Writing OpenCL™ for FPGAs

IWOCL
Agenda

- Intel® FPGA SDK for OpenCL™
- Optimizing ND Range Kernels
- Single Work-Item Execution
- Using Channels / Pipes
- Optimizing Memory
Innovation Across the Board

- FPGA/CPLD: Lowest Cost, Lowest Power
- FPGA: Cost/Power Balance SoC & Transceivers
- FPGA: Mid-range FPGAs SoC & Transceivers
- FPGA: Optimized for High Bandwidth
- PowerSoCs: High-efficiency Power Management

**Embedded Soft and Hard Processors**
- Nios® II
- Arm*

**Design Software**
- Intel® Quartus® Prime Design Software
- Intel FPGA SDK for OpenCL™

**Development Kits**

**Intellectual Property (IP)**
- Industrial
- Computing
- Enterprise
Intel® FPGA SDK for OpenCL™
Intel® FPGA SDK for OpenCL™ Section Agenda

- Introduction
- Intel® FPGA SDK for OpenCL™ Usage
- Overview of Debug and Optimizing Reports

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Intel® FPGA SDK for OpenCL™ Usage

Intel® FPGA SDK for OpenCL™

OpenCL Host Program

Standard C Compiler

Executable File

Offline Compiler (OpenCL Kernel Compiler)

Binary Programming File

Intel FPGA OpenCL Libraries
FPGA Architecture

- **Massive Parallelism**
  - Millions of logic elements
  - Thousands of embedded memory blocks
  - Thousands of Variable Precision DSP blocks
  - Programmable routing
  - Dozens of High-speed transceivers
  - Various built-in hardened IP

- **FPGA Advantages**
  - Custom hardware!
  - Efficient processing
  - Low power
  - Ability to reconfigure
  - Fast time-to-market
FPGA Architecture for OpenCL™ Implementation

Precompiled periphery (BSP)

FPGA

Processor

Host Interface

External Memory Controller & PHY

External Memory Controller & PHY

DDR

Custom Built Kernel System

Global Memory Interconnect

On-Chip Memory

Kernel Pipeline

Kernel Pipeline

On-Chip Memory

Local Memory Interconnect

Local Memory Interconnect

Global Memory Interconnect

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos
OpenCL™ Kernels to Dataflow Circuits

Each kernel is converted into custom dataflow hardware (Compute Unit)

- Gain the benefits of FPGAs without the length design process
- Implement C operators as circuits
  - HDL code located in `<SDK Installation>\ip`
  - Load Store units to read/write memory
  - Arithmetic units to perform calculations
  - Flow control units
  - Connect circuits according to data flow in the kernel
- May replicate circuit to accelerate algorithm

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Compilation Example

Kernel compiled into dataflow circuit with flow control

- Includes branch and merge units

```c
__kernel void my_kernel ( __global float *a,
                        __global float *b,
                        __global float *c,
                        int N)
{
    int i;
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```
Pipeline Execution of NDRange Kernels and Loops

- For NDRange work-items and loop iterations
- On each cycle the portions of the pipeline are processing different threads
- While work-item 2 is being loaded, work-item 1 is being added, and work-item 0 is being stored

Example Workgroup with 8 work-items

Thread IDs

```
7 7 7 7 7 6 5 4
3 2 3 2 1 0
```
Simultaneous Multithreading Execution Model

Tasks distributed through multiple queues can run in parallel

- Same device or multiple devices
- AOC implements dedicated compute units for each kernel
  - Different kernels can run in parallel

Implicit Parallelism in Algorithm

Sequential execution with one queue

Task Parallelism in OpenCL™ implementation

```
u = foo(x);
y = bar(x);
```

```
Q1.clEnqueueNDrangeKernel(cl_foo, ...)
Q2.clEnqueueNDrangeKernel(cl_bar, ...)
```

```
Device
foo_CU
bar_CU
```

```
K1  K2
```

```
cl_foo
```

```
cl_bar
```

```
K1  K2
```

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Intel® FPGA SDK for OpenCL™ Section Agenda

- Introduction

- Intel® FPGA SDK for OpenCL™ Usage
  - SDK Content
  - Kernel Compilation
  - Host Compilation
  - AOCL Utility
  - Runtime
  - Libraries

- Overview of Debug and Optimizing Reports
SDK Components

▪ Offline Compiler (AOC)
  – Translates your OpenCL™ C kernel source file into an Intel® FPGA hardware image
  – Requires Intel Quartus® Development Environment

▪ Host Libraries
  – Provides the OpenCL host API to be used by OpenCL host applications

▪ AOCL Utility
  – Perform various tasks related to the board, drivers, and compile process

▪ Intel Code Builder for OpenCL API with FPGA kernel development framework
  – Provides Microsoft* Visual Studio or Eclipse-based IDE for code development
Compiling Kernels

Run the Offline Compiler

- `aoc -list-boards`
  - List available boards within the current board package
- `aoc -board=<board> <kernel file>`
  - Compile the kernel to a board in the board package
  - Generates the kernel hardware system and compiles it using the Intel® Quartus® Prime software to target a specific board

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

Create libraries from RTL or OpenCL™ source and call those library functions from User OpenCL code

See the Intel® FPGA SDK for OpenCL Programming Guide for detailed examples
aoc Output Files

- `<kernel file>.aoco`
  - Intermediate object file representing the created hardware system

- `<kernel file>.aocx`
  - Kernel executable file used to program FPGA

- Inside `<kernel file>` folder
  - `<kernel file folder>\reports\report.html`
    - Interactive HTML report
    - Static report showing optimization, detailed area, and architectural information
  - `<kernel file>.log` compilation log
  - Intel® Quartus® Prime software generated source and report files
Intel FPGA Preferred Board for OpenCL

- Intel® FPGA Preferred Board for OpenCL™
  - Available for purchase from preferred partners
  - Passes conformance testing

- Download and install Intel FPGA OpenCL compatible BSP from vendor
  - Supplies board information required by the offline compiler
  - Provides software layer necessary to interact with the host code including drivers

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Custom Platform

Framework of host software and FPGA interface design to enable the use of OpenCL™ on a custom board

- FPGA design, software, and board bring up skills required
- Custom BSP provides
  - Timing-closed Hardware
  - MMD software layer
  - Some AOCL utility function
Compiling the Host Program

- **Include** `CL/opencl.h` or `CL/cl.hpp`

- **Use a conventional C compiler (Visual Studio*/GCC)**

- **Add** `$INTELFPGAOCLSDKROOT/host/include` to your file search path
  - Recommended to use `aocl compile-config`

- **Link to Intel® FPGA OpenCL™ libraries**
  - Link to libraries located in the
    `$INTELFPGAOCLSDKROOT/host/<OS>/lib` directory
    - Recommended to use `aocl link-config`

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

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Intel® FPGA SDK for OpenCL™ Section Agenda

- Introduction
- Intel® FPGA SDK for OpenCL™ Usage
- Overview of Debug and Optimizing Reports
Kernel Development Flow and Tools

1. Modify kernel.cl
2. Emulator (secs)
   - Functional bugs?
3. HTML Report (~1 min)
   - Loop Optimization Report
   - Detailed Area Report
   - Architectural Viewer
   - Loop inefficiencies?
   - Undesired hardware structure?
   - Sub-optimal memory interconnect?
4. Profiler (full compile time)
   - Poor performance?
5. Done
Debugging Kernels Using `printf`

`printf` instructions in kernels are supported

- Conforms to OpenCL™ 1.2 specification
  - No usage limitations
    - Can use inside if-then-else statements, loops, etc.

- Order of concurrent calls (from different work-items) are not guaranteed

- `aoc` allocates 64kB global buffer for `printf`s
  - Once kernel execution completes, contents are printed to standard output
  - If the buffer overflows, kernel execution stalls until the host reads and prints the buffer contents

- Due to global memory use, `printf` will impede performance

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Enable kernel functional debug on x86 systems

- Quickly generate x86 executables that represent the kernel

```
aoc -march=emulator <kernel file>
```

- Debug support for
  - Standard OpenCL™ syntax, Channels, Printf statements

- Set environment prior to executing host application

```
set CL_CONTEXT_EMULATORDEVICE_INTELFPGA=<target board>
```
HTML Report

Static report showing optimization, area, and architectural information

- Automatically generated with the object file ($aoc \ -c$)
  - Located in `<kernel file folder>\reports\report.html`

- Dynamic reference information to original source code

- Sections
  - Loop Analysis
  - Area Report
  - Architectural Viewer
  - Kernel Memory Viewer
HTML Loop Analysis Optimization Report

- Actionable feedback on pipeline status of loops
  - Shows loop carried dependencies and bottlenecks
  - Especially important for single work-item kernels since they have an outer loop
- Shows loop unrolling status
- Shows loop nesting relationship
HTML Area Report

Generate detailed estimated area utilization report of kernel code

- Detailed breakdown of resources by source line or by system blocks
- Provides architectural details of HW
  - Suggestions to resolve inefficiencies
HTML System Viewer

- Displays kernel pipeline implementation and memory access implementation
- Visualize
  - Off-chip memory
    - Load-store units
    - Accesses
  - Stalls
  - Latencies
  - On-chip memory
    - Implementation
    - Accesses
HTML Kernel Memory Viewer

Helps you identify data movement bottlenecks in your kernel design. Illustrates:

- Memory replication
- Banking
- Implemented arbitration
- Read/write capabilities of each memory port
1. Compile kernel with `-profile` option
   - Inserts profiling counters into the HW

2. Run host application
   - Generates `profile.mon` file

3. View data using the profiler GUI

