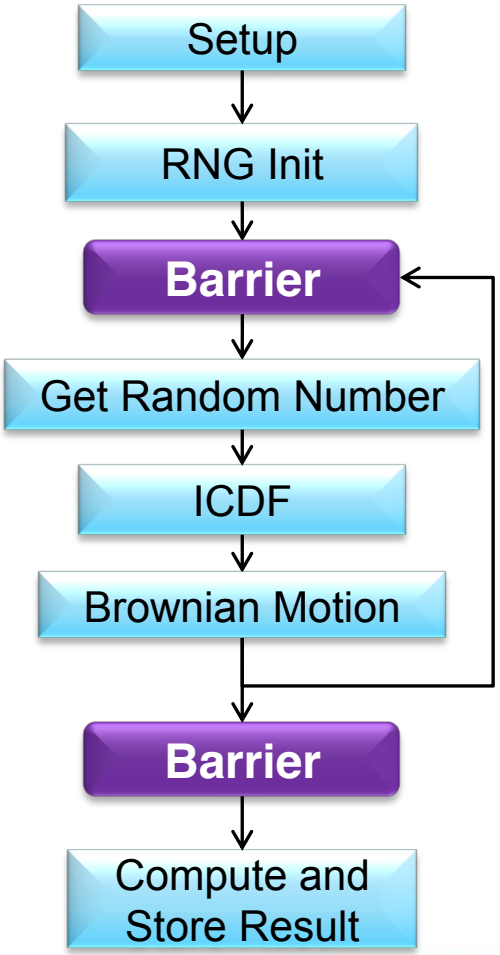
Each thread now performs the work of four!

SGEMM – Vectorization

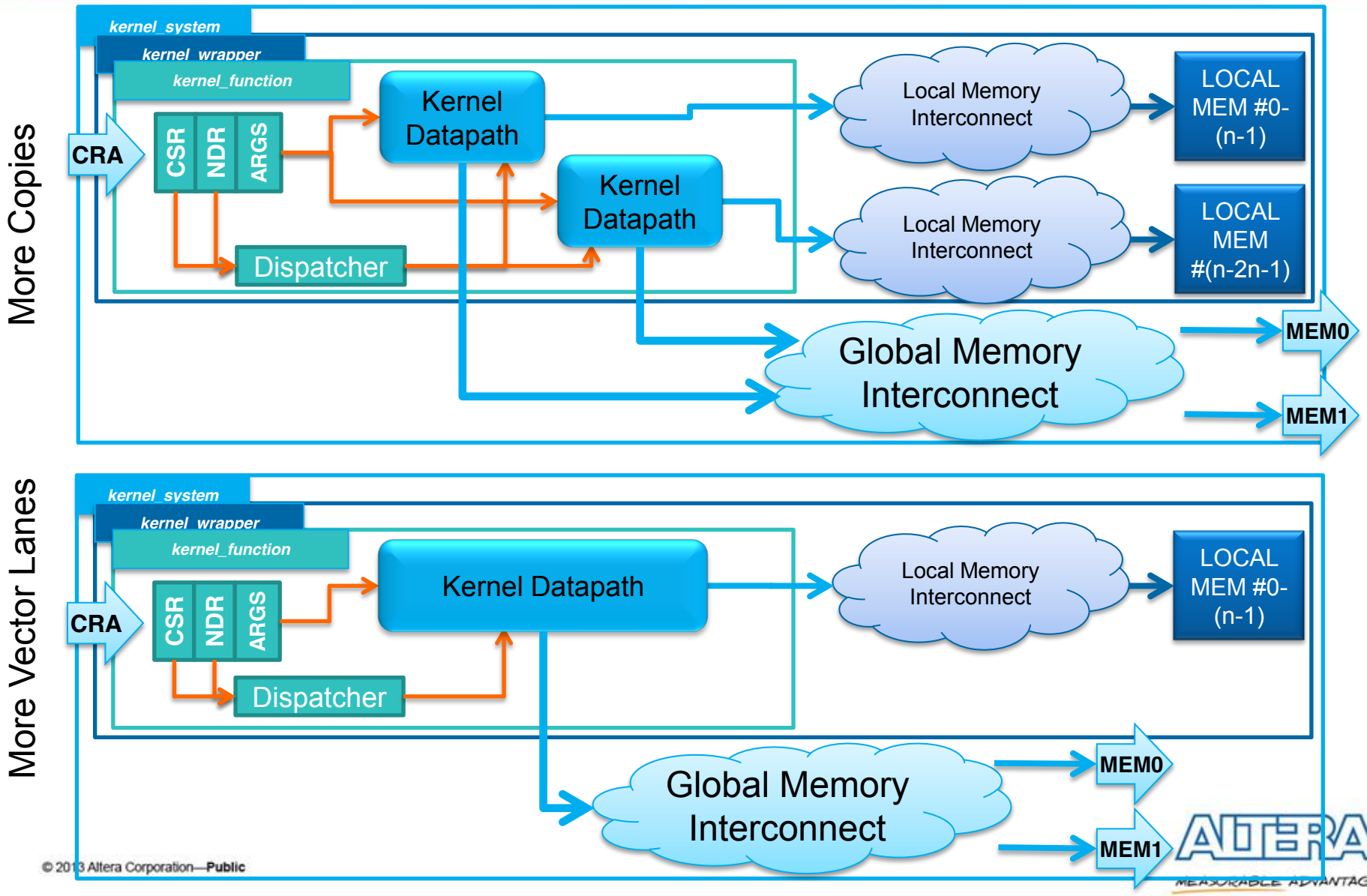


Replication Example - MCBS

```
__kernel void __attribute((reqd_work_group_size(128,1,1)))
             __attribute((num_copies(4)))
             black_scholes( int m, int n, float K1, float K2, float S_0, float K, unsigned int seed, __global float *result)
{
  const float two_to_the_neg_32 = (float)(0.00000000023283064365386962 8f);
  int num_sim, num_pts, i, counter = 0;
  float value, z, diff, sum  = 0.0f, S = S_0;
  int local_id = get_local_id(0), running_offset = get_local_id(0);
  __local unsigned int mt[1024];
  __local float s_partial[128];
  if (local_id == 0)  RNG_init(seed, mt);
  for(num_sim=0;num_sim<m;num_sim++) {
    barrier(CLK_LOCAL_MEM_FENCE);
    unsigned int u = RNG_read_int32(running_offset, mt);
    if (u==0) u++;
    running_offset = (running_offset + get_local_size(0)) & 1023;
    value = ((float)u) * two_to_the_neg_32;
    if (value >= 1.0f) value = 0.999f;
    z = icdf(value);
    S *= exp(K1 + K2*z);
    counter = (counter + 1) % n;
    if (counter == 0) {
      diff = S - K;
      if(diff > ((float)0.0))  sum += diff;
      S = S_0;
    }
  }
  s_partial[local_id] = sum;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (local_id == 0) {
    for (i = 1; i < get_local_size(0); i++) sum = sum + s_partial[i];
    result[get_group_id(0)] = sum;
  }
}
```



