Introduction to OpenCL™ for Intel® FPGAs - Part II

Single Work-Item Execution

Single Work-Item Execution Agenda

- **Introduction**
- Understanding execution models and optimization reports
- Resolving common dependency issues
  - Exercise 1
- Good design practices
- Advanced uses
- Application examples
  - Exercise 2

Single Work-Item Execution

- Launching kernels with global size of (1,1,1)
  - A kernel executed on a compute unit with exactly one work-item
  - Or use `cl::CommandQueue::enqueueTask`
- Defined as a **Task** in OpenCL™
- **Loops in kernels automatically parallelized by the Intel® FPGA OpenCL Offline Compiler**
  - **Single work-item kernels almost always have an outer loop**
  - ***Entire kernel gets pipeline parallelized!***
- Intel FPGA specific feature that wouldn't run well on other architectures

Single-Threaded Kernels Motivation

- Data parallelism isn't always easy to extract
- NDRange execution may not be suitable for certain situations
  - Difficulties partitioning data into workgroups
  - Streaming application where data cannot arrive in parallel
- Some algorithms that are inherently sequential and depend on previous results
  - E.g. FIR filters, compression algorithms
- Sequential programming model of tasks more similar to C programming
  - Certain usage scenario more suited for sequential programming model
  - Easier to port

Data Parallelization Review

OpenCL™ NDRange execution best suited for applications where each loop iteration is independent

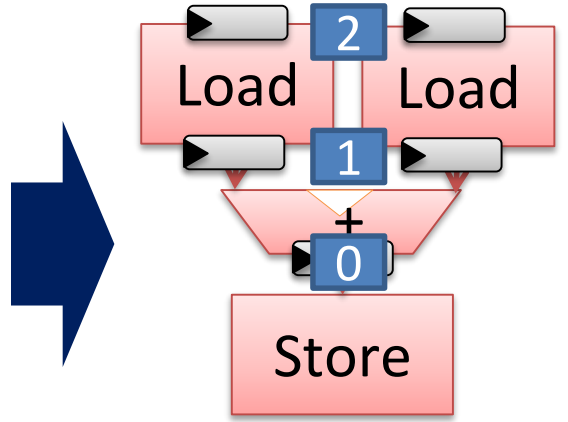
Algorithm

```
for (int i=0; i < n; i++)
    answer[i] = a[i] + b[i];
```

OpenCL™ Implementation

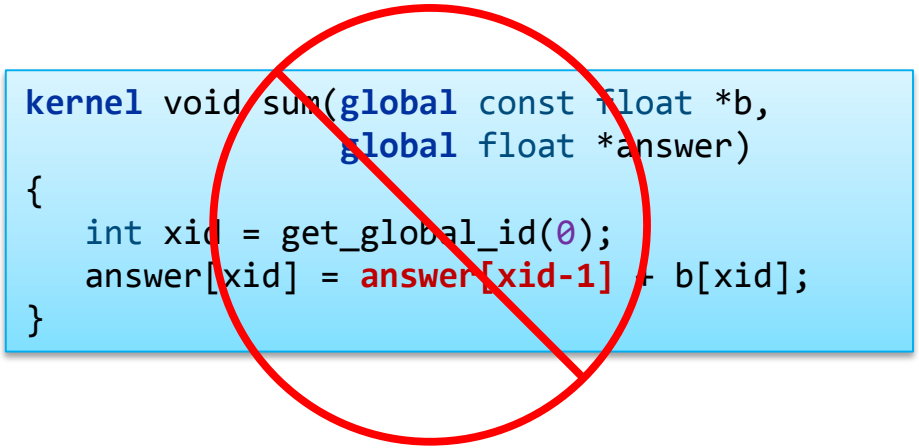
```
__kernel void sum(__global const float *a,
                  __global const float *b,
                  __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

FPGA Acceleration through Pipelined Execution



Data-Parallel Execution with Dependencies

- Difficult to express programs which have partial dependencies during execution
- Does this work?

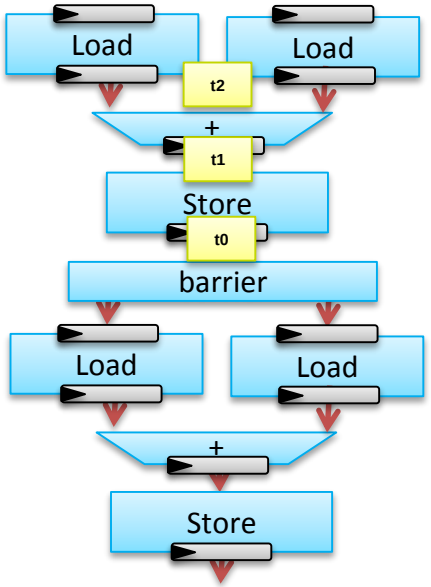


- No
  - Would require new language semantics to describe the desired behavior
    - With possibly complicated supporting hardware

Sharing State – NDRange Kernels

- Barriers define memory synchronization points
- Shared data must be buffered up

```
int xid = get_global_id(0);
c[xid] = a[xid] + b[xid];
barrier();
c[N-xid] = c[M-xid]+a[xid];
```

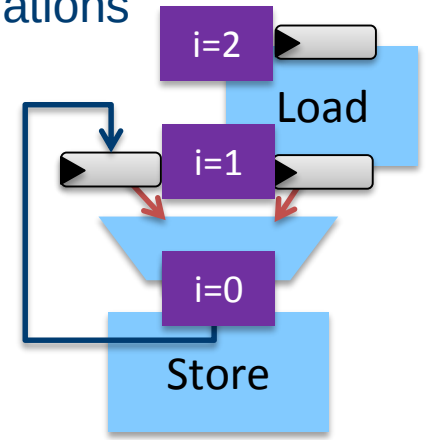


Solution: Tasks and Loop-pipelining

Allow users to express programs as a single-thread kernel

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

- Compiler will infer pipelined parallel execution across loop iterations
  - Efficiently execute multiple loop iterations
  - Dependencies resolved by the compiler
  - Values transferred between loop iterations with FPGA resources
    - No need to buffer up data
    - Easy and cheap to share data through feedbacks in the pipeline

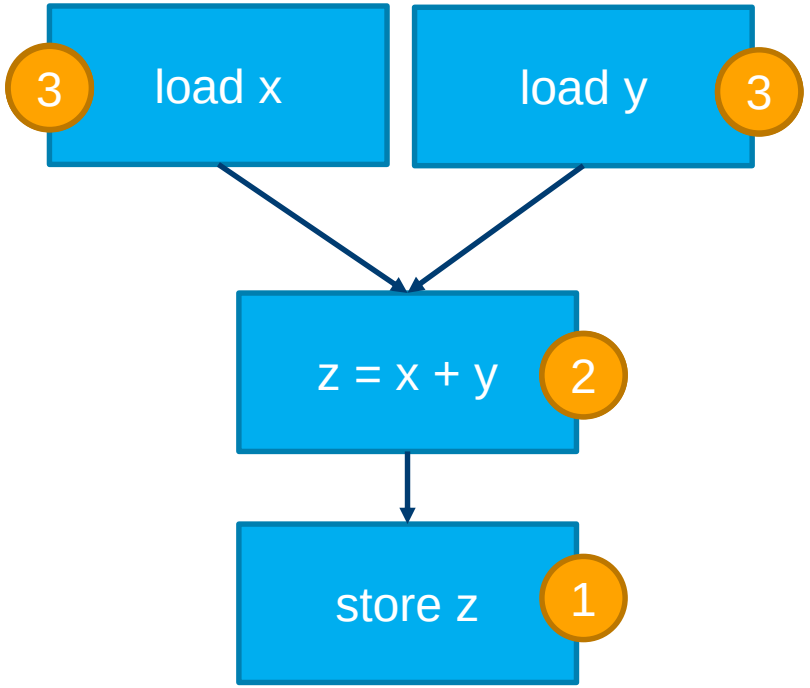


Pipeline Parallel Execution for Loops

- AOC will infer pipelined parallel execution across loop iterations
- Example Pseudo-code:

```
__kernel void sum(__global const float *x,
                  __global const float *y,
                  __global float *z)
{
  for (int i=0; i < n; i++)
    z[i] = a[i] + b[i];
}
```

Loop remains in the kernel, like in traditional C programming code



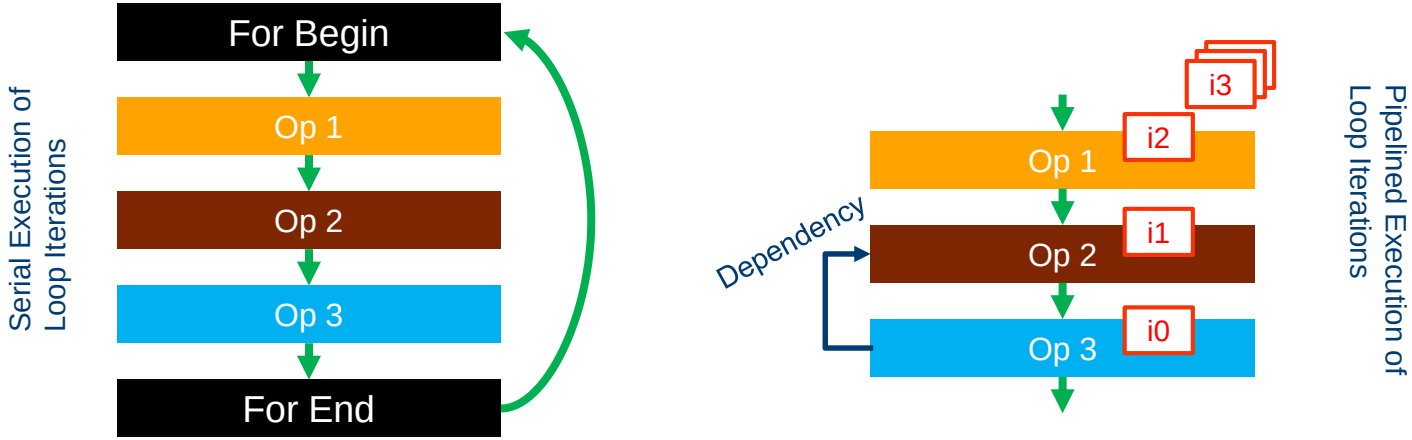
n - Iteration number

Note: each instruction may require several stages in the pipeline

Loop Pipelining vs Serial Execution

Loop pipelining: Launch loop iterations as soon as dependency is resolved

- Initiation interval(II): launch frequency (in cycles) of a new loop iteration
  - II=1 is optimally pipelined
    - No dependency or dependencies can be resolved in 1 cycle



Loop Pipelining

AOC will pipeline each iteration of the loop for acceleration

- Analyze any dependencies between iterations
- Schedule these operations and make copies of hardware if needed
- Launch the next iteration as soon as possible

```
float array[M];
for (int i=0; i < n; i++)
{
   for (int j=0; j < M-1; j++)
        array[j] = array[j+1];
   array[M-1] = a[i];

   for (int j=0; j < M; j++)
      answer[i] += array[j] * coefs[j];
}
```

Shift Register `array`
(Dependency for next iteration)

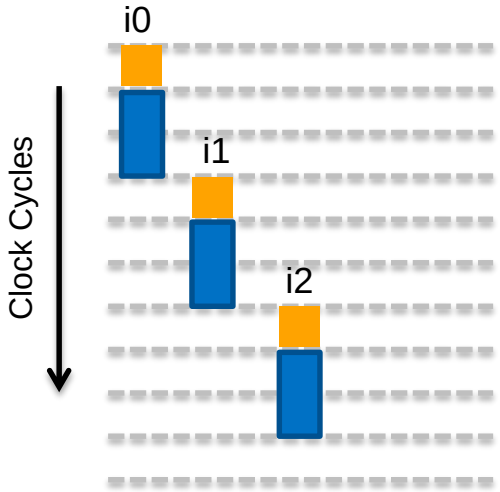
Reduction on `array`
(Not a dependency)

At this point, launch the next iteration of outer loop

(Copies of shift registers made automatically)

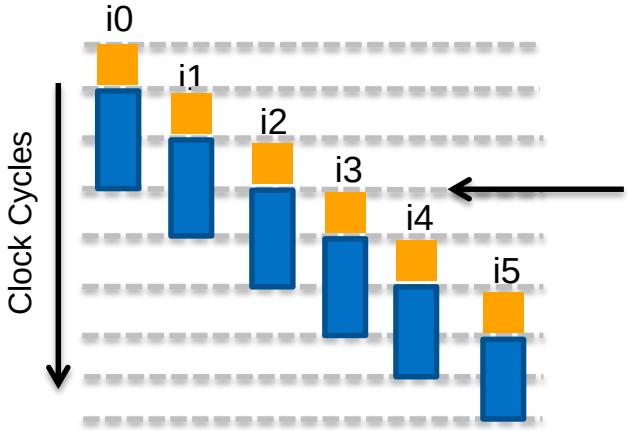
Loop Pipelining Example

No Loop Pipelining



No Overlap of Iterations!

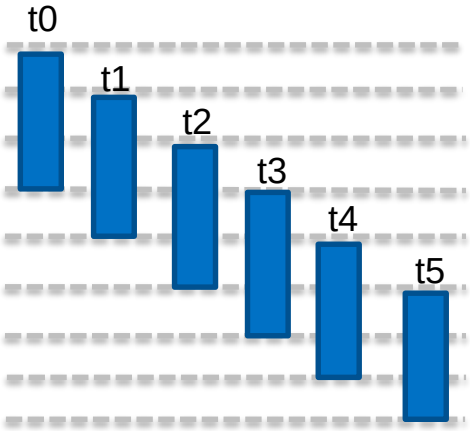
With Loop Pipelining



Finishes Faster because Iterations Are Overlapped

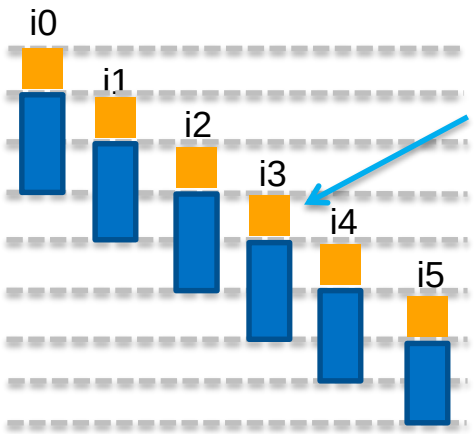
Parallel Threads vs Loop Pipelining

NDRange Parallel Threads



Parallel threads launch 1 thread per clock cycle in pipelined fashion

Loop Pipelining



If loop dependency resolved in 1 clock cycle

- Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
  - If dependency resolved in 1 clock cycle, then the throughput is the same
  - *Data dependency resolved without adding extra compute time!*

Advantages of Loop Pipelining

- Some applications require loop pipelining to be efficient
  - E.g. streaming applications
    - When data cannot arrive in parallel

- Loop Pipelining automatically extracts parallelism within a kernel
- Memory accesses may also be optimized
- No need to partition data into workgroups explicitly
- Makes porting of C-based applications easier

Single Work-Item vs. NDRange Kernels

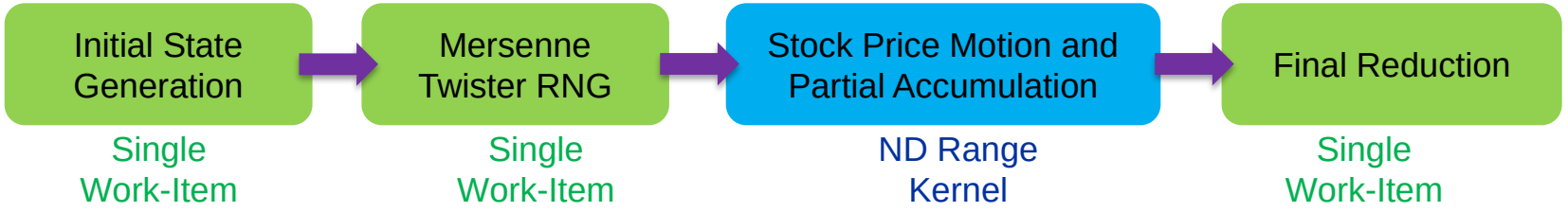
One approach is not better than the other

- Create single work-item kernels if
  - Data processing sequencing is critical
  - Algorithm can't easily break down into work-items due to data dependencies
  - Not all data available prior to kernel launch
  - Data cannot be easily partitioned into workgroups
- Create NDRange kernels if
  - Kernel does not have loop and memory dependencies
  - Kernel can execute multiple work-items in parallel efficiently
    - Able to take advantage of SIMD processing

Single Work-Item vs. NDRange Kernels Example

Able to mix kernel types in the same application depending on algorithm

- Monte Carlo Asian Options Simulation Example
  - Difficult to parallelize random number generation using an NDRange kernel
    - Express as single work-item kernel and rely on compiler to infer parallelism
  - Partial accumulation can be easily expressed with NDRange kernel



Recognition of Single Work-Item Kernels

AOC assumes single work-item kernels if kernel code does not query any work-item information

- No `get_global_id(), get_local_id(), or get_group_id()` calls
- Enables AOC to automatically perform loop pipelining and memory dependence analysis on the kernel
- Many C-based algorithms can directly compile to an OpenCL™ Task

```
__kernel void mykernel (…) {
     for (i=0; i< FFT_POINTS; i++) {
          …
     }
}
```

Launching Single Work-Item Kernels (Tasks)

- Use `cl::CommandQueue::enqueueNDRangeKernel` with `global_work_size` and `local_work_size` set to 1
- Or `cl::CommandQueue::enqueueTask` in host code
  - Equivalent to the above `enqueueNDRangeKernel` call
  - Simpler, but deprecated in OpenCL™ 2.0

Host Code

```
setup_memory_buffers();
transfer_data_to_fpga();

myqueue.enqueueTask(mykernel, …);

read_data_from_fpga();
```

Test Your Knowledge

What's the best kernel type to implement? ND Range or Single work-item?

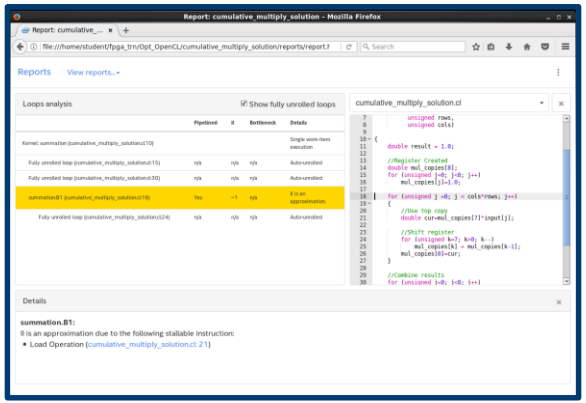
Kernel Algorithm	Kernel Type
Gzip Compression Data is traversed byte-by-byte and is compared against previous bytes to look for matches	
Matrix Multiplication Calculation tiled into dimensions of BLOCK_SIZE x BLOCK_SIZE	
Document Filtering Apply bloom filter to an existing set of independent documents to find the documents of interest to a user	
OPRA FAST Parser Parses streaming data from UDP offload engine and returns a subset of fields over Ethernet	

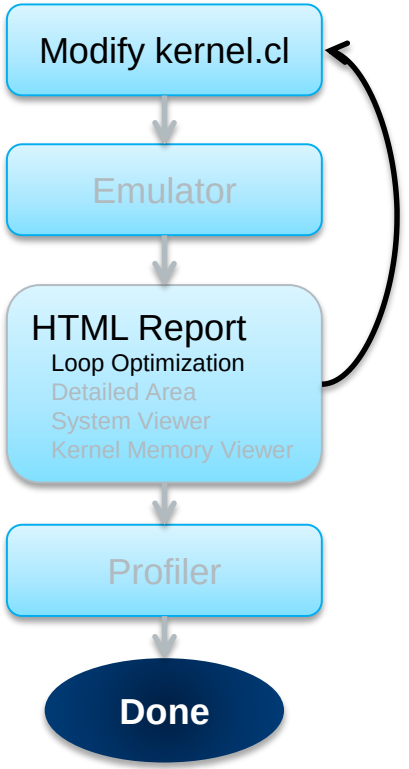
Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
  - Lab 1
- Good design practices
- Advanced uses
- Application examples
  - Lab 2

Loop Analysis for Single Work-Item Kernels

- Automatically Generated
- Reports status of loop pipelining
- Displays dependency information
- Part of HTML Report
  - **<kernel file folder>\reports\report.html**





Loop Pipelining Optimization Report

Report shows pipeline status of each single-work item kernel loop

- Initiation Interval (II) = launch frequency of loop iterations
  - Cycles between loop iteration launches
- Minimizing II is the key to single work-item performance optimization
- Report shows
  - If loops are pipelined
  - Initiation interval of pipelined loops

Loop Pipeline Status

Three possible scenarios

- Loop iterations launched every clock cycle (II=1)