```
aoc -profile <kernel file>
aocl report <kernel file>.aocx profile.mon
```
Profiler Reports – Source Code Tab

Displays statistics about memory and channel accesses

<table>
<thead>
<tr>
<th>List</th>
<th>Source Code</th>
<th>Attributes</th>
<th>Stall%</th>
<th>Occupancy%</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>0.13%</td>
<td>0.100%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0.25%</td>
<td>0.100%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>0.37%</td>
<td>0.100%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>0.25%</td>
<td>0.100%</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>0.00%</td>
<td>0.000%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>0.00%</td>
<td>0.000%</td>
<td></td>
</tr>
</tbody>
</table>

- **Stall%**: Percentage of time current data access is causing pipeline stalls
- **Occupancy%**: Percentage of overall profile time when the current data access is active
- **Bandwidth**: Average memory bandwidth for the current memory access
- **Efficiency**: % of data acquired that the kernel program actually uses

Tooltip available also shows: Cache Hit %, Unaligned Access %, Coalesced, Average Burst Size, and Activity%
Profiler Reports – Kernel Execution Tab

- Illustrates the execution time of each kernel
- Shows interactions between different kernel executions
- May display memory transfers between the host and devices
  - To enable, set the environment variable ACL_PROFILE_TIMER to 1
### Profiler Reports – Kernel Summary Tab

- Reports memory bursts, stalls and bandwidth
- Each kernel has a separate memory tab
Matrix Multiplication Design Example

- Demonstrates concepts in this class
- Located on the website

Matrix-matrix multiply mathematics
- A is an n x m matrix
- B is an m x p matrix

\[
A = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1m} \\
A_{21} & A_{22} & \cdots & A_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
A_{n1} & A_{n2} & \cdots & A_{nm}
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
B_{11} & B_{12} & \cdots & B_{1p} \\
B_{21} & B_{22} & \cdots & B_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
B_{m1} & B_{m2} & \cdots & B_{mp}
\end{pmatrix}
\]

- Product (AB) is an n x p matrix

\[
AB = \begin{pmatrix}
(AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\
(AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
(AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np}
\end{pmatrix}
\]

Equation

\[
(AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}
\]

https://www.altera.com/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html
Matrix Multiplication Naïve Implementation

- NDR.arange implementation of (2048x1024) x (1024x1024) matrix multiply
- Each work-item calculates one result in the product matrix

```c
#define WIDTH 1024
void matrixMul(__global float *restrict C,
              __global float *restrict A,
              __global float *restrict B)
{
    float Csub = 0.0f;
    int x = get_global_id(0);
    int y = get_global_id(1);

    for (int i = 0; i < WIDTH; i++) {
        Csub += A[y * WIDTH + i] * B[x + WIDTH * i];
    }

    C[y * WIDTH + x] = Csub;
}
```

Loops across matrix A and down matrix B for each result
Matrix Multiplication Naïve Implementation

- One Compute Unit created
- 1 multiplication and 1 adder created
- At 400Mhz, would result in 0.8 GFLOPs
  - Theoretical maximum computation bandwidth of circuit
- And that’s not even the bottleneck
  - next slide

![Diagram of matrix multiplication process]
Matrix Multiplication (Naïve) Profiler Report

- Profiler ran for execution on Stratix® V board
  - 11 seconds to execute
  - Total amount of data read: 11s x (1,300 MB/s + 7400 MB/s) ≈ 95GB
    - Total input size = 3M floats x 4 bytes/float = 12 MB
    - Data being accessed repeatedly (~8000x)

- Issues with initial implementation: High stall, medium occupancy, low efficiency

- Profiling Store: Extremely low occupancy, rarely-used LSU, Don’t Care

```c
// Store result in matrix C
C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = runni...  // __global[DDR],write) (7.34%) (0.1%)  (1.2MB/s, 66.67%Efficiency)
```
Optimizing ND Range Kernels
Optimizing ND Range Kernel Execution Agenda

- Workgroup Size
- Loop Unrolling
- Kernel Vectorization
- Kernel Compute Unit Replication
Workgroup Characteristics

- Work-items within a workgroup can share local data and synchronize

- OpenCL™ workgroup size rules
  - NDrange must be evenly divisible by workgroup size in each dimension
  - Set at kernel launch time by the host local_work_size argument in the clEnqueueNDRangeKernel call
  - All work items from the same workgroup assigned to the same CU at the same time

- Optimal workgroup size determined by the hardware

- FPGA compute unit workgroup limit can be set by kernel attributes
Specifying Work-Group Size Attributes

Allow AOC to allocate the optimal amount of hardware resources to manage and synchronize the work-items in a workgroup

- Allows work-group size optimized code

- \texttt{max\_work\_group\_size}(N)
  - Specifies the maximum number of work-items in a workgroup

- \texttt{reqd\_work\_group\_size}(X,Y,Z)
  - Specifies the required work-group size

\begin{verbatim}
#include<amath.h>
__attribute__((reqd_work_group_size(64,64,1)))
__kernel void mykernel (...) {
  ...
}
\end{verbatim}

\begin{verbatim}
#include<amath.h>
__attribute__((max_work_group_size(256)))
__kernel void mykernel (...) {
  ...
}
\end{verbatim}
Query Kernel CU Workgroup Requirements

Use `clGetKernelWorkGroupInfo` to query Kernel CU workgroup size limit

- Use the following `param_names`
  - `CL_KERNEL_WORK_GROUP_SIZE`
    - Maximum workgroup size the compute unit supports
  - `CL_KERNEL_COMPILE_WORK_GROUP_SIZE`
    - Work-group size specified by kernel attribute `reqd_work_group_size(X,Y,Z)`
    - If none exist, will return (0,0,0)

```
cl::Kernel::getWorkGroupInfo (mydeviceId, CL_KERNEL_WORK_GROUP_SIZE, &param_value)
```

`param_value`: pointer to return value
Recommended to specify the workgroup size when launching kernels on the Intel® FPGA platform

- Setting `local_work_size` to NULL may result in an undesirable workgroup size

```c
// 1D Work-Group Example
int err;
size_t const globalWorkSize = 1920;
size_t const localWorkSize = 8;
err = myqueue.enqueueNDRangeKernel(1dkernel, cl::NullRange, cl::NDRange(globalWorkSize),
                                  cl::NDRange(localWorkSize));

// 3D C Work-Group Example
err = myqueue.enqueueNDRangeKernel(3dkernel, cl::NullRange, cl::NDRange(512, 512, 512),
                                  cl::NDRange(16, 8, 2));
```
Matrix Multiplication Design: Analyze Memory Access Pattern

- **Bottleneck:** Memory controller can’t keep up (high stall, medium occupancy)

- **Problem**
  - Each input value is accessed repeatedly (~8000x)
  - Input data size is 12MB yet we’re reading 95GB of data from global memory

```c
for (int i = 0; i < WIDTH; i++) {
    Csub += A[y * WIDTH + i] * B[x + WIDTH * i];
}
C[y * WIDTH + x] = Csub;
```

- **Code analysis:** repeated access
  - Reads an entire row of A and an entire column of B to calculate each value of C
  - Adjacent threads read much of the same data (row from matrix A or a column from matrix B)
Matrix Multiplication Design: Tiling / Blocking

- Tiling is buffering data onto fast on-chip storage where it will be repeatedly accessed (caching)
  - Very common technique
  - Used when multiple threads need to access overlapping parts of data set

- Data must be partitioned into blocks to fit into local memory
  - Only work-items within a workgroup can share data
  - Local memory size and geometry set at compile time
  - Workgroup sizes (block sizes) must be known at compile time
Matrix Multiplication Design: Tiling / Blocking

- Set required workgroup size using attribute
- Set local memory size based on block size

```c
#define BLOCK_SIZE 64
#define WIDTH 1024
__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
void matrixMul(__global float *restrict C, __global float *restrict A, __global float *restrict B) {
    __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
    __local float B_local[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_y][local_x] = B[b + WIDTH * local_y + local_x];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += A_local[local_y][k] * B_local[k][local_x];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[get_global_id(1) * WIDTH + get_global_id(0)] = Csub;
}
```

Loop through elements in a BLOCK to cache in data
Loop through BLOCK width to calculate partial result
Matrix Multiplication Design: Tiling / Blocking

\[ C = A \times B \]

Store C

Load A

Load B

On-chip Memory

On-chip Memory

Accumulator
Matrix Multiplication: Block Size vs Performance

- Workgroup size and local memory requirement increases **quadratically** with Block Size (BS)

- Global demand and kernel time drops **linearly** with block size

- For block size 64, read data ~23x times from global

- Eventually problem changes from memory-bound to **compute-bound** and **area-bound**

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Local Mem Size (floats)</th>
<th>Global Reads (floats)</th>
<th>Kernel Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4,294,967,296</td>
<td>11,224</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2,147,483,648</td>
<td>3,313 (-70%)</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1,073,741,824</td>
<td>1,683 (-49%)</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>536,870,912</td>
<td>900 (-47%)</td>
</tr>
<tr>
<td>16</td>
<td>512</td>
<td>268,435,456</td>
<td>438 (-51%)</td>
</tr>
<tr>
<td>32</td>
<td>2,048</td>
<td>134,217,728</td>
<td>218 (-50%)</td>
</tr>
<tr>
<td>64</td>
<td>8,192</td>
<td>67,108,864</td>
<td>151 (-31%)</td>
</tr>
<tr>
<td>BS</td>
<td>2 * BS²</td>
<td>2 * N³ / BS</td>
<td>--</td>
</tr>
</tbody>
</table>

Matrix: (2048 x 1,024) x (1024 x 1024) = (2048 x 1024)
Optimizing ND Range Kernel Execution Agenda

- Optimization Overview
- Workgroup Size
- Loop Unrolling
- Kernel Vectorization
- Kernel Compute Unit Replication
unroll kernel pragma

#pragma unroll <N> instructs AOC to attempt to unroll a loop <N> times

- Without <N>, AOC will attempt to unroll the loop fully
- Warning issued if AOC unable to unroll