Vectorization vs. Replication

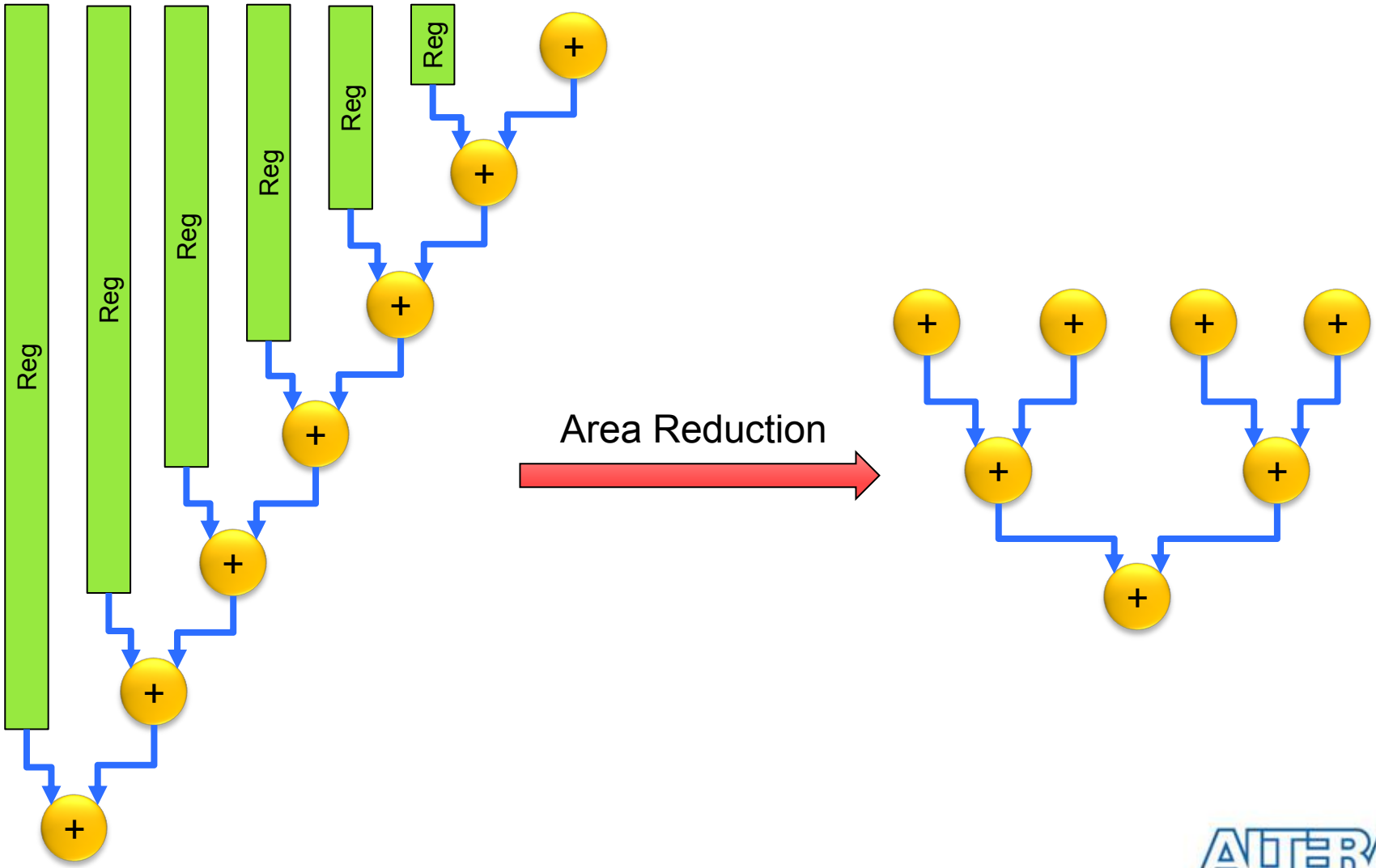


FLOATING POINT OPTIMIZATIONS

Tree-Balancing

- **Floating-point operations are generally not reordered**
  - This is because rounding after each operation affects the outcome
    - ie. ((a+b) + c) != (a+(b+c))
- **This usually creates an imbalance in a pipeline, which costs area**
- **We can remedy the problem by telling the compiler that it is acceptable to balance operations**
  - For example, create a tree of floating-point additions in SGEMM, rather than a chain
  - Use **--fp-relaxed=true** flag in OpenCL SDK

Example – FP balancing



Why is FP large?

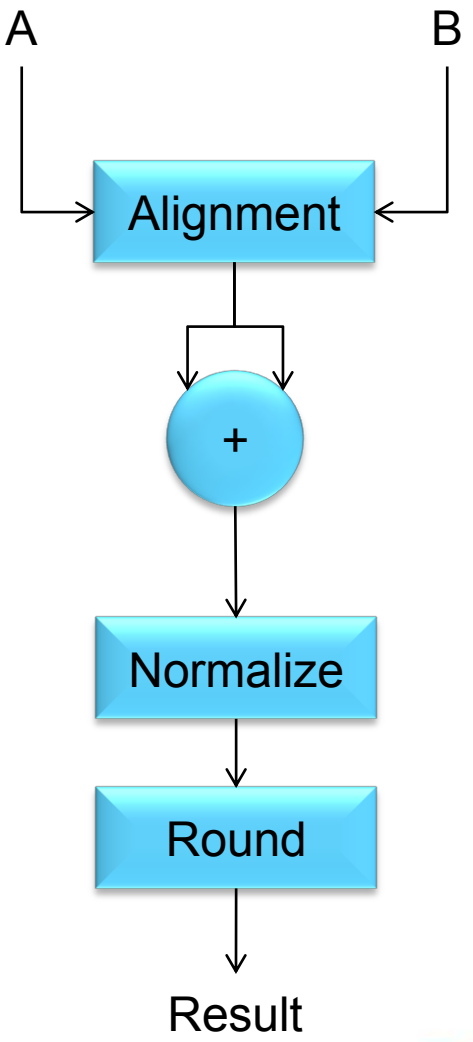
- **Breaking up an number into exponent and mantissa requires pre- and post-processing**

- **Some operations are easier than others**
  - Multiplication requires the addition of exponents, but mantissas are multiplied as though they were integers
  - Addition requires barrel shifters to align inputs such that the mantissas are added when exponents are equal