```
|-+ Loop "for.body" (file test.cl line 10)
    Pipelined well. Successive iterations are launched every cycle.
```

- Failed loop pipelined execution, iterations will execute serially

```
+ Loop "Block1" (file b.cl line 4)
| NOT pipelined due to:
```

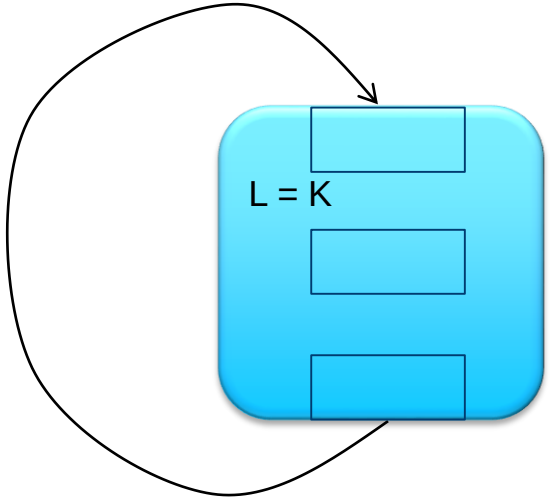
- Pipeline execution inferred but II > 1 due to dependencies

```
+ Loop "for.body8" (file test.cl line 138)
   Pipelined with successive iterations launched every 7 cycles due to:
```

Loop Pipeline Single Loop Execution

Basic case – single loop

```
kernel void test() {
     for (i=0; i<N; i++) {
        …
     }
}
```



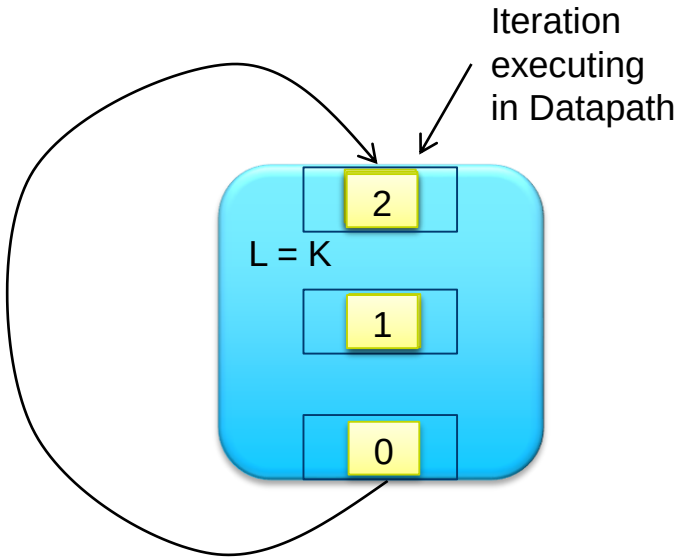
L : Latency of the loop
(clock cycles or pipeline stages)

K: Constant value

Loop Pipeline Single Loop Execution

Basic case – single loop

```
kernel void test() {
     for (i=0; i<N; i++) {
        …
     }
}
```



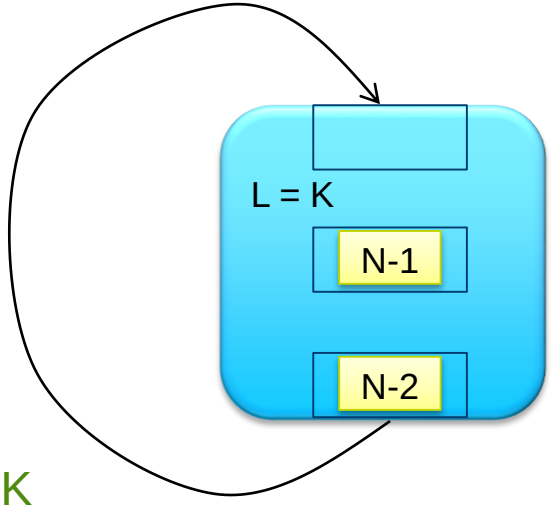
Loop Analysis Report: II=1

With II = 1, iterations launched every clock cycle one after another

Loop Pipeline Single Loop Execution

Basic case – single loop

```
kernel void test() {
      for (i=0; i<N; i++) {
         …
      }
}
```



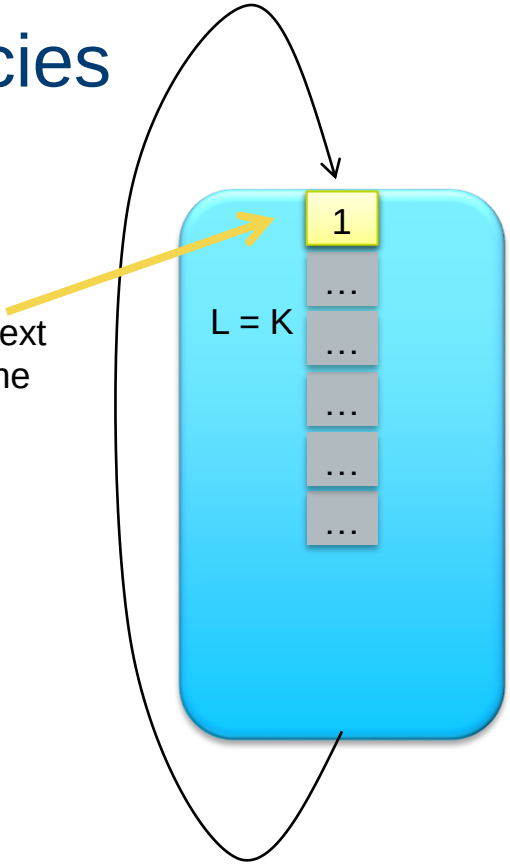
- Total number of clock cycle to run kernel is about N + K
  - K typically in the order of 100s of clock cycles
  - N: Iterations based on data, usually orders of magnitudes larger than K
  - So: Number of total clock cycles ≈ N
  - Throughput can be estimated without actually running the kernel!

Single Loop with Complex Dependencies

- II > 1, caused by complex data or memory dependencies
  - Dependencies not resolved in 1 cycle

```
kernel void test() {
     for ( … ) {
         A[x] = A[y];
         …
     }
}
```

Loop Analysis Report: II=6



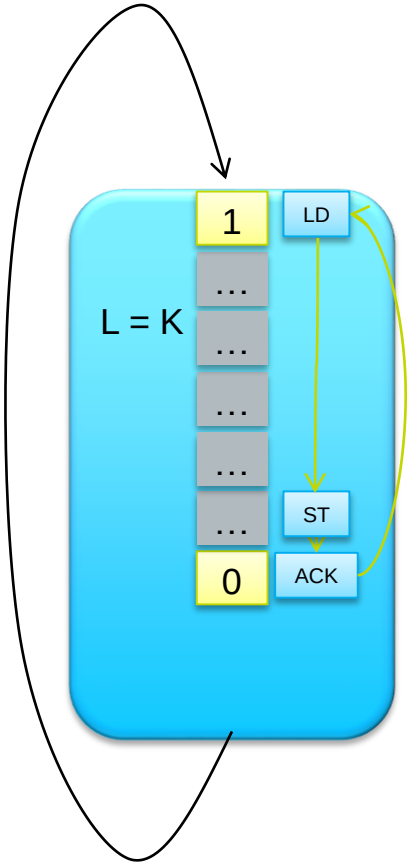
- Total number of cycles to run is about N*6 + K ≈ 6*N

Single Loop with Complex Dependencies

- II > 1
- Hardware created to stall the pipeline until dependency is resolved

```
kernel void test() {
     for ( … ) {
         A[x] = A[y];
         …
     }
}
```

- Total number of cycles to run kernel is about N*II + K ≈ II*N
- Key to single work-item kernel throughput is reducing II
  - Minimize stalls



Memory Dependency

Loop-carried dependency where a memory operation cannot occur before dependent memory operation from a previous iteration

```
+ Loop "for.body8" (file test.cl line 138)
   Pipelined with successive iterations launched every 7 cycles due to:

      Memory dependency on Load Operation from: (file test.cl line 140)
        Store Operation (file test.cl line 140)
      Largest Critical Path Contributors:
          73%: Load Operation  (file test.cl line 140)
          26%: Store Operation  (file test.cl line 140)
```

- Largest Critical Path Contributor
  - Specifies the operations that contribute to the delay

Data Dependency

Loop-carried dependency where a variable is dependent on the result from a computation in the previous iteration

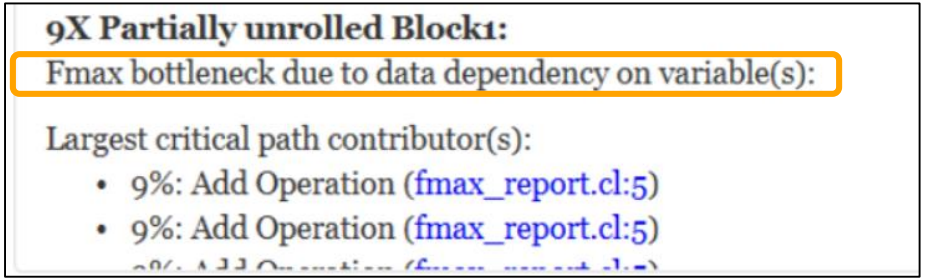
```
+ Loop "for.body" (file float.cl line 5)
  Pipelined with successive iterations launched every 9 cycles due to:

    Data dependency on variable sum (file float.cl line 6)
    Largest Critical Path Contributor:
        96%: Fadd Operation  (file float.cl line 6)
```

- Largest Critical Path Contributor
  - Specifies the operations that contribute to the delay

Speed-Limiting Constructs

AOC may make trade-offs between circuit frequency and II value

- Forcing dependency calculations to complete in fewer cycles may result in hardware operating at a lower frequency
  - Less pipelining then AOC would otherwise like to use



- Check Fmax in `<kernel file>/acl_quartus_report.txt`
- Consider simplifying the loop-carried dependency calculation

Resolving Loop Exit Condition at Iteration Initiation

- During loop pipelining exit condition evaluated at the beginning of the loop
- Exit condition evaluated at the end for serial loops
- If exit condition cannot be moved to the beginning, loop pipelining fails
- Usually caused by an exit condition containing a memory operation or other complex operations

```
while( input[i++] < N )  {
      …
}
```

```
+ Loop "for.cond.preheader" (file test.cl line 9)
| NOT pipelined due to:
|   Loop exit condition unresolvable at iteration initiation.
|   Simplify loop exit condition to fix this problem.
```

Loop Block Contains Non-Linear Execution

Loop pipelining require linear execution

- Non-linear loop block execution prevent loop pipelining

```
for ( unsigned i = 0; i < N; i++ ) {
     if ( (i & 3) == 0 )
          for (…)
               …
     else
          for (…)
               …
}
```

```
+ Loop "Block1" (file b.cl line 4)
| NOT pipelined due to:
|
|   Loop structure: loop contains divergent inner loops.
|   Making all inner loops unconditional should fix this problem.
```

Loop Block Contains Non-Linear Execution (Possible Solution)

- Always take both execution paths, and select between the result dependent on the nonlinear comparison at the end

```
for ( unsigned i = 0; i < N; i++ ) {
     for (…) {
          …
          result1=…
     }
     for (…) {
          …
          result2=…
     }

     if ( (i & 3) == 0 )
          result = result1;
     else
          result = result2;
}
```

Loop Pipeline with Nested Loops

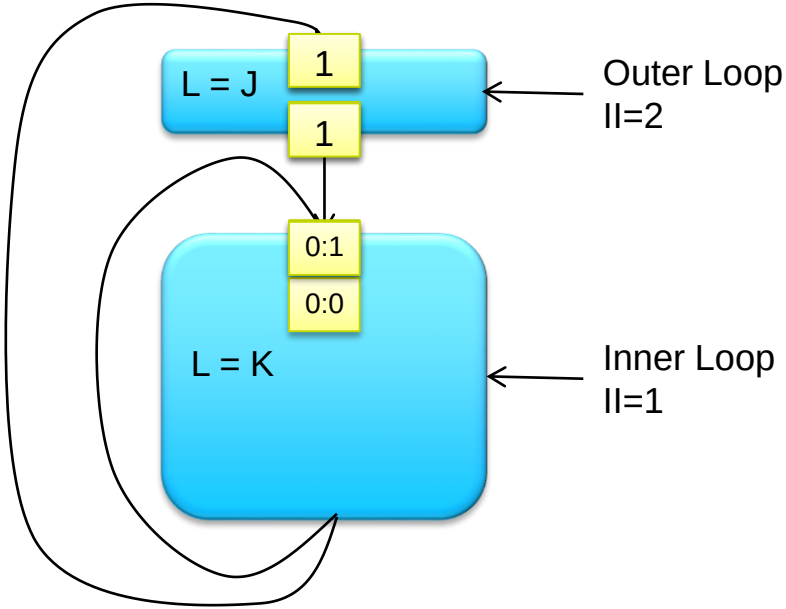
"Critical Loop" determines performance, non-critical loops can have poor II

```
kernel void test() {
    while (i < M) {
        …
        for ( j=0; j<N; j++) {
```

Loop Analysis Report:

Outer Loop: Pipelined, II >=2

Inner Loop: Pipelined, II=1



Total run = M*(N*1) + K + J

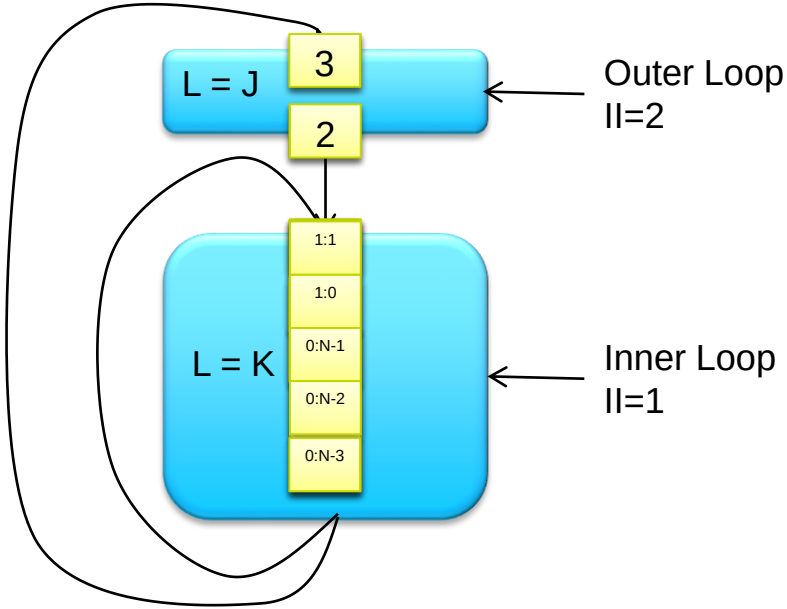
Critical Loop II

Loop Pipeline with Nested Loops

"Critical Loop" determines performance, non-critical loops can have poor II

```
kernel void test() {
    while (i<M) {
        …
        for ( j=0;j<N;j++) {
```

- Outer loop iterations now blocked because inner loop is busy
- II on outer loop doesn't impact performance
- Outer loop II only an issue if
  - N * II_inner_loop < II_outer_loop

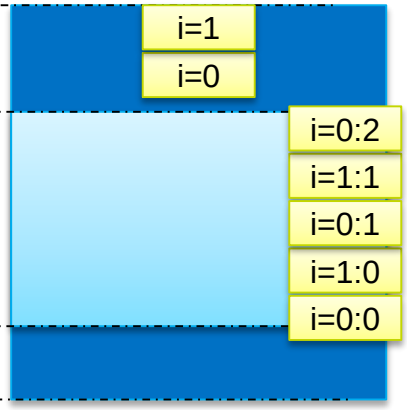


Out-of-Order Loop Execution

Nested loops where the number of iterations of the inner loop varies among outer loop iterations

- Outer loop iteration could become out-of-order

```
for (i=0; i<N; i++) {
    …
    MV_done = false;
    do {
      SADsMB(refBuf, MB, … );
      …
      if ( check( MB ) ) {
            MV_done = true;
      }
    } while (!MV_done);

}
```



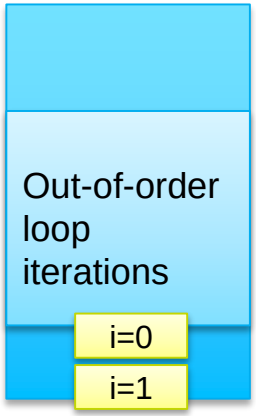
Out-of-order loop iterations

Out-of-Order Loop Iterations

```
for ( i=0; i < N; i++ )
     for ( j=0; j < N-i ; j++ ){
          …
     }
}
```

- Common coding style
- Compiler analyzes impact of out-of-order iterations on functionality
  - Check for independence of iterations
  - Loop pipelining still inferred if functionality not affected
- If out-of-order iterations may lead to incorrect result
  - Loop NOT pipelined

```
Loop Report:

+ Loop "for.cond1.preheader" (file test.cl line 38)
| NOT pipelined due to:
|
|   Loop iteration ordering: iterations may get out of order with respect to the
|   listed inner loop, as the number of iterations of the listed inner loop may be
|   different for different iterations of this loop.
|       Loop "for.body3" (file test.cl line 39)
```



Serial Region Execution

Optimization Report

Outer Loop: II = 2

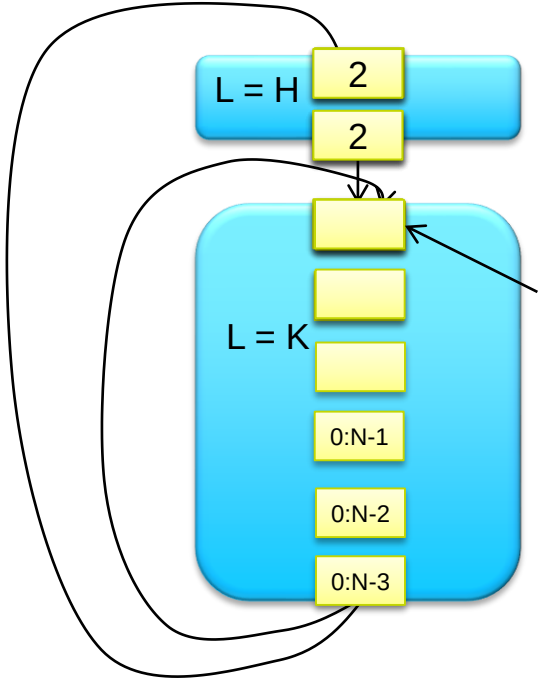
Serial execution around: Inner Loop

Inner Loop: II=1

- Serial region can occur with nested loops
  - An inner loop access causing an outer loop dependency
  - Inner loop becomes a serial region in the outer loop iteration

```
kernel void test() {
     int a[1024];
     while (i<M) {
        for ( j=0; j<N; j++) {
          a[X] = b[X];
          process(a);
        }
     }
}
```

Access to **a** can not be made until all previous outer iterations have completed



Iteration 1 cannot enter inner loop because it is a serial region

Iteration 1 enters inner loop after all iteration 0 inner iterations have exited

Serial Regions

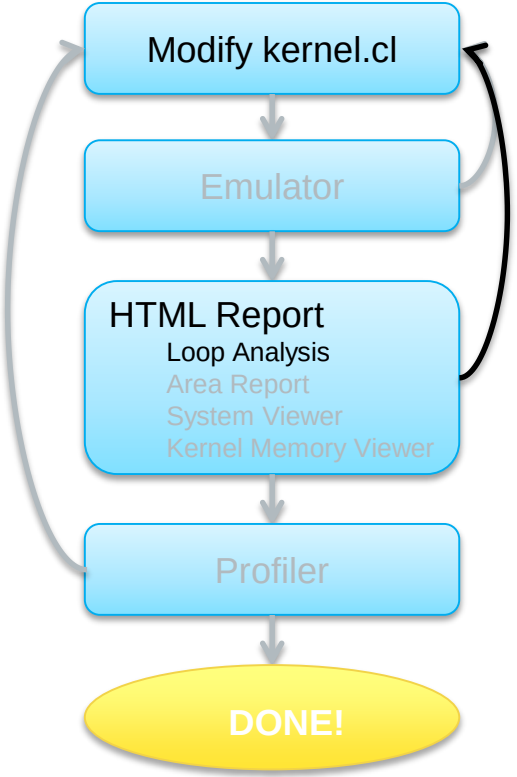
- Significant issue if inner loop II>1
- Not an issue if inner loop trip count is high relative to latency of inner loop
- II of both inner and outer loops not affected
- Optimization report will state data or memory dependency causing the serial region