```c
#pragma unroll 2
for (size_t k=0; k<4; k++) {
    mac += data_in[(gid*4)+k] * coeff[k];
}
```

- Control the amount of hardware used for loops
  - Trading off between performance and area
  - If performance is exceeded, reducing loop unrolling factor can help reduce area
  - Force compiler to not unroll by using #pragma unroll 1
Loop Unrolling Example

- Sum of 4 values for every work-item
- Store a new result every 4 iterations

```c
accum = 0;
for (size_t i=0; i<4; i++)
{
    accum += data_in[(gid*4)+i];
}
sum_out[gid] = accum;
```
Loop Unrolling Example: Unroll 2

- Unroll factor of 2
  - 2 iterations of the loop performed for every forward execution
- Store a new result every 2 iterations

```c
accum = 0;
#pragma unroll 2
for (size_t i=0; i<4; i++)
{
    accum += data_in[(gid*4)+i];
}
sum_out[gid] = accum;
```
Loop Unrolling Example: Fully Unrolled

- Unroll every iteration of the loop
- Store a new result every clock cycle

```c
accum = 0;
#pragma unroll
for (size_t i=0; i<4; i++)
{
    accum += data_in[(gid*4)+i];
}
sum_out[4*gid] = accum;
```

Additional Optimizations Shown:
1. `accum` register removed
2. Order of operation optimization done if allowed
3. Operators removed if not needed
   - There would be 4 adders created if initial value of `accum` is not 0.
Loop Unrolling in the HTML Report

- Loop unrolling reported in loop analysis section of the HTML report
  - `<kernel file folder>\reports\report.html`
  - Also in `<kernel file>.log`

- Reported information
  - Loop location
  - Nesting relationship
  - Requested unroll factor
  - Achieved unroll factor
Matrix Multiplication: Initial Implementation

- 1 multiplication and 1 adder created
- Need to try loop unrolling to increase compute

```c
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += A_local[local_y][k] + B_local[k][local_x];
```
Matrix Multiplication: Improved Implementation

```
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += A_local[local_y][k] + B_local[k][local_x];
```
Optimizing ND Range Kernel Execution Agenda

- Optimization Overview
- Dynamic Profiler Overview
- Workgroup Size
- Loop Unrolling
- Kernel Vectorization
- Kernel Compute Unit Replication
Kernel Vectorization

Widen the pipeline to achieve higher throughput

- Allow multiple work-items from the same workgroup to execute in Single Instruction Multiple Data (SIMD) fashion

- Translate scalar operations into SIMD vectored operations
Vectorize Kernel Code Manually

- Replicate operations in the kernel manually
  - Must also adjust NDRange in host application

```c
__kernel void mykernel (...) {
    size_t gid = get_global_id(0);
    result[gid] = in_a[gid] + in_b[gid];
}
```

```c
__kernel void mykernel (...) {
    size_t gid = get_global_id(0);
    result[gid*4+0] = a[gid*4+0] + b[gid*4+0];
    result[gid*4+1] = a[gid*4+1] + b[gid*4+1];
    result[gid*4+2] = a[gid*4+2] + b[gid*4+2];
    result[gid*4+3] = a[gid*4+3] + b[gid*4+3];
}
```
Vectorize Kernel - Memory Coalescing

Vectorize a kernel using OpenCL™ vectored data types

- Elements of vectored data types always in consecutive memory locations
  - e.g. float4, int8, etc
  - Accesses can be coalesced (Wider accesses results in fewer accesses)

```c
__kernel void mykernel(
    __global const float4 * restrict in_a,
    __global const float4 * restrict in_b,
    __global float4 * restrict result)
{
  size_t gid = get_global_id(0);
  result[gid] = in_a[gid] + in_b[gid];
}
```

![result[gid].x = in_a[gid].x + in_b[gid].x; result[gid].y = in_a[gid].y + in_b[gid].y; result[gid].z = in_a[gid].z + in_b[gid].z; result[gid].w = in_a[gid].w + in_b[gid].w;](image)
Automatic Kernel Vectorization

Use attribute to enable automatic kernel compute unit vectorization

- Without modifying the kernel body
- Memory accesses automatically coalesced
- No need to adjust NDRange in host application

- **num_simd_work_items** attribute
  - Specify the SIMD factor (# of work-items in the same workgroup executed in parallel)
    - Hardware operators automatically vectorized
    - Vectorization takes affect in the X dimension of the workgroup

```c
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void mykernel (...
...```
Automatic SIMD Vectorization Limitations

- `num_simd_work_items` must be 2, 4, 8, or 16
- `reqd_work_group_size` must be evenly divisible by `num_simd_work_items` in the X dimension
- If a control path depends on `get_global_id` or `get_local_id`, that branch will not be vectorized
  - The rest of the kernel will be
- Use manual vectorization or kernel replication (next section) in these situations
Matrix Multiplication: SIMD Vectorization w/Unrolling

```c
#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 A_local[BLOCK_SIZE][BLOCK_SIZE];
    __local float B_local[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_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];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[get_global_id(1) * WIDTH + get_global_id(0)] = Csub;
}
```

Dynamic Profiler Results

<table>
<thead>
<tr>
<th>SIMD_WORK_ITEMS</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>151</td>
</tr>
<tr>
<td>2</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>53</td>
</tr>
</tbody>
</table>

Original design time: 11224 ms
Dynamic Profiler
Benefits of Tiling, SIMD, and Loop Unrolling

Naïve Kernel: BLOCK_SIZE=1, SIMD=1, No Unrolling, Time = 11,224 ms

Improved Kernel: BLOCK_SIZE=64, SIMD=4, Loop Unrolled, Time = 53ms

- Conclusion (212x Performance Improvement)
  - Stall / Occupancy are similar, memory efficiency improved
  - SIMD Vectorization and BLOCKING improves memory access efficiency while reducing global memory access requirement
  - SIMD Vectorization and Loop Unrolling improves computational bandwidth
  - Know your algorithm! Think about your algorithm before low-level system issues
Optimizing ND Range Kernel Execution Agenda

- Optimization Overview
- Dynamic Profiler Overview
- Workgroup Size
- Loop Unrolling
- Kernel Vectorization
- Kernel Compute Unit Replication
Default Compute Unit Created

- Only one compute unit per kernel created by default
- Workgroups distributed to compute unit in sequence
Multiple Compute Units

- **num_compute_unit** kernel attribute specifies number of CUs to generate
  - **num_compute_units**(N) or **num_compute_units**(X,Y,Z)
    - N or X*Y*Z compute units created
  - Entire compute unit including all local memory, control logic, and operators replicated
    - Each compute unit functionally identical
  - Kernel usage not limited, limited only by FPGA resource

- **Workgroups** from the same NDRRange kernel launch are distributed to available compute units and processed in parallel
  - Need at least three times as workgroups as compute units to effectively utilize all hardware

```c
__attribute__((num_compute_units(3)))
__kernel void ...
```
num_compute_units Applied

```c
__attribute__((num_compute_units(3)))
__kernel void ...
```
Memory Considerations - CU Replication vs. SIMD

### num_compute_units
- Increases **number** of global memory accesses
- May lead to poor access patterns
  - Random accesses
  - Possible contention

### num_simd_work_items
- Increases **width** of global memory accesses
- Coalescing of memory accesses
  - Wide accesses
  - Burst accesses
Compute Unit Replication vs. SIMD Vectorization

- Try SIMD vectorization first
  - Usually leads to more efficient hardware than compute unit replication

- May combining SIMD vectorization with computer unit replication
  - Possibly required to achieve best performance and/or fit
  - 4 copies of 4-lane-wide CUs **may or may not** be better than 2 8-lane-wide CUs

<table>
<thead>
<tr>
<th>num_compute_units</th>
<th>num_simd_work_items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designed to increase throughput by increasing kernel hardware</td>
<td>Increases the # of work-items from the same workgroup to be processed in parallel in a CU</td>
</tr>
<tr>
<td>Increase # of compute unis where workgroups can be scheduled</td>
<td>Kernel control logic shared across each SIMD vector lane</td>
</tr>
<tr>
<td>Entire CU including control logic replicated (more resource usage)</td>
<td>Usage only limited by FPGA resources</td>
</tr>
<tr>
<td>Kernel code and resource restrictions</td>
<td></td>
</tr>
</tbody>
</table>
Example: Combining Replication and Vectorization