- **Handle corner cases**
  - +/- Infinity
  - NaN (not-a-number)
  - Denormalized numbers (exponent = 0)

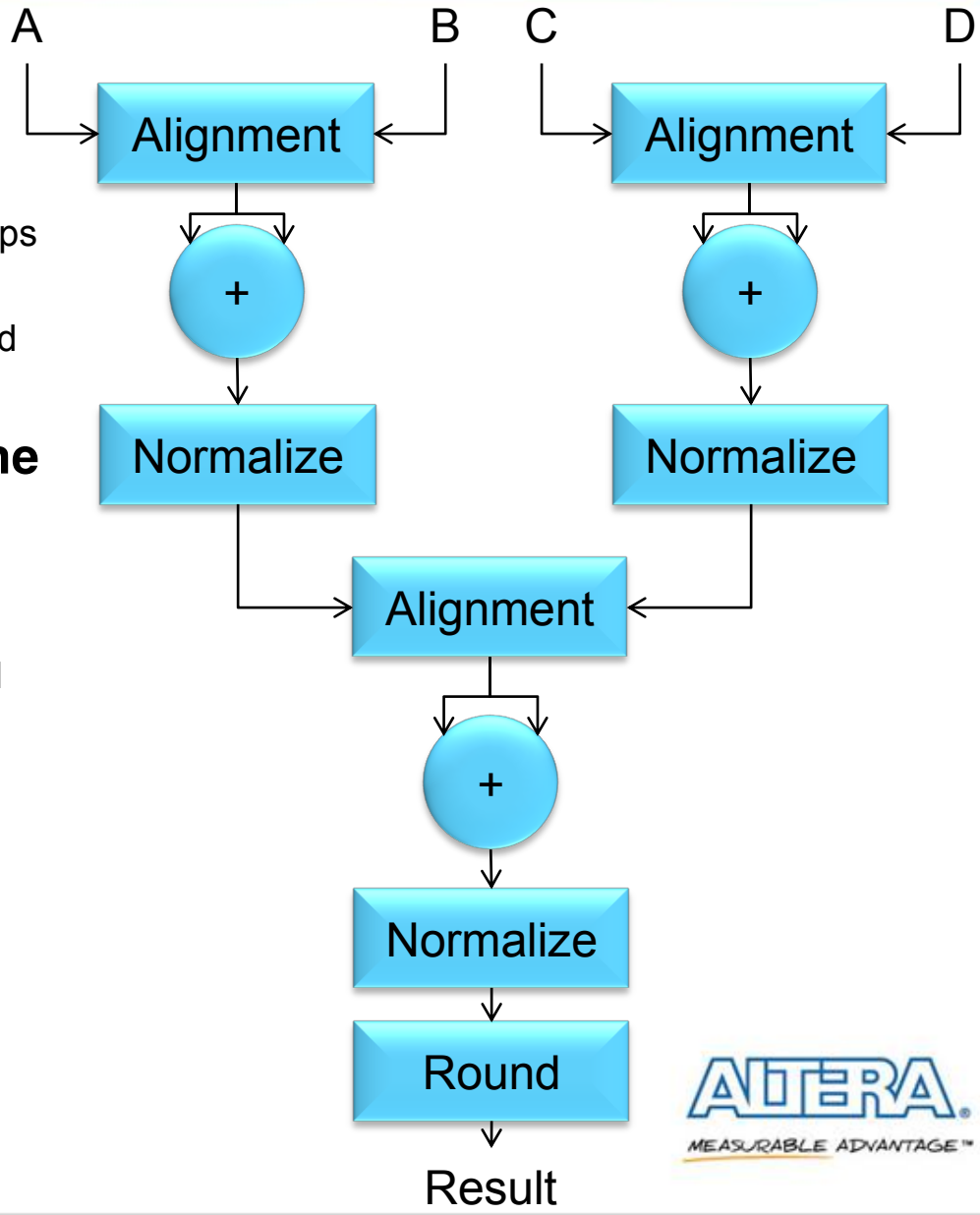
Hardware circuit for FP adder

- **Comprises**
  - Alignment (100 ALMs)
  - Operation (21 ALMs)
  - Normalization (81 ALMs)
  - Rounding (50 ALMs)
- **Normalization and rounding together occupy half of the circuit area**



What can we do?

- **In a chain of operations it is possible to reduce the circuit area**
  - Extending the size of the mantissa helps preserve the precision, but costs less than keeping rounding modules around
- **Rounding can be applied at the end of a chain**
- **Remove support for NaN and Inf**
  - Reduces the size of normalization and alignment logic
- **To enable FPC in Altera's OpenCL SDK**
  - --opt-arg -fpc=true



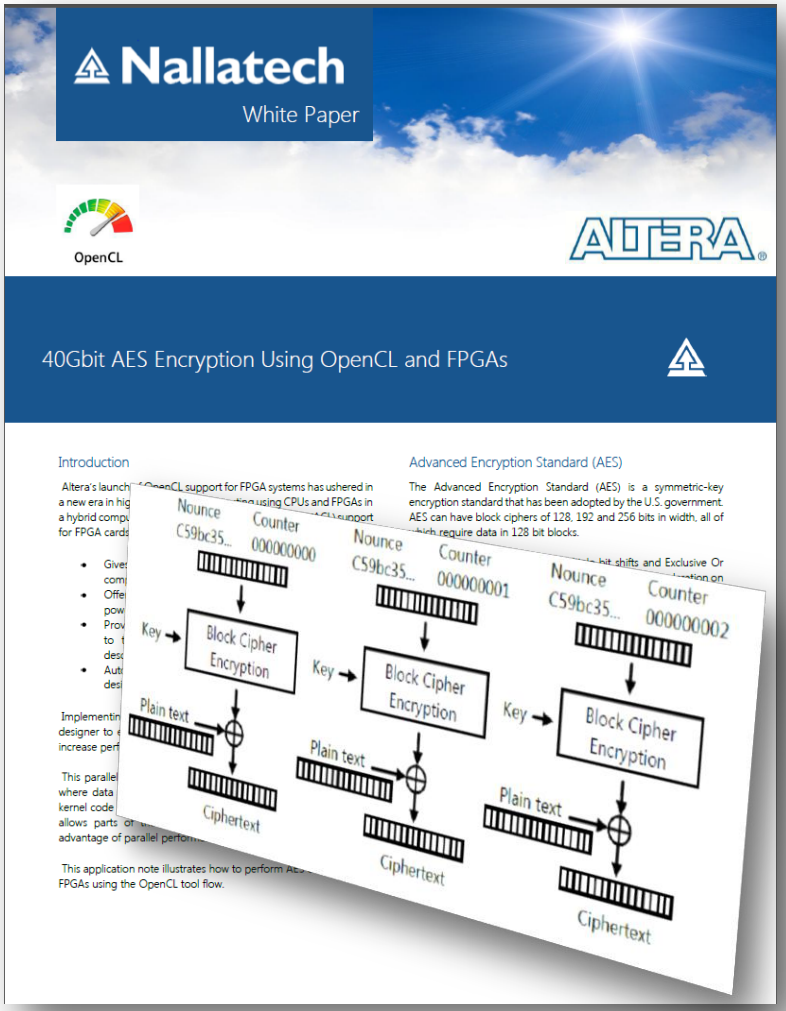
CASE STUDIES

AES Encryption

- **Counter (CTR) based encryption/decryption**
  - 256-bit key
- **Advantage FPGA**
  - Integer arithmetic
  - Coarse grain bit operations
  - Complex decision making
- **Results**