```
| Iterations executed serially across the region listed below.
| Only a single loop iteration will execute inside the listed region.
| This will cause performance degradation unless the region is pipelined well
| (can process an iteration every cycle).
|
|     Loop "Block2" (file singlethreaded.cl line 10)
|     due to:
|     Data dependency on variable
```

Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
  - Lab 1
- Good design practices
- Advanced uses
- Application examples
  - Lab 2

Minimize Pipeline Stalls



Improve the performance of single work-item kernels by addressing loop-carried dependencies

- Techniques
  - Remove dependency
  - Relaxing dependency
  - Simplifying dependency
  - Transferring dependency to local memory
  - Remove dependency using a pragma

Removing Loop-Carried Dependency (Unoptimized)

- Outer loop launches every cycle
  - Not the critical loop

- Each inner iteration requires `sum` from the previous outer iteration
  - Becomes serial region

- Inner loop pipelined well!

Unoptimized

```
int sum = 0;
for (unsigned i=0; i<N; i++) {
      for (unsigned j=0; j<N; j++) {
            sum += A[i*N+j];
      }
      sum += B[i];
}
```

```
| *** Loop Analysis Report ***


Loop "Block1":
      Pipelined with II>=1
      Serial Region across Loop "Block2"
      due to dependency on variable sum

Loop "Block2":
      Pipelined with II=1
```

Removing Loop-Carried Dependency (Optimized)

To remove the dependency and thus serial region

- Accumulate using local variable for inner loop (**sum2**)
  - Instead of using the same **sum** as outer loop
- Add the local sum2 to sum at the end of each outer iteration

Optimized

```
int sum = 0;
for (unsigned i=0; i<N; i++) {
      int sum2 = 0;
      for (unsigned j=0; j<N; j++) {
            sum2 += A[i*N+j];
      }
      sum += sum2;
      sum += B[i];
}
```

```
| *** Loop Analysis Report ***


Loop "Block1":
      Pipelined with II>=1

Loop "Block2":
      Pipelined with II=1
```

Relaxing Loop-Carried Dependency (Unoptimized)

- Floating point multiply here takes 6 cycles
  - Data dependency on `mul` every cycle means II needs to be 6

- Strategy: Increase the distance of the dependency to be more than 1 iteration

Unoptimized

```
float mul = 1.0f;
for (unsigned i = 0; i < N; i++)
{
      mul = mul * A[i];
}
```

```
| *** Loop Analysis Report ***

Loop "Block1"
    Pipelined, II=6 due to Data
    dependency on variable mul
    Largest Critical Path Contributor:
          100%: Fmul Operation
```

Relaxing Loop-Carried Dependency (Optimized)

- Relax the dependency over `M` iterations to match latency of dependent operation
- Instead of 1 result variable, use `M` copies
  - Number of copies depend on the initial II
  - `M` copies implemented as shift register
- Top copy used in multiplication
- Shift values
  - Result goes to the bottom of shift register
- Reduce all the copies to one result

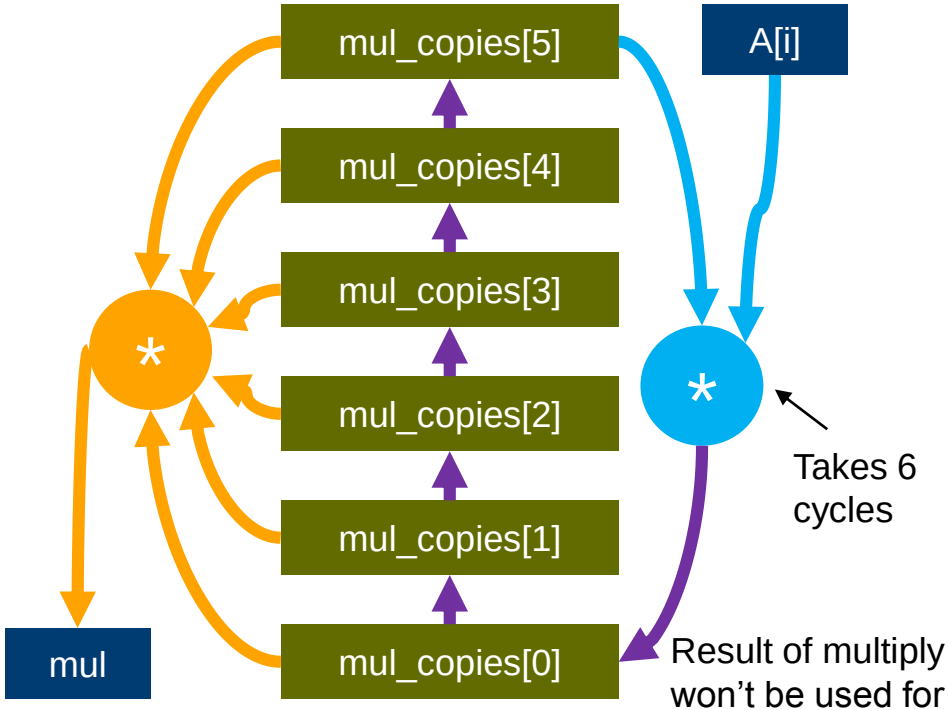
`#pragma unroll` signals compiler to flatten the loop structure and execute all iterations of the loop in one feed forward path

Optimized

```
#define M 6
float mul = 1.0f;
float mul_copies[M];
for (unsigned i = 0; i < M; i++)
    mul_copies[i] = 1.0f;

for (unsigned i = 0; i < N; i++) {
    float cur = mul_copies[M-1]*A[i];

    #pragma unroll
    for (unsigned j = M-1; j >0; j--)
        mul_copies[j] = mul_copies[j-1];
    mul_copies[0] = cur;
}

#pragma unroll
for (unsigned i = 0; i < M; i++)
    mul = mul * mul_copies[i];
```

```
*** Loop Analysis Report ***

Loop "Block 1"
     Pipelined. II=1
```

Relaxing Loop-Carried Dependency (Optimized)



Optimized

```
#define M 6
float mul = 1.0f;
float mul_copies[M];
for (unsigned i = 0; i < M; i++)
     mul_copies[i] = 1.0f;

for (unsigned i = 0; i < N; i++) {
     float cur = mul_copies[M-1]*A[i];

     #pragma unroll
     for (unsigned j = M-1; j >0; j--)
          mul_copies[j] = mul_copies[j-1];
     mul_copies[0] = cur;
}

#pragma unroll
for (unsigned i = 0; i < M; i++)
     mul = mul * mul_copies[i];
```

```
*** Loop Analysis Report ***

Loop "Block 1"
     Pipelined. II=1
```

Relaxing Loop-Carried Dependency Generically

Symptom:

- One operation taking multiple cycles
  - Causing data dependency
  - Here, double floating point add takes 11 cycles

Solution:

- Infer shift register to relax the dependency

Unoptimized

```
double sum = 0;
for (int i = 0; i < N; ++i)
      sum += input[i];
```

```
*** Loop Analysis Report ***

Loop "Block1"
  Pipelined, II=11
  Data dependency on sum
  Largest Critical Path Contributor:
      97%: Fadd operation
```

Relax Loop-Carried Dependency Generically (Solution)

Optimized

Approach:

- Create shift register with II_CYCLES+1 elements
- Perform operation with input element and bottom value of shift register and store it at the top
- Shift every element in the register down
- Reduce shift register values into one value

```
#define II_CYCLES 12

double shift_reg[II_CYCLES+1];
for (int i = 0; i < II_CYCLES+1; i++)
     shift_reg[i] = 0;

for(int i = 0; i < N; ++i) {
     shift_reg[II_CYCLES] = shift_reg[0] + input[i];
     #pragma unroll
     for(int j = 0; j < II_CYCLES; ++j)
          shift_reg[j] = shift_reg[j + 1];
}

double sum = 0;
#pragma unroll
for(int i = 0; i < II_CYCLES; ++i)
      sum += shift_reg[i];
```

Optimization report:

- Indicates main loop with shift register has II=1
- All other loops are successfully unrolled (not shown)

```
 *** Loop Analysis Report ***
Loop "Block1"
      Pipelined. II=1
```

Transferring Loop-Carried Dependency to Local Memory (Unoptimized)

System memory accesses may have long latencies, move dependencies to local memory

- Example:
  - Dependency on Global variable A

Unoptimized

```
__kernel void mykernel (int* restrict A) {
      for (unsigned i = 1; i < N; i++)
          A[N-i] = A[i];
}
```

```
 *** Loop Optimization Report ***

Loop "Block1":
  Pipelined with II >= <some value>
  Due to Stallable Load Operation
```

Transferring Loop-Carried Dependency to Local Memory (Optimized)

Solution: Move array A[i] from system to local memory

- Copy global A[] to local B[]
- Execute the loop on local array B[i]
- Copy local B[] back to global A[]

- Dependency now on local array B[]
  - Successive iterations launched every cycles

Optimized

```
__kernel void mykernel(int* restrict A) {
      int B[N];
      for (unsigned i = 0; i < N; i++)
            B[i] = A[i];

      for (unsigned i = 1; i < N; i++)
            B[N-i] = B[i];

      for (unsigned i = 0; i < N; i++)
            A[i] = B[i];
}
```

```
 *** Loop Optimization Report ***
…
Loop "Block1"
  Pipelined. II=1

Loop "Block2":
  Pipelined with II = 1

Loop "Block3":
  Pipelined. II=1
```

Removing Memory Access Loop-Carried Dependency

- `ivdep` pragma asserts memory array accesses will not cause dependencies
  - Apply to loops
  - Removes constraints from otherwise dependent load and store instructions
  - Applies to `private`, `local`, and `global` arrays and pointers
  - Reduces logic utilization and lowers the II value
  - User responsible for functionality!

```
#pragma ivdep
for (unsigned i = 1; i < N; i++)
    A[i] = A[i - X[i]];
```

- Example
  - `X[i]` unknown at compile time, compiler assumes dependency across iterations
  - With `#pragma ivdep`, compiler assumes accesses to memory in this loop will not cause dependencies

`ivdep` Pragma

- `#pragma ivdep`
  - Dependencies ignored for all accesses to memory arrays

```
#pragma ivdep
for (unsigned i = 1; i < N; i++) {
      A[i] = A[i – X[i]];
      B[i] = B[i – Y[i]];
}
```

Dependency ignored for A and B array

- `#pragma ivdep array(`*`array_name`*`)`
  - Dependency ignored for only `array_name` accesses

```
#pragma ivdep array(A)
for (unsigned i = 1; i < N; i++) {
      A[i] = A[i – X[i]];
      B[i] = B[i – Y[i]];
}
```

Dependency ignored for A array
Dependency for B still enforced

`ivdep` Pragma with `safelen`

- `#pragma ivdep safelen (<N>) array(`*`array_name`*`)`
  - `safelen` specifies dependence distance in iterations between load/store in the loop
    - i.e. maximum number of consecutive loop iterations without loop-carried dependency
    - Not specifying `safelen: safelen=∞`

```
#pragma ivdep safelen(32)
for (unsigned i = 1; i < N; i++) {
     …
}
```

Promises that the iteration that is 32 iterations away is the closest iteration that could be dependent on this iteration

Exercise 1

Relax Data Dependencies

Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
  - Lab 1
- Good design practices
- Advanced uses
- Application examples
  - Lab 2

Good Design Practices for Tasks

Follow guidelines here to allow AOC to effectively analyze and pipeline loops

- Recommendations
  - Avoid Pointer Aliasing
  - Construct “Well-Formed” Loops
  - Minimize loop-carried dependencies
  - Avoid complex loop exit conditions
  - Convert nested loops into a single loop
  - Declare Variables in the Deepest Scope Possible
  - Remove unnecessary operations out of loop

Pointer Aliasing

Multiple pointers could be pointing to the same content

```
kernel void mykernel(global float *a, global float *b)
```

```
   mykernel.setArg(0, (void *)&globalbuffer);
   mykernel.setArg(1, (void *)&globalbuffer);
```

- AOC assumes dependency between pointers that may be aliased
  - Limits pipeline parallelism
  - May increase loop initiation interval

```
for (…) {
     …=a[i];
     b[i]=…;
}
```

This is a dependency if pointer aliasing is possible

Avoid Pointer Aliasing

- Avoid using pointers that alias other pointers
  - Only one pointer variable is used to access the contents
- Use the `restrict` keyword whenever possible

```
__kernel void my_kernel (__global int * restrict A,
                         __global int * restrict B) {
```

  - C keyword specify to the compiler no aliasing for a pointer
  - Prevents AOC from assuming memory dependencies between pointers accesses
  - May cause functional errors if used with pointers that aliases other pointers
- Warning issued if kernel global pointer argument without `restrict`

```
temp.cl:3:40: warning: declaring kernel argument with no 'restrict' may lead to low kernel performance
```

Construct “Well-Formed” Loops

- Exit condition that compares against an integer bound
- Simple induction increment
  - E.g. one per iteration
  - Though not a requirement
- Easier for AOC to analyze efficiently and to maximize performance

Example of a “well-formed” nested loop structure

```
for( int i=0; i<N; i++) {
   //Statements
   for (int j=0; j<M; j++) {
     //Statements
   }
}
```

Minimize Loop-Carried Dependencies

- *Loop-carried dependency*: reading of data written by the previous iteration
  - Read operation needs to wait for the write operation of the previous iteration
  - May decreases the performance of the kernel

```
for( int i=1; i<N; i++) {
     A[i] = A[i-1] + i;
}
```

- Key to single-work item kernel performance
- AOC performs static dependency analysis of loops
- AOC assume dependency if it cannot deduce dependencies due to use of unknown variables or complex addressing

Minimize Loop-Carried Dependency Guidelines

- Avoid pointer arithmetic

```
int t = *(A++);
*A = t;
```

Here, A is a pointer. Avoid this.

- Avoid complex array indexes
  - Avoid nonconstants in array indexes.
    - E.g. A[K+i] where i is the loop index and K is an unknown variable
  - Avoid multiple index variables in the same subscript location
    - E.g. A[i+2*j] where i and j are loop indexes for a double nested loop
      - A[i][j] can be analyzed much more efficiently (different subscripts)
  - Avoid nonlinear indexing
    - E.g. A[i & c]
- Use loops with constant bounds in the kernel when possible

Avoid Complex Loop Exit Conditions

- Loop exit condition analyzed to determine if subsequent loop iterations can enter loop pipeline
  - Can be a dependency

- Complex exit conditions that require accesses to memory and/or perform complex operations delays the launch of subsequent iterations
  - Decreasing loop pipeline performance

Convert Nested Loops into Single Loop

Combine nested loops to save resources and improve performance

- Consider using the `loop_coalesce` pragma

Nested Loop

```
for(i=0; i<N; i++)
{
   //statements
   for (j=0; j<M; j++)
   {
     //statements
   }
}
```

Converted Single Loop

```
for( i=0; i< N*M; i++)
{
   //Statements
}
```

Loop Coalescing Notes

Rule of thumb: coalesce loops when possible

- Nested loops have more logic than a coalesced loop
  - Saves area
- Nested loops have more latency than a coalesced loop
  - More computational overhead
  - Coalescing may save a few cycles

`loop_coalesce` Pragma

Directs compiler to coalesce nested loops into a single loop

- Helps reduce overhead needed for loops
  - Reduces area and latency of component
- In certain cases may lengthen critical loop II

```
#pragma loop_coalesce
for (…) {
     for (…) {
          …
     }
}
```

Compiler attempts to coalesce all nested loops

Nesting Level

```
#pragma loop_coalesce 2
for (A)
     for (B)
          for (C)
     for (D)
```

Compiler attempts to coalesce only loops A, B, and D

Declare Variables in the Deepest Scope Possible

Declare variables in the deepest scope possible

- Minimizes data dependencies and hardware usage
- Avoid preservation of variables across loops that don't use it

```
1  int a[N];
2  for (int i = 0; i < m; ++i) {
3     int b[N];
4     for (int j = 0; j < n; ++j) {
5           // statements
6     }
7  }
```

- Declare `b` (line 3) just outside the loop where it's used
  - Unless data required to be maintained across the outer loop
  - Array `a` (line 1) requires more resources to implement than array `b`

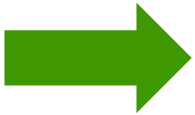
Move Unnecessary Operations Out of the Loop

If operation contributes to II but doesn't need to be inside the loop, move it outside the loop

- Ensure functionality

Possible if operations are associative

```
data = input;
for (int i = 0; i < N; ++i)
{
      process(data);
      data=data*constant;
      …
}
```



```
data=input;
data=data*constant;
for (int i = 0; i < N; ++i)
{
      process(data);
      …
}
```

Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
  - Lab 1
- Good design practices
- Advanced uses
- Application examples
  - Lab 2

`ii` Pragma

Forces the loop to have a specific initiation interval (II) in clock cycles

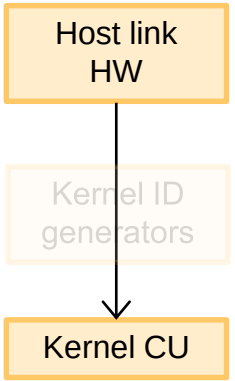
- `#pragma ii <N>`
  - Tell compiler to set the loop II to <N>
    - If specified II cannot be achieved, compile errors out
  - Usually used to specify a **higher II** than what the compiler chooses to increase f_{max}
    - Allows more pipelining on non-critical loops

```
#pragma ii 4
for (unsigned i = 1; i < N; i++)
{
     // Loop body
}
```

Reducing Kernel Hardware Overhead with `max_global_work_dim(0)`

- Single Work-Item Kernels are not dispatched across work-items/workgroups
- Kernel attribute `max_global_work_dim(0)` removes dispatch HW logic
  - Saves resources
  - Removes logic that generate threads IDs for specified kernel
    - global ID, local ID, group ID
  - Other number of dimensions values are allowed (up to 3)
    - But results in no resource savings

```
__attribute__((max_global_work_dim(0)))
__kernel void mykernel (…) {
    for(…
}
```



`max_global_work_dim(0)` Recommendation

Recommended to be used for **ALL** single work-item kernels (Tasks)

- Compiler does not perform this by default in order to conform to OpenCL™ standards

- Once set, multi-threaded (more than 1 work-item) launch of the kernels will result in error
- Once set, overhead omission reflected for the kernel in the HTML Area Report

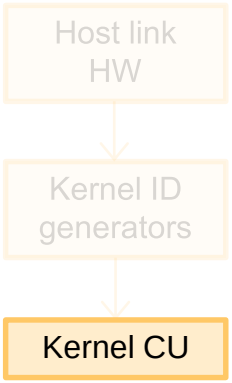
[-] gl_test (Logic: 2%)	6297 (1%)	8526 (1%)	72 (3%)	0 (0%)	
Function overhead	1570	1685	0	0	• Kernel dispatch logic.
[+] Block0 (Logic: 1%)	4727 (1%)	6841 (1%)	72 (3%)	0 (0%)	

Kernels That Runs Without the host

Mark kernels that runs automatically without the host with `autorun` attribute

```
__attribute__((autorun))
```

- Starts kernel execution automatically once FPGA is configured without the host
  - Restarts automatically if it finishes execution
- Saves resources
  - Omits logic used for communication with the host
  - Omits logic that dispatches work-items (ID generators)



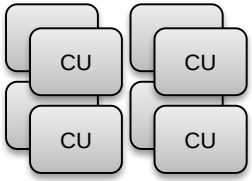
`autorun` Kernel Requirements

- Must use either the `max_global_work_dim(0)` or `reqd_work_group_size(X,Y,Z)` attribute
  - Fixed number of threads launched every time
- Must not have any argument
  - No communication with the host
- I/O channels not supported
  - Cannot guarantee data is not dropped at startup
  - Kernel-to-kernel channels allowed
- Typically for kernels that processes data from channels and write to channels

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
void kernel mykernel()
```