- Resource estimates of 16 SIMD lanes indicate “no fit”
- Resource estimates of 8 SIMD lanes suggest 12 lanes may fit
  - Automatic vectorization only supports 2, 4, 8 and 16 lane configurations
- Generate 12 lanes by combining `num_simd_work_items` and `num_compute_units`

```c
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(8,8,1)))
__kernel void mykernel (...) {
  ...
```
Exercise 4

Optimizing an NDRRange Kernel
Single Work-Item Execution
Single Work-Item Execution Agenda

- **Introduction**
- Understanding execution models and optimization reports
- Resolving common dependency issues
- **Advanced Uses**
  - 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™

- Single work-item kernels almost always have an outer loop
  - Loops in kernels automatically parallelized by the Intel® FPGA OpenCL Offline Compiler
    - *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

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Tasks and Loop-pipelining

- NDRange Kernels can’t handle dependencies across work-items well

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

- Solution: Tasks
  - 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

Share data through feedbacks 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

```c
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];
}
```

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

Looks like multi-threaded execution!
Parallel Threads vs Loop Pipelining

- 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!*

Parallel threads launch 1 thread per clock cycle in pipelined fashion.

If loop dependency resolved in 1 clock cycle.
Loop Unrolling in Time vs Pipelining

Unroll

Pipelining
Single Work-Item vs. NDRange Kernels

One approach is not better than the other, can have both types of kernels in the same application

- 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
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

```c
__kernel void mykernel (...) {
  for (i=0; i< FFT_POINTS; i++) {
    ...
  }
}
```
Launching Single Work-Item Kernels (Tasks)

- Single work item kernels assumed when there are no `get_global_id()`, `get_local_id()`, or `get_group_id()` calls
- **Use** `cl::CommandQueue::enqueueNDRangeKernel` with `global_work_size` and `local_work_size` set to 1
- **Or** `cl::CommandQueue::enqueueTask` in host code

**Host Code**

```c
setup_memory_buffers();
transfer_data_to_fpga();

myqueue.enqueueTask(mykernel, ...);

read_data_from_fpga();
```
Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
- Advanced Uses
  - 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`
- Also part of the log file
  - `<kernel file folder>\<kernel file>.log`
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
    - Ideal II = 1
Loop Pipeline Single Loop Execution

Basic case – single loop

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

$L = K$

$L$: 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

```c
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 $\approx N$
  - 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 $N*6 + K \approx 6*N$
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 $N \times II + K \approx II \times N$
- 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

- Largest Critical Path Contributor
  - Specifies the operations that contribute to the delay
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 = $M^*(N^*1) + K + J$
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

  L = J

  L = K

  Outer Loop II=2

  Inner Loop II=1
Loop Pipeline with Nested Loops

Which loop is the critical loop?

- Depends on the value of $N$ and $P$
- If $P$ is much smaller than $N$, II for $P$ loop doesn’t matter
  - If $P \times 8 < N$

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

Loop Analysis Report:
- M loop: II $\geq 1$
- N loop: II = 1
- P loop: II = 8
Interleaving of Outer Iterations in the Inner Loop

- When Inner Loop II>1 and inner loop is not a serial region (discussed later)

```java
for (...) {
  ...
  for (...) {
    ...
  }
}
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

```python
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

- 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

```c
for ( i=0; i < N; i++ )
    for ( j=0; j < N-i; j++ ){
        ...
    }
```
Serial Region Execution

- 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

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

**Optimization Report**

Outer Loop: II = 2
Serial execution around: Inner Loop
Inner Loop: II=1

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
- Advanced Uses
  - 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 cycles
  - Not the critical loop

- Each inner iteration requires \( \text{sum} \) from the previous outer iteration
  - Becomes serial region

- Inner loop pipelined well!

```c
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];
}
```
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

```c
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];
}
```

<table>
<thead>
<tr>
<th>*** Loop Analysis Report ***</th>
</tr>
</thead>
</table>
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 \texttt{mul} every cycle means II needs to be 6

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

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

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

Result of multiply won’t be used for 6 cycles
Takes 6 cycles
Optimized
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

```c
component void mycomp (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

```c
component void mycomp(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!

- **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

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

- Dependencies ignored for all accesses to memory arrays

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

#pragma ivdep array(array_name)

- Dependency ignored for only array_name accesses

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

- **ivdep and structs**

  ```c
  #pragma ivdep array(S.A)
  for (unsigned i = 0; i < N; i++)
  ```

  No dependencies for array A inside struct S

  ```c
  #pragma ivdep array(S->A)
  for (unsigned i = 0; i < N; i++)
    S->A[i] = S->A[i-X[i]]
  ```

  No dependencies for A inside the struct pointed to by S

- **ivdep applies to all arrays that may alias with specified pointer**

  ```c
  int *ptr = select ? A : B;
  #pragma ivdep array(ptr)
  for (unsigned i = 0; i < N; i++){
    A[i] = A[i - X[i]];
    B[i] = B[i - Y[i]];
  }
  ```

  No dependencies for A and B array
Convert Nested Loops into Single Loop

Combine nested loops to save resources and improve performance

- Consider using the `loop_coalesce` pragma

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

- Nested loops have more logic and latency than a coalesced loop

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

```c
#pragma loop_coalesce
for (...) {
    for (...) {
        ...
    }
}
```

Compiler attempts to coalesce all nested loops

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

Compiler attempts to coalesce only loops A, B, and D
Single Work-Item Execution Agenda

- Introduction
- Understanding execution models and optimization reports
- Resolving common dependency issues
- Advanced uses
  - Lab 2
Reducing Kernel Hardware Overhead with \( \text{max\_global\_work\_dim}(0) \)

- Single Work-Item Kernels are not dispatched across work-items/workgroups
- **Kernel attribute** `\( \text{max\_global\_work\_dim}(0) \)` 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 result in no resource savings

```c
__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
Kernels That Runs Without the host

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

```c
_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**
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
    - More on this in the Optimizing NDRange kernels section
  - 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 \texttt{get\_compute\_id}

Using compute ID to determine channel usage

```c
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)
        ...
}
```

Diagram showing the channels and pipes between PEs.
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", Divider-and-conquer
  - Call each of these Processing Elements Kernels (PE)
Convolutional Neural Network (CNN) Example

Convolution Operation

Ping Pong Buffer: Output of one stage becomes the input of the subsequent stage

Processing Element == Convolution Operation

Filter Buffer
Matrix Multiply in OpenCL™ – Small 4x4 variant

- 2D Systolic Array
  - Each PE a dot product
  - DSP blocks chained together
- Regular array topology
Exercise 2

Relax Data Dependencies
Reducing Communication Latency with Pipes/Channels
Traditional OpenCL™: Host-Centric Architecture

All communication to/from kernels done through global memory
Idea: Communication without Global Memory

- Kernel-to-kernel communication done directly on-chip using FIFOs
- IO-to-kernel communication done without the host
- Enabled through Intel FPGA Channels / OpenCL Pipes
Channels / Pipe Features

- Provides FIFO-like communication mechanism
- Each call site is unidirectional
- Allows BSP-specific I/O communication with kernel compute units

Advantages
- Leverage internal bandwidth of the FPGA
- Avoid the bottleneck of using off-chip memory
- Reduces overall latency by allowing concurrent Kernel execution
- Reduce storage requirements when data is consumed as it is produced
Kernel-to-Kernel Channel Performance Gains

- **Standard**
  - If communication between kernels is required, host forced to launches kernels sequentially
    - Kernel 1 writes to global memory, kernel 2 reads from global memory

- **With channels**
  - Host can launch kernels in parallel
    - kernel 1 writes to channel as kernel 2 reads from it
IO Channel Performance Gains

- **Standard**
  - Data needs to be written to global memory first before kernel can process it and then read back after processing
  - Limited by PCIe* bandwidth and memory throughput

- **With IO channels**
  - Kernel can run while data flows across network interface
  - System running at speed of network interface
Channel Declaration

- Enable the Intel® FPGA extension for channels

  ```
  #pragma OPENCL EXTENSION cl_intel_channels : enable
  ```

- Declare file-scope channel handle along with type
  - Supports any built-in OpenCL™ or user defined types
    - structs, char, uchar, short, ushort, int, uint, long, ulong, float, vector data types
    - Type must be 1024 bits or less
  - Optionally specify depth of FIFO (Buffered Channel)
  - Declaring an array of channels produces independent channels

```cpp
channel int a; // Channel ‘a’ for ints
channel long b __attribute__((depth(8))); //buffered channel b
channel float4 c[2]; //Creates 2 float4 channels, c[0] and c[1]
```
Blocking Channel Reads and Writes

Function Prototypes

```
void write_channel_intel(channel <type> channel_id, const <type> data);

<typename> read_channel_intel(channel <type> channel_id);
```

- Each write adds a single piece of data to the channel
  - `write_channel_intel(a_channel, (float4) x);`

- Each read removes a single piece of data from the channel
  - `int x = read_channel_intel(b_channel);`

- `channel_id` identifies the buffer

- `write_channel_intel` blocks if the channel is full

- `read_channel_intel` blocks if the channel is empty

- `<type>` must match between reads and writes and channel handle
Non-Blocking Channel Reads and Writes

Function Prototype