Platform	Throughput (GB/s)
E5503 Xeon Processor	0.01 (single core)
AMD Radeon HD 7970	0.33
PCIe385 A7 Accelerator	5.20

42% utilization (2 kernels)
•Power conservation
•Fill up for even higher performance



ALTERA
MEASURABLE ADVANTAGE™

Multi-Asset Barrier Option Pricing

- **Monte-Carlo simulation**
  - Heston Model

$$dS_t = \mu S_t dt + \sqrt{v_t} S_t dW_t^S$$
$$dv_t = \kappa(\theta - v_t)dt + \xi\sqrt{v_t} dW_t^v$$

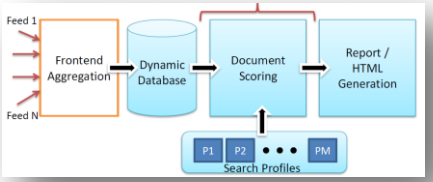
  - ND Range
    - Assets x Paths (64x1000000)
- **Advantage FPGA**
  - Complex Control Flow
- **Results**

Platform	Power (W)	Performance (Msims/s)	Msims/W
W3690 Xeon Processor	130	32	0.25
nVidia Tesla C2075	225	63	0.28
PCIe385 D5 Accelerator	23	170	7.40

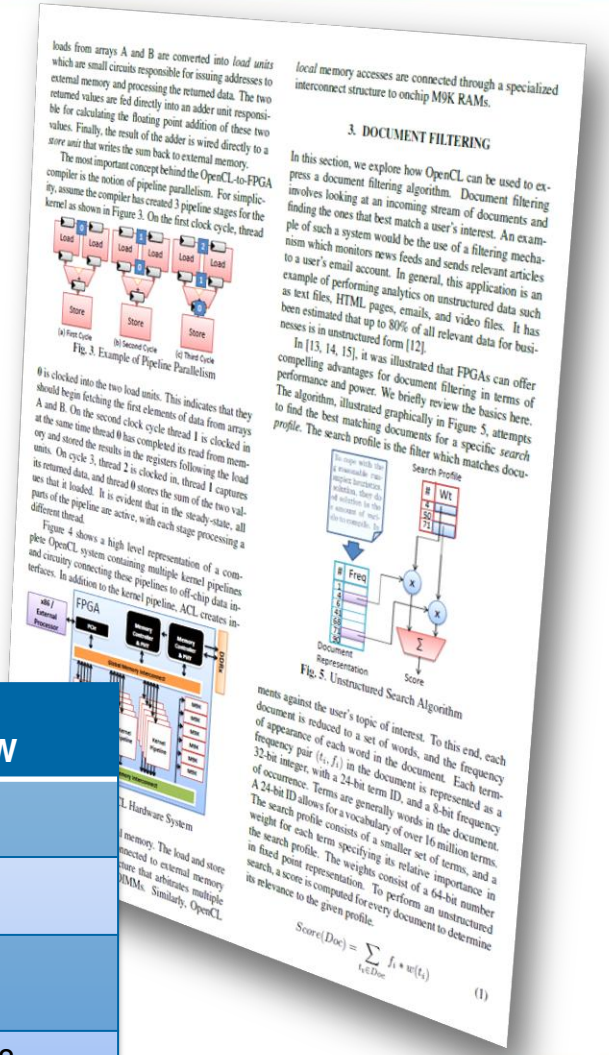
ALTERA
MEASURABLE ADVANTAGE™

Document Filtering

- **Unstructured data analytics**
  - Bloom Filter



- **Advantage FPGA**
  - Integer Arithmetic
  - Flexible Memory Configuration
- **Results**



3. DOCUMENT FILTERING

In this section, we explore how OpenCL can be used to express a document filtering algorithm. Document filtering involves looking at an incoming stream of documents and finding the ones that best match a user's interest. An example of such a system would be the use of a filtering mechanism which monitors news feeds and sends relevant articles to a user's email account. In general, this application is an example of performing analytics on unstructured data such as text files, HTML pages, emails, and video files. It has been estimated that up to 80% of all relevant data for businesses is in unstructured form [12].

In [13, 14, 15], it was illustrated that FPGAs can offer compelling advantages for document filtering in terms of performance and power. We briefly review the basics here. The algorithm, illustrated graphically in Figure 5, attempts to find the best matching documents for a specific *search profile*. The search profile is the filter which matches documents against the user's topic of interest. To this end, each document is reduced to a set of words, and the frequency of appearance of each word in the document. Each term-frequency pair (t_i, f_i) in the document is represented as a 32-bit integer, with a 24-bit term ID, and a 8-bit frequency of occurrence. Terms are generally words in the document. A 24-bit ID allows for a vocabulary of over 16 million terms, and a The search profile consists of a smaller set of terms, and a weight for each term specifying its relative importance in the search profile. The weights consist of a 64-bit number in fixed point representation. To perform an unstructured search, a score is computed for every document to determine its relevance to the given profile.

$$Score(Doc) = \sum_{t_i \in Doc} f_i * w(t_i) \quad (1)$$

Fig. 3. Example of Pipeline Parallelism

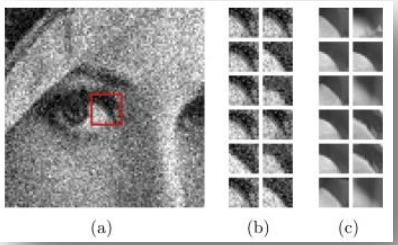
Fig. 5. Unstructured Search Algorithm

Platform	Power (W)	Performance (MTs)	MTs/W
W3690 Xeon Processor	130	2070	15.92
nVidia Tesla C2075	215	3240	15.07
DE4 Stratix IV-530 Accelerator	21	1755	83.57
PCIe385 A7 Accelerator	25	3602	144.08

ALTERA
MEASURABLE ADVANTAGE™

Fractal Video Compression

- **Best matching codebook**
  - Correlation with SAD



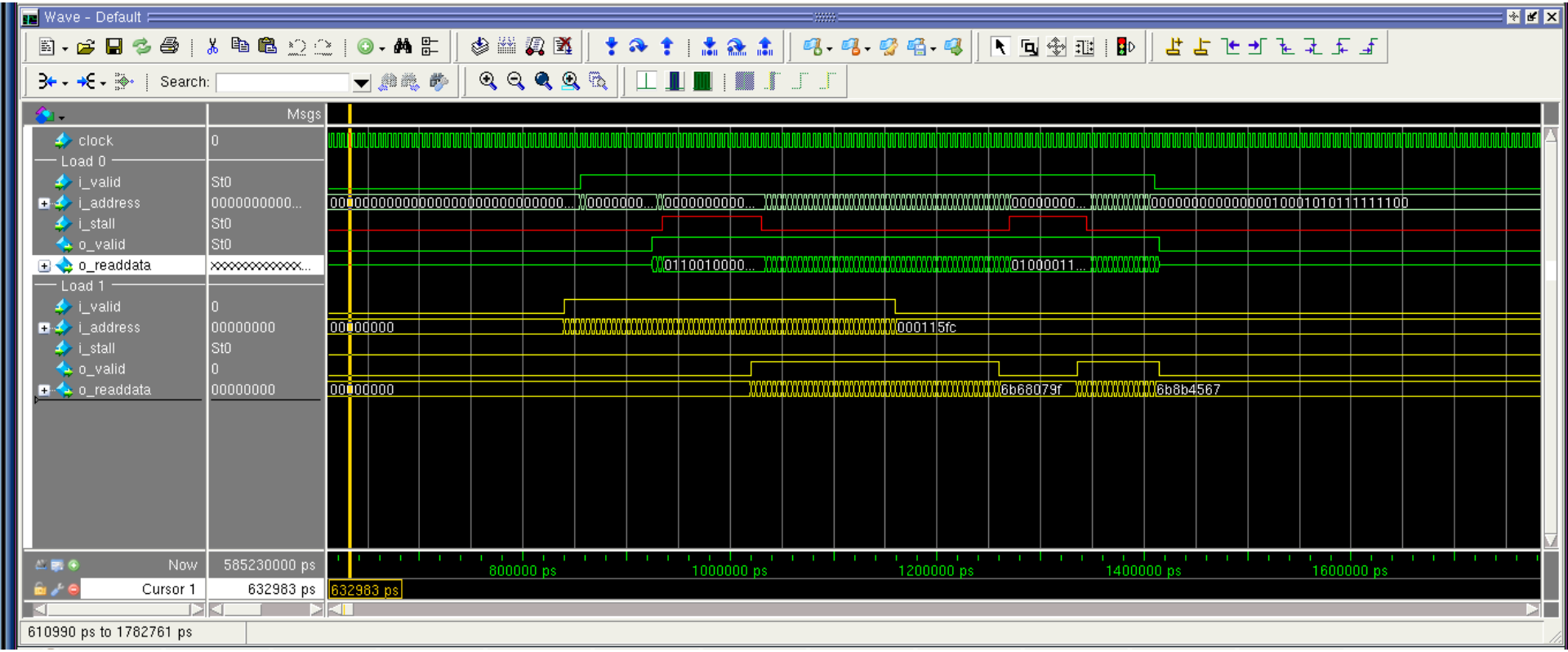
- **Advantage FPGA**
  - Integer Arithmetic
- **Results**

Platform	Power (W)	Performance (FPS)	FPS/W
W3690 Xeon Processor	130	4.6	.035
nVidia Tesla C2075	215	53.1	.247
DE4 Stratix IV-530 Accelerator	21	70.9	2.976
PCIe385 A7 Accelerator	25	74.4	2.976

ALTERA®
MEASURABLE ADVANTAGE™

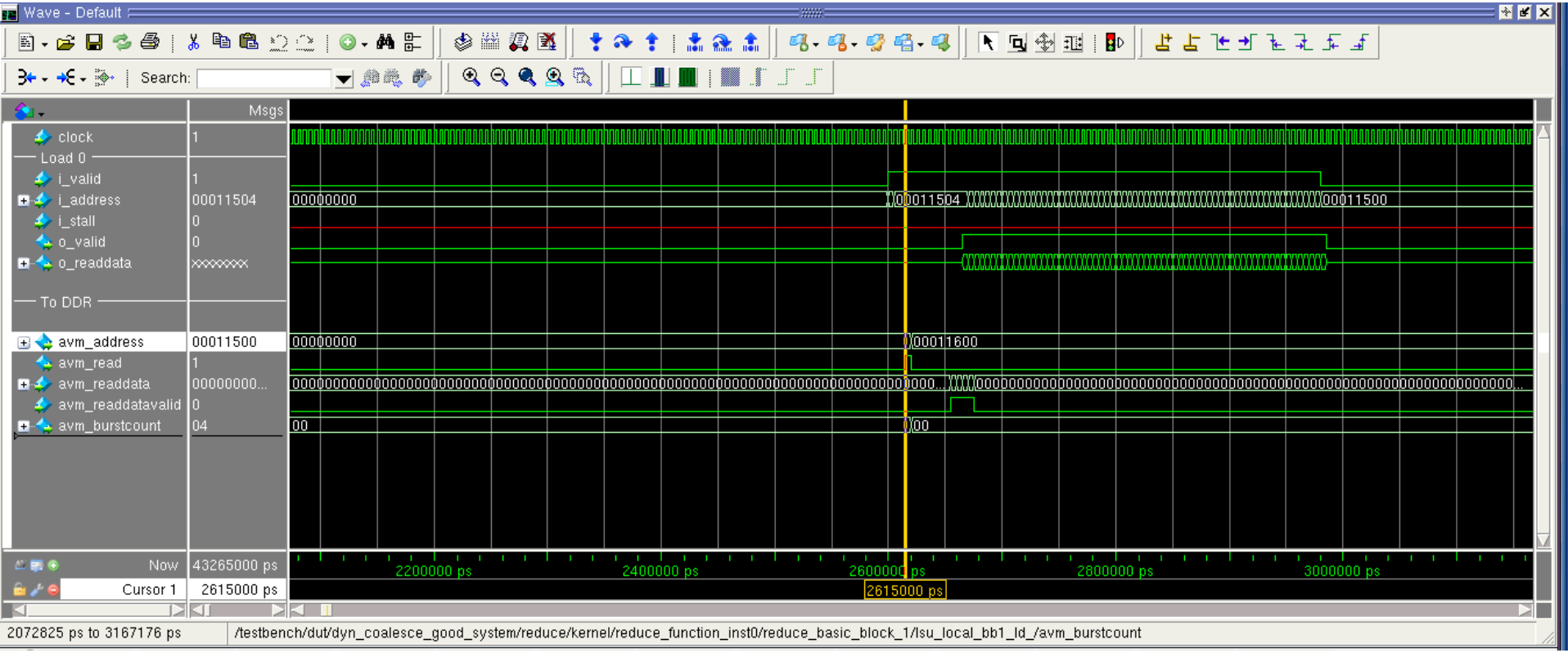
Thank You

ModelSim Example – Load Contention



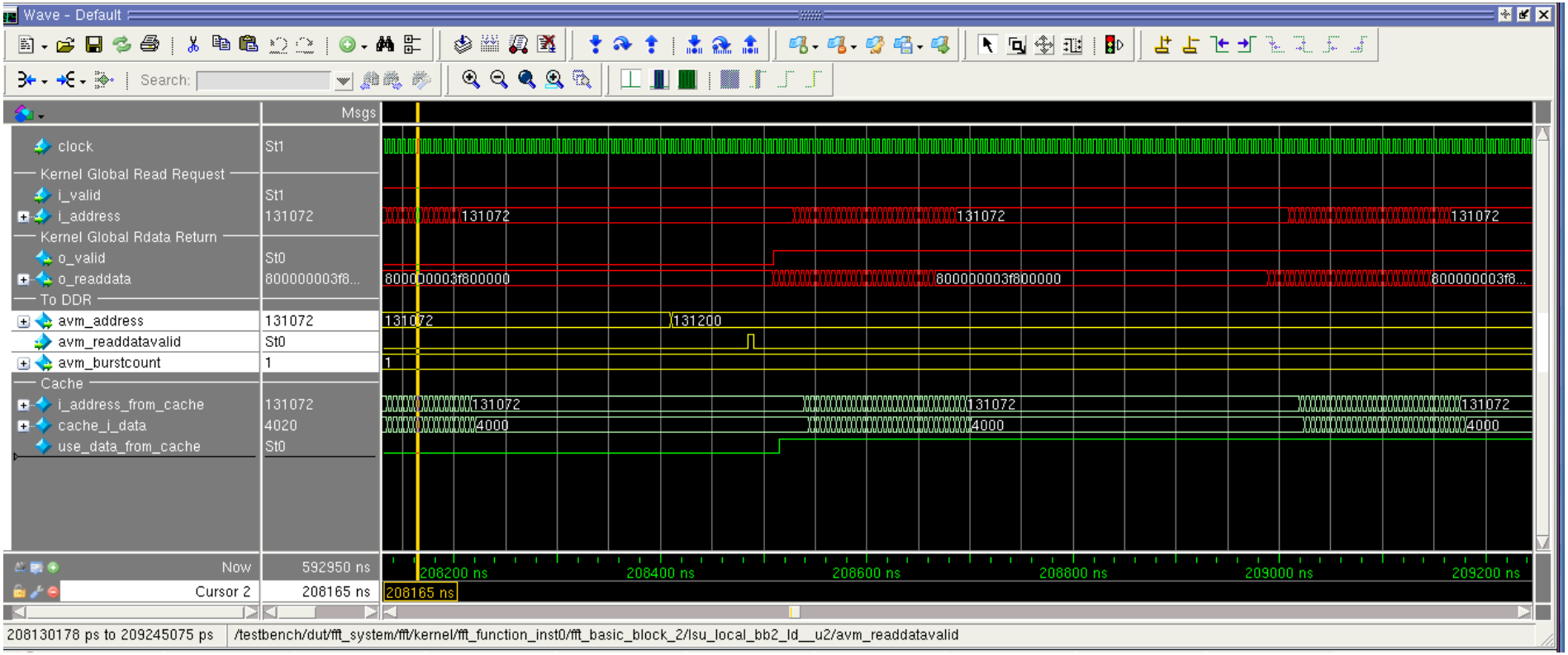
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – No Load Contention



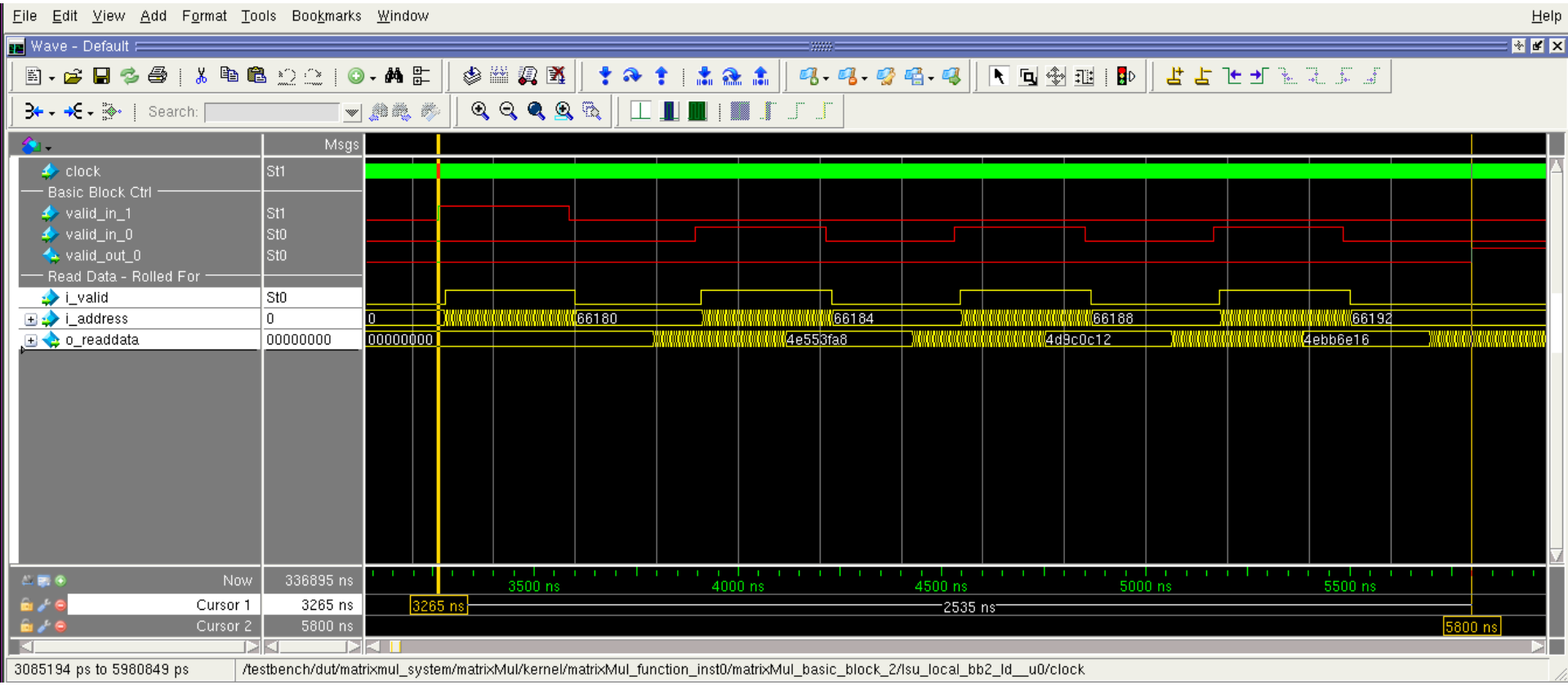
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – FFT Constant Cache



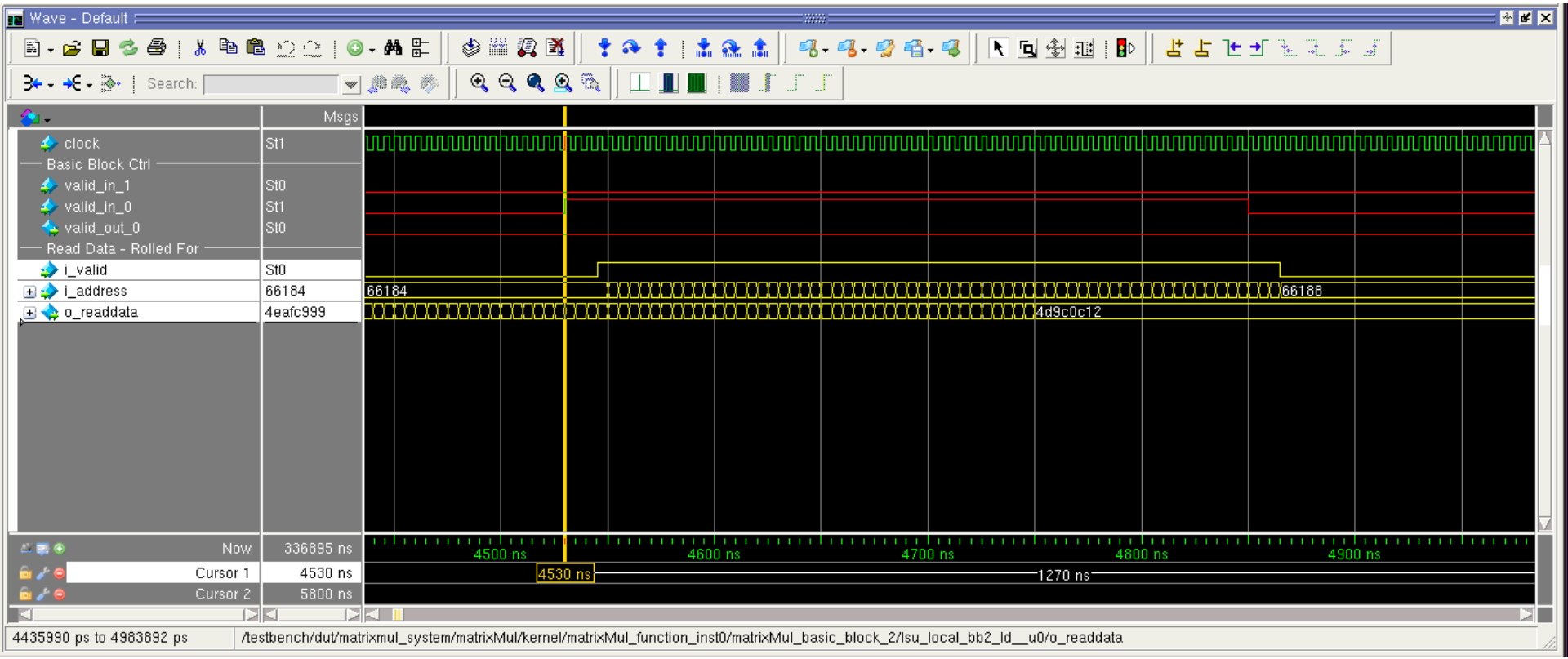
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – Loop Rolled



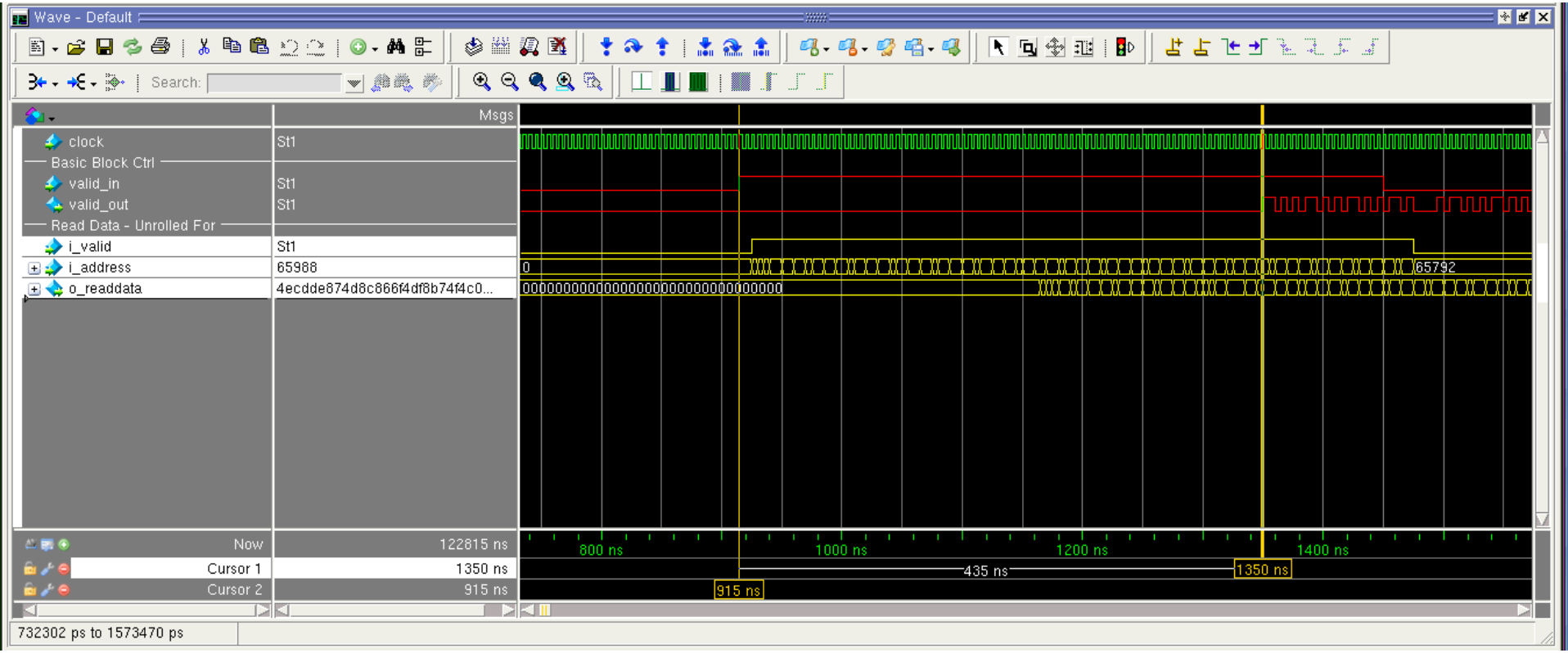
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – Loop Rolled Zoomed



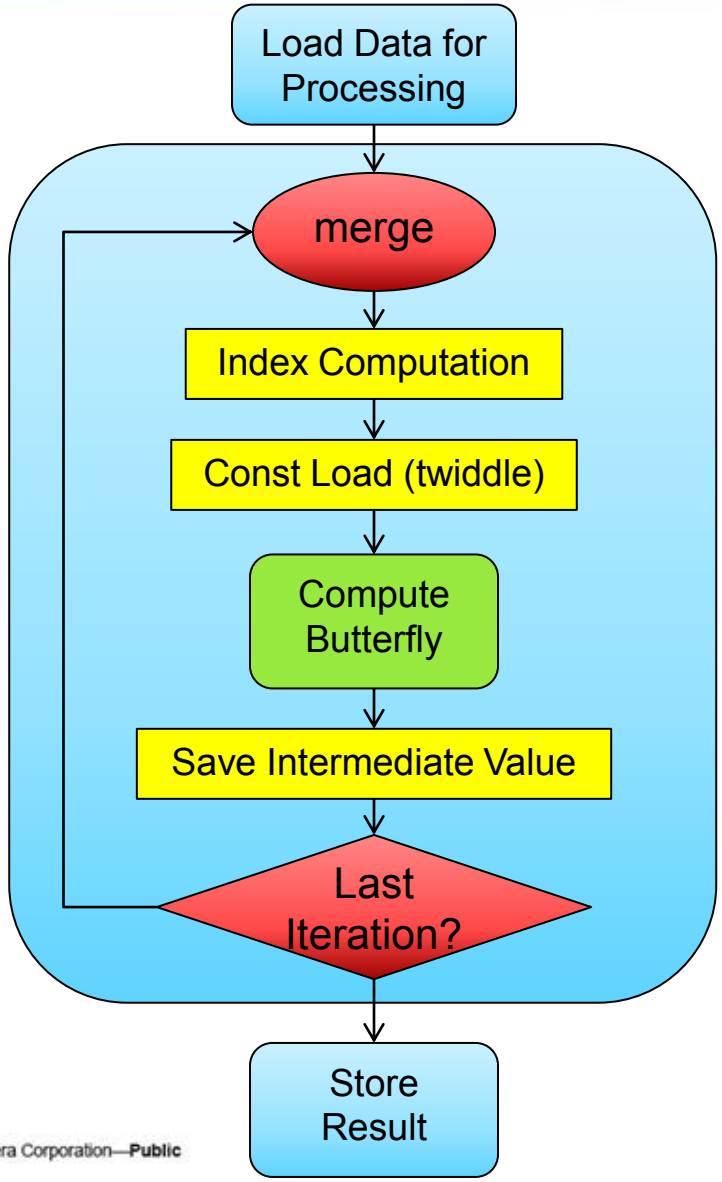
ALTERA
MEASURABLE ADVANTAGE™

ModelSim Example – Loop Unrolled

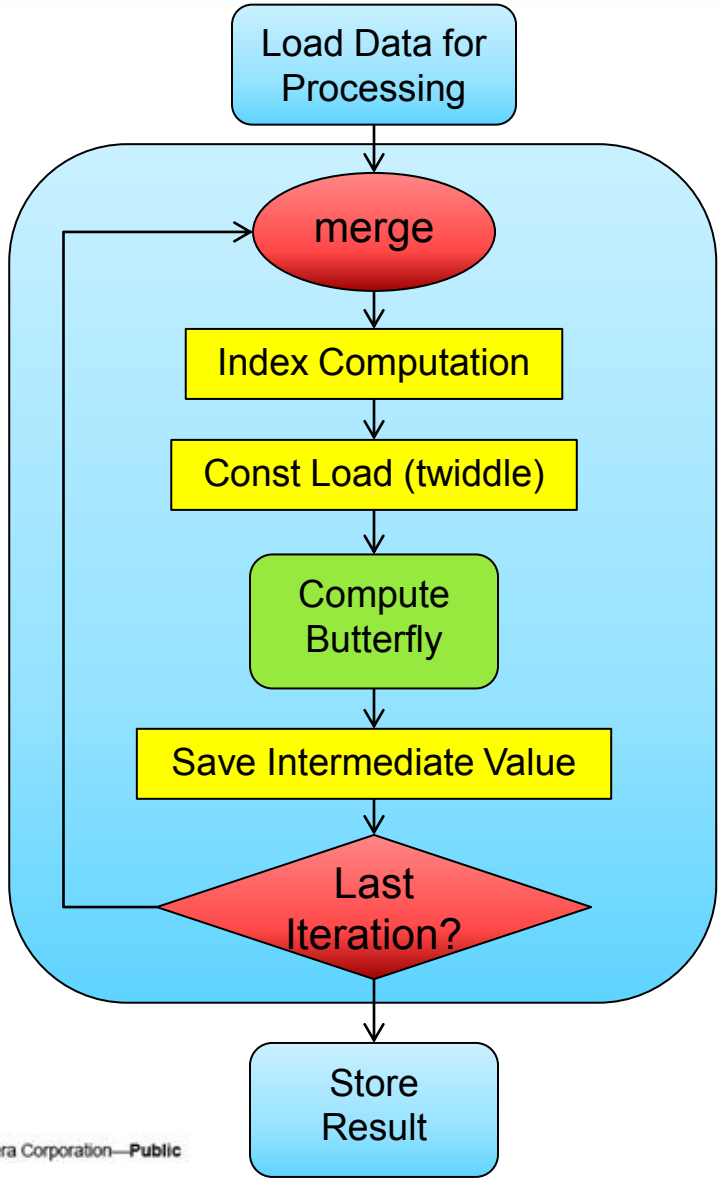


ALTERA
MEASURABLE ADVANTAGE™

Example – FFT; Block Diagram



Example – FFT; High-level Data Flow Graph



Constant Load will generate a cache to reduce demand on DDRx bandwidth