Creating an Array of Compute Units

Replicate kernel hardware with `num_compute_units(X,Y,Z)` attribute



- Creates X*Y*Z copies of kernel pipeline
  - Increases throughput
  - For NDRange kernels, CU's are used to execute multiple workgroups in parallel
  - Consumes X*Y*Z times more resources for that kernel compute unit
- With single work-item kernels, AOC allows customization of kernel compute units using the `get_compute_id()` function
  - Create compute ID dependent logic

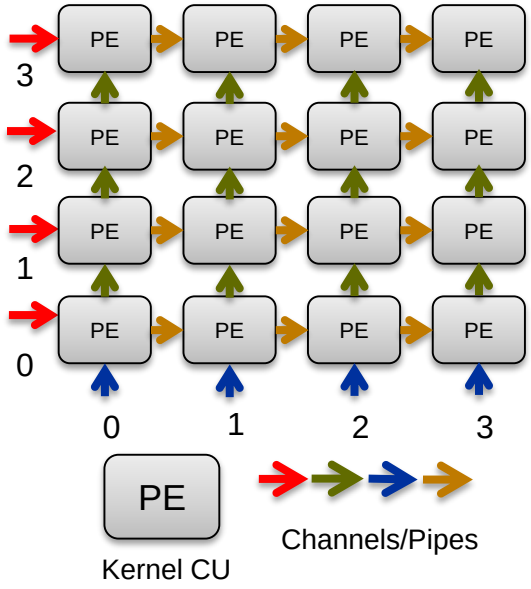
get_compute_id() Function Usage

- Each replicated compute unit assigned a compute ID
- `get_compute_id(`*`dim`*`)` call retrieves the unique index of each compute unit in the specified dimension during compilation
  - Compute IDs are static values
  - *`dim`*: 0 = X, 1 = Y, 2 = Z
- `autorun` and `max_global_work_dim(0)` attributes required!
- Alternative to replicating the kernel source code and specializing for each copy
- Allows compiler to generate unique hardware for each compute unit
  - e.g. `if (get_compute_id(0) == X)` then do something
  - Often used to customize computations or control flow

Example with `get_compute_id`

Using compute ID to determine channel usage

```
channel float4 ch_PE_row[3][4];
channel float4 ch_PE_col[4][3];
channel float4 ch_PE_row_side[4];
channel float4 ch_PE_col_side[4];

__attribute__((autorun))
__attribute__((max_global_work_dim(0)))
__attribute__((num_compute_units(4,4)))
kernel void PE() {
     float4 a,b;
     if (get_compute_id(0)==0) //First PE of row
          a = read_channel(ch_PE_col_side[col]);
     else
          a = read_channel(ch_PE_col[row-1][col]);

     if (get_compute_id(1)==0)
     …
```



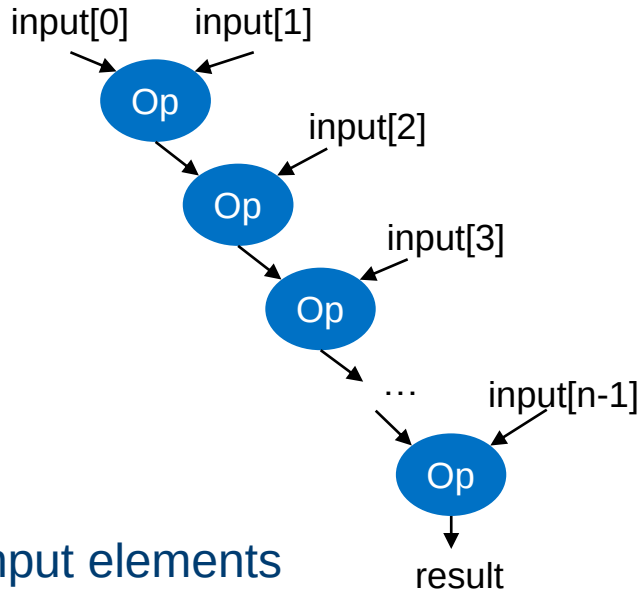
Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Good design practices
- Resolving common dependency issues
  - Lab 1
- Advanced uses
- Application examples
  - Reduction, Scan, TDFIR Example, Systolic Arrays
  - Lab 2

Reduction

Reduce [(set of input elements), operator]

- Operator Requirement
  - Binary: 2 input, 1 output
  - Associative:
    - (A op B) op C = A op (B op C)
  - Example: *, +, min, logical OR, bitwise AND
- Result: cumulatively apply the operator to all the input elements
- Serial implementation of reduce:



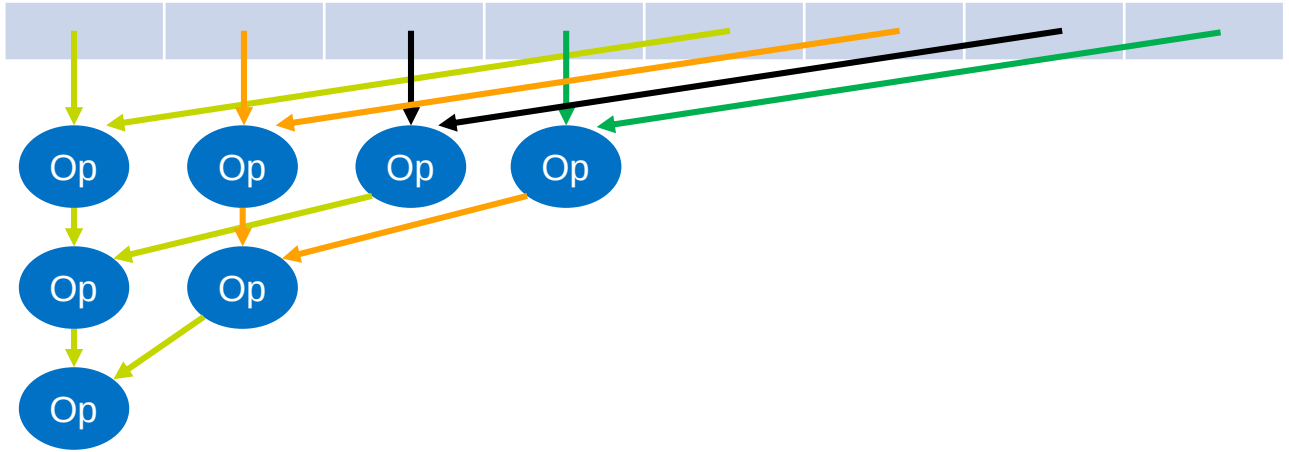

```
sum=0;
for (i=0; i<length; i++)
    sum = sum + input[i];
```

SIMD Parallel Reduce

Idea: regroup data

- Take advantage of associative property and perform as much operations as possible in parallel
- (a+b)+c)+d ➔ (a+b) + (c+d)

Algorithm:



NDRange OpenCL™ Parallel Reduce

```
__kernel void reduce(   __global float* buffer,
            __local float* scratch,
            __const int length,
            __global float* result)
{
      int global_index = get_global_id(0);
      int local_index = get_local_id(0);

      scratch[local_index] = buffer[global_index];
      barrier(CLK_LOCAL_MEM_FENCE);

      for (int offset = get_local_size(0) / 2; offset > 0; offset >>= 1) {
            if (local_index < offset )
                  scratch[local_index] += scratch[local_index + offset];

            barrier(CLK_LOCAL_MEM_FENCE);
      }
      if (local_index == 0)
            result[get_group_id(0)] = scratch[0];
}
```

Load data in to local memory

Loop through offsets, starting from the largest

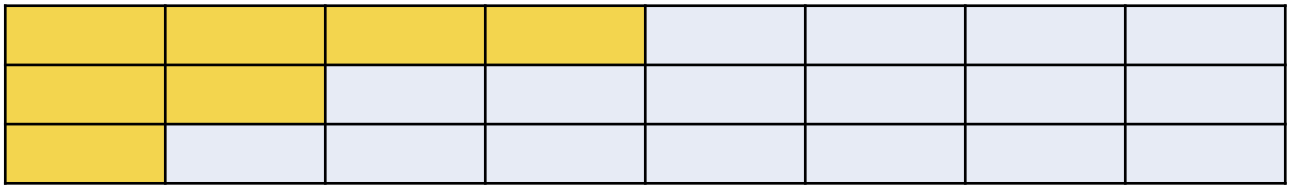
Perform Operation

Make sure ops for all workitems are done

Commit final result for workgroup to memory

Parallel Reduction Issues

- SIMD utilization on vectored architectures
  - Less than 1/3 of SIMD operators actually doing any useful work



- Other ways to improve performance
  - Two Stage Reduction
    - Perform serial reduction on independent groups of data in parallel, then parallel reduction
    - More complex to implement

Reduction with Single Work-Item Kernels

- Write reductions in a natural way
- AOC compiler generates efficient hardware by pipelining the loop

```
__attribute__((max_global_work_dim(0)))
kernel void reduce (  global int *a,
                      int n,
                      global int *result)
{
     int local_result = 0;
     for (int i=0; i<n; i++) {
          local_result += a[i];
     }
     *result = local_result;
}
```

Scan

- Scan [(set of input elements), operator]
  - Binary associative operator
  - Result: cumulative application of the operator up until that point
  - AKA: Prefix sum

Input	a_0	a_1	a_2	a_3	…	a_{n-1}
Output	a_0	a_0 op a_1	a_0 op a_1 op a_2	a_0 op a_1 op a_2 op a_3	…	a_0 op a_1 op a_2 op…a_{n-1}

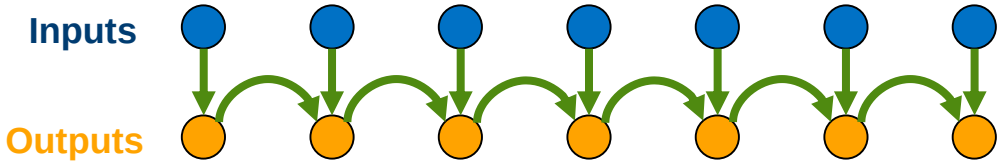
- Example:

Input	2	1	4	6	3	4	9	5
Max Scan Output	2	2	4	6	6	6	9	9
Sum Scan Output	2	3	7	13	16	20	29	34

Scan Implementation

```
output[0] = input[0];
for (i=1; i<length; i++) {
     output[i] = input[i] op output[i-1];
}
```

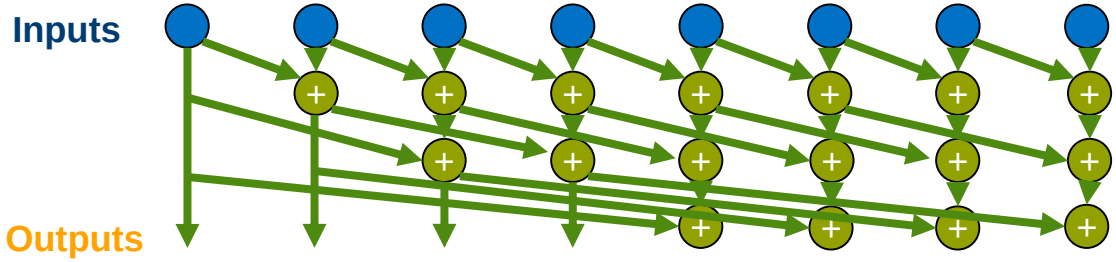
- One of the most important parallel primitives
- Many algorithms display simililar behavior and can be expressed using scan
  - Therefore can be parallelized with scan if scan can be efficiently parallelized



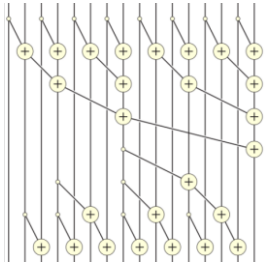
- Useful for
  - Histograms, graph reversal, data compression, collision detection, sort, etc

Efficient NDRange Parallel Implementation

- Hillis/Steele Inclusive Scan
  - Starting with Step 0: on Step i, add yourself to your 2^i left neighbor, if such neighbor doesn't exist just copy the existing value
    - Step efficient, NOT work efficient, better if you have more processors then data



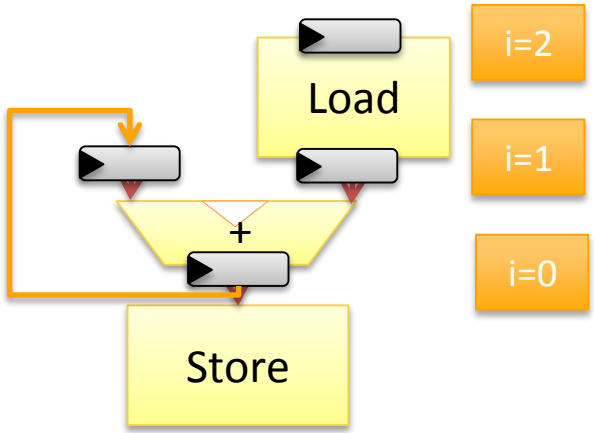
- Blelloch Exclusive Scan
  - Reduce (calculate partial sum) then downsweep (combine them)
  - More work efficient, better if you have more data then processor



Scan with Single Work-Item Kernels

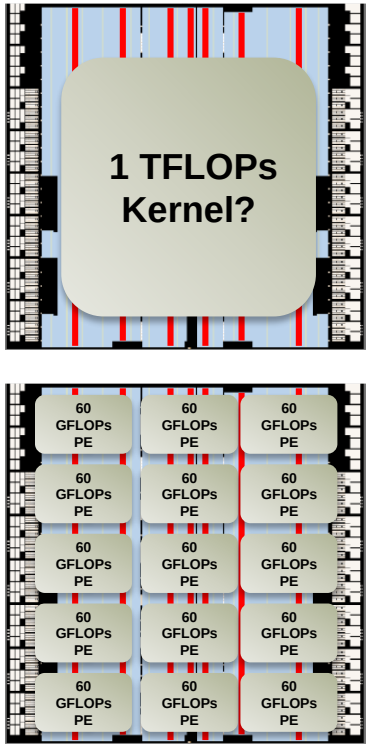
- Write scan in a natural way
- AOC compiler generates efficient hardware by pipelining the loop
  - Achieve the work efficiency of a serial implementation but with pipeline parallelism
  - *Very* work-efficient

```
__attribute__((max_global_work_dim(0)))
kernel void sum_scan (global int *input,
                      int length,
                      global int *output)
{
     output[0] = input[0];
     for (i=1; i<length; i++) {
          output[i] = input[i] + output[i-1];
     }
}
```



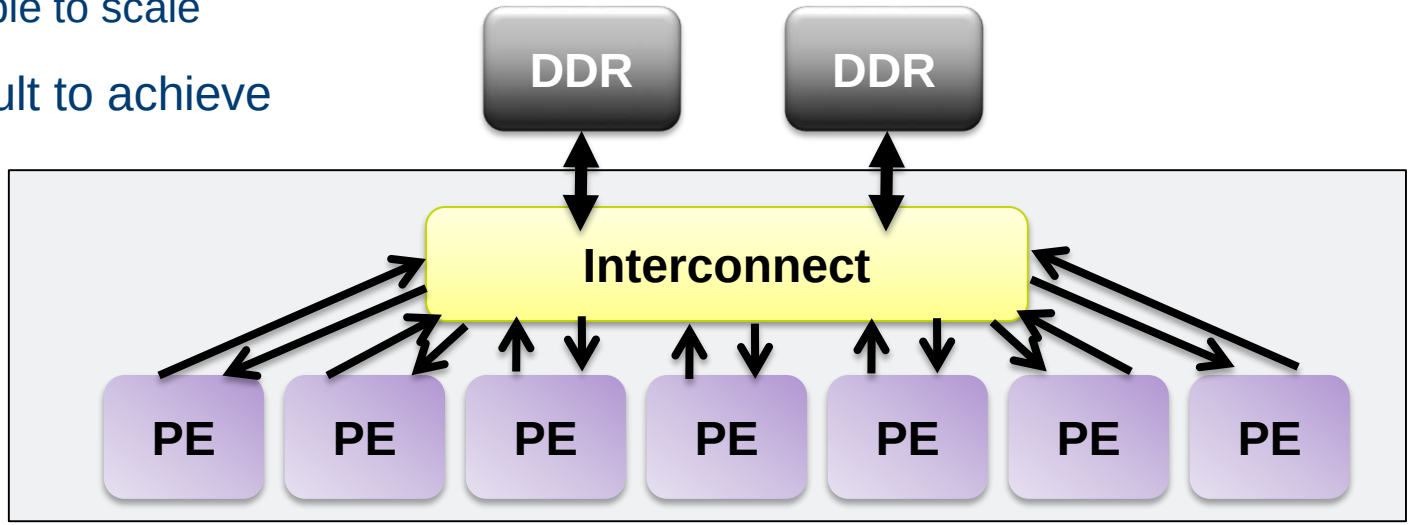
Systolic Array Motivation

- Key to peak device performance
  - Highest possible frequency / Keep FPGA resource busy
- Approach 1: Single large kernel
  - “CPU coding style”, difficult to generate efficient HW
- Approach 2: Utilize small kernels
  - Easier to optimize and generate efficient HW
  - Then replicate kernels
  - “FPGA coding style”, Divide-and-conquer
  - Call each of these Processing Elements Kernels (PE)



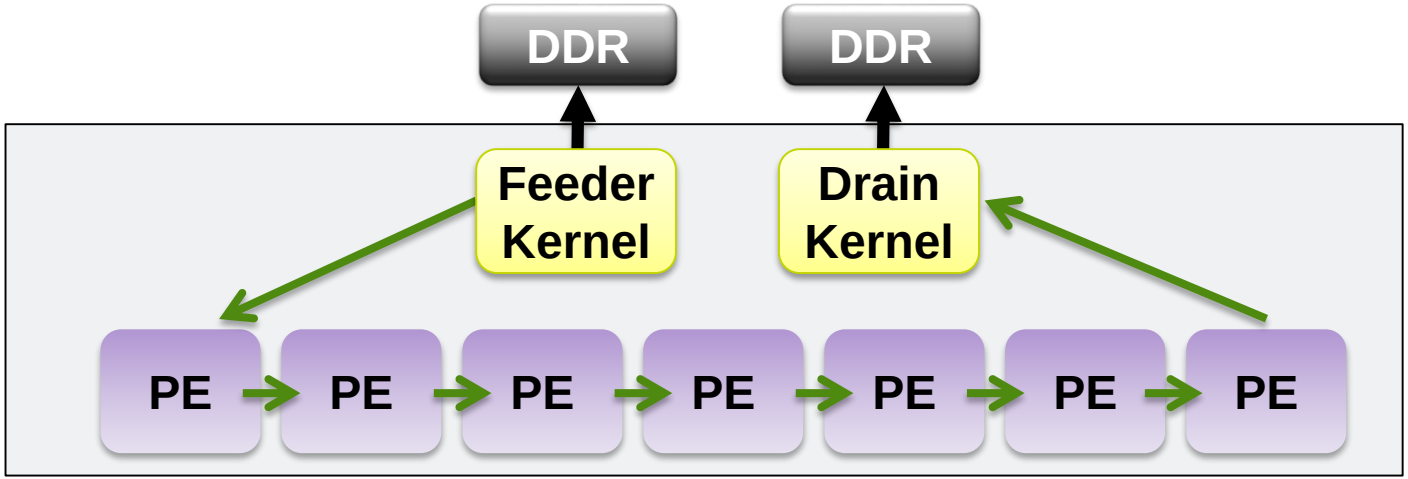
Data Transfer with Traditional OpenCL™

- System performance can only be achieved with effective memory usage
- Need throughput to feed all Processing Elements
  - Able to scale
- Difficult to achieve



Effective Kernel-to-Kernel Communication

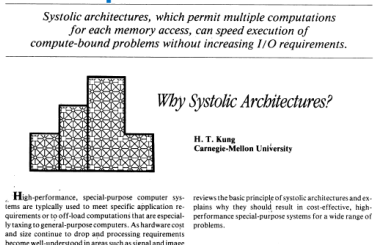
- Take advantage of the customizable on-chip bandwidth of FPGAs
- Use channels / pipes!
- Use FPGA's ability to partition data in off-chip memory



Systolic Arrays

Using a network of processing units each receives data from upstream neighbors, processes it, and passes it downstream

- Concept invented in 1982



Systolic architectures, which permit multiple computations for each memory access, can speed execution of compute-bound problems without increasing I/O requirements.

Why Systolic Architectures?