```c
bool write_channel_nb_intel( channel <type> channel_id, const <type> data);
<type> read_channel_nb_intel(channel <type> channel_id, bool * valid);
```

- Like blocking calls except functions does not block, pipeline not stalled
- Functions returns bool value indicating if operation took place successfully
  - ```int x = read_channel_nb_intel(a_channel, &valid);```
  - ‘x’ gets data if ‘valid’ is true
  - ```valid = write_channel_nb_intel(b_channel, x);```
  - ‘b_channel” contains ‘x’ if ‘valid’ is true
- Useful if operation may not occur, when dealing with I/O channels, or to facilitate work distribution
Kernel Concurrency

- Channels designed to work with reading & writing kernels executing in parallel
  - Limited storage in the channel
  - Not the standard model for OpenCL™ kernels

- May require changes to the host code

- Use a separate command queue for each kernel
  - To allow for parallelism with in-order queues

```cpp
#define NUM_KERNELS ...
std::vector<cl::Kernel> kernels;
std::vector<cl::CommandQueue> myqueue;
```
Buffered Channels

- Default channels are 0-depth, i.e. no storage, read and write happens together
- Use the depth attribute to specify a minimum depth for the channel
- Use buffered channels if there are temporary imbalances btw. reads and writes
  - Prevents stall (profiler can detect stalls)
  - Conditional reads/writes may cause imbalance between reads/writes

```c
channel int c __attribute__((depth(20)));
__kernel void producer (...) {
    if (...) 
        write_channel_intel(c, ...)
}
__kernel void consumer (...) {
    if (...) 
        val=read_channel_intel(c)
}
```
I/O Channels

- Channels used with input or output features of a board
  - E.g., network interfaces, PCIe\* interfaces, camera interfaces, etc.
- Behavior defined by the Board Support Package (check board_spec.xml)

```
<channels>
  <interface name="udp_0" port="udp0_out" type="streamsource" width="256" chan_id="eth0_in"/>
  <interface name="pcie" port="tx" type="streamsink" width="32" chan_id="pcie_out" />
</channels>
```

- Declaration of I/O channel using the io attribute
  ```
  channel QUDPWord udp_in_IO __attribute__((io("eth0_in")));
  channel float data __attribute__((io("pcie_out")));
  ```

- Usage same as other channels
  ```
  data = read_channel_intel(udp_in_IO);
  ```
Implementing OpenCL™ Pipes

Implement pipes instead of channels for compatibility with other SDKs

- AOC implements pipes as a wrapper around channels
  - Channels are statically inferred from pipe arguments
  - Kernel CUs are connected via name matching
  - All rules that apply to channels also apply to pipes
    - Types supported, size limit, blocking/non-blocking behavior, etc.

- AOC does not support the entire pipes specification
  - Not fully OpenCL™ 2.0 conformant
Pipe Syntax, Kernel Side

- Pipes are specified as a kernel argument with the keyword `pipe`
  - `read_only` or `write_only` qualifier and data type required in declaration
- Read / Write to the pipe using `read_pipe()` and `write_pipe()` calls
  - Specify pipe name and address of variable to read/write

```c
__kernel void producer (write_only pipe uint p0) {
    for (...) {
        error = write_pipe(p0, &data);
    }
}

__kernel void consumer (read_only pipe uint p0) {
    for (...) {
        error = read_pipe(p0, &value);
    }
}
```

Compiler looks for matching pipe ID to form a HW connection.
Pipe Syntax, Host Side

- Use `clCreatePipe` to create the pipe object
  - Similar to `clCreateBuffer`, returns `cl_mem` object
- Use `clSetKernelArg` to map pipe to appropriate read and write kernel args
- Both of these functions has no affect on the creation of the pipe hardware
- Needs to be called to conform to the OpenCL™ standard

```c
cl_mem pipe = clCreatePipe(context, 0, sizeof(float), SIZE, NULL, &status);
clSetKernelArg(producer_kernel, 0, sizeof(cl_mem), &pipe);
clSetKernelArg(consumer_kernel, 0, sizeof(cl_mem), &pipe);
```
Pipe Attributes

- Apply `__attribute__((blocking))` for blocking behavior
  - Pipes are non-blocking by default

```c
__kernel void producer (write_only pipe uint __attribute__((blocking)) p0)
__kernel void consumer (read_only pipe uint __attribute__((blocking)) p0)
```

- Use `depth` attribute to specify the minimum depth of a pipe
  - If read and write depths differ, AOC uses the larger depth of the two

```c
#define SIZE 100
__kernel void producer (write_only pipe uint __attribute__((depth(SIZE))) p1)
__kernel void consumer (read_only pipe uint __attribute__((depth(SIZE))) p1)
```

- I/O Pipes with `io` attribute

```c
__kernel void myk (read_only pipe QUDPWord __attribute__((io("eth0_in))) UDP_in)
```
Channels / Pipes in the Area Report

Channel / pipe implementation shown in the detailed HTML area report

- Width implemented, Depth implemented (vs depth requested)
- Resources used

<table>
<thead>
<tr>
<th></th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Total (Logic: 15%)</td>
<td>50471</td>
<td>64586</td>
<td>349</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Board interface</td>
<td>38282</td>
<td>44528</td>
<td>257</td>
<td>0</td>
<td>Platform interface logic.</td>
</tr>
<tr>
<td>Global interconnect</td>
<td>5034</td>
<td>9568</td>
<td>52</td>
<td>0</td>
<td>Global interconnect for 1 global load and 1 global store.</td>
</tr>
<tr>
<td>Channel(_acl_p1_pipe_channel)</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>Channel is implemented 32 bits wide by 0 deep.</td>
</tr>
<tr>
<td>channels:ct:4 (c0)</td>
<td>49</td>
<td>157</td>
<td>1</td>
<td>0</td>
<td>Channel is implemented 32 bits wide by 256 deep. Requested depth was 128.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Channel depth was changed for the following reasons:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- instruction scheduling requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- nature of underlying FIFO implementation</td>
</tr>
</tbody>
</table>
NDRange and Single Work-Item Kernel Interaction with Channels/Pipes

- Single Work-Item and NDRange Kernel can interact predictably
- Algorithm may naturally split into both single work-item and NDRange kernels
- Ex. Generating random data for a Monte Carlo simulation:

```c
kernel void rng(int seed) {
    int r = seed;
    while(true) {
        r = rand(r);
        write_channel_intel(RAND, r);
    }
}
```

```c
kernel void sim(...) {
    int gid = get_global_id(0);
    int rnd = read_channel_intel(RAND);
    out[gid] = do_sim(data, rnd);
}
```
Arbitration with Non-Blocking Channels/Pipes

```c
kernel void arb2to1(...) {
    bool v = false;
    while(true) {
        int d = read_channel_nb_intel(C_IN1, &v);
        if(!v)
            d = read_channel_nb_intel(C_IN2, &v);
        if(v)
            write_channel_intel(C_OUT, d);
    }
}
```
Channel / Pipe Example Application

- Three Kernels:
  - Read Kernel -- (Transfers data from DDR to channel)
  - Streamer Kernel -- (Reads from input channel, processes data, and writes to output pipe)
  - Write Kernel -- (Transfers data from pipe to DDR)

- Separate queues needed to launch kernels in parallel
#pragma OPENCL EXTENSION cl_intel_channels : enable

channel uint c0 __attribute__((depth(128)));

kernel void host_reader( global const uint *src) {
    size_t gID = get_global_id(0);
    write_channel_intel(c0, src[gID]);
}

kernel void streamer( write_only pipe uint p1 __attribute__((blocking)), int N) {
    uint iData;
    for (unsigned i=0; i<N; i++) {
        iData = read_channel_intel(c0);
        iData = word_convert(iData);
        write_pipe(p1, &iData);
    }
}

kernel void host_writer( global uint *dst, read_only pipe uint p1 __attribute__((blocking))) {
    size_t gID = get_global_id(0);
    uint value = 0;
    read_pipe(p1, &value);
    dst[gID] = value;
}

This NDRange kernel reads data from the host and sends it to channel c0.

This single work-item kernel processes data from c0 and passes it to p1.

This NDRange kernel reads data from pipe p1 and writes data to host.
Host Pipes

- Allow host to send/receive data to/from the kernels without global memory
  - Performance advantage
  - Achieve peak host-to-device bandwidth

Host Code

```c
#pragma OPENCL EXTENSION cl_intel_fpga_host_pipe : enable
kernel void reader(__attribute__((intel_host_accessible))
  __read_only pipe ulong4 host_in) {....}
kernel void writer(__attribute__((intel_host_accessible))
  __write_only pipe ulong4 device_out) {....}
```

Kernel Code

```c
cl_mem read_pipe = clCreatePipe( context,CL_MEM_HOST_READ_ONLY, ...);
cl_mem write_pipe = clCreatePipe( context, CL_MEM_HOST_WRITE_ONLY, ...);
clReadPipeIntelFPGA (read_pipe, &val);
clWritePipeIntelFPGA (write_pipe, &val);
```
Pipes vs Channels

- Most cases they are the same
  - Usage and Performance

- Use Pipes
  - Partially conformant to OpenCL™ standards
    - Needs modification from OpenCL 2.0 Pipes

- Use Channels
  - With autorun kernels
  - Use model more aligned with FPGA implementation
    - Pipe usage more verbose, especially on the host side

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
Optimizing Memory Accesses
Optimizing Memory Accesses Agenda

- Overview
- Global/constant memory
- Local memory
- Private memory
- Host memory
OpenCL™ Memory Model

- **Global Memory**
  - Off-chip memory (DDR / QDR / HMC)
  - Slow for non-sequential access

- **Constant Memory**
  - Visible to all workgroups
  - Accessed through shared cache

- **Local Memory**
  - Shared within workgroup
  - FPGA on-chip memory
  - Much higher bandwidth and lower latency than global memory

- **Private Memory**
  - Unique to a work-item
  - FPGA registers or on-chip memory

- **Host Memory (Separate CPU Memory)**
Need to Optimize Memory Accesses

- In many real-world algorithms, data movement through memory is often the bottleneck

- Memory access efficiency often determine overall performance of a kernel
  - Large performance gains can be achieved from optimization effort

- Global Memory
  - Maximum global memory BW is much smaller than maximum local memory BW
  - Maximum computational BW of the FPGA is much larger than the global memory BW
  - Increases in kernel performance leads to increases in global memory BW requirements
HTML Report: System Viewer and Memories

- Stall point graph that include load and store information between kernel pipeline and memories
- Verify memory replication
- Identify stallable loads and stores
- See type of LSU implemented
System Viewer: Visualize Memory Accesses

- Visualize Connections from each load/store to local and global memory
HTML Area Report for Memory Implementation

- Shows global and constant cache interconnect implemented
- Reports type of global load store unit implemented
- Local memory implementation reported
  - Overall state: Optimal, Good but replicated, Potentially inefficient
  - Total size, replication factors, stallable/stall-free, merging, banking, number of reads and writes
- Shows private variable implementation

![HTML Area Report Table]

<table>
<thead>
<tr>
<th>Area Report</th>
<th>(all utilization values are estimated)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LSs</td>
</tr>
<tr>
<td>System Total</td>
<td>0.01%</td>
</tr>
<tr>
<td>Local intensive</td>
<td>0.02%</td>
</tr>
<tr>
<td>Global interconnect</td>
<td>0.04%</td>
</tr>
<tr>
<td>Constant cache interconnect</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

- LSs: Load Store Unit
- PPs: Processor Part
- RAs: Repeater Array
- DPs: Data Part

Legend:
- Total size
- Replication factors
- Stallable/stall-free
- Merging
- Banking
- Number of reads and writes

Note: This report is based on estimated values and may not reflect actual system performance.

Legend for System Total:
- 0.01% of total

Legend for Local intensive:
- 0.02% of total

Legend for Global interconnect:
- 0.04% of total

Legend for Constant cache interconnect:
- 0.00% of total
HTML Kernel Memory Viewer

Displays the local memory present in your design

Illustrates:

- Memory replication
- Banking
- Implemented arbitration
- Read/write capabilities of each memory port
Dynamic Profiler and Memory Accesses

- Displays statistics about each memory accesses on source code tab
  - Entry shows type of access: global / local
  - At access location, displays pipeline stall %, occupancy %, average bandwidth, efficiency%, cache hit%, non-aligned access, burst, and coalescing

Kernel tab shows overall statistics
Optimizing Memory Accesses Agenda

- Overview
- Global/constant memory
- Local memory
- Private memory
- Host memory
Global Memory in OpenCL™

- **global address space**
  - Used to transfer data between host and device
  - Used for kernel-to-kernel communication
  - Shared by all work-items in all workgroups

- Generally allocated on host as `cl::Memory` object
  - Created/allocated with `cl::Buffer` constructor
  - Data transferred using `cl::enqueue[Read/Write]Buffer` method
  - Object assigned to global pointer argument of kernels

```cpp
__kernel void add(__global float* a, __global float* b, __global float* c)
{
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos*
OpenCL™ BSP Global Memory

- Global memory controllers and devices defined by the Board Support Package
- Global memory interconnect built by the kernel compiler
Compiler Generated Hardware

- Custom global interconnect created
- LSU type selected by the compiler
  - Performs Width Adaptation
    - User data (e.g. 32-bit int) to memory word (512-bit DRAM word)
    - Coalesced to avoid wasted bandwidth

```c
#include <stdio.h>

int main() {
    global int* x;
   ...
    int y = x[k];
    return 0;
}
```
LSU Types

- **Burst-Coalesced**
  - Most common global memory LSU
  - Specialized LSU to groups loads/stores into bursts
  - LSU for load can cache/re-use data
    - Private caching is applied heuristically

- **Streaming**
  - Simplified version of burst-coalesced LSU that supports only completely linear accesses

- **Pipeliined**
  - Used for local memory

- **And others**
Global Memory Load Store Units in the Area Report

Implementation of LSUs annotated with source line

- Include size of cache, situations when cache is created, and other tips

```c
kernel void gl_test(global int * restrict in, global int * restrict out) {
    int i = get_global_id(0);
    int idx = out[i]; // idx is data-dependent

    int cached_value = in[idx]; // this is a cached LSU (burst-coalesced-cached).

    out[i] = cached_value;
}
```

- Load with a private 512 kilobit cache. Cache is not shared with any other load. It is flushed on kernel start. Use Dynamic Prefetch to verify cache effectiveness. Other kernels should not be updating the data in global memory while this kernel is using it. Cache is created when memory access pattern is data-dependent or appears to be repetitive. Simplify access pattern or mark pointer as ‘volatile’ to disable generation of this cache.

<table>
<thead>
<tr>
<th>Block0 (Logic: 1%)</th>
<th>4727 (1%)</th>
<th>6841 (1%)</th>
<th>72 (3%)</th>
<th>0 (0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>caching.cl3</td>
<td>354</td>
<td>402</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>caching.cl5</td>
<td>3094</td>
<td>4489</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>caching.cl7</td>
<td>1247</td>
<td>1918</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>
Arbitration Interconnect to Global Memory

- Generated automatically by the compiler
- Arbitrate to physical interfaces
  - Tree interconnect (high bandwidth)
    - OR
      - Ring interconnect (high $f_{max}$)
        - Increase reliance on large bursts
      - Arbitration type chosen base on # of LSUs
- Distribute (load balance) across physical interfaces
Kernel Argument Constant Memory

- Written to global memory and likely constant cache by the host
  - Can be modified later by the host, shared by all work-groups

- Use for read-only data that all work-groups access
  - E.g. high-bandwidth table lookups

- Constants kernel arguments are also stored in on-chip memory if possible
  - Optimized for 100% cache hit performance
  - Default size is 16kB
    - Shared by all constant arguments
    - Can be set at kernel compile time

__kernel void my_kernel(__constant float * restrict coef) ...
...
Complete Picture

Pipeline

Load Unit

decoupled

stream

Load Unit

Coalesce

Load Unit

Cache

Constant Load Unit

high BW

Constant Load Unit

Constant Cache

Arbitration

Low Bandwidth

Global Memory
Global / Constant Cache Interconnect Area Report

- **Global interconnect** – accessing external memory (e.g. DDR4)
  - Number of global loads and stores affects area

- **Constant cache interconnect** – accessing memory marked as `constant`
  - Number of reads affects replication which affects area
  - Include tips for improving performance

<table>
<thead>
<tr>
<th>Area Report (area utilization values are estimated)</th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Total (Logic: 16%)</td>
<td>50902</td>
<td>72669</td>
<td>391</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Board Interface</td>
<td>39232</td>
<td>44526</td>
<td>257</td>
<td>0</td>
<td>• Platform Interface logic.</td>
</tr>
<tr>
<td>Global Interconnect</td>
<td>6034</td>
<td>9568</td>
<td>52</td>
<td>0</td>
<td>• Global Interconnect for 0 global loads and 2 global stores.</td>
</tr>
<tr>
<td>Constant cache interconnect</td>
<td>894</td>
<td>9500</td>
<td>44</td>
<td>0</td>
<td>• 16384 bytes constant cache accessible to all kernels and is persistent across kernel invocations. Data inside the cache is replicated 2 times to support 4 reads. Cache optimised for hits, misses incur a large penalty. If amount of data in the cache is small, consider passing it by value as a kernel argument. Use Dynamic Profiler to check stats on accesses to the cache to assess the cache's effectiveness. Profiling actual cache hit rate is currently not supported.</td>
</tr>
</tbody>
</table>

```c
// kernel void A(constant int *src, global int *dst) {
    int i = get_global_id(0);
    dst[i] = src[i] + src[i + 1] + src[i >> 1];
}

// kernel void B(constant int *src, global int *dst) {
    int i = get_global_id(0);
    dst[i] = src[i] + src[i + 1] + src[i >> 1];
}
```
File Scope __constant

- File scope __constant variables supported
- Dedicated on-chip ROM resources allocated for each variable
  - Not shared with __constant arguments, not stored in global memory
  - In-lined into the kernel compute unit

```c
__constant int my_array[4] = {0x0, 0x1, 0x2, 0x3};

__kernel void my_kernel (__global int * my_buffer)
{
    size_t gid = get_global_id(0);
    my_buffer[gid] += my_array[gid % 4];
}
```
Heterogeneous Memory

- Some BSPs offer more than one type of global memory
  - DDR, QDR, HMC, etc.
- Memory location can be set per kernel argument using
  - Using `buffer_location` ("MEMORY_NAME")

```c
__kernel void foo (  
  global int *x, // Default memory location (usually DDR)  
  global __attribute__((buffer_location("DDR"))) int *y,  
  global __attribute__((buffer_location("QDR"))) int *z,  
  global __attribute__((buffer_location("HMC"))) int *x  )

cl::Buffer mybuf(context, CL_MEM_HETEROGENEOUS_INTEL, size, NULL, &errNum);
```
Global Memory Banking Optimizations

- Global memory addresses can be set as interleaved or partitioned by bank (controller)
- Burst-interleaved is the default
  - Best for sequential traffic and for load balancing between memory banks
  - Same behavior as GPUs
- Interleaving granularity set by BSP in XML
  - Usually $\text{width} \times \text{maxburst}$

```xml
<!-- DDR3-1600 -->
<global_mem name="DDR" max_bandwidth="25600" interleaved_bytes="1024" config_addr="0x018">
  <interface name="board" port="kernel_mem0" type="slave" width="512" maxburst="16" address="0x00000000" size="0x100000000" latency="240"/>
  <interface name="board" port="kernel_mem1" type="slave" width="512" maxburst="16" address="0x100000000" size="0x100000000" latency="240"/>
</global_mem>
```

- Address
  - 0x7FF_FFFF
  - 0x7FFFFC00
  - 0x7FFF_F800
  - 0x0000_0800
  - 0x0000_0400
  - 0x0000_0000
Manually Partitioning Global Memory

- Turn off interleaving
  - `aoc <kernel file>.cl -no-interleaving <memory_type>`

- Allocate each memory buffer to one of the banks
  - **Use** `CL_CHANNEL... flags`
  - Allocate each buffer to designated memory bank only

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bank Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CL_CHANNEL_1_INTELFPGA</code></td>
<td>Allocates to lowest available memory region</td>
</tr>
<tr>
<td><code>CL_CHANNEL_2_INTELFPGA</code></td>
<td>Allocates to the second memory bank</td>
</tr>
<tr>
<td><code>CL_CHANNEL_n_INTELFPGA</code></td>
<td>Allocates to the n\textsuperscript{th} bank (as long as the board supports it)</td>
</tr>
</tbody>
</table>

```
c::Buffer mybuf(context, CL_CHANNEL_2_INTELFPGA, size, 0, 0);
```
Matrix Multiplication: Global Memory (default)

for (int i = 0; i < WIDTH; i++) {
    Csub += A[y * WIDTH + i] * B[x + WIDTH * i];
}
C[y * WIDTH + x] = Csub;
Matrix Multiplication: Global Memory (partitioned)

- Optimize matrix A and B access
  - By using separate banks
- C is rarely accessed so don’t care

```c
for (int i = 0; i < WIDTH; i++) {
    Csub += A[y * WIDTH + i] * B[x + WIDTH * i];
}
C[y * WIDTH + x] = Csub;
```
Optimizing Memory Accesses Agenda

- Overview
- Global/constant memory
- Local memory
- Private memory
- Host memory
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

```c
__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

```c
__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
}
```

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

Local memory pointer argument
`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

```c
__kernel void my_kernel (  
    __local float* A,  
    __attribute__((local_mem_size(1024))) __local float* B,  
    __attribute__((local_mem_size(32768))) __local float* C)  
{  
    ...  
}
```

- 16kB allocated for A
- 1kB allocated for B
- 32kB allocated for C
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 accommodates $\#\text{loads} + \#\text{stores} \leq 2$

- Kernels may require many complex accesses

- Compiler optimizes kernel pipeline, interconnect and memory system
  - Through splitting, coalescing, banking, replication, double-pumping, port sharing
Interconnect: Port Sharing

- 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
Automatic Double Pumping

1x

Port 1
Port 2

Block RAM

Port 1
Port 2
Port 3
Port 4

2xCLK

2x

Port 1
Port 2
Port 3
Port 4

2xCLK

Memory
2x clock

store
load
load
load
load
load
load

array
Bank 0

LD
R

LD
R

ST
W

ST
W
Replication

Up to four ports with doublepump

1-3 write

Y-read

store
load
load
load
load
load
load
load
load
load
load

Port0
Port1
Port2
Port3

Memory
2x clock

Port0
Port1
Port2
Port3

Memory
2x clock
Local Memory Replication Example

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

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

    calc = res;
}
```

1 write port, 9 read ports  
Up to 3 read ports, 1 write port per replicant (double pump)  
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 access patterns and decomposition of the address

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

- Example, $B[i][j]$ accesses address =
  - $B + ((i \times 32 + j) \times \text{sizeof(float)})$
  - Access is always at a 32-bit boundary
  - More powerful information inferred from related accesses
Static Coalescing

- Components often access consecutive addresses (variable A)

```c
__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
```c
__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

- 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

```c
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];
    ...
}
```
Banking

Use multiple banks on lower bits to implement the memory

```c
__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]),
}
```
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:

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

- 2D, coalesced

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

- 2D, banked

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

```
<table>
<thead>
<tr>
<th>free_replication.cl.9 (lmem)</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
```

- 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 – 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

<table>
<thead>
<tr>
<th>31mem_nospilt.cl:9 (imem0)</th>
<th>33</th>
<th>512</th>
<th>96</th>
<th>0</th>
</tr>
</thead>
</table>

- 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: 31mem_nospilt.cl:10, 31mem_nospilt.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

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

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

### Control Memory Architecture Using Attributes

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

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

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

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

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

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

```
```
__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**

```c
__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)];
}
```

Simultaneous Accesses, No arbitration needed with optimal banking.
Local Memory Attribute Example

- Using attributes to control replication factor

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

- No replication needed

```c
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
  - $\leq 4$ 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

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

```
//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];
```
Matrix Multiplication: Swapping Indices

- Convert the access to local memory $B_{\text{local}}$ to be linear and thus much easier for the compiler to analyze

```c
B_{\text{local}}[local_y][local_x] = B[b + WIDTH * local_y + local_x];
...
Csub += A_{\text{local}}[local_y][k] * B_{\text{local}}[k][local_x];
```

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

```c
B_{\text{local}}[local_x][local_y] = B[b + WIDTH * local_y + local_x];
...
Csub += A_{\text{local}}[local_y][k] * B_{\text{local}}[local_x][k];
```
Matrix Multiplication: Local Memory Optimized

```c
#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.

<table>
<thead>
<tr>
<th>Kernel System (logic: 585)</th>
<th>ALUs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board interface</td>
<td>3262</td>
<td>44528</td>
<td>257</td>
<td>0</td>
<td>Platform i...</td>
</tr>
<tr>
<td>Global Interconnect</td>
<td>8779</td>
<td>12545</td>
<td>78</td>
<td>0</td>
<td>Global Int...</td>
</tr>
<tr>
<td><strong>matrixMult</strong></td>
<td>139810 (27%)</td>
<td>208050 (208)</td>
<td>722 (208)</td>
<td>264 (138)</td>
<td>Achieved k... Number of...</td>
</tr>
<tr>
<td>Data control overhead</td>
<td>192</td>
<td>5225</td>
<td>14</td>
<td>0</td>
<td>State + Fe...</td>
</tr>
<tr>
<td>Function overhead</td>
<td>1700</td>
<td>1702</td>
<td>0</td>
<td>0</td>
<td>Kernel diss...</td>
</tr>
<tr>
<td>matrix_mult.cl:111 (A_local)</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>0</td>
<td>Local mem...</td>
</tr>
<tr>
<td>matrix_mult.cl:112 (B_local)</td>
<td>0</td>
<td>0</td>
<td>256</td>
<td>0</td>
<td>Local mem...</td>
</tr>
<tr>
<td><strong>No Source Line</strong></td>
<td>848</td>
<td>3685</td>
<td>17</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**matrix_mult.cl:112 (B_local):**
- Local memory: Optimal.

Requested size 16784 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
Optimizing Memory Accesses Agenda

- Overview
- Global/constant memory
- Local memory
- Private memory
- Host memory
Private Memory Implemented as Registers

- Private variables and arrays can be implemented as:
  - On-chip memory systems.
  - Pipeline registers or FIFOs

- Unless the private variables match a register conversion rule, the result is equivalent to local memory
  - All tradeoffs, reports, and discussion about local memory applies

- Scalar variables (float, int, char, etc.) almost always implemented in registers

- Aggregate types (arrays, struct and vectors) can be converted to registers
  - If members accessed can be determined at compile-time.

```c
__kernel void MyKernel(...) {
  __private float pData[4];
  ...
}
```
Private Memory Implemented in RAM

- If accesses are not constant, memory implemented in on-chip RAM
  - `temp` is implemented in RAM
  - loads/stores are used to access data

```c
kernel void foo(global int* restrict A, global int* restrict B) {
    int temp[20];

    for(unsigned i = 0; i < 20; i++) {
        temp[i] = A[i];
    }

    for(unsigned i = 0; i < 20; i++) {
        B[i] = temp[i] + temp[N-1-i];
    }
}
```
Private Memory Implemented as Registers (Constant access)

- Each element of `temp` becomes a register

```c
int temp[20];
#pragma unroll
for(unsigned i = 0; i < 20; i++)
    temp[i] = A[i];
#pragma unroll
for(unsigned i = 0; i < 20; i++)
    B[i] = temp[i] + temp[N-1-i];
```

```c
int temp[20];
#pragma unroll
for(unsigned i = 0; i < 20; i++)
    temp[i] = A[i];
```
Private Memory Implemented as Registers (Size Requirement)

- Private memory of size < 64 bytes always converted to registers
  - Compiler heuristic
  - `temp` becomes a 160-bit register
  - Shift operations are used to extract the 32-bit data to operate on

```c
kernel void foo(global int* restrict A, global int* restrict B) {
    int temp[5];

    for(unsigned i = 0; i < 5; i++) {
        temp[i] = A[i];
    }

    for(unsigned i = 0; i < 5; i++) {
        B[i] = temp[i] + temp[N-1-i];
    }
}
```
Private Memory Describing Shift Registers

- Shift register inferred

```c
pixel_t sr[2*W+3];
while (keep_going) {
    // Shift data in
    #pragma unroll
    for(int i=1; i<2*W+3; ++i)
        sr[i] = sr[i-1];
    sr[0] = data_in;
    ...  
    // Tap output data
    data_out = {sr[ 0], sr[  1], sr[  2],
                sr[ W], sr[ W+1], sr[ W+2],
                sr[2*W], sr[2*W+1], sr[2*W+2]}
    ...
}
```
Shift Register Implementation

- Inference result from access pattern
- Each element of the shift register is converted from memory to register
- All registers are then clustered together into 1 or several shift registers
- Shift registers can be backed by any array shape
  - The compiler will infer shift registers after the arrays are broken into individual elements

Shift register has frequent accesses

- If conversion to shift registers fails, due to the coding style, a large number of loads and stores to memory will be instantiated
Area Report: Private Variables Implemented as Registers

- Private variables implemented as registers annotated

```
#define FF_SIZE (64)

kernel void t(global int * restrict src, global int * restrict dst, int N) {
    int delay_fifo[FF_SIZE];
    #pragma unroll
    for (int k = 0; k < FF_SIZE; ++k) {
        delay_fifo[k] = k;
    }
    #pragma unroll
    for (int i = 0; i < N; ++i) {
        dst[i] = delay_fifo[0];
        #pragma unroll
        for (int k = 0; k < FF_SIZE - 1; ++k) {
            delay_fifo[k] = delay_fifo[k + 1];
        }
        delay_fifo[FF_SIZE - 1] = src[i];
    }
}
```

<table>
<thead>
<tr>
<th>Private Variable:</th>
<th>304</th>
<th>4528</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>'delay_fifo'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(not_shift_reg.cl.3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Implemented using registers of the following size:
  - 64 registers of width 32 and depth 0
Area Report: Private Variables Implemented as Shift Registers

- Private variables implemented as shift registers reported
  - See details about the individual registers used to implement the whole array

<table>
<thead>
<tr>
<th>Private Variable: <em>sr</em> (shift_reg.cl(5))</th>
<th>168</th>
<th>363</th>
<th>36</th>
<th>0</th>
</tr>
</thead>
</table>

- Implemented as a shift register with 6 or fewer tap points. This is a very efficient storage type. Implemented using registers of the following sizes:
  - 1 register of width 15 and depth 1
  - 3 registers of width 32 and depth 1
  - 1 register of width 32 and depth 10
  - 1 register of width 32 and depth 14
  - 1 register of width 32 and depth 10001
  - 1 register of width 32 and depth 15380
Area Report: Private Variables Implemented as Barrel Shifters

- Arrays that are indexed dynamically may be implemented as a high-overhead barrel shifters
- Warning issued
  - Static indexing would yield much better results
Area Report: Private Variables Implemented as ROM

- Private large constant array can be implemented as ROM
- ROMs are replicated for each read
- Resources used are shown on lines where accesses occur

<table>
<thead>
<tr>
<th>[] Block0 (Logic: 1%)</th>
<th>LEs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rom.cl:4</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>rom.cl:5</td>
<td>16</td>
<td>0</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>rom.cl:6</td>
<td>1220</td>
<td>2022</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>No Source Line</td>
<td>908</td>
<td>1918</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Two accesses to private constant `tbl[]`

```c
#include "tbl.h"

kernel void f(global int * dst) {
  int i = get_global_id(0);
  int res = tbl[i];
  res += tbl[i + 23];
  dst[i] = res;
}
```
Data Type Optimizations
Floating-Point Optimizations

- Apply to half, float and double data types
- AOC has the ability to optionally optimize for floating-point operations
  - Optimizations will cause small differences in floating-point results
    - Not IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) compliant
- AOC floating-point optimizations:
  - Tree Balancing
  - Reducing Rounding Operations
- Other optimizations
  - Floating-point vs. fixed-point representations
  - Use a device with hard floating point
Tree-Balancing

- Floating-point operations are not associative
  - Rounding after each operation affects the outcome
  - ie. \(((a+b) + c) \neq (a+(b+c))\)

- By default the compiler doesn’t reorder floating-point operations
  - May creates an imbalance in a pipeline, costs latency and possibly area

- Manually enable compiler to balance operations
  - For example, create a tree of floating-point additions in SGEMM, rather than a chain
  - Use `-fp-relaxed=true` flag when calling `aoc`
Arithmetic Order of Operation Rules

- Strict order of operation rules apply in OpenCL™
- By default, AOC honors those rules
  - May lead to long, unbalanced, slower, less-efficient floating-point operations
- Example:

\[ \text{Result} = (((A \times B) + C) + (D \times E)) + (F \times G) \]
Tree Balancing

- Allow AOC to reorder operations to convert into a tree pipeline structure
  - Possibly affects the precision, not consistent with IEEE 754
- Enable AOC tree balancing with `-fp-relaxed` option
  - Design needs to tolerate the small differences in floating-point results

```
result = (((A * B) + C) + (D * E)) + (F * G)
```

**Same Operation, Balanced Tree Implementation**
Tree Balancing and Resource Savings
Rounding Operations

- For a series of floating-point operations, IEEE 754 require multiple rounding operation
- Rounding can require significant amount of hardware resources
- Fused floating-point operation
  - Perform only one round at the end of the tree of the floating-point operations
  - Leads to more accurate results
  - Other processor architectures support certain fused instructions such as fused multiply and accumulate (FMAC)
  - AOC can fuse any combination of floating-point operators
Reducing Rounding Operations

- AOC will not reduce rounding operations by default

- Enable AOC rounding reduction with \(-fpc\) option
  - Not IEEE 754 compliant
  - Use when program can tolerate these differences in floating-point results

1. Removes floating-point rounding operations whenever possible
   - Round floating-point operation only once at the end of the tree of operations
     - Applies to \(*\), \(+\), and \(-\)

2. Carry additional mantissa bits to maintain precision
   - Carries additional bits through calculations, removed at the end of the tree of operations

3. Changes rounding mode to round toward zero

\(\text{aoc} \ -fpc \ <\text{kernel_file}.\cl\)
Implementing Arbitrary Precision Integers

- Include the library in your .cl file: `#include "ihc_apint.h"

- Aoc run with the option: `-l $INTELFPGAOCLSDKROOT/include/kernel_headers`

```c
#include "ihc_apint.h"

__kernel void fixed_point_add(__global const unsigned int * restrict a,
                               __global const unsigned int * restrict b,
                               __global unsigned int * restrict result)
{
    size_t gid = get_global_id(0);
    ap_uint10 temp, temp2;
    ap_uint20 temp_result;
    temp = a[gid]; temp2 = b[gid];
    temp_result = ((int20_t)a) * b;
    result[gid] = temp_result;
}
```

Datatypes available are `ap_uint<bit size>` and `ap_int<bit size>`

Make sure to cast one of the arguments to account for bit growth to prevent overflow.
Summary

- NDRange kernel attribute customizes Compute Unit architecture
- Effective Loop Pipelining
- Communication through Channels / Pipes
- Memory Optimizations
- Data Type Considerations
References

▪ Intel® OpenCL™ collateral ([www.altera.com/OpenCL](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
Follow-on Training

- Single-Threaded vs. Multi-Threaded Kernels online training
- Building Custom Platforms online training
Many Ways to Learn

**Videos**
FREE
Always available
~4 minutes long
YouTube videos

**Online Training**
FREE
Always available
~30 minutes long
>200 topics
English, Chinese, Japanese

**Virtual Classes**
Live over WebEx*
Training Center
Ask questions to Intel® FPGA expert
Hands on labs
Taught in ½ day sessions
Class schedules at
[www.altera.com/training](http://www.altera.com/training)

**Instructor-led Training**
In-person
Ask questions to Intel® FPGA expert
Hands on labs
1 day long
Class schedules at
[www.altera.com/training](http://www.altera.com/training)
Instructor-Led and Virtual Training Curriculum

Programmable Solutions Group

239

*Other names and brands may be claimed as the property of others
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
Exercise 4

Optimizing the Hough Transform
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 Corporation