Intel® FPGA SDK for OpenCL™ Pro Edition

Best Practices Guide

Updated for Intel® Quartus® Prime Design Suite: **22.4**

Online Version

Send Feedback

UG-OCL003

ID: 683521

Version: 2022.12.19

Contents

1. Introduction to Intel® FPGA SDK for OpenCL™ Pro Edition Best Practices Guide............ 5
1.1. FPGA Overview..5
1.2. Pipelines...7
1.3. Single Work-Item Kernel versus NDRange Kernel.. 9

2. Reviewing Your Kernel's report.html File.. 15
2.1. High-Level Design Report Layout...15
2.2. Reviewing the Summary Report.. 16
2.3. Viewing Throughput Bottlenecks in the Design.. 18
2.4. Using Views.. 19
2.4.1. Features of the System Viewer...19
2.4.2. Features of the Kernel Memory Viewer 26
2.4.3. Features of the Schedule Viewer.. 31
2.5. Analyzing Throughput..32
2.5.1. Reviewing Loop Information... 33
2.6. Reviewing Area Information.. 36
2.6.1. Area Report Message for Board Interface.. 38
2.6.2. Area Report Message for Function Overhead... 38
2.6.3. Area Report Message for State..38
2.6.4. Area Report Message for Feedback... 38
2.6.5. Area Report Messages for Private Variable Storage............................... 38
2.7. Optimizing an OpenCL Design Example Based on Information in the HTML Report........ 40
2.8. Accessing HLD FPGA Reports in JSON Format...49

3. OpenCL Kernel Design Concepts.. 51
3.1. Kernels...51
3.2. Global Memory Interconnect..52
3.3. Local Memory..54
3.3.1. Changing the Memory Access Pattern Example.....................................61
3.4. Loops in a Single Work-Item Kernel.. 66
3.4.1. Trade-Off Between Initiation Interval and Maximum Frequency..................... 67
3.4.2. Loop-Carried Dependencies that Affect the Initiation Interval of a Loop........... 68
3.4.3. Nested Loops.. 70
3.4.4. Loop Speculation... 78
3.4.5. Loop Fusion..80
3.4.6. Loop Bottlenecks... 81
3.5. Channels.. 84
3.6. Load-Store Units..85
3.6.1. Load-Store Unit Types.. 85
3.6.2. Load-Store Unit Modifiers..87
3.6.3. Controlling the Load-Store Units... 89
3.6.4. When to Use Each LSU... 90

4. OpenCL Kernel Design Best Practices...92
4.1. Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes...............92
4.1.1. Characteristics of Channels and Pipes... 93
4.1.2. Execution Order for Channels and Pipes.. 95
4.1.3. Optimizing Buffer Inference for Channels or Pipes................................. 96

4.1.4. Best Practices for Channels and Pipes.. 97
4.2. Unrolling Loops..97
4.3. Optimizing Floating-Point Operations.. 99
4.3.1. Floating-Point versus Fixed-Point Representations.....................................101
4.4. Allocating Aligned Memory.. 102
4.5. Aligning a Struct with or without Padding... 103
4.6. Maintaining Similar Structures for Vector Type Elements...105
4.7. Avoiding Pointer Aliasing.. 105
4.8. Avoid Expensive Functions...106
4.9. Avoiding Work-Item ID-Dependent Backward Branching.. 107

5. Profiling Your Kernel to Identify Performance Bottlenecks....................................... 108

5.1. Best Practices for Profiling Your Kernel... 109
5.2. Instrumenting the Kernel Pipeline with Performance Counters (-profile)....................109
5.3. Obtaining Profiling Data During Runtime...110
5.3.1. Invoking the Profiler Runtime Wrapper 110
5.3.2. Viewing Profiling Data Using Intel VTune™ Profiler.....................................111
5.4. Reducing Area Resource Use While Profiling.. 112
5.5. Temporal Performance Collection...112
5.5.1. Profiling Autorun Kernels.. 113
5.6. Performance Data Types..114
5.7. Interpreting the Profiling Information..114
5.7.1. Stall, Occupancy, Bandwidth.. 115
5.7.2. Stalling Channels.. 116
5.7.3. Channel Depths.. 117
5.8. Profiler Analyses of Example OpenCL Design Scenarios 117
5.8.1. High Stall Percentage..117
5.8.2. Low Occupancy Percentage...118
5.8.3. High Stall and High Occupancy Percentages.. 118
5.8.4. No Stalls, Low Occupancy Percentage, and Low Bandwidth......................... 119
5.8.5. No Stalls, High Occupancy Percentage, and Low Bandwidth.........................120
5.8.6. High Stall and Low Occupancy Percentages...121
5.9. Intel FPGA Dynamic Profiler for OpenCL Limitations.. 121

6. Strategies for Improving Single Work-Item Kernel Performance.............................. 123

6.1. Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback...123
6.1.1. Removing Loop-Carried Dependency...124
6.1.2. Relaxing Loop-Carried Dependency...127
6.1.3. Transferring Loop-Carried Dependency to Local Memory............................. 129
6.1.4. Relaxing Loop-Carried Dependency by Inferring Shift Registers................... 130
6.1.5. Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays 132
6.2. Good Design Practices for Single Work-Item Kernel...134

7. Strategies for Improving NDRange Kernel Data Processing Efficiency....................... 137

7.1. Specifying a Maximum Work-group Size or a Required Work-Group Size................... 137
7.2. Kernel Vectorization...139
7.3. Multiple Compute Units.. 140
7.3.1. Compute Unit Replication versus Kernel SIMD Vectorization........................ 141
7.4. Combination of Compute Unit Replication and Kernel SIMD Vectorization.................. 143
7.5. Reviewing Kernel Properties and Loop Unroll Status in the HTML Report................... 144

8. Strategies for Improving Memory Access Efficiency..145
8.1. General Guidelines on Optimizing Memory Accesses...146
8.2. Optimize Global Memory Accesses..146
8.2.1. Contiguous Memory Accesses..148
8.2.2. Manual Partitioning of Global Memory..149
8.2.3. Optimizing for Two or More Banks of Global Memory.................................150
8.3. Performing Kernel Computations Using Constant, Local or Private Memory................151
8.3.1. Constant Cache Memory...151
8.3.2. Preloading Data to Local Memory..151
8.3.3. Storing Variables and Arrays in Private Memory...153
8.4. Improving Kernel Performance by Banking the Local Memory.................................154
8.4.1. Optimizing the Geometric Configuration of Local Memory Banks Based on Array Index..156
8.5. Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor.... 157
8.6. Minimizing the Memory Dependencies for Loop Pipelining.......................................159
8.7. Static Memory Coalescing...160

9. Strategies for Optimizing FPGA Area Usage.. 162
9.1. Compilation Considerations...162
9.2. Board Variant Selection Considerations..162
9.3. Memory Access Considerations...163
9.4. Arithmetic Operation Considerations..164
9.5. Data Type Selection Considerations..165

10. Strategies for Optimizing Intel Stratix 10 OpenCL Designs..................................... 166
10.1. Reducing Channel Overhead...166
10.1.1. Reducing the Number of Kernels...166
10.1.2. Using a Single Kernel to Describe Systolic Arrays....................................167
10.1.3. Using Non-Blocking Channels...168
10.2. Optimizing Loop Control...169
10.2.1. Simplifying Loop-Carried Dependencies in Intel Stratix 10 OpenCL Designs.. 171
10.3. Simplifying Memory Access to Local Memories...172
10.4. On-Chip Storage of Reused Data...173
10.5. Optimizing Data Path Control 174
10.6. Creating RTL Modules...177
10.6.1. Reset Recommendations...178

11. Strategies for Improving Performance in Your Host Application............................. 180
11.1. Multi-Threaded Host Application..180
11.2. Utilizing Hardware Kernel Invocation Queue..181
11.2.1. Double Buffered Host Application Utilizing Kernel Invocation Queue.............183

12. Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide Archives......................185

A. Document Revision History for the Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide...186

1. Introduction to Intel® FPGA SDK for OpenCL™ Pro Edition Best Practices Guide

The *Intel® FPGA SDK for OpenCL™ Pro Edition Best Practices Guide* provides guidance on leveraging the functionalities of the Intel FPGA Software Development Kit (SDK) for OpenCL[1] to optimize your OpenCL[2] applications for Intel FPGA products.

This document assumes that you are familiar with OpenCL concepts and application programming interfaces (APIs), as described in the *OpenCL Specification version 1.0* by the Khronos Group. It also assumes that you have experience in creating OpenCL applications.

To achieve the highest performance of your OpenCL application for FPGAs, familiarize yourself with details of the underlying hardware. In addition, understand the compiler optimizations that convert and map your OpenCL application to FPGAs.

For more information about the OpenCL Specification version 1.0, refer to the OpenCL Reference Pages on the Khronos Group website. For detailed information on the OpenCL APIs and programming language, refer to the *OpenCL Specification version 1.0*.

Tip: If you are looking for guidance on leveraging the functionalities of Data Parallel C++ (DPC++) to optimize your FPGA designs, then refer to the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

Related Information

- OpenCL Reference Pages
- OpenCL Specification version 1.0

1.1. FPGA Overview

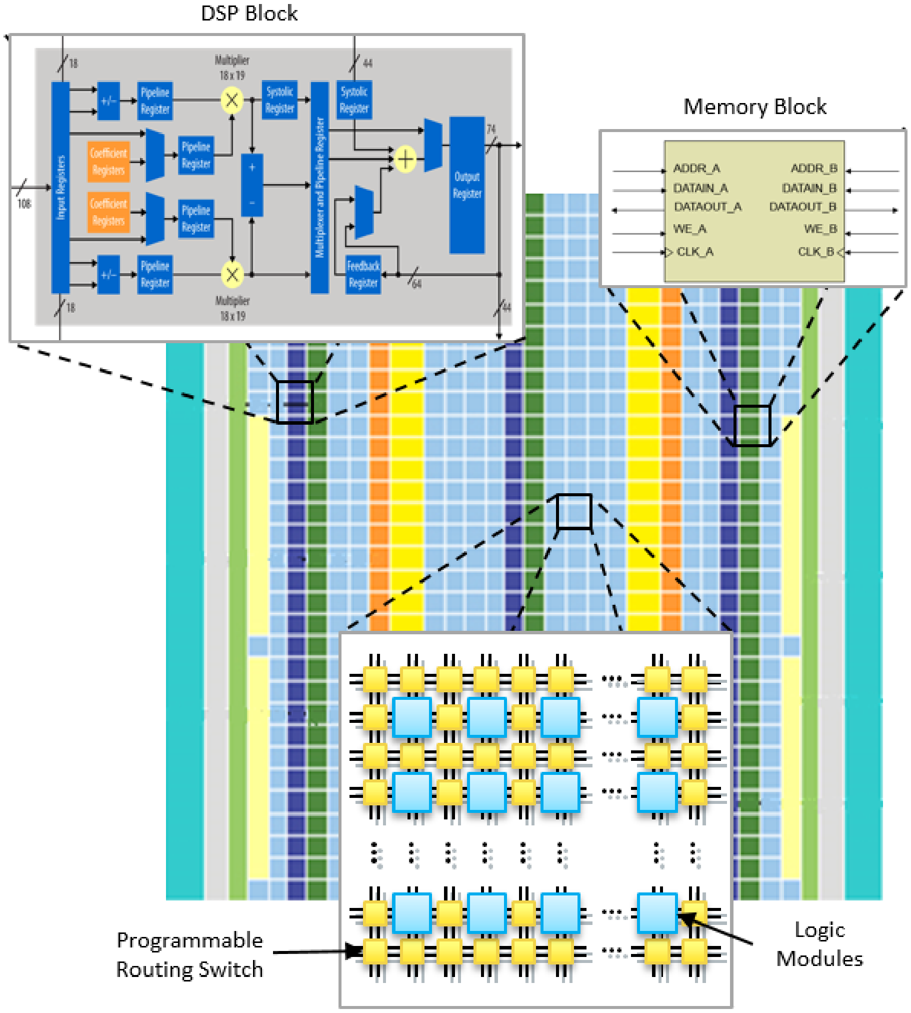
Field-programmable gate arrays (FPGAs) are integrated circuits that you can configure repeatedly to perform an infinite number of functions.

An FPGA consists of several small computational units. Custom datapaths can be built directly into the fabric by programming the compute units and connecting them as shown in the following figure. Data flow is programmed directly into the architecture.

[1] The Intel FPGA SDK for OpenCL is based on a published Khronos Specification, and has passed the Khronos Conformance Testing Process. Current conformance status can be found at www.khronos.org/conformance.

[2] OpenCL and the OpenCL logo are trademarks of Apple Inc. and used by permission of the Khronos Group™.

Figure 1. FPGA Architecture



With FPGAs, low-level operations like bit masking, shifting, and addition are all configurable. Also, you can assemble these operations in any order. To implement computation pipelines, FPGAs integrate combinations of lookup tables (LUTs), registers, on-chip memories, and arithmetic hardware (for example, digital signal processor (DSP) blocks) through a network of reconfigurable connections. As a result, FPGAs achieve a high level of programmability. LUTs are responsible for implementing various logic functions. For example, reprogramming a LUT can change an operation from a bit-wise AND logic function to a bit-wise XOR logic function.

The key benefit of using FPGAs for algorithm acceleration is that they support wide, heterogeneous and unique pipeline implementations. This characteristic is in contrast to many different types of processing units such as symmetric multiprocessors, DSPs, and graphics processing units (GPUs). In these types of devices, parallelism is achieved by replicating the same generic computation hardware multiple times. In FPGAs, however, you can achieve parallelism by duplicating only the logic that your algorithm exercises.

A processor implements an instruction set that limits the amount of work it can perform each clock cycle. For example, most processors do not have a dedicated instruction that can execute the following C code:

```
E = (((A + B) ^ C) & D) >> 2;
```

Without a dedicated instruction for this C code example, a CPU, DSP, or GPU must execute multiple instructions to perform the operation. In contrast, you may think of an FPGA as a hardware platform that can implement any instruction set that your software algorithm requires. You can configure an FPGA to perform a sequence of operations that implements the code example above in a single clock cycle. An FPGA implementation connects specialized addition hardware with a LUT that performs the bit-wise XOR and AND operations. The device then uses its programmable connections to perform a right shift by two bits without consuming any hardware resources. The result of this operation then becomes a part of subsequent operations to form complex pipelines.

1.2. Pipelines

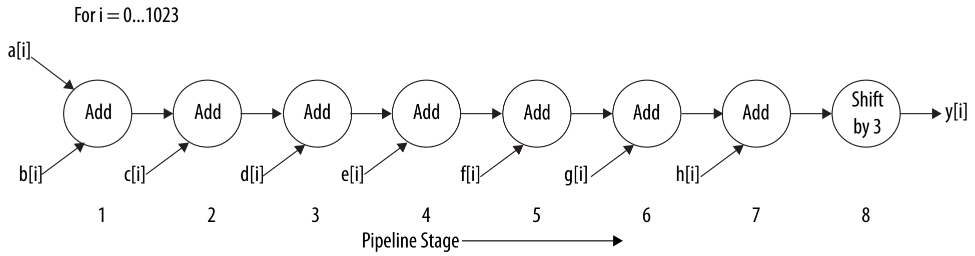
In a pipelined architecture, input data passes through a sequence of stages. Each stage performs an operation that contributes to the final result, such as memory operation or calculation.

The designs of microprocessors, digital signal processors (DSPs), hardware accelerators, and other high performance implementations of digital hardware often contain pipeline architectures.

For example, the diagram below represents the following example code fragment as a multistage pipeline:

```
for (i = 0; i < 1024; i++)
{
   y[i] = (a[i] + b[i] + c[i] + d[i] + e[i] + f[i] + g[i] + h[i]) >> 3;
}
```

Figure 2. Example Multistage Pipeline Diagram



With a pipelined architecture, each arithmetic operation passes into the pipeline one at a time. Therefore, as shown in the diagram above, a saturated pipeline consists of eight stages that calculate the arithmetic operations simultaneously and in parallel. In addition, because of the large number of loop iterations, the pipeline stages continue to perform these arithmetic instructions concurrently for each subsequent loop iteration.

Intel FPGA SDK for OpenCL Pipeline Approach

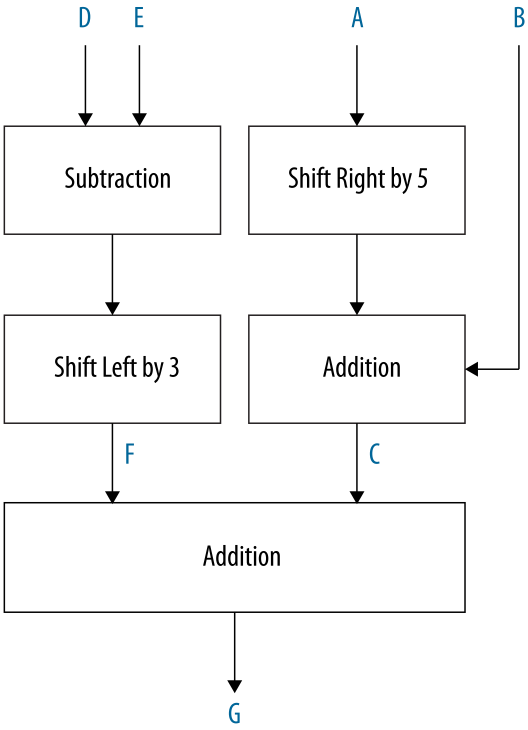
A new pipeline is constructed based on your design. As a result, it can accommodate the highly configurable nature of FPGAs.

Consider the following OpenCL code fragment:

```
C = (A >> 5) + B;
F = (D – E) << 3;
G = C + F;
```

You can configure an FPGA to instantiate a complex pipeline structure that executes the entire code simultaneously. In this case, the SDK implements the code as two independent pipelined entities that feed into a pipelined adder, as shown in the figure below.

Figure 3. Example of the SDK's Pipeline Approach



The Intel FPGA SDK for OpenCL Offline Compiler provides a custom pipeline structure that speeds up computation by allowing operations within a large number of work-items to occur concurrently. The offline compiler can create a custom pipeline that calculates the values for variables *C*, *F* and *G* every clock cycle, as shown below. After a ramp-up phase, the pipeline sustains a throughput of one work-item per cycle.

Figure 4. An FPGA Pipeline with Three Operations Per Clock Cycle

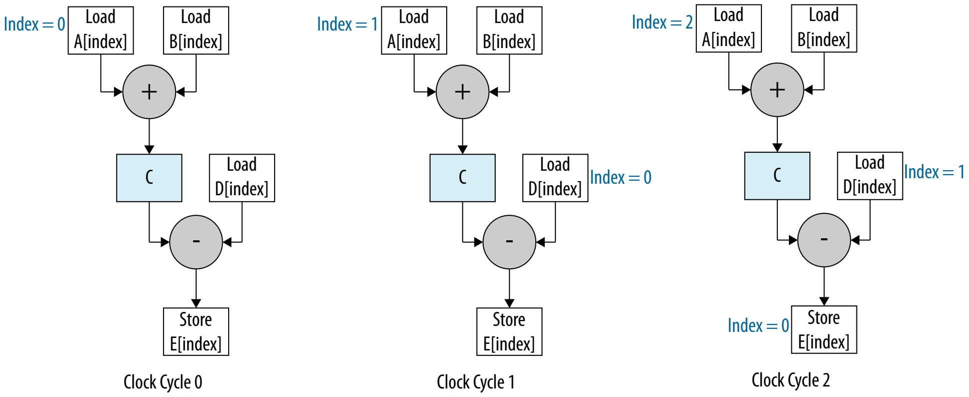
C	$(A_0 >> 5) + B_0$	$(A_1 >> 5) + B_1$	$(A_2 >> 5) + B_2$	$(A_3 >> 5) + B_3$	$(A_4 >> 5) + B_4$	$(A_5 >> 5) + B_5$
F	$(D_0 - E_0) << 3$	$(D_1 - E_1) << 3$	$(D_2 - E_2) << 3$	$(D_3 - E_3) << 3$	$(D_4 - E_4) << 3$	$(D_5 - E_5) << 3$
G		$C_0 + F_0$	$C_1 + F_1$	$C_2 + F_2$	$C_3 + F_3$	$C_4 + F_4$
	0	1	2	3	4	5

Time in Clock Cycles

A traditional processor has a limited set of shared registers. Eventually, a processor must write the stored data out to memory to allow more data to occupy the registers. The offline compiler keeps data "live" by generating enough registers to store the data for all the active work-items within the pipeline. The following code example and figure illustrate a live variable *C* in the OpenCL pipeline:

```
size_t index = get_global_id(0);

C = A[index] + B[index];
E[index] = C – D[index];
```

Figure 5. An FPGA Pipeline with a Live Variable C



1.3. Single Work-Item Kernel versus NDRange Kernel

Intel recommends that you structure your OpenCL kernel as a single work-item, if possible. However, if your kernel program benefits from explicitly describing multiple concurrent threads, you can structure your application as an NDRange kernel because the kernel can execute multiple work-items concurrently.

When a kernel describes a single work item, the Intel FPGA SDK for OpenCL host can execute the kernel as a single work-item, which is equivalent to launching a kernel with an NDRange size of (1, 1, 1). The compiler tries to accelerate the single work item for best performance.

The OpenCL Specification version 1.0 describes this mode of operation as *task parallel programming*. A *task* refers to a kernel executed with one work-group that contains one work-item.

Generally, the host launches multiple work-items in parallel. However, this data parallel programming model is not suitable for situations where fine-grained data must be shared among parallel work-items. In these cases, you can maximize throughput by expressing your kernel as a single work-item. Unlike NDRange kernels, single work-item kernels follow a natural sequential model similar to C programming. Particularly, you do not have to partition the data across work-items.

To ensure high-throughput single work-item-based kernel execution on the FPGA, the Intel FPGA SDK for OpenCL Offline Compiler must process multiple pipeline stages in parallel at any given time. This parallelism is realized by pipelining the iterations of loops.

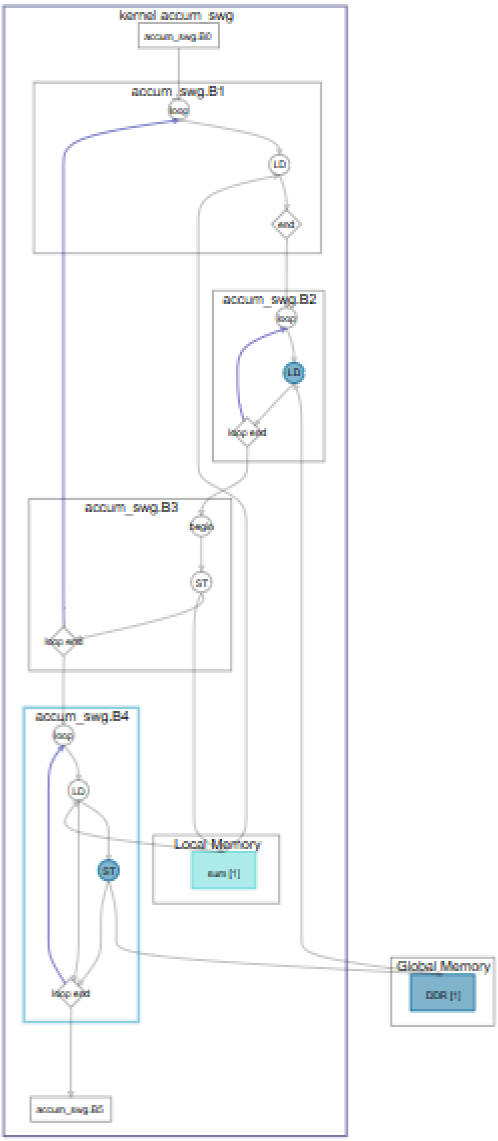
Consider the following simple example code that shows accumulating with a single-work item:

```
1 kernel void accum_swg (global int* a,
                         global int* c,
                         int size,
                         int k_size) {
2     int sum[1024];
3     for (int k = 0; k < k_size; ++k) {
4        for (int i = 0; i < size; ++i) {
5            int j = k * size + i;
6            sum[k] += __prefetching_load(&a[j]);
7        }
8     }
9     for (int k = 0; k < k_size; ++k) {
10       c[k] = sum[k];
11    }
12 }
```

During each loop iteration, data values from the global memory a is accumulated to `sum[k]`. In this example, the inner loop on line 4 has an initiation interval value of 1 with a latency of 11. The outer loop also has an initiation interval value greater than or equal to 1 with a latency of 8.

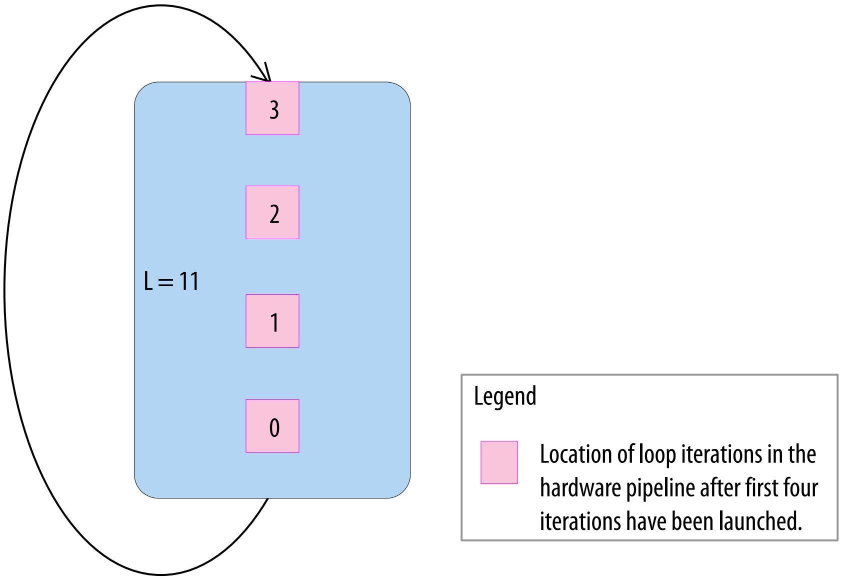
Note: The launch frequency of a new loop iteration is called the initiation interval (II). II refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. An optimally unrolled loop has an II value of 1 because one loop iteration is processed every clock cycle.

Figure 7. System View of Single-Work Item Kernel



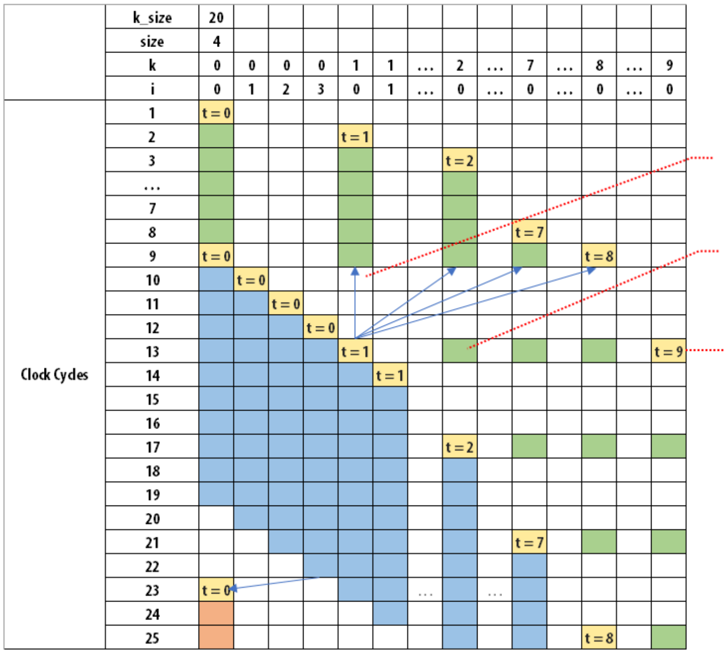
The following figure illustrates how each iteration of i enters into the block:

Figure 8. Inner Loop `accum_swg.B2` Execution


3
2
L = 11
1
0
Legend
Location of loop iterations in the hardware pipeline after first four iterations have been launched.

When you observe the outer loop, having an II value of 1 also means that each iteration of the thread can enter at every clock cycle. In the example, `k_size` of 20 and `size` of 4 is considered. This is true for the first eight clock cycles as outer loop iterations 0 to 7 can enter without any downstream stalling it. Once thread 0 enters into the inner loop, it takes four iterations to finish. Threads 1 to 8 cannot enter into the inner loop and they are stalled for four cycles by thread 0. Thread 1 enters into the inner loop after thread 0's iterations are completed. As a result, thread 9 enters into the outer loop on clock cycle 13. Threads 9 to 20 enters into the loop at every four clock cycles, which is the value of `size`. Through this example, you can observe that the dynamic initiation interval of the outer loop is greater than the statically predicted initiation interval of 1 and it is a function of the trip count of the inner loop.

Figure 9. Single Work-Item Execution



Stalled until the inner loop has finished its computations.

Thread 2 can continue to pipeline through outer loop and stall in the next cycle until thread 1 is done.

Thread 9 enters into outer loop on cycle 13.

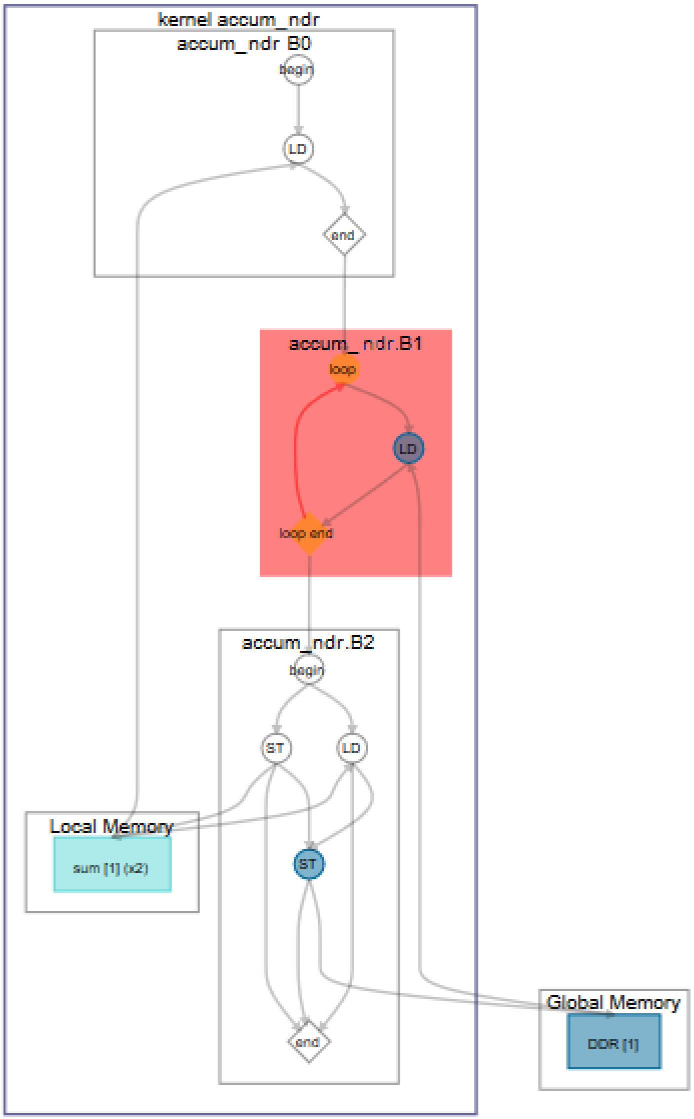
Important:

- Using any of the following functions causes your kernel to be interpreted as an NDRange:
  - `get_local_id()`
  - `get_global_id()`
  - `get_group_id()`
  - `get_local_linear_id()`
  - `barrier`
- If the `reqd_work_group_size` attribute is specified to be anything other than (1, 1, 1), your kernel is interpreted as an NDRange. Otherwise, your kernel is interpreted as a single-work-item kernel.

Consider the same accumulate example written in NDRange:

```
kernel void accum_ndr (global int* a,
                       global int* c,
                       int size) {
   int k = get_global_id(0);

   int sum[1024];
   for (int i = 0; i < size; ++i) {
     int j = k * size + i;
     sum[k] += a[j];
   }
   c[k] = sum[k];
}
```

Figure 11. System View of the NDRange Kernel



Limitations

The OpenCL task parallel programming model does not support the notion of a barrier in single-work-item execution. Replace barriers (`barrier`) with memory fences (`mem_fence`) in your kernel.

Related Information

- Multiple Work-Item Ordering for Channels
- Loops in a Single Work-Item Kernel on page 66
- Nested Loops on page 70

2. Reviewing Your Kernel's report.html File

The <*your_kernel_filename*>`/reports/report.html` file provides you with kernel analytical data such as area and memory usages, as well as loop structure and kernel pipeline information.

Tip: If you are looking for oneAPI DPC++-specific instructions, then refer to Review the `report.html` File chapter in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

2.1. High-Level Design Report Layout

The summary and analysis reports in the High-Level Design Reports (`report.html` have four main sections:

- Reports menu
- Analysis pane
- Source code pane
- Details pane

Reports Menu

The Reports menu provides three hierarchical categories of views. You can select a report to view an analysis of different parts of your kernel design. All reports are interlinked.

- **Summary** gives you a quick overview of the results of compiling your design including a summary of all kernels in your design along with an estimate of resources used by the kernels. To navigate to a particular section of the summary report, use the left-hand list pane, which you can show or hide by selecting the **List** button on the top-right corner.
- **Bottlenecks** gives a quick overview of the throughput bottlenecks in the design. You can show or hide the Bottlenecks viewers by selecting the **List** button on the top-right corner.
- **Throughput Analysis** helps you in optimizing your design based on f_{MAX} and bottleneck summary, result from Intel FPGA dynamic profiler for OpenCL, loop analysis, and latency estimator.
- **Area Analysis** helps you in locating the area inefficiency. It provides information about resource utilization of the system, incremental compile, Intel Quartus resource summary, and wasted RAM bits.
- **System Viewers** provide a graphical representation of the generated hardware to supplement the throughput and area analysis. Each viewer shows different information about kernels, channels, global memory, blocks, clusters, and more.

Analysis Pane

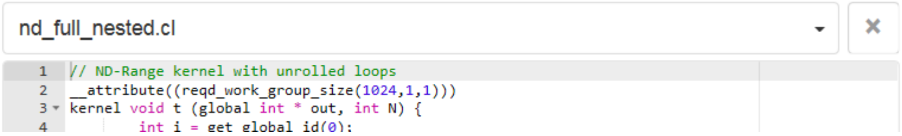
The analysis pane displays detailed information of the report you selected from the reports menu.

Source Code Pane

The source code pane displays the code for all the source files in your kernel.

To select between different source files in your kernel, click the pull-down menu at the top of the source code pane. To collapse the source code pane, do one of the following actions:

- Click the **X** icon beside the source code pane pull-down menu.



```
// ND-Range kernel with unrolled loops
__attribute((reqd_work_group_size(1024,1,1)))
kernel void t (global int * out, int N) {
        int i = get_global_id(0);
```

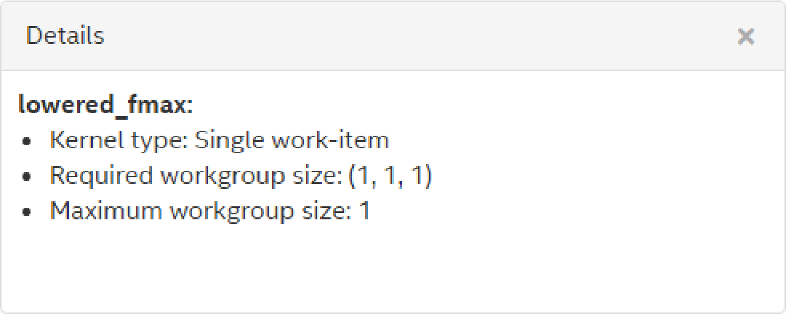
- Select the **Source Code** button on the top-right corner of the report to show or hide the source code pane.

The source code is displayed when you have not specified the `-g0` compiler command option when you compiled your code.

Details Pane

For each line that appears in a loop analysis or area report, the Details pane shows additional information, if available, that elaborates on the comment in the Details column report. To collapse the Details pane, do one of the following actions:

- Click the **X** icon on the top-right corner side of the Details pane.



lowered_fmax:
- Kernel type: Single work-item
- Required workgroup size: (1, 1, 1)
- Maximum workgroup size: 1

- Select the **Details** button on the top-right corner of the report to show or hide the Details pane.

2.2. Reviewing the Summary Report

The Summary Report gives you a quick overview of the results of compiling your design including a summary of each kernel in your design and a summary of the estimated resources that each kernel in your design uses.

The Summary report is divided into the following sections based on the order of compilation:

- Compile Info
- Kernels Summary
- Clock Frequency Summary
- System Resource Utilization Summary
- Quartus Fitter Resource Utilization Summary
- Compile Estimated Kernel Resource Utilization Summary
- Warnings Summary

Compile Info

The Compile Info section shows general information about the compile including the following items:

- Name of the project
- Target FPGA family, device, and board
- Intel Quartus® Prime version
- Intel FPGA SDK for OpenCL Offline Compiler version
- The date and time at which the reports are generated

Kernels Summary

The Kernels Summary lists each kernel in your design, and some information about each of the kernels, including the following items:

- Line number in the source code
- Whether the kernel is an NDRange or a single work-item kernel
- Whether the autorun attribute is used
- The required workgroup size for the kernel
- The number of compute units

When you select a kernel in the list, the Details pane shows additional information about the kernel:

Clock Frequency Summary

After you compile your design with Intel Quartus Prime software, the Clock Frequency Summary shows the following:

- Quartus Fitter Clock Frequency
- Compiler Target Frequency (MHz)
- Compiler estimated frequency (MHz)

The Quartus Fitter clock frequency is the maximum clock frequency that can be achieved for the design. When the compiler estimates a lower frequency than the targeted frequency, the frequency value is highlighted in red.

Both the Kernels Summary and Clock Frequency Summary displays the target clock frequency applied at the source on the kernel. When the values of the source is different than the compilation flag you applied, the Clock Frequency Summary Compiler Target Frequency shows “Various” instead of reporting a number.

System Resource Utilization Summary

This displays a summary of device, static partition, kernel system total usage and total estimate of the resources used including ALMs, RAMs and DSPs. The device utilization and static partition constants are grayed out to indicate that you cannot change them in your design since these numbers map to the device datasheet and BSP, respectively.

Quartus Fitter Resource Utilization Summary

After you compile your design with Intel Quartus Prime software, this section is populated with the compilation results. The Quartus Fit Resource Utilization Summary section shows the total resource utilization both for the entire design, and for the device.

Compile Estimated Kernel Resource Utilization Summary

The Estimated Resource Usage section shows a summary of the estimated resources used by each kernel in your design, as well as the estimated resources used for all channels, estimated resources for the global interconnect, constant cache, and board interface.

Warnings Summary

The Warnings Summary section shows some of the compiler warnings generated during the compilation.

2.3. Viewing Throughput Bottlenecks in the Design

The Bottlenecks viewer, when used with the Loop Analysis and Schedule Viewerreports, provides information about the throughput bottlenecks in your design. This viewer lists all loops that result in a bottleneck for the current selected system, kernel, or task. You can select these loops to view more details about the bottleneck in the Details pane. For more information about the concept of bottlenecks, refer to Loop Bottlenecks on page 81.

The Bottlenecks viewer identifies the following categories of bottlenecks:

- F_{MAX} reduced or II increased, or both
- Compiler applied bottlenecks (private copies set to 1 on local memory)
- Bottlenecks due to the pragmas or attributes you apply on a loop
- Concurrency limiter bottlenecks

Here is an example of data dependency:

```
kernel void lowered_fmax (global int *dst, int N) {
    int res = N;
    #pragma unroll  9
    for (int i = 0; i < N; i++) {
        res += 1;
        res ^= i;
    }
    dst[0] = res;
}
```

The Bottlenecks viewer displays the following message:

```
9X Partially unrolled lowered_fmax.B1:
Compiler failed to schedule this loop with smaller II due to data dependency on
variable(s):
  res (Unknown location)
Most critical loop feedback path during scheduling:
  Number of nodes in critical path exceeded what the compiler has captured. Only
the top 19 failing nodes are listed.
    1.00 clock cycle 32-bit Select Operation (fmax_ii.cl: 2, fmax_ii.cl: 6)
```

In the Bottlenecks viewer, you can then select the loop to display more information in the Details pane, which you can use to investigate why and what caused this bottleneck. For additional information about the bottlenecks, refer to the System Viewer and the Schedule Viewer. The System Viewer provides information about the isolated failing path and bottleneck type. The Schedule Viewer displays the bottleneck path for the variable.

2.4. Using Views

From the Report's **Views** drop-down menu, you can analyze your OpenCL system using the following reports:

- **System Viewer**: Shows a hierarchical graphical report consisting of system, global memory, block, and cluster views of your OpenCL system. It allows you to review information such as the sizes and types of loads and stores between kernels and different memories, channels connected between kernels, loops, stalls, and latencies, and view all variables inside a cluster that have loop-carried dependency.
- **Kernel Memory Viewer**: Shows how the offline compiler interprets the data connections across the memory system of your kernel.
- **Schedule Viewer**: Displays a static view of the scheduled cycle and latency of a clustered group of instructions in your design.

2.4.1. Features of the System Viewer

The System Viewer is an interactive graphical report of your OpenCL system that allows you to review information such as the sizes and types of loads and stores, stalls, latencies, load and store information between kernels and different memories, channels connected between kernels, and loops. You can access this report by selecting **System Viewer** from the **Views** drop-down menu. This viewer mainly helps in viewing the connectivity between global memory and channels.

Tip: To understand the hierarchy within a kernel, refer to Kernels on page 51.

The System Viewer shows an abstracted netlist of your OpenCL system. Reviewing the graphical representation of your OpenCL design allows you to verify memory replication, identify any load and store instructions that are stallable, inspect the connectivity of clusters, and review stallable and stall-free instructions.

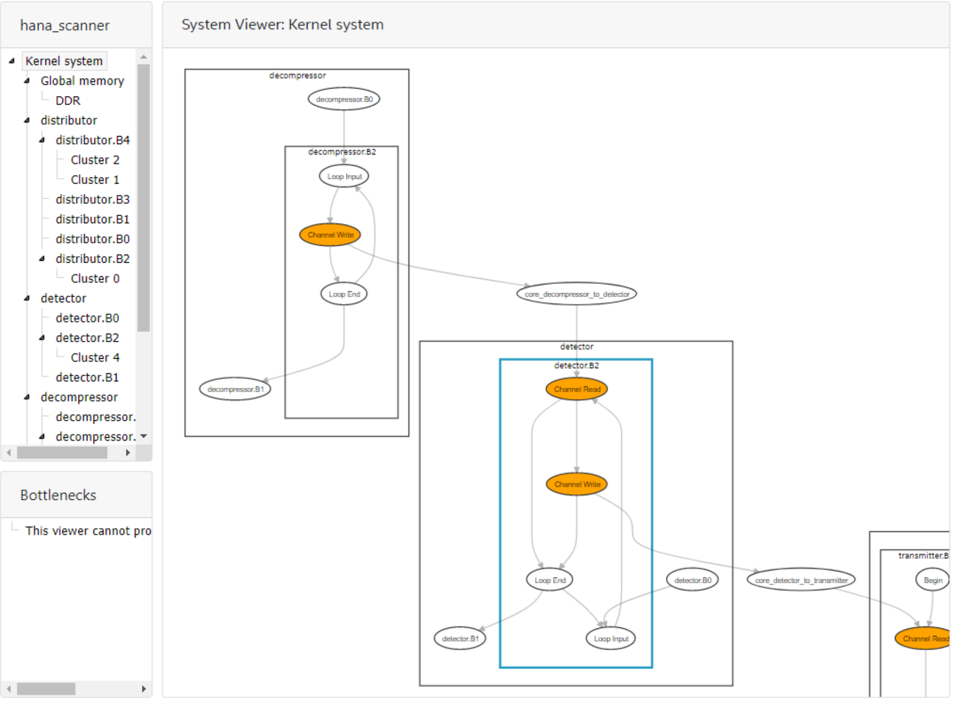
You can interact with the System Viewer in the following ways:

- Use the mouse wheel to zoom in and out within the System Viewer.
- Hover over any node to view information on that node in the Details pane.

2.4.1.1. Reviewing System Information

Use the system view of the System Viewer report to view various kernels in your OpenCL system. The system view illustrates connections between your kernels and connections from kernels to memories. In addition, the system view shows the connection of blocks within a kernel and highlights blocks with a high initiation interval (II).

Figure 12. Kernel System View of the System Viewer Report



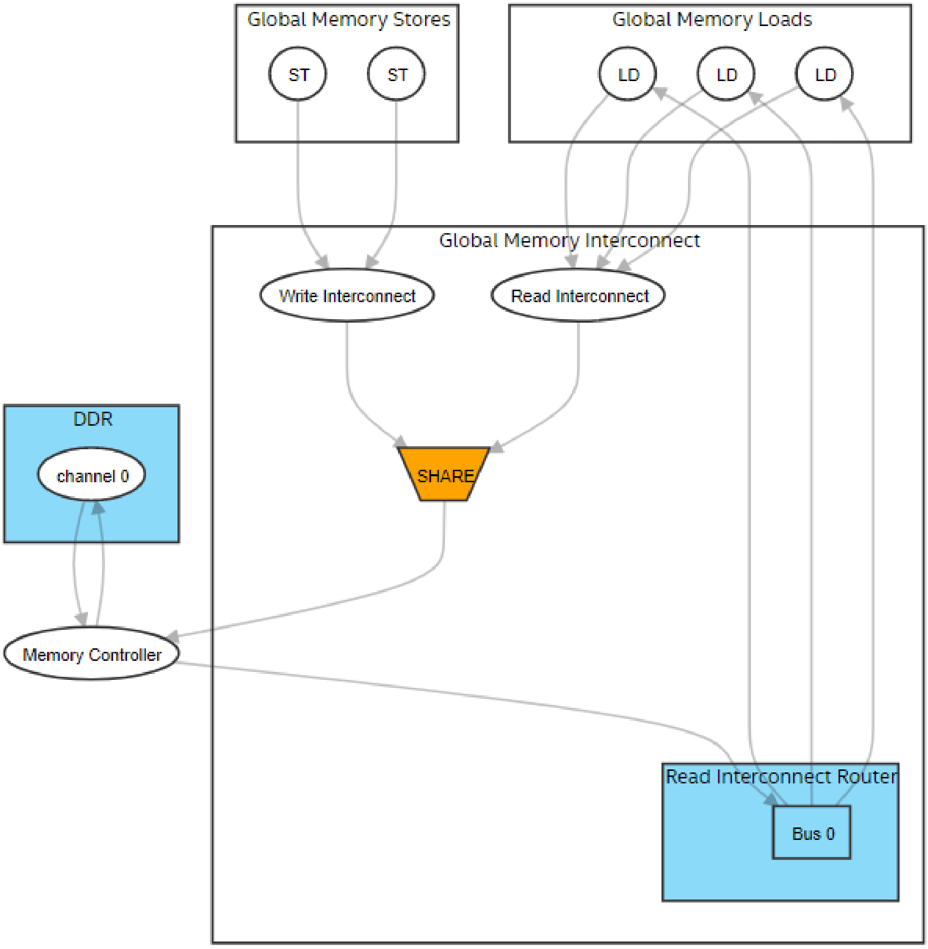
2.4.1.2. Reviewing Global Memory Information

The global memory view of the System Viewer provides a list of all global memories in the design. The global memory view shows the following:

- Connectivity within the system showing data flow direction between global memory and kernels
- Memory throughput bottlenecks
- Status of the offline compiler flags, such as `-num-reorder` and `-force-single-store-ring`
- Global load-store unit (LSU) types
- Type of write/read interconnects
- Number of write rings
- Number and connectivity of read-router buses

The following image is an example of the global memory view of the System Viewer:

Figure 13. Graphical Representation of the Global Memory in the System Viewer



In Graphical Representation of the Global Memory in the System Viewer:

- When you select stores or loads, you can view respective lines in the source code and details about the LSU type and LSU-level bandwidth.
- For the write interconnect block, you can view the interconnect style, number of writes to the global memory, status of the `-force-single-store-ring` compiler flag, and number of store rings.
- For the read interconnect block, you can view the interconnect style and number of reads from the global memory.
- For the read interconnect router block, you can view the status of the `-num-reorder` flag, total number of buses, and all connections between buses and load LSUs. Buses in this block provide read data from the memory to load LSUs.

- For the global memory (DDR in Graphical Representation of the Global Memory in the System Viewer), you can view the status of interleaving, interleaving size, number of channels, maximum bandwidth the BSP can deliver, and channel width.
- For the memory controller block, you can view the maximum bandwidth the BSP can deliver, sum of the load/store throughput, and read/write bandwidth. For additional information about how global memory bandwidth use is calculated, refer to Global Memory Bandwidth Use on page 147 in this guide. It describes the formulas used in calculating the bandwidth.
- LSUs using USM pointers show up twice in both host and device global memory views as they can access both memories.

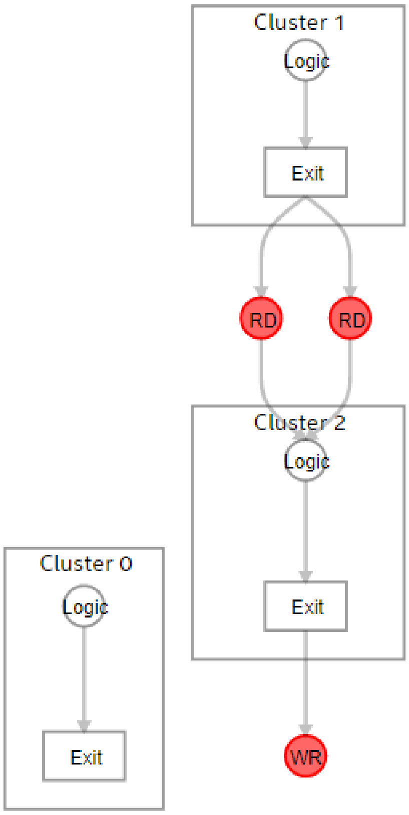
2.4.1.3. Reviewing Block Information

The block view of the System Viewer provides a more granular graph view of the kernel. This view shows the following:

- Fine grained details within kernels (including instructions and dependencies of the instructions) of the generated datapath of computations. The Intel FPGA SDK for OpenCL Offline Compiler encapsulates maximum instructions in clusters for better QoR. The System Viewer shows clusters, instructions outside clusters and their connections.
- Linking from the instruction back to source line by clicking the instruction node.
- Various information about the instructions, such as data width, node’s schedule information in start cycle and latency are provided, if applicable.

The following image is an example of the block view of the System Viewer:

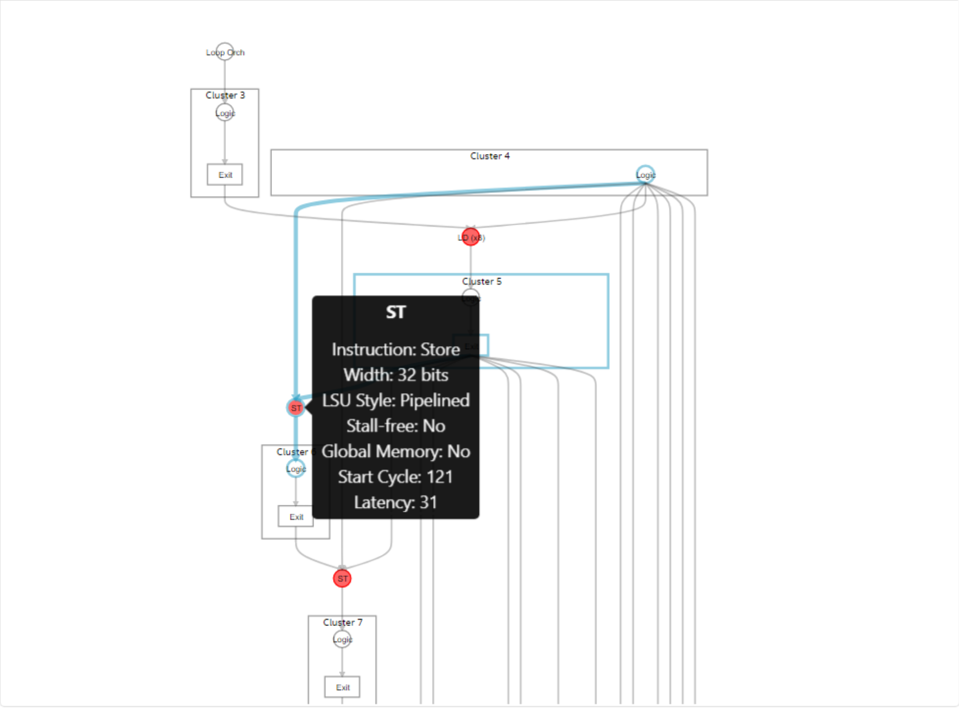
Figure 14. Granular Graphical View of a Block



If your design has loops, the Intel FPGA SDK for OpenCL Offline Compiler encapsulates the loop control logic into loop orchestration nodes and the initial condition of the loops into loop input node and their connection to the datapath.

Inside a block, there are often stallable channel RD/WR or memory LD/ST nodes connecting to computation nodes or clusters. You can click different nodes and view the Details pane (or hover over the nodes) to see detailed information about the instruction. For example, you can click the LD/ST nodes to view attributes such as instruction type, width, LSU style, stall-free, global memory, scheduled start cycle, and estimated latency. For stallable nodes, the latency value provided is an estimate. Perform a simulation or hardware run for more accurate latency values.

Figure 15. Instruction Node with Details in a Tooltip Box

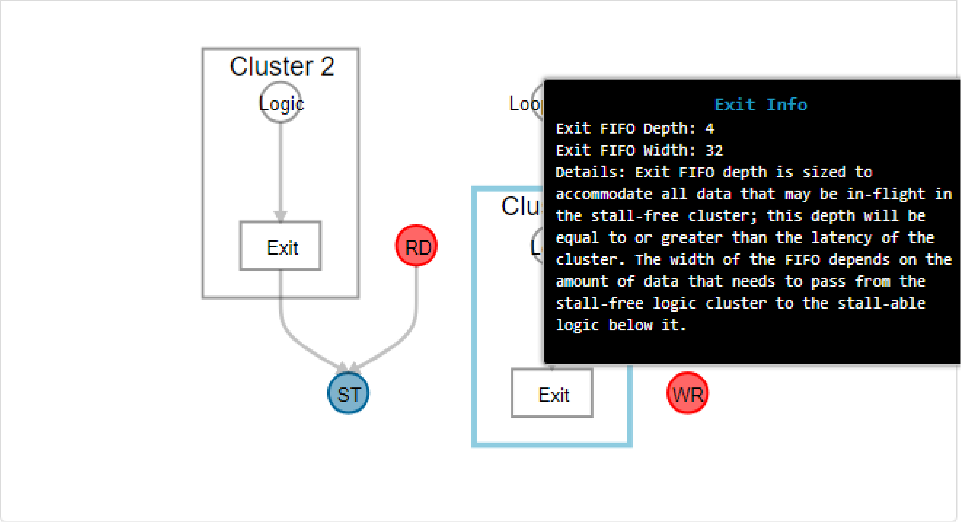


ST

Instruction: Store
Width: 32 bits
LSU Style: Pipelined
Stall-free: No
Global Memory: No
Start Cycle: 121
Latency: 31

A cluster has a FIFO in its exit node to store any pipelined data in-flight. You can click the cluster exit node to find the exit FIFO width and depth attribute. The cluster exit FIFO size is also available in the cluster view of the System Viewer. Refer to the following image for cluster exit FIFO details in a block view example.

Figure 16. Cluster Node With FIFO Details



```
Exit Info
Exit FIFO Depth: 4
Exit FIFO Width: 32
Details: Exit FIFO depth is sized to
accommodate all data that may be in-flight in
the stall-free cluster; this depth will be
equal to or greater than the latency of the
cluster. The width of the FIFO depends on the
amount of data that needs to pass from the
stall-free logic cluster to the stall-able
logic below it.
```

2.4.1.4. Reviewing Cluster Information

The cluster view of the System Viewer provides more granular graph views of the system or kernel. It helps in viewing clusters inside a block and all variables inside a cluster that have loop-carried dependency.

This view shows the following:

- Fine grained details within clusters (including instructions and dependencies of the instructions) of the generated datapath of computations.
- Linking from the instruction back to source line by clicking the instruction node.
- Various information about the instructions, such as data width, node's schedule information in start cycle and latency are provided, if applicable.

A cluster starts with an entry node and ends with an exit node. The cluster exit node has a FIFO of depth greater than or equal to the latency of the cluster to store any data in-flight. You can find the size of the cluster exit FIFO by clicking on the exit node. The cluster exit FIFO size information is also available in the block view of the System Viewer.

Figure 17. Granular Graphical View of a Cluster

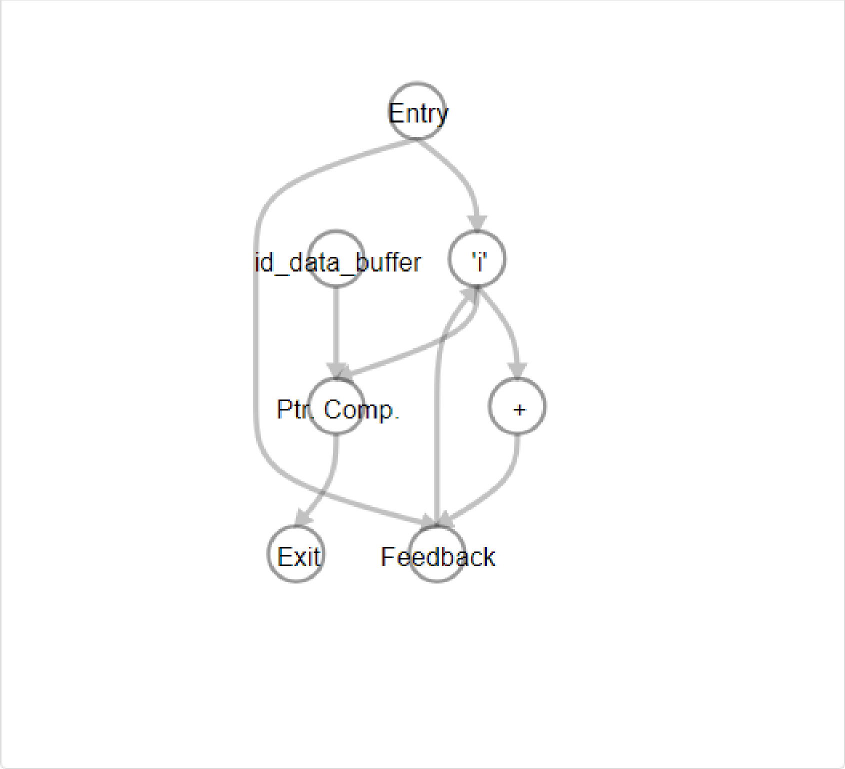
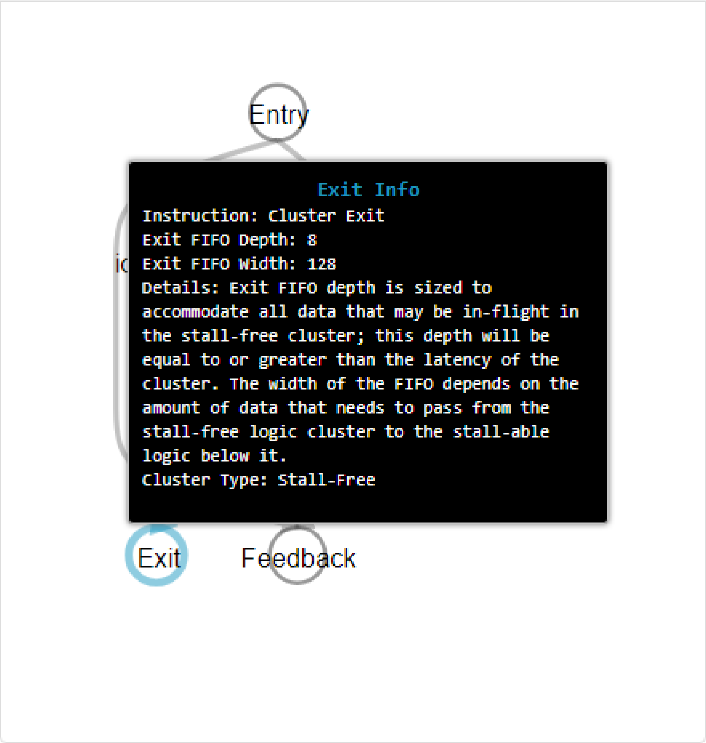


Figure 18. Cluster Exit Node Example



Exit Info
Instruction: Cluster Exit
Exit FIFO Depth: 8
Exit FIFO Width: 128
Details: Exit FIFO depth is sized to accommodate all data that may be in-flight in the stall-free cluster; this depth will be equal to or greater than the latency of the cluster. The width of the FIFO depends on the amount of data that needs to pass from the stall-free logic cluster to the stall-able logic below it.
Cluster Type: Stall-Free

Besides computation nodes, when your design contains loops, you can see loop orchestration nodes and variable nodes along with their Feedback nodes. The compiler generates the loop orchestration logic for loops in your design. This logic is represented by loop orchestration nodes in the cluster view of the System Viewer. A variable node corresponds to a variable that has loop-carried dependency in your design. A variable node goes through various computation logic and finally feeds to a Feedback node that connects back to the variable node. This back edge means that the variable is passed to the next iteration after the new value is evaluated. Scan for loop-carried variables that have a long latency to the Feedback nodes as they can be the II bottlenecks. You can cross-check by referring to the Loop Analysis report for more information about the II bottleneck. The Feedback node has a FIFO to store any data in-flight for the loop and is sized to `d*II` where d is the dependency distance and II is the initiation interval. You can find the size of the cluster exit FIFO by clicking on the feedback node and looking at the Details pane or the tooltip box. Refer to Cluster Exit Node Example for cluster exit FIFO details in the cluster view example.

Note: The dependency distance is the number of iterations between successive load/store that depends on each other.

2.4.2. Features of the Kernel Memory Viewer

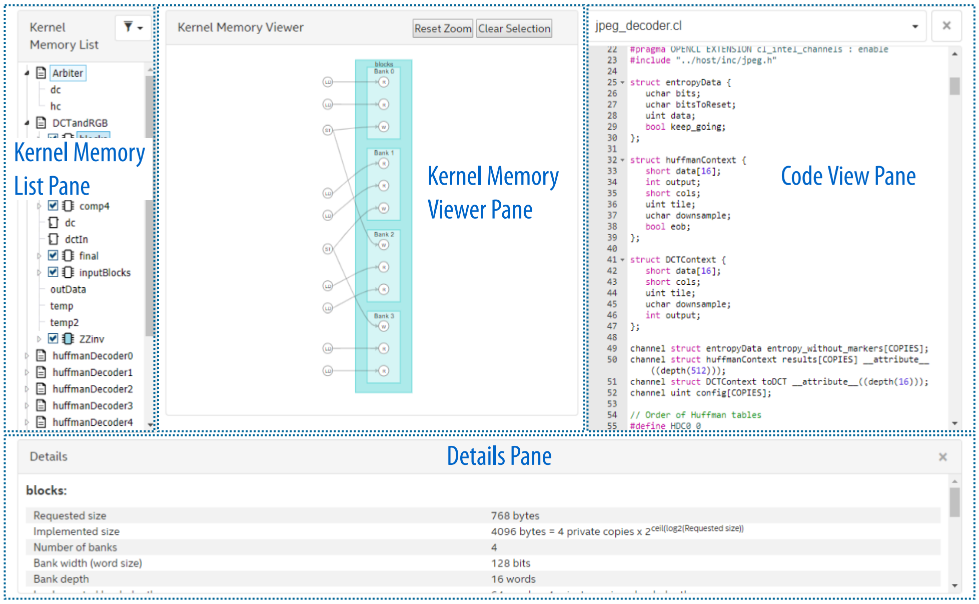
Data movement is a bottleneck in many algorithms. The Kernel Memory Viewer in the High Level Design Report (`report.html`) shows you how the Intel FPGA SDK for OpenCL Offline Compiler interprets the data connections and synthesizes memory for your kernel. Use the Kernel Memory Viewer to help you identify data movement bottlenecks in your kernel design.

Some patterns in memory accesses can cause undesired arbitration in the load-store units (LSUs), which can affect the throughput performance of your kernel. Use the Kernel Memory Viewer to identify unwanted arbitration in the LSUs.

Access the Kernel Memory Viewer by clicking **System Viewers ➤ Kernel Memory Viewer**.

The following image illustrates the layout of the Kernel Memory Viewer:

Figure 19. Kernel Memory Viewer Layout



```
#pragma OPENCL EXTENSION cl_intel_channels : enable
#include "../host/inc/jpeg.h"

struct entropyData {
   uchar bits;
   uchar bitsToReset;
   uint data;
   bool keep_going;
};

struct huffmanContext {
   short data[16];
   int output;
   short cols;
   uint tile;
   uchar downsample;
   bool eob;
};

struct DCTContext {
   short data[16];
   short cols;
   uint tile;
   uchar downsample;
   int output;
};

channel struct entropyData entropy_without_markers[COPIES];
channel struct huffmanContext results[COPIES] __attribute__
    ((depth(512)));
channel struct DCTContext toDCT __attribute__((depth(16)));
channel uint config[COPIES];

// Order of Huffman tables
#define HDC0 0
```

blocks:

Requested size	768 bytes
Implemented size	4096 bytes = 4 private copies x $2^{\text{ceil(log2(Requested size))}}$
Number of banks	4
Bank width (word size)	128 bits
Bank depth	16 words

The Kernel Memory Viewer has the following panes:

Kernel Memory List Lists all memories present in your design. When you select a memory name, you can view its graphical representation in the Kernel Memory Viewer pane.

Kernel Memory Viewer Shows a graphical representation of the memory system or memory bank selected in the Kernel Memory List pane.

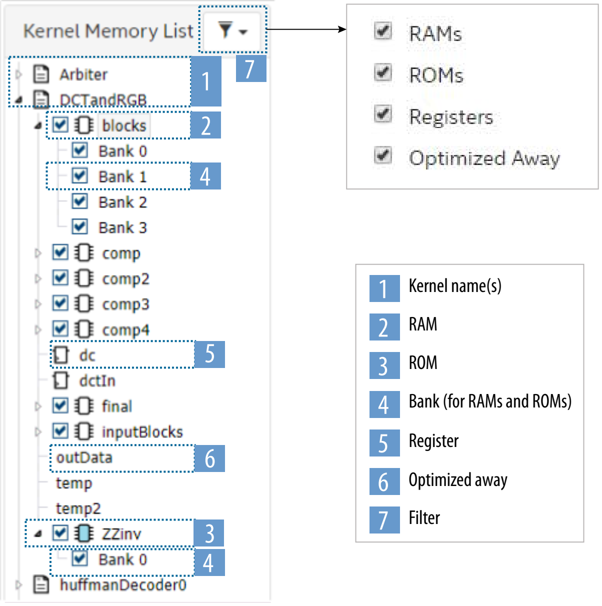
Code View Shows the source code file for which the reports are generated.

Details Shows the details of the memory system or memory bank selected in the Kernel Memory List pane.

Kernel Memory List

The Kernel Memory List pane displays a hierarchy of kernels with memories synthesized (RAMs, ROMs, and registers) and optimized away in that kernel.

Figure 20. Features and Details of the Kernel Memory List Pane



The following table describes each numbered feature highlighted in the above image:

Table 1. Kernel Memory List Pane Icons and Labels

	Icon or Label	Name	Description
1	📄	Kernel name	The list of memories in your kernel can be expanded or collapsed. Memories that do not belong to any kernel are shown under **(Other)**.
2	▯	RAM	A RAM is a memory that has at least one write to it. The name of the RAM memory is same as its name in your design. When you select a memory name, you can view a logical representation of the RAM in the Kernel Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select banks that you want the Kernel Memory Viewer pane to display: 1. Expand the memory name. 2. Clear the memory name check box to collapse all memory banks in the view. 3. Select the memory name check box to show all memory banks in the view.
3	🟦	ROM	A ROM is a memory that is read-only. The name of the ROM memory is same as its name in your design. When you select a memory name, you can view a logical representation of the ROM in the Kernel Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select banks that you want the Kernel Memory Viewer pane to display: 1. Expand the memory name. 2. Clear the memory name check box to collapse all memory banks in the view. 3. Select the memory name check box to show all memory banks in the view.

continued...

	Icon or Label	Name	Description
4	**Bank #num**	Bank	A memory bank is always associated with a RAM or a ROM. Each bank is named as **Bank #num**, where #num is the ID of the memory bank starting from 0. • Click the bank name to display the bank view in the Kernel Memory Viewer pane, which displays a graphical representation of the bank with all of its replicates and private copies. This view can help you focus on specific memory banks of a complex memory design. • Clear the memory bank name check box to collapse the bank in the logical representation of the memory. • Select the memory bank name check box to display the bank in the logical representation of the memory.
5	▯	Register	A register is a kernel variable that is carried through the pipeline in registers (rather than being stored in a RAM or ROM). The name of the register is same as its name in your design. You can implement a register variable either exclusively in FFs or in a combination of FFs and RAM-based FIFOs.
6	**text label**	Optimized Away	A kernel variable can be optimized away because it is unused in your design, or compiler optimizations have transformed all uses of the variable such that it is unnecessary. The name of the optimized away variable is same as its name in your design.
7	▼	Filter	Use the Kernel Memory List filter to selectively view the list of RAMs, ROMs, registers, and optimized away variables in your design. When you clear the check box associated with an item in the filter, you hide all occurrences of that kind of item in the Kernel Memory List. Filter your Kernel Memory List to help you focus on a specific type of memory in your design.

Kernel Memory Viewer

In the Kernel Memory Viewer pane, you can view connections between loads and stores to specific logical ports on the banks in a memory system. You can also view the number of replicates and private copies created per bank for your memory system. You can see the following types of nodes in the Kernel Memory Viewer pane, depending on the kernel memory system and what you have selected in the Kernel Memory List pane:

Table 2. Node Types Observed in the Kernel Memory Viewer Pane

Node Type	Description
Memory node	The memory system for a given variable in your design.
Bank node	A bank in the memory system. A memory system contains at least one bank. A memory bank can connect to one or more port nodes. Only banks selected in the Kernel Memory List pane are shown.
Replication node	A replication node shows memory bank replicates that are created to efficiently support multiple accesses to a local memory. A bank contains at least one replicate. You can view replicate nodes only when you view a memory bank by clicking its name in the Kernel Memory List pane.
Private-copy node	A private-copy node shows private copies within a replicate that are created to allow simultaneous execution of multiple loop iterations. A replicate contains at least one private copy. You can view private-copy nodes only when you view a memory bank by clicking its name in the Kernel Memory List pane.

continued...

Node Type	Description
Port node	Each read or write access to a local memory is mapped to a port. The logical port for a bank. There are three types of port: • **R**: A read-only port • **W**: A write-only port • **RW**: A read and write port
LSU node	A store (ST) or load (LD) node connected to the memory through port nodes.
Arbitration node	An arbitration (ARB) node shows that LSUs compete for access to a shared port node, which can lead to stalls.
Port-sharing node	A port-sharing node (SHARE) shows that LSUs have mutually exclusive access to a shared port node, so the load-store units are free from stalls.

Within the graphical representation of a memory in the Kernel Memory View pane, you can do the following:

- Hover over any node to view the attributes of that node.
- Hover over an LSU node to highlight the path from the LSU node to all ports that the LSU connects to.
- Hover over a port node to highlight the path from the port node to all LSUs that read or write to the port node.
- Click a node to select it and display the node attributes in the Details pane.

The following images illustrate examples of what you see in the Kernel Memory Viewer:

Figure 21. Logical Representation of a Memory in Kernel Memory Viewer Pane