H. T. Kung
Carnegie-Mellon University

High-performance, special-purpose computer systems are typically used to meet specific application requirements or to off-load computations that are especially taxing to general-purpose computers. As hardware cost and size continue to drop and processing requirements become well-understood in areas such as signal and image

reviews the basic principle of systolic architectures and explains why they should result in cost-effective, high-performance special-purpose systems for a wide range of problems.

- Useful for many applications
  - Filters, FFTs, convolutions, correlation, matrix “anything”, sorting, searching, etc
- AOC has all the tools to implement an effective systolic array architecture
  - Custom implementation of kernel compute units
  - Channels / pipes
  - num_compute_unit / get_compute_id
  - etc.

Exercise 2

Optimize a Single Work-Item Kernel

Optimizing Local Memory

Local Memory and Hardware

- Most efficient way to communicate between work-items in a workgroup
  - Smaller capacity than global memory
  - Lower latency than global memory
    - Random access in 2 cycles
  - Very high bandwidth ~8TB/s
- Implemented using M20K blocks or MLABs in FPGA
  - 53 MB of M20K RAM in Arria® 10 platform
  - 12.7 MB of MLAB RAM in Arria 10 platform

Arria® 10 GX FPGA Development Kit

Device: 10AX115S2F45I1SG

	Product Line	GX 1150
Resources	Part number reference	10AX115
	LEs (K)	1,150
	Adaptive logic modules (ALMs)	427,700
	Registers	1,708,800
	M20K memory blocks	2,713
	M20K memory (MB)	53
	MLAB memory (MB)	12.7
	Hardened single-precision floating-point multipliers/adders	1,518/1,518
	18 x 19 multipliers	3,036
	Peak GMACS	3,340
	GFLOPS	1,366

On-chip Memory Systems

- "Local" and some "private" memories use on-chip RAM resources
  - Much better performance than global memories
- Local memory system is customized to your application at compile time
  - Dependent on data type and usage
  - Banking configuration (number of banks, width), and interconnect customized to minimize contention
  - Big advantage over fixed-architecture accelerators
    - If your code is optimized for another architecture, undo the fixed-architecture workaround

Statically Allocating Local Memory

Statically allocate local pointer

Cache data in local memory

Barrier ensures all work-items in the workgroup have loaded data into cache before moving on. Discussed in an upcoming slide.

```
__kernel void mykernel (__global float* ina, …) {

  __local float ina_local[64];

  ina_local[get_local_id(0)] = ina[get_global_id(0)];

  barrier(CLK_LOCAL_MEM_FENCE);
  …
  // Usage of any element of ina_local
}
```

Dynamically Allocated Local Memory

- Not preferred
- For Intel® FPGA, a static amount is always allocated at compile time
  - Dynamically allocated size must be <= statically allocated size

Kernel Code

```
__kernel void mykernel (__global float* ina, __local float *ina_local…) {

   ina_local[get_local_id(0)] = a[get_global_id(0)];
   barrier(CLK_LOCAL_MEM_FENCE);
   …
   // Usage of any element of ina_local
}
```

local memory pointer argument

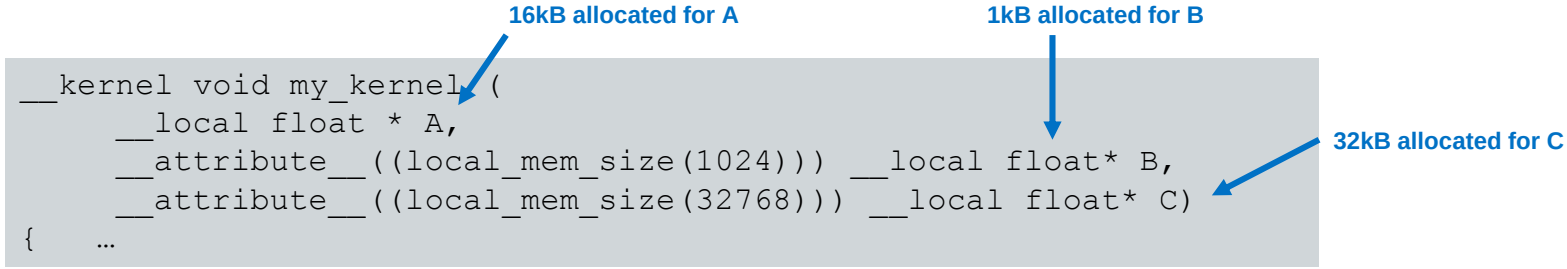
Host Code

```
cl::Kernel::setArg(0, &global_mem_buffer);
cl::Kernel:setArg(1, NULL)
```

arg_value must be NULL when argument is local!

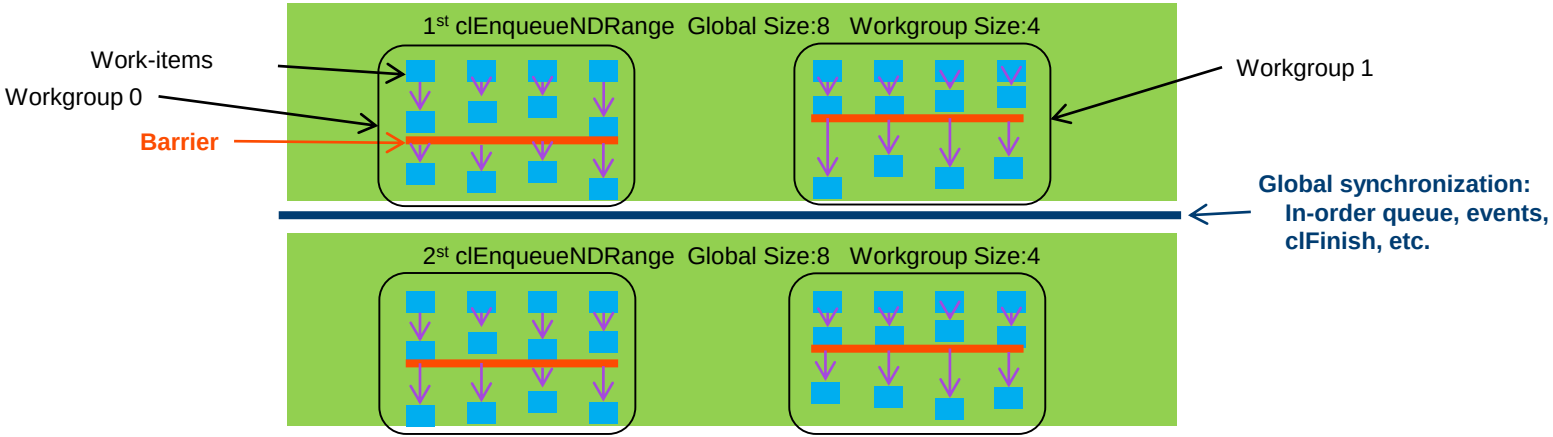
Local Memory Kernel Argument Allocation

- Physical pointer kernel arguments size set at compile time
- By default 16kB of local memory is allocated for each variable
- `cl::Kernel::setArg()` cannot request data larger than the statically allocated size
- Use `local_mem_size` attribute to manually set size, **must be power of 2**
  - Specify a pointer size in bytes



Local Memory Access Synchronization

- OpenCL™ device relaxed consistency
  - Within a work-item, operations are ordered predictably
  - Among work-items in a workgroup, memory guaranteed to be consistent at barrier
  - Between workgroups, no guarantee until completion of kernel execution



Memory Fences

- Ensure reads and/or writes issued before the fence will complete before reads and/or writes after the fence
  - Apply to local memory, global memory, or channels
  - Orders loads and stores within a single work-item executing a kernel
- API
  - `void read_mem_fence(cl_mem_fence_flags flags)`
    - Orders only loads
  - `void write_mem_fence(cl_mem_fence_flags flags)`
    - Orders only stores
  - `void mem_fence(cl_mem_fence_flags flags)`
    - Orders loads and stores

Valid fence flags
`CLK_LOCAL_MEM_FENCE`
`CLK_GLOBAL_MEM_FENCE`
`CLK_CHANNEL_MEM_FENCE`

Barriers

- Require all work-items in a workgroup to execute function before continuing
  - Implies `mem_fence`, waits until access to memory prior to the barrier are committed
  - Enable sharing of data
    - Usually in local memory
  - Each barrier must be encountered by all or none of the work-items in a workgroup
    - Applies to conditional barriers as well
    - In a loop, all or none of the work-items should hit barrier for every iteration
    - Otherwise kernel will hang
- `void barrier(cl_mem_fence_flags flags)`
  - Same flag options as memory fence calls

Workgroup Boundary and Caching

May need to cache more data than the workgroup size

- Example: 3 element moving average

```
__kernel void MovingAv (__global float* ina, __global float* output)
{
      int local_id = get_local_id(0);
      int global_id = get_global_id(0);
      __local int ina_local[LOCAL_GROUP_SIZE+2];

      ina_local[local_id] = ina[global_id];
      if ((local_id == get_local_size(0)-1) && (global_id < get_global_size(0)-1)) {
            ina_local[local_id+1] = ina[global_id+1];
            ina_local[local_id+2] = ina[global_id+2];
      }
      barrier(CLK_LOCAL_MEM_FENCE);
      if (global_id < get_global_size(0) - 2)
            out[global_id]=(ina_local[local_id] + ina_local[local_id+1] + ina_local[local_id+2])/3;
}
```

Local memory has to hold workgroup size +2 elements

Every work-item caches one element

Last work-item in each workgroup cache 2 extra value, except last group

Take average for each work-item, except very last two

Workgroup Boundary and Caching Diagram (1)

```
ina_local[local_id] = ina[global_id];
```

Global ID	0	1	2	3	4	5	6	7	8	9	10	11
Group ID	0	0	0	0	1	1	1	1	2	2	2	2
Local ID	0	1	2	3	0	1	2	3	0	1	2	3
Value	00	11	22	33	44	55	66	77	88	99	AA	BB

Local ID	0	1	2	3		
Value	00	11	22	33		

Local ID	0	1	2	3		
Value	44	55	66	77		

Local ID	0	1	2	3		
Value	88	99	AA	BB		

get_work_dim () = 1
get_local_size(0) = 4
get_global_size(0) = 12

Workgroup Boundary and Caching Diagram (2)

```
if ((local_id == get_local_size(0)-1) && (global_id < get_global_size(0)-1)) {
      ina_local[local_id+1] = ina[global_id+1];
      ina_local[local_id+2] = ina[global_id+2];
}
```

Global ID	0	1	2	3	4	5	6	7	8	9	10	11
Group ID	0	0	0	0	1	1	1	1	2	2	2	2
Local ID	0	1	2	3	0	1	2	3	0	1	2	3
Value	00	11	22	33	44	55	66	77	88	99	AA	BB

Local ID	0	1	2	3		
Value	00	11	22	33	44	55

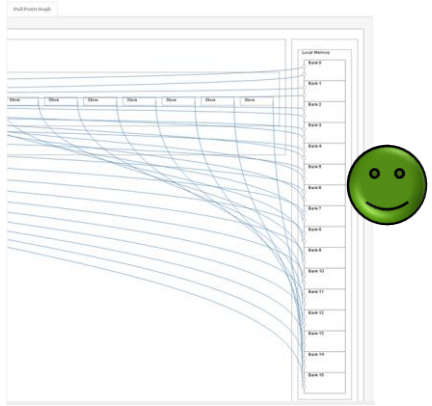
Local ID	0	1	2	3		
Value	44	55	66	77	88	99

Local ID	0	1	2	3		
Value	88	99	AA	BB		

get_work_dim () = 1
get_local_size(0) = 4
get_global_size(0) = 12

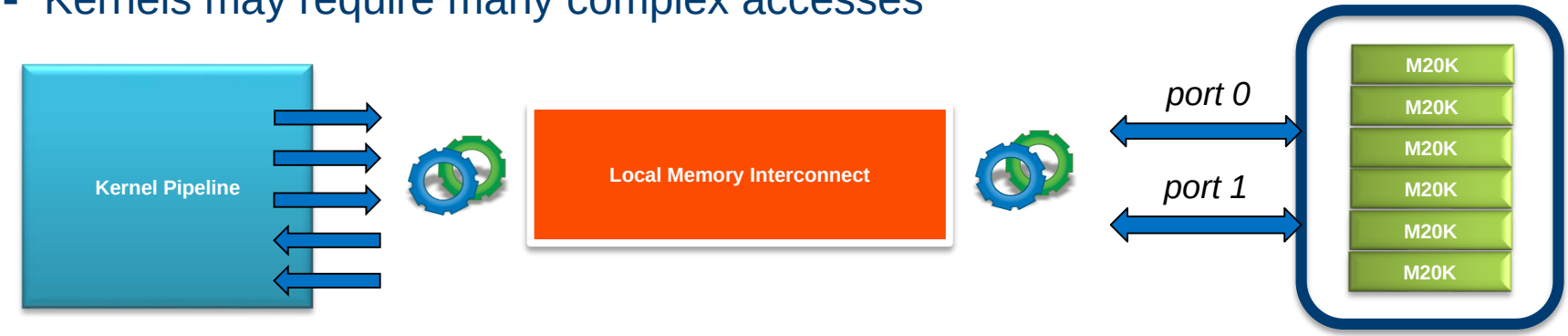
Efficient On-chip Memory Systems

- Loads/stores with **stall-free** properties ideal
  - Have fixed latency
  - Access latency is lower
  - Use less resources
  - Can be included in stall-free execution regions of the pipeline
- Lead to simpler interconnect
  - No arbitration is needed
- Can be scheduled more efficiently
  - See discussions on dependencies



On-chip memory architecture

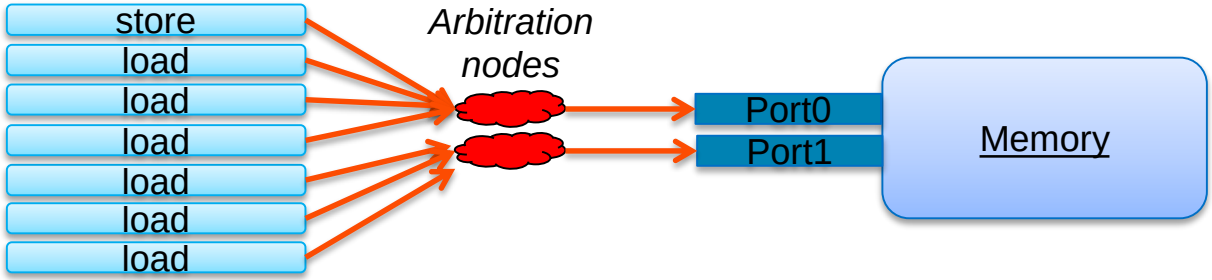
- Basic memory architectures map to dual-ported M20Ks
  - Concurrently accomodates `#loads + #stores ≤ 2`
- Kernels may require many complex accesses



- Compiler optimizes kernel pipeline, interconnect and memory system
  - Through *splitting*, *coalescing*, *banking*, *replication*, *double-pumping*, *port sharing*

Interconnect

- Interconnect includes access arbitration to memory ports



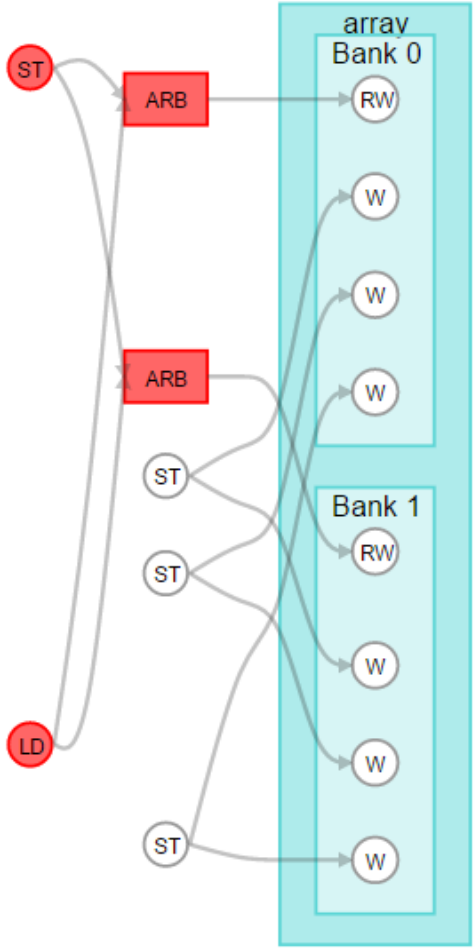
- With no optimization, sharing ports destroys performance
  - Pipeline stalls due to arbitration for concurrent accesses
  - Unless mutually exclusive accesses
- Key to high local-memory efficiency is stall-free memory accesses
  - Concurrent memory accesses can access memory without contention

Port Sharing: Arbitration

Arbitration logic is required if there are more read/write sites than ports available.

```
__kernel
void foo_arb(int ind1, int ind2,
    int ind3, int val1, int val2, int calc)
{
  __local int array[1024];
  array[ind1] = val1;
  array[ind1+1] = val1;
  array[ind2+1] = val2;
  array[ind2] = val2;
  calc=array[ind3];
}
```

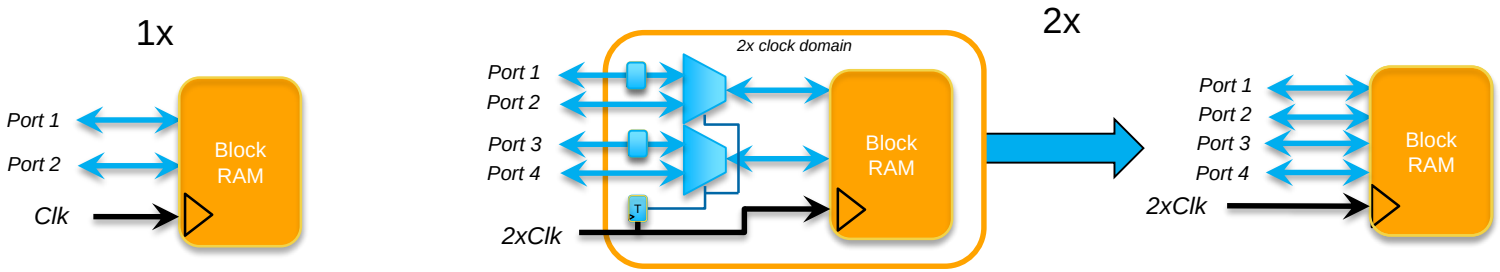
ST
ST
ST
ST
LD

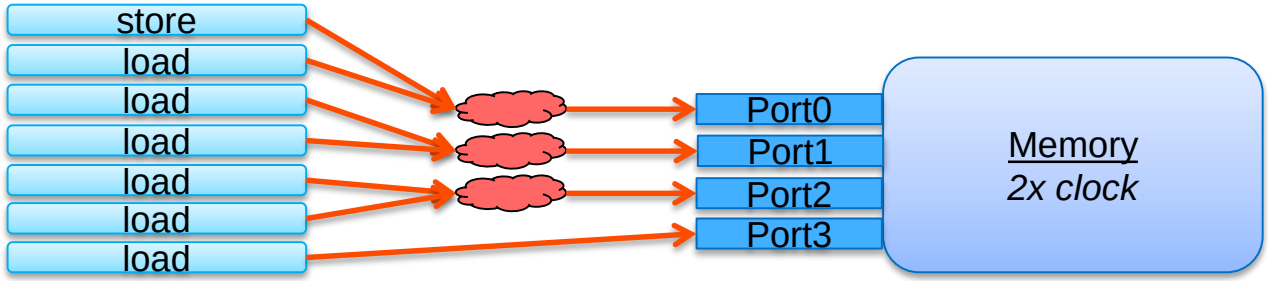


Compiler Choice of Local Memory System Geometry

- For each address space, the compiler iterates over multiple local memory configurations and chooses the “best” configuration
  - Number of banks
  - Bank width (determines the maximum coalescing width)
  - 1x vs. 2x clock memory
  - Replication possibility
- Uses a heuristic-based cost function, Priorities:
  1. Stall-free load/stores
  2. Fmax
  3. Resource Utilization

Automatic Double Pumping

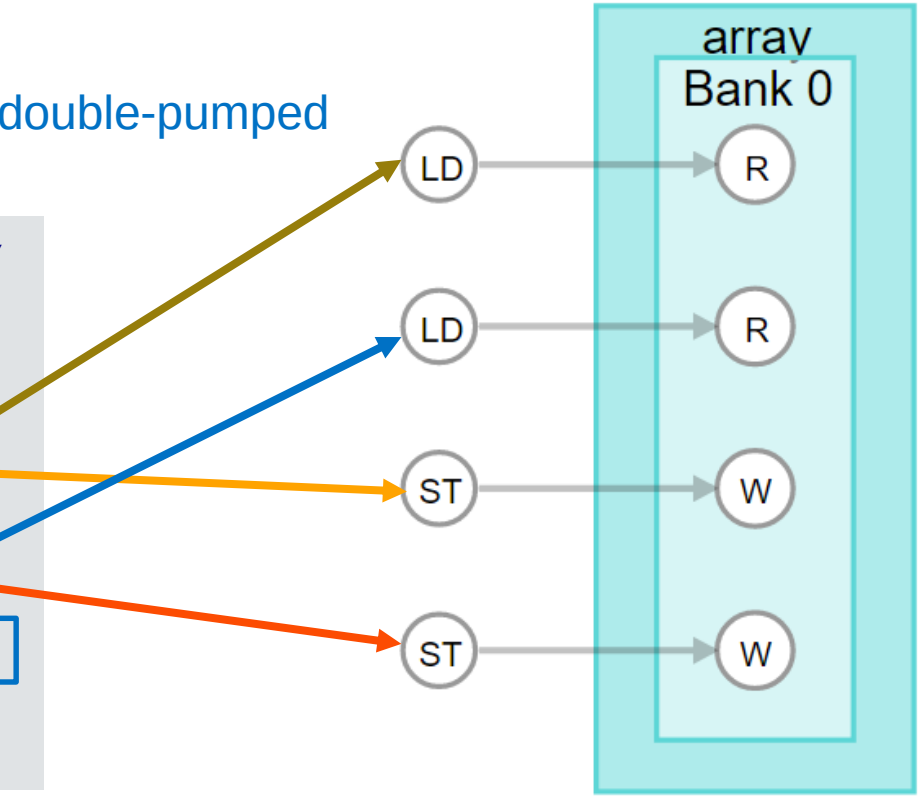




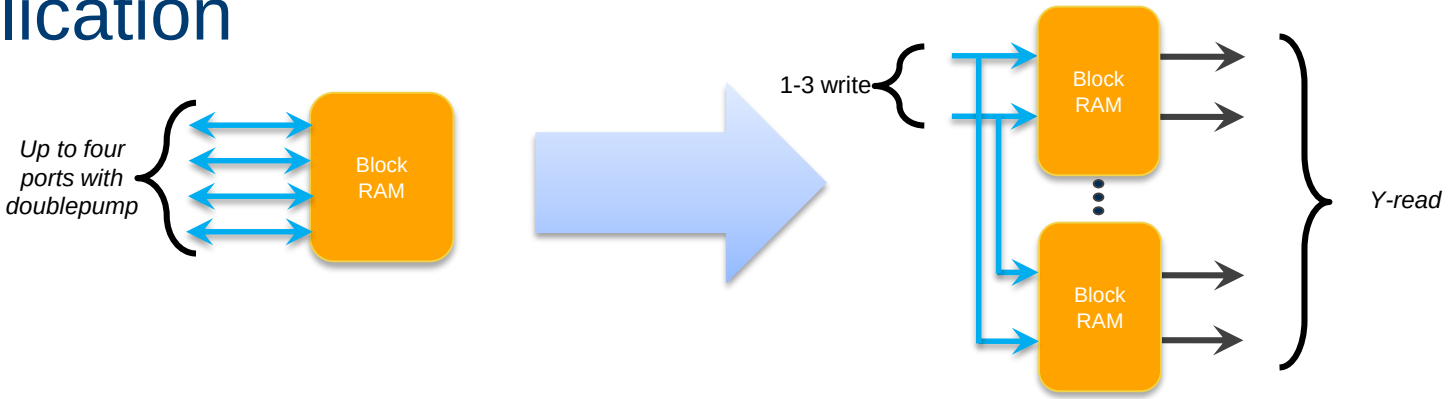
Double-Pumped Memory Example

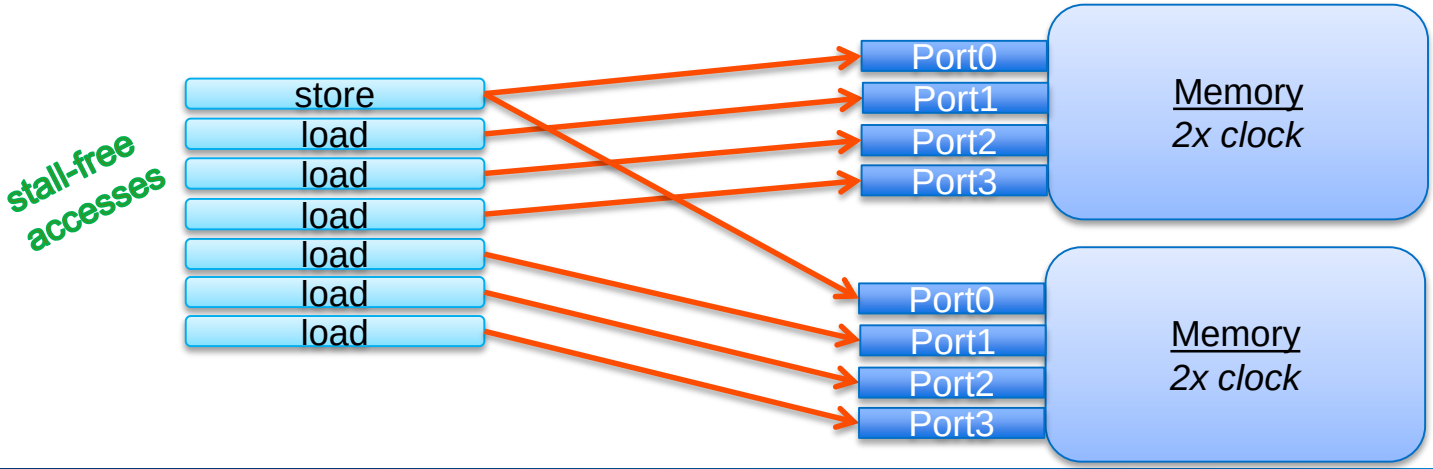
Compiler may automatically implement double-pumped memory

```
__kernel void bar (int ind1, int ind2,
         int val, int calc)
{
  __local int array[1024];

  array[ind1] = val;

  array[ind1+1] = val;

  calc = array[ind2] + array[ind2+1];
}
```



Replication





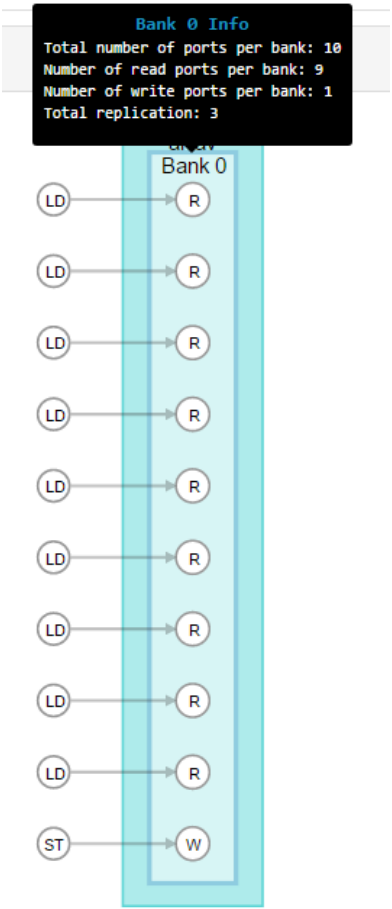
Replication Notes

- Stores must be connected to every replicated RAM and not suffer contention
  - Memory never replicated if there are > 3 simultaneous stores
- Works great in cases where 1 store used in conjunction with many loads
  - Numerous copies can be made for loads
- Pros:
  - Transparent to the user
  - Fmax not negatively impacted
  - Simpler interconnect

- Cons:
  - Entire memory system gets replicated (potentially large increase in M20Ks)

Local Memory Replication Example

```
__kernel
void foo_replication (int ind1, int ind2, int val, int calc) {
    __local int array[1024];
    int res = 0;

 ST array[ind1] = val;
    #pragma unroll
    for (int i = 0; i < 9; i++)
 LD     res += array[ind2+i];

    calc = res;
}
```



1 write port, 9 read ports
Up to 3 read ports, 1 write port per replicant
Therefore, replication factor = 3 needed for stall free accesses

Compiler Code Analysis

- Double pumping/replication done with minimal understanding of kernel pipeline
  - Just assume that ALL loads and stores are concurrent
- Compiler analyzes kernel code for more advanced optimizations
  - Based on access patterns and decomposition of the address

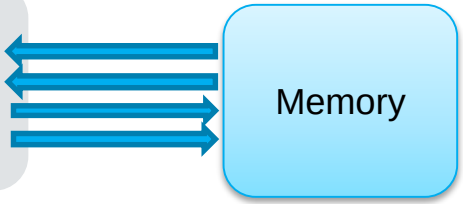
```
local float B[1024][32];
…
B[i][j] = …
```

- Example, `B[i][j]` accesses address =
  - `B + ((i * 32 + j) * sizeof(float))`
  - Access is always at a 32-bit boundary
  - More powerful information inferred from related accesses

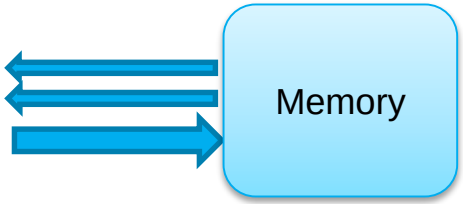
Static Coalescing

- Components often access consecutive addresses (variable A)

```
__kernel void example() {
   __local int A[32][2], B[32][2];
   …
   A[lid][0] = B[lid][0];
   A[lid][1] = B[lid + x][1];
```



- Code specifies 2 consecutive stores to array A
- Compiler merges consecutive memory accesses into a wider accesses
  - Leads to fewer ports used and therefore less contention
  - One wider store to A

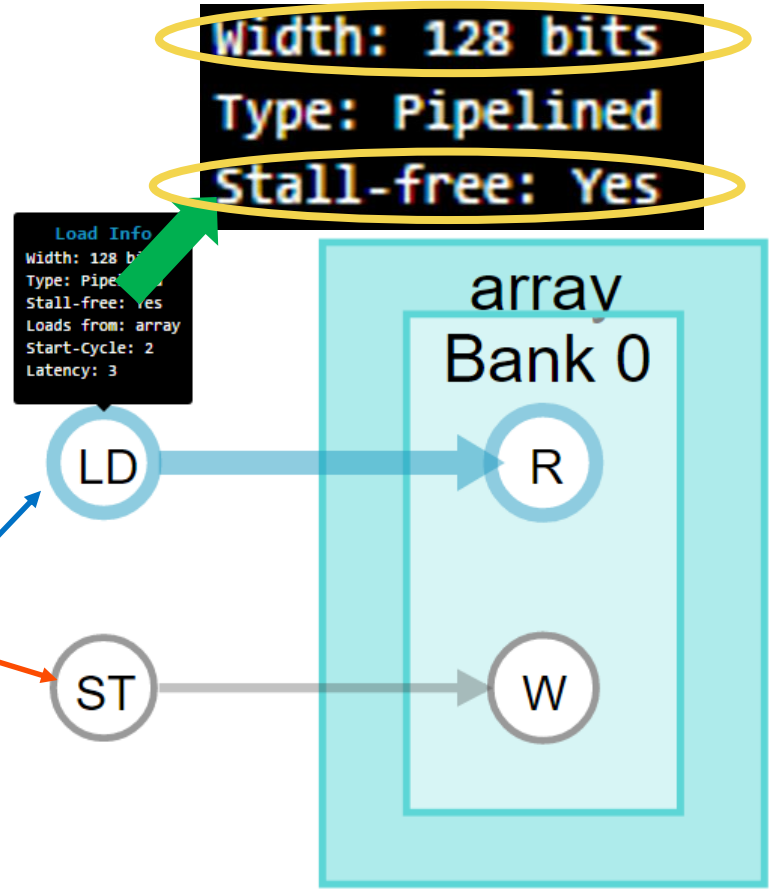


Automatic Static Coalescing Notes

- Fewer accesses competing for memory ports
  - Simpler memory systems, with fewer ports
- Before coalescing
  - 2 stores for array A → double pumping is needed
- After coalescing
  - 1 store for array A → 1x memory system is possible
  - Compiler will combine / widen automatically for you when beneficial
- Not all accesses have to match widths
- Leads to wider shallower memories

Coalescing

```
__kernel
void foo_coal (int ind1, int ind2, int val,
     int calc)
{
     __local int array[1024];
     int res = 0;

     #pragma unroll
     for (int i = 0; i < 4; i++)
          array[ind1*4 + i] = val;

     #pragma unroll
     for (int i = 0; i < 4; i++)
     res += array[ind2*4 + i];

  calc = res;
}
```



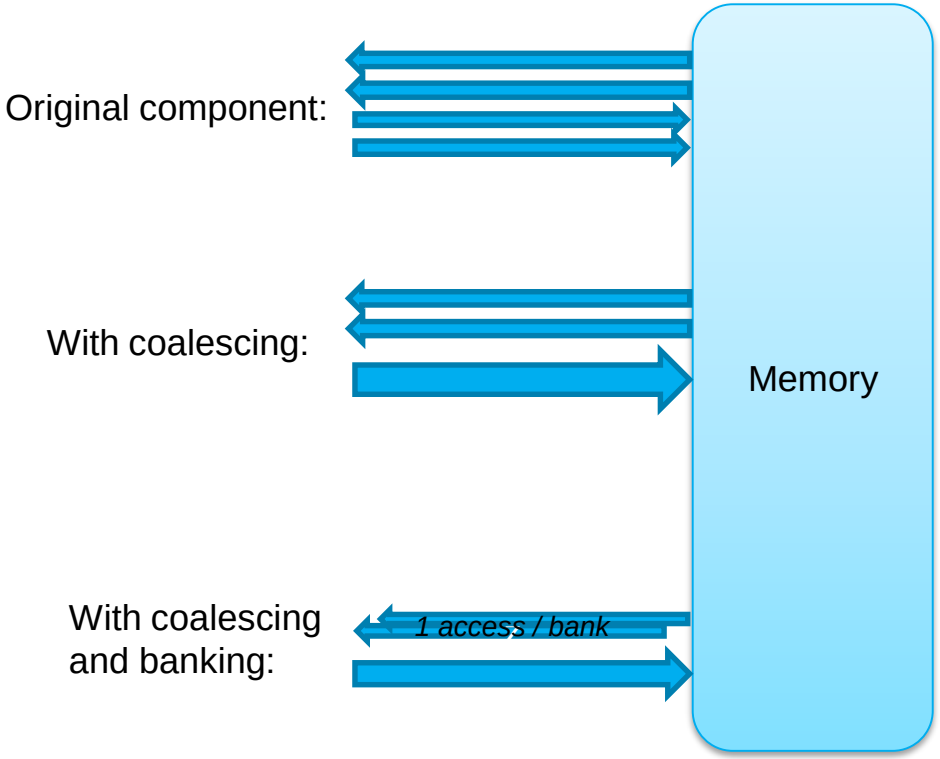
Automatic Banking

```
kernel void example() {
  local int A[32][2], B[32][2];
  …
  int lid = get_local_id(0);
  A[lid][0] = B[lid][0];
  A[lid][1] = B[lid + x][1];
  …
}
```

- Can the compiler do better for access to array B?
  - Currently 2 loads: `B[lid][0]` and `B[lid + x][1]`
  - The loads will access two disjoint partitions of the memory
- Solution: Compiler can partition memory into multiple banks to create concurrent accesses
  - Create separate memories for B with individual set of ports

Automatic Memory Coalescing and Banking

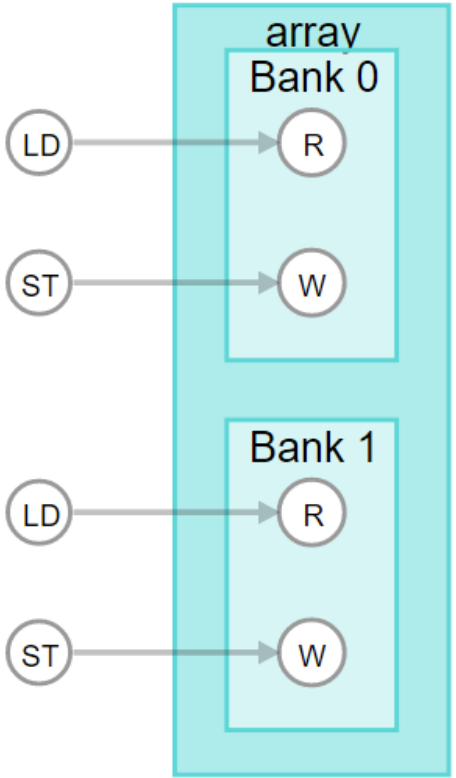
```
__kernel void example() {
  __local int A[32][2];
  __local int B[32][2];
  …
  A[lid][0] = B[lid][0];
  A[lid][1] = B[lid + x][1];
  …
}
```



Banking

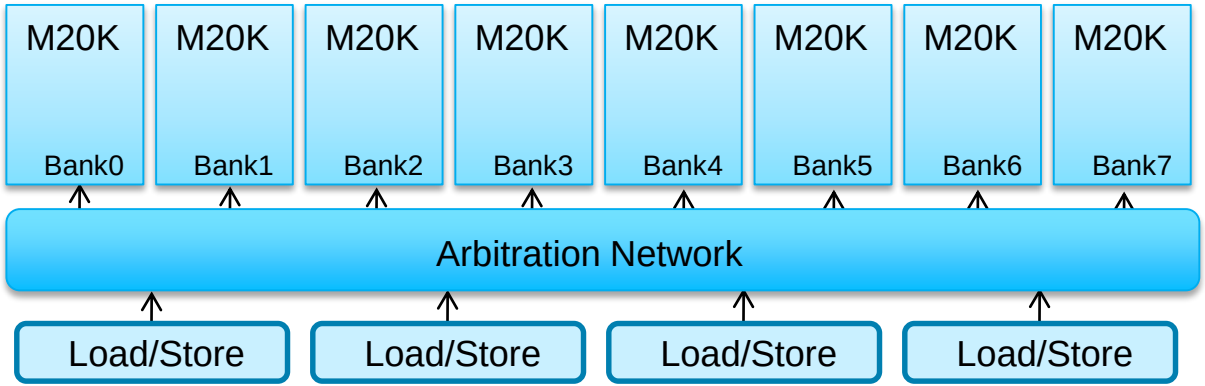
Use multiple banks on lower bits to implement the memory

```
__kernel
void foo_banking (int ind1, int ind2,
        int val1, int val2, int calc) {
  __local int array[1024][2];

  array[ind1][0] = val1;
  array[ind2][1] = val2;

  calc =  (array[ind2][0] +
           array[ind1][1]);
}
```



Automatic Banking

- Memory banked into logical banks
  - An N-bank configuration can handle 4*N-requests per clock cycle as long as each request addresses a different bank
    - Assuming x2 clock, dual port, and no replication
- Compiler uses lower address indexes/bits of memory access for bank selection

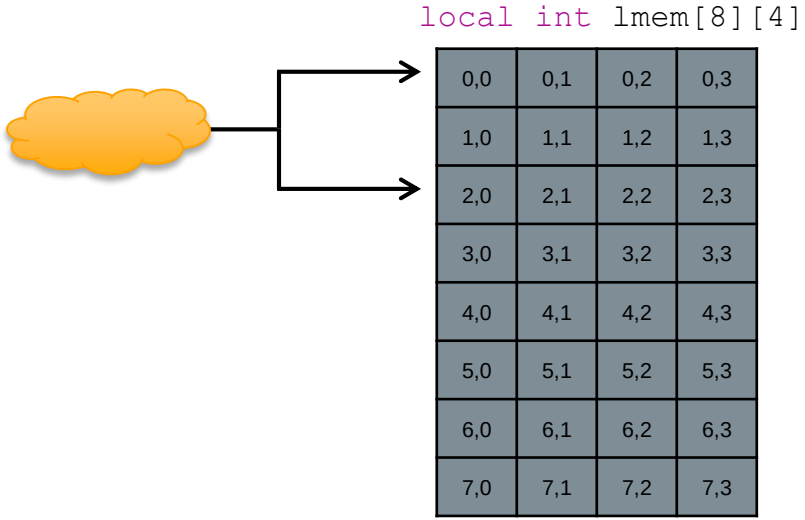


Local Memory Limitations

- Banking / coalescing implementation cannot be determined when lowest dimension is unknown
  - Here variable x is unknown and compiler must accept out-of-bound values

```
local int lmem[8][4];

#pragma unroll
for(int i = 0; i<4; i+=2)
{
  lmem[i][x] = …;
}
```

Unrolling creates two simultaneous accesses

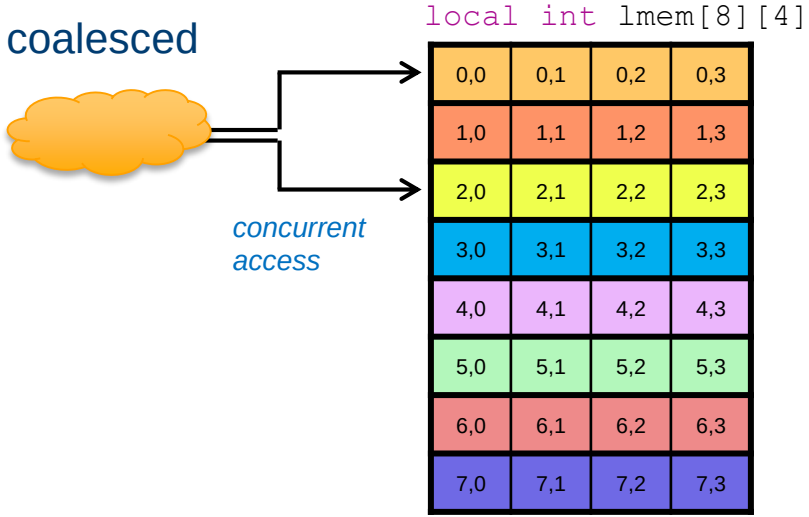


Overcoming Compiler Limitations (Coalescing and Banking)

- Assisting compiler analysis by restricting the range of lower index
  - By masking the variable x
- Wide memory: lowest dimension size dictates coalescing
  - Mask the lower index to bound it so it can be coalesced

```
local int lmem[8][4];

#pragma unroll
for(int i = 0; i<4; i+=2)
{
  lmem[i][x & 0x3] = …;
}
```

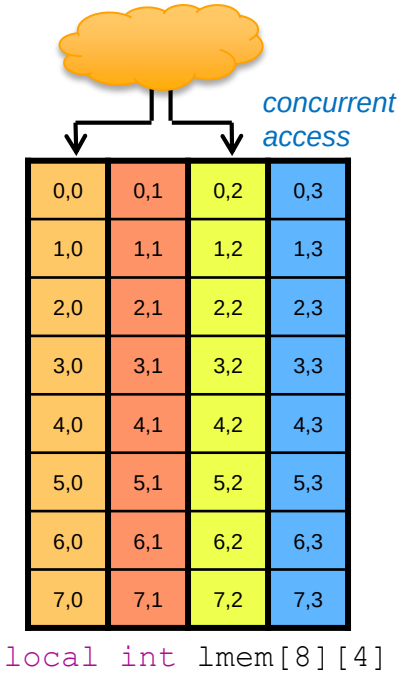
Banking of wider accesses achieved



Overcoming Compiler Limitations (Banking)

- Use known index as lower index to allow compiler to infer better banking structure
  - 4 banks created here

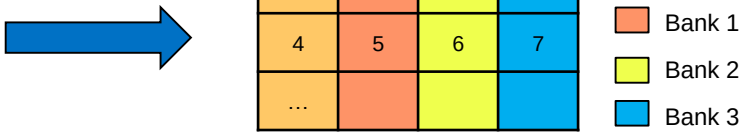
```
local int lmem[8][4];

#pragma unroll
for(int i = 0; i<4; i+=2)
{
    lmem[x][i] = …;
}
```



Memory Geometry Unrelated to Array Shape

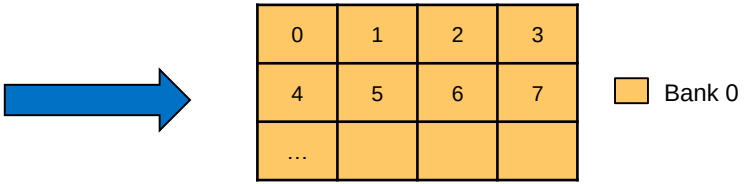
- Compiler creates memory geometry based on how an array is accessed, not how it's declared
- Array could be banked:

```
local int lmem[N];
```



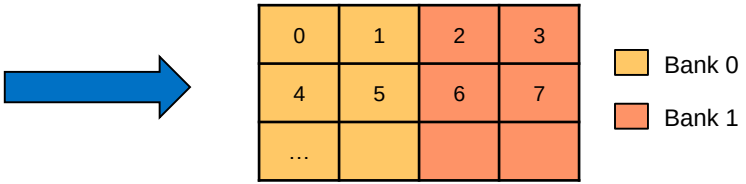
- Coalesced

```
local int lmem[N];
```



- Or coalesced and banked:

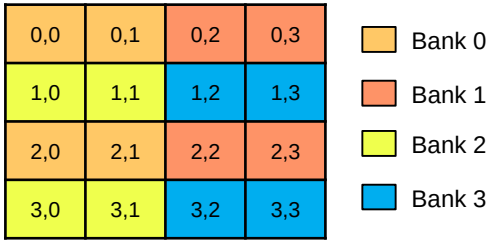
```
local int lmem[N];
```



2D Possible Geometries

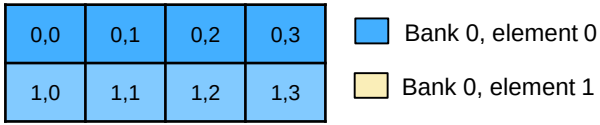
- 2D, coalesced and banked:

```
local int lmem[N][4];
```



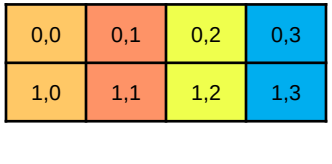
- 2D, coalesced

```
local int lmem[N][4];
```



- 2D, banked

```
local int lmem[N][4];
```



Automatic Memory Splitting

- Each private/local variable will ideally get its own memory system
  - Separate addresses and ports
  - If compiler can distinguish accesses to the separate variables
    - Compiler will split arrays only when it can prove that a pointer refers to exactly one array
- Benefits: Separate interconnect, dedicated loads and stores
- Impact of Not Splitting:
  - Complicates banking analysis, dependence analysis, etc.
  - More complicated local memory system, hence, stallable loads/stores.
  - More RAM usage if memory is replicated.

Automatic Memory Splitting Failure

- Pointer arithmetic and dynamic accesses may break the analysis
  - Pointer aliasing forces compiler to merge memory systems

Pointer temp can access either temp1 or temp2 content so temp1 and temp2 are merged

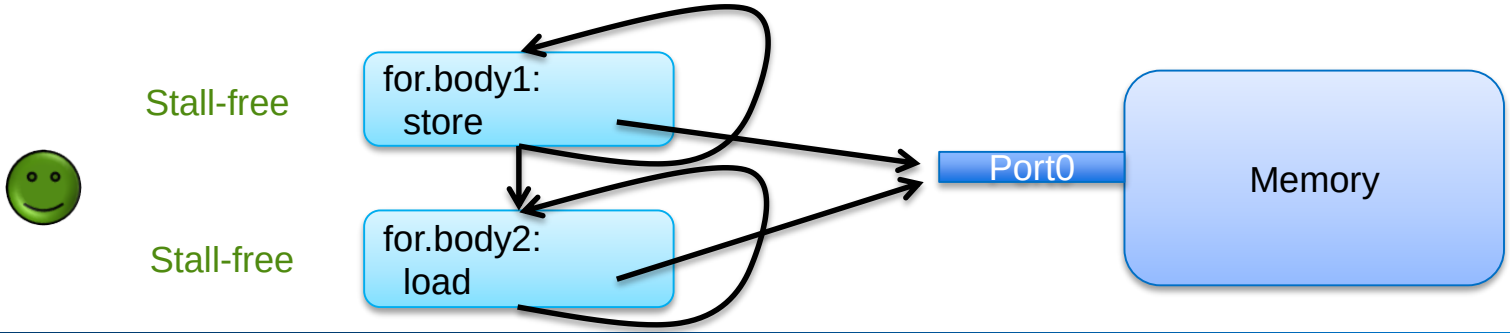
```
int temp1[20];
int temp2[20];
…
for(unsigned i = 0; i < 20; i++) {
      int* temp = ( i % 2 ) ? temp1 : temp2;
      *C += temp[i];
}
```

- FYI
  - Use dynamic indexing, instead of dynamic pointers, if possible
  - Avoid pointer aliasing

```
int temp1[20];
int temp2[20];
…
for(unsigned i = 0; i < 20; i++) {
      *C += ( i % 2 ) ? temp1[i] : temp2[i];
}
```

Automatic Port Sharing Optimization

- The compiler can decide that two loads/stores can be connected to the same port and STILL be stall-free
- Condition: accesses are mutually exclusive, i.e. not active in the same cycle.
  - In single-work item kernels, 2 loads/stores in 2 separate loops can share a port
    - if there is no outer loop or outer loop is not pipelined
  - 2 loads/stores in the same basic block (branch-free section) can never share ports



Memory Dependencies and Loop Pipelining

- Compiler ensure accesses from the same thread respect program order
- Loop dependencies may introduce bottlenecks for single work-item kernels
  - Memory operation will wait until a dependent memory operation completes
    - Due to latency associated with memory accesses
    - Could impact loop initiation interval (II)
  - Compiler indicates memory dependencies in the optimization report
- What can be done:
  - Understand what situation can be efficiently handled by the compiler
  - Ensure that the compiler doesn't assume false dependencies
  - Override dependency analysis with `#pragma ivdep` (covered earlier)

Minimize Impact of Dependencies for Loop Pipelining

- Unless proven otherwise, compiler assumes dependency and acts accordingly
  - Impact of dependencies is lower if memory system is stall-free
  - Read after write (control dependency) can be fully resolved by the compiler
    - Take advantage M20K behavior

Local Memory in the Area Report

- Many different local memory properties shown in HTML area report
  - Overall state:
    - Optimal : Stall-free, no replication or replication did not use extra block RAM
    - Good but replicated: Stall-free
    - Potentially inefficient: Possible stalls
  - Total size, replication factors, stallable/stall-free, merging, banking, # reads + writes
  - Full details of each reported property in *Best Practices Guide*
  - Private variables implemented in on-chip RAM reported as local

free_replication.cl:9 (lmem)	0	0	1	0	• Local memory: Optimal. Requested size 512 bytes (rounded up to nearest power of 2), implemented size 1024 bytes, replicated 2 times total, stall-free, 1 read and 1 write. Additional information: - Replicated 2 times to efficiently support multiple simultaneous workgroups. This replication resulted in no increase in actual block RAM usage.

Local Memory - Merging

- Local arrays are merged due to pointer mixing
  - Here created three local memory areas (lmem0, lmem1, lmem2) but assigned to each other

3lmem_nosplit.cl:10 (lmem1)	0	0	0	0	• Merged with another memory system declared at 3lmem_nosplit.cl:9. See that entry for details. Avoid mixing pointers from multiple memory systems, doing pointer arithmetic, or storing pointers to these memory systems to fix this.
3lmem_nosplit.cl:11 (lmem2)	0	0	0	0	• Merged with another memory system declared at 3lmem_nosplit.cl:9. See that entry for details. Avoid mixing pointers from multiple memory systems, doing pointer arithmetic, or storing pointers to these memory systems to fix this.
3lmem_nosplit.cl:9 (lmem0)	33	512	96	0	• Local memory: Good but replicated. Requested size 16384 bytes (rounded up to nearest power of 2), implemented size 147456 bytes, replicated 9 times total, stall-free, 3 reads and 3 writes. Additional information: - Merged with memory systems declared at: 3lmem_nosplit.cl:10, 3lmem_nosplit.cl:11. - Replicated 3 times to efficiently support multiple simultaneous workgroups. This replication resulted in 4 times increase in actual block RAM usage. Reducing the number of barriers or increasing max_work_group_size may help reduce this replication factor. - Replicated 3 times to efficiently support multiple accesses. To reduce this replication factor, reduce number of read and write accesses.

```
#define NUM_READS   2
#define NUM_WRITES  1
#define NUM_BARRIERS 1
#define ALLOW_SPLITTING 0

__attribute((reqd_work_group_size(1024,1,1)))
kernel void big_lmem(global int* restrict in, global int* restrict out) {

  local int lmem0[1024];
  local int lmem1[1024];
  local int lmem2[1024];

  int gi = get_global_id(0);
  int gs = get_global_size(0);
  int li = get_local_id(0);

  int res = in[gi];
  #pragma unroll
  for (int i = 0; i < NUM_WRITES; ++i) {
    lmem0[li - i] = res;
    lmem1[li - i] = res;
    lmem2[li - i] = res;
    res >>= 1;
  }

  // successive barriers are not optimized away
  #pragma unroll
  for (int i = 0; i < NUM_BARRIERS; ++i) {
    barrier(CLK_GLOBAL_MEM_FENCE);
  }

  res = 0;
  #pragma unroll 1
  for (int i = 0; i < NUM_READS; ++i) {
    local int *l0 = lmem0;
    local int *l1 = lmem1;
    local int *l2 = lmem2;
    #if ALOW_SPLITTING==0
      if (i % 3 == 1) {
        l0 = lmem1;
        l1 = lmem2;
        l2 = lmem0;
      }
    #endif
    res ^= l0[li - i];
    res ^= l1[li - i];
    res ^= l2[li - i];
  }

  out[gi] = res;
}
```

Local Memory – Replication

- Replication applied to achieve a stall-free access
  - Message: `Local memory: Good but replicated.`
- Local memory systems with replication can still be optimal if no additional block RAMs are used
  - Replicated using unused depth in block RAM

3lmem_nosplit.cl:9 (lmem0)	33	512	96	0	• Local memory: Good but replicated. Requested size 16384 bytes (rounded up to nearest power of 2), implemented size 147456 bytes, replicated 9 times total, stall-free, 3 reads and 3 writes. Additional information: - Merged with memory systems declared at: 3lmem_nosplit.cl:10, 3lmem_nosplit.cl:11. - Replicated 3 times to efficiently support multiple simultaneous workgroups. This replication resulted in 4 times increase in actual block RAM usage. Reducing the number of barriers or increasing max_work_group_size may help reduce this replication factor. - Replicated 3 times to efficiently support multiple accesses. To reduce this replication factor, reduce number of read and write accesses.

Local Memory - Banking

- Proper banking can help solve stalls
- Inefficient local memory constructs flagged

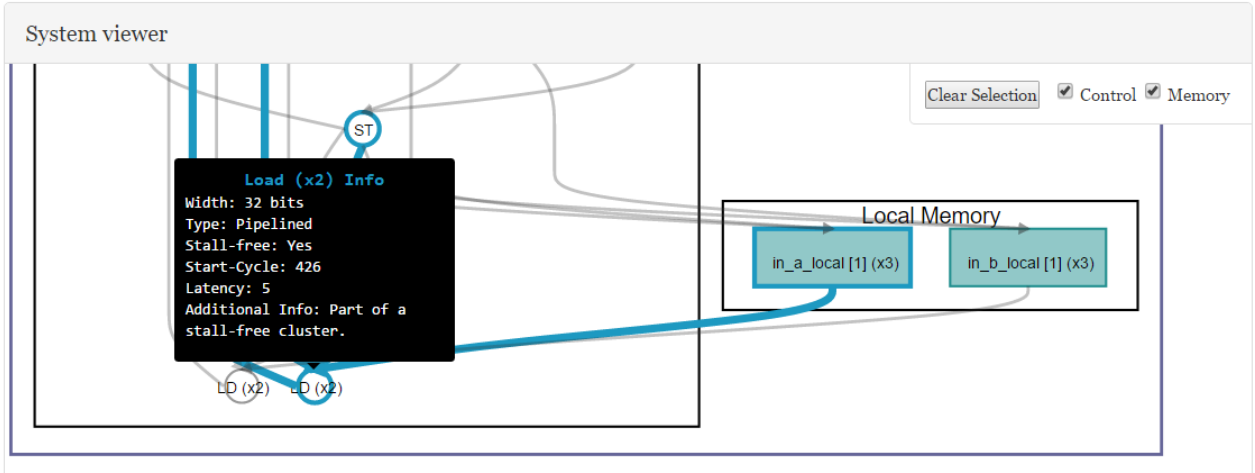
not_banked_2d.cl:12 (lmem)	66	512	16	0	Local memory: Potentially inefficient configuration. Requested size 16384 bytes (rounded up to nearest power of 2), implemented size 32768 bytes, replicated 2 times total, **stallable**, 4 reads and 4 writes. Additional information: - Reduce the number of write accesses or fix banking to make this memory system stall-free. Banking may be improved by using compile-time known indexing on lowest array dimension. - Replicated 2 times to efficiently support multiple simultaneous workgroups. This replication resulted in 2 times increase in actual block RAM usage. Reducing the number of barriers or increasing max_work_group_size may help reduce this replication factor. - Banked on lowest dimension into 2 separate banks (this is a good thing).
banked_2d.cl:10 (lmem)	0	0	16	0	Local memory: Good but replicated. Requested size 16384 bytes (rounded up to nearest power of 2), implemented size 32768 bytes, replicated 2 times total, stall-free, 4 reads and 4 writes. Additional information: - Replicated 2 times to efficiently support multiple simultaneous workgroups. This replication resulted in 2 times increase in actual block RAM usage. Reducing the number of barriers or increasing max_work_group_size may help reduce this replication factor. - Banked on lowest dimension into 4 separate banks (this is a good thing).

Area report messages will often contain suggestions on fixing problems in your design

```
8    // Banking only works on lowest dimensio
9   #if ALLOW_SPLIT
10    local int lmem[1024][NUM_READS];
11  #else
12    local int lmem[NUM_READS][1024];
13  #endif
14
15    int gi = get_global_id(0);
16    int gs = get_global_size(0);
17    int li = get_local_id(0);
18    int ls = get_local_size(0);
19
20    int res = in[gi];
21    #pragma unroll
22    for (int i = 0; i < NUM_READS; ++i) {
23      #if ALLOW_SPLIT
24        lmem[(li * i) % ls][i] = res;
25      #else
26        lmem[i][(li * i) % ls] = res;
27      #endif
28      res >>= 1;
29    }
```

HTML System Viewer – Local Memory

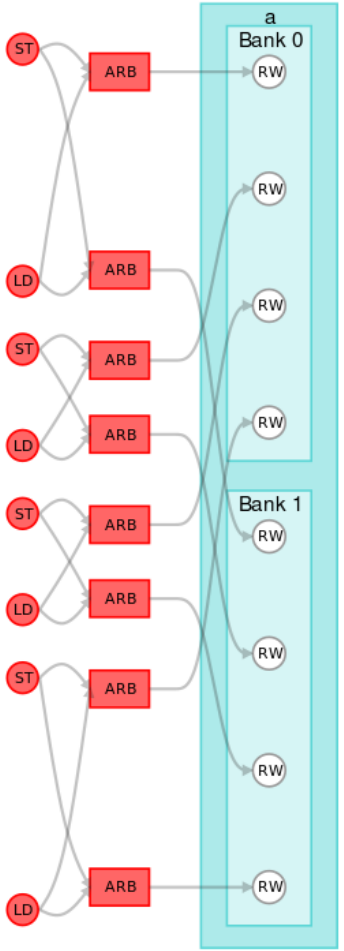
- Examine each load or store unit
  - Type, stall-free status, latency
- View memory implementation
  - Banking
  - Replication
- Visualize each access



Kernel Memory Viewer

Displays detailed information of memory layout

- Select memories and banks to show
- Shows number/type of ports, and sharing/arbitration logic if any
- Shows each read/write site
  - Includes access width
  - Stall-free or stallable (Red indicates stallable)



Local Memory Access Guidelines

- Certain optimization techniques, such as loop unrolling, may lead to more concurrent memory access
  - Too many memory accesses can complicate and degrade memory system performance
- Direct AOC to set the local memory to required size
  - For function scope local data, size statically specified
    - Preferred
  - For pass-by-pointer local data, use attribute
- Remove code for GPU-specific local memory bank conflicts
  - AOC generated hardware avoids local memory bank conflicts
  - Local memory in FPGAs implemented very differently from GPUs

Local Memory Configuration with Attributes

- Use attributes to force the compiler to choose a certain local memory configuration
- Use when compiler unable to infer optimal implementation

Example

```
int __attribute__((memory,
                   numbanks(2),
                   bankwidth(32),
                   doublepump,
                   numwriteports(1)
                   numreadports(4))) lmem[128];
```

Local Memory Attributes

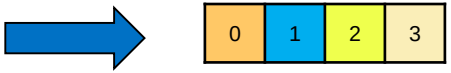
Control Memory Architecture Using Attributes

Attribute	Effect
`register/memory`	Controls whether a register or onchip memory implementation is used
`numbanks(N)`	Sets the number of banks
`bankwidth(N)`	Sets the bank width in bytes
`singlepump/doublepump`	Controls whether the memory is single- or double-pumped
`numreadports(N)`	Specifies that the memory must have N read ports
`numwriteports(N)`	Specifies that the memory must have N write ports
`merge("label", "direction")`	Forces two or more variables to be implemented in the same memory system
`bank_bits(b0,b1,…,bn)`	Forces the memory system to split into 2n banks, with {b0, b1, ..., bn} forming the bank-select bits

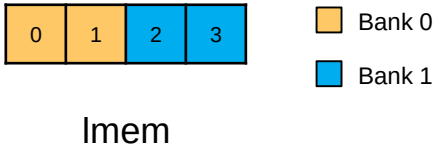
numbanks(N) and bankwidth(N) Memory Attribute Usage

- Same local memory integer array `lmem[4]` implemented in different configurations

```
__local int
__attribute__((numbanks(4),
               bankwidth(4)))
               lmem[4];
```



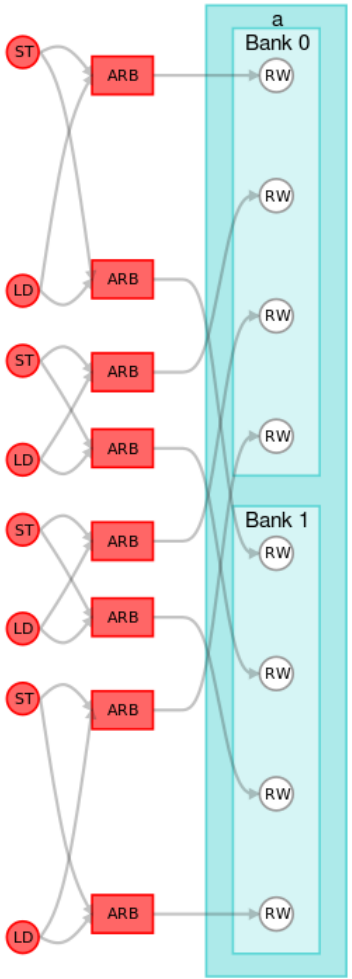

```
__local int
__attribute__((numbanks(2),
               bankwidth(8)))
               lmem[4];
```



Bank Bits Example: Default Implementation

```
__kernel void bank_arb_consecutive_multidim (
    int raddr, int waddr,
    int wdata, int upperdim, int rdata) {

    __local int a[2][4][128];

     #pragma unroll
     for (int i = 0; i < 4; i++)
          a[upperdim][i][(waddr & 0x7f)] = wdata + i;

     int rdata = 0;
     #pragma unroll
     for (int i = 0; i < 4; i++)
          rdata += a[upperdim][i][(raddr & 0x7f)];
}
```

Simultaneous Accesses
Default banking on lower bits.
Arbitration needed on the multiple middle index accesses

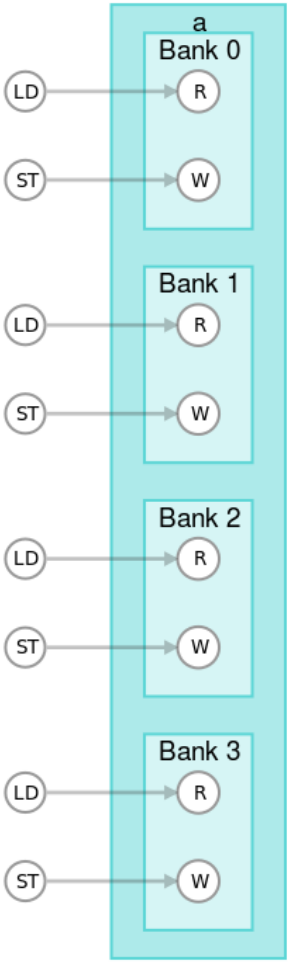


Bank Bits Example: `bankbits` Solution

```
__kernel void bank_arb_consecutive_multidim (
    int raddr, int waddr,
    int wdata, int upperdim, int rdata) {

    __local int __attribute__((bank_bits(8,7),bankwidth(4)))
    a[2][4][128];

    #pragma unroll
    for (int i = 0; i < 4; i++)
        a[upperdim][i][(waddr & 0x7f)] = wdata + i;

    int rdata = 0;
    #pragma unroll
    for (int i = 0; i < 4; i++)
        rdata += a[upperdim][i][(raddr & 0x7f)];
}
```

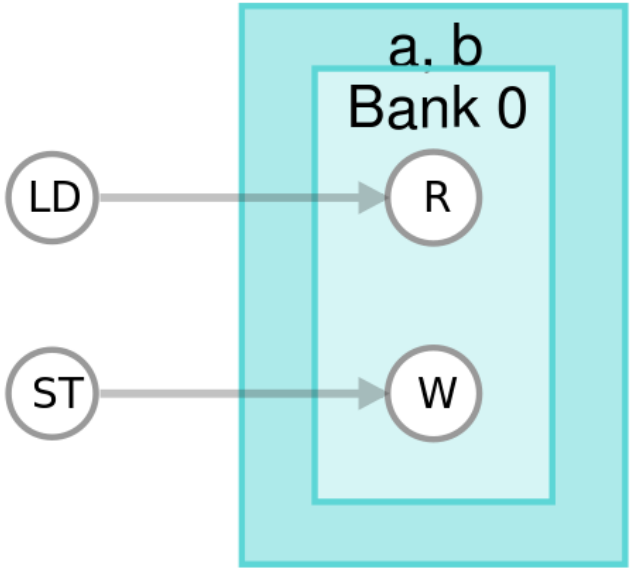
Lower dimensions of array need to be power of 2

Simultaneous Accesses, No arbitration needed with optimal banking



Local Memory Merging: Depth-wise

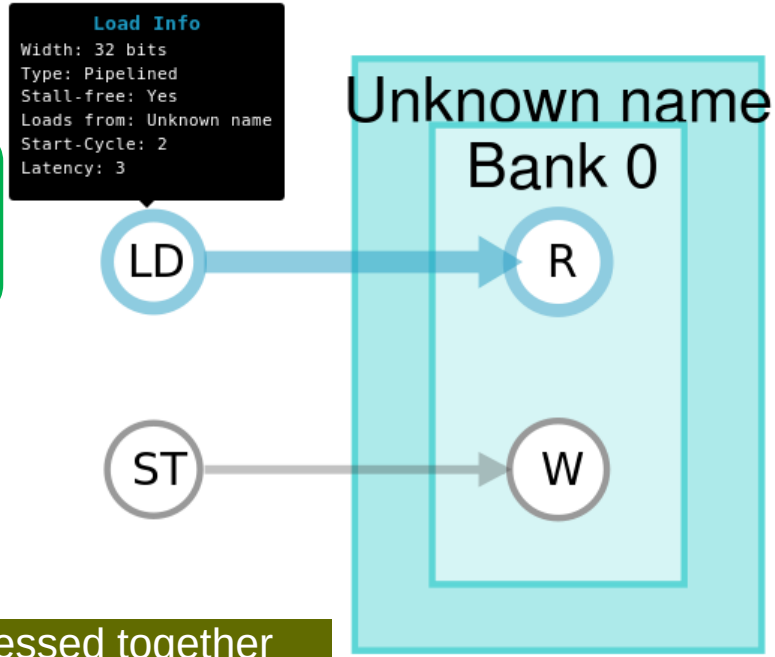
```
__kernel void depth_manual (bool use_a, int raddr,
  int raddr2, int waddr, int waddr2, int wdata, int rdata) {

    __local int
    __attribute__((merge("mem_name","depth")))
    int a[128];
    __local int
    __attribute__((merge("mem_name","depth")))
    int b[128];

    int rdata;
    // mutually exclusive write
    if (use_a)
        a[waddr] = wdata;
    else
        b[waddr2] = wdata;

    // mutually exclusive read
    if (use_a)
        rdata = a[raddr];
    else
        rdata = b[raddr2]; ….
```



- `a` and `b` never accessed simultaneously.
- Depth-wise merging allows them to share bank/memory block/ports.
- No arbitration needed.
- Saves resources.

Local Memory Merging: Width-wise

```
__kernel void width_manual (int raddr, int waddr,
                short wdata, short rdata) {

  __local short __attribute__((merge("mem_name", "width")))
     a[256];
  __local short __attribute__((merge("mem_name", "width")))
     b[256];

  short rdata = 0;

  // Lock step write (could have constant offset)
  a[waddr] = wdata;
  b[waddr] = wdata;

  // Lock step read (could have constant offset)
  rdata += a[raddr];
  rdata += b[raddr];

}
```

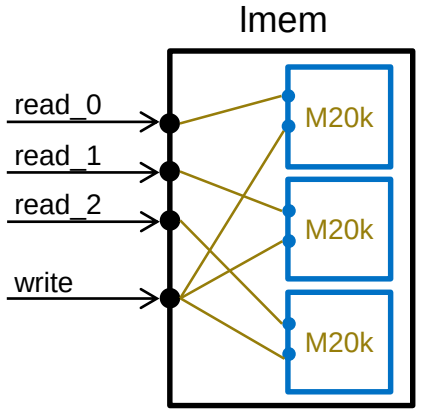


- `a` and `b` always accessed together
- Width-wise merging coalesces them together
- Saves resources

Local Memory Attribute Example

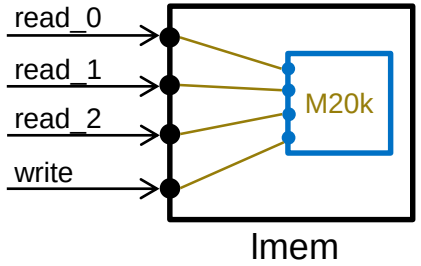
- Using attributes to control replication factor

```
local int
__attribute__((singlepump,
               numreadports(3),
               numwriteports(1)))
               lmem[16];
```



- No replication needed

```
local int
__attribute__((doublepump,
               numreadports(3),
               numwriteports(1)))
               lmem[16];
```



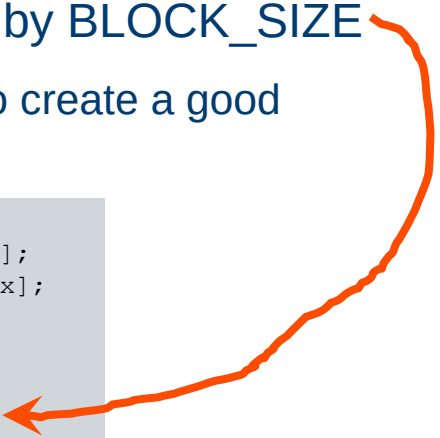
Conclusions

- Memory systems and interconnects customized for your kernel
- Write simple code, especially memory indexing
  - More likely to be statically decomposed
  - Be aware of implemented banking
  - Possible to transpose array to infer better banked behavior
- Be aware of loads/stores to the same bank
  - <= 4 will get never-stall without replication (double pumped)
- Enable replication by limiting number of stores

Matrix Multiplication Design Example: Analyze Local Memory Access Pattern

- Non-linear access of local array `B_local`
- For each iteration of k, pointer for array `B_local` jumps by BLOCK_SIZE
  - Large stride on each access makes it difficult for compiler to create a good coalesced/banked local memory configuration

```
//Loop through block and doing the following
A_local[local_y][local_x]= A[a + WIDTH * local_y + local_x];
B_local[local_y][local_x] = B[b + WIDTH * local_y + local_x];
barrier(CLK_LOCAL_MEM_FENCE);
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
      Csub += A_local[local_y][k] * B_local[k][local_x];
```



- Local memory access pattern is important, dictates implementation of local memory

Matrix Multiplication: Swapping Indices

- Convert the access to local memory `B_local` to be linear and thus much easier for the compiler to analyze

```
B_local[local_y][local_x] = B[b + WIDTH * local_y + local_x];
…
      Csub += A_local[local_y][k] * B_local[k][local_x];
```

```
B_local[local_x][local_y] = B[b + WIDTH * local_y + local_x];
…
      Csub += A_local[local_y][k] * B_local[local_x][k];
```

- Sometimes the compiler will figure this out for you, but if in doubt you can always do this easily in your source code

Matrix Multiplication: Local Memory Optimized

```
#define BLOCK_SIZE 64
#define WIDTH 1024
__kernel  __attribute((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
void matrixMul(__global float *restrict C, __global float *restrict A,
               __global float *restrict B)
{
      __local float As[BLOCK_SIZE][BLOCK_SIZE];
      __local float Bs[BLOCK_SIZE][BLOCK_SIZE];
// Initialize x(gid(0)), y(gid(1)), local_x, local_y, aBegin, aEnd, aStep, bStep (Hidden)
      float Csub = 0.0f;
      for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
            A_local[local_y][local_x]= A[a + WIDTH * local_y + local_x];
            B_local[local_x][local_y) = B[b + WIDTH * local_y + local_x];
            barrier(CLK_LOCAL_MEM_FENCE);
            #pragma unroll
            for (int k = 0; k < BLOCK_SIZE; ++k)
                  Csub += A_local[local_y][k] * B_local[local_x][k];
            barrier(CLK_LOCAL_MEM_FENCE);
      }
      C[get_global_id(1) * WIDTH + get_global_id(0)] = Csub;
}
```

Note the difference in A_local and B_local addressing scheme.

Matrix Multiplication: Area Report - Local Memory

Area report (source view)
(area utilization values are estimated)
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 58%)	186851 (36%)	265123 (25%)	1056 (41%)	264 (13%)	
Board interface	38262	44528	257	0	• Platform i...
Global interconnect	8779	12545	78	0	• Global int...
matrixMult	139810 (27%)	208050 (20%)	721 (28%)	264 (13%)	• Achieved k... • Number of ...
Data control overhead	1982	5225	14	0	• State + Fe...
Function overhead	1706	1762	0	0	• Kernel dis...
matrix_mult.cl:111 (A_local)	0	0	64	0	• Local memo...
matrix_mult.cl:112 (B_local)	0	0	256	0	• Local memo...
No Source Line	848	3605	17	0	

matrix_mult.cl

```
90  //
91  //  The combination of these values determines the number of floating-point
92  //  operations per cycle.
93
94  #include "../host/inc/matrixMult.h"
95
96  #ifndef SIMD_WORK_ITEMS
97  #define SIMD_WORK_ITEMS 4 // default value
98  #endif
99
100 __kernel
101 __attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
102 __attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
103 void matrixMult( // Input and output matrices
104                  __global float *restrict C,
105                  __global float *A,
106                  __global float *B,
107                  // Widths of matrices.
108                  int A_width, int B_width)
109 {
110     // Local storage for a block of input matrices A and B
111     __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
112     __local float B_local[BLOCK_SIZE][BLOCK_SIZE];
113
114     // Block index
```

Details

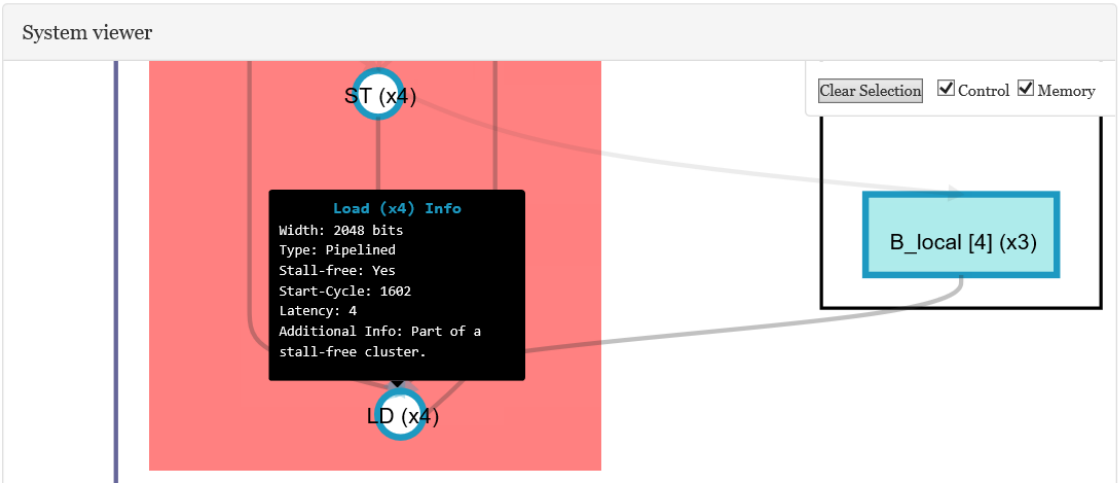
matrix_mult.cl:112 (B_local):

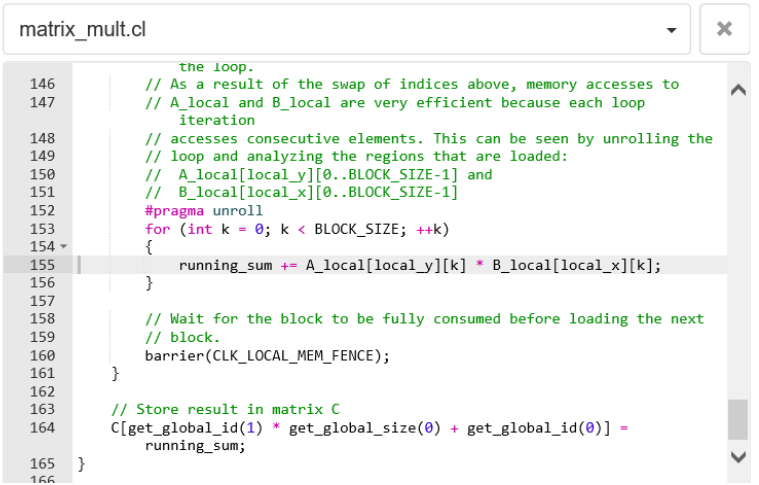
- Local memory: Optimal.

Requested size 16384 bytes (rounded up to nearest power of 2), implemented size 49152 bytes, replicated 3 times total, stall-free, 4 reads and 4 writes. Additional information:
- Replicated 3 times to efficiently support multiple simultaneous workgroups. This replication resulted in no increase in actual block RAM usage.
- Banked on lowest dimension into 4 separate banks (this is a good thing).

Matrix Multiplication Design Example: HTML System Viewer - Local Memory

- Looking at load unit for B_local
  - 2048 Bits, Pipelined, Stall-free





Exercise 5

Local Memory Optimizations

References

- Intel® OpenCL™ collateral (www.altera.com/OpenCL)
  - White papers
  - Demos and Design Examples
  - Intel FPGA SDK for OpenCL Getting Started Guide
  - **Intel FPGA SDK for OpenCL Programming Guide**
  - **Intel FPGA SDK for OpenCL Best Practices Guide**
  - Free Intel FPGA OpenCL Online Trainings
- Khronos* Group OpenCL Page
- OpenCL 1.2 Reference Card
  - https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

Intel® FPGA Technical Support Resources

- Intel FPGA Technology Landing Pages
  - Single page collecting resources related to particular FPGA topics and applications
- Intel® FPGA Technical Training materials
- Intel Programmable Solutions Group (PSG) community forum for self-help
- Intel PSG wiki site for design examples
- Intel PSG Knowledge Base Solutions
- Intel PSG Self Servicing License Center
- Please contact your sales and field support if you need further assistance

Legal Disclaimers/Acknowledgements

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel, the Intel logo, Intel Inside, the Intel Inside logo, MAX, Stratix, Cyclone, Arria, Quartus, HyperFlex, Intel Atom, Intel Xeon and Enpirion are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

OpenCL is the trademark of Apple Inc. used by permission by Khronos

*Other names and brands may be claimed as the property of others


intel
experience
what's inside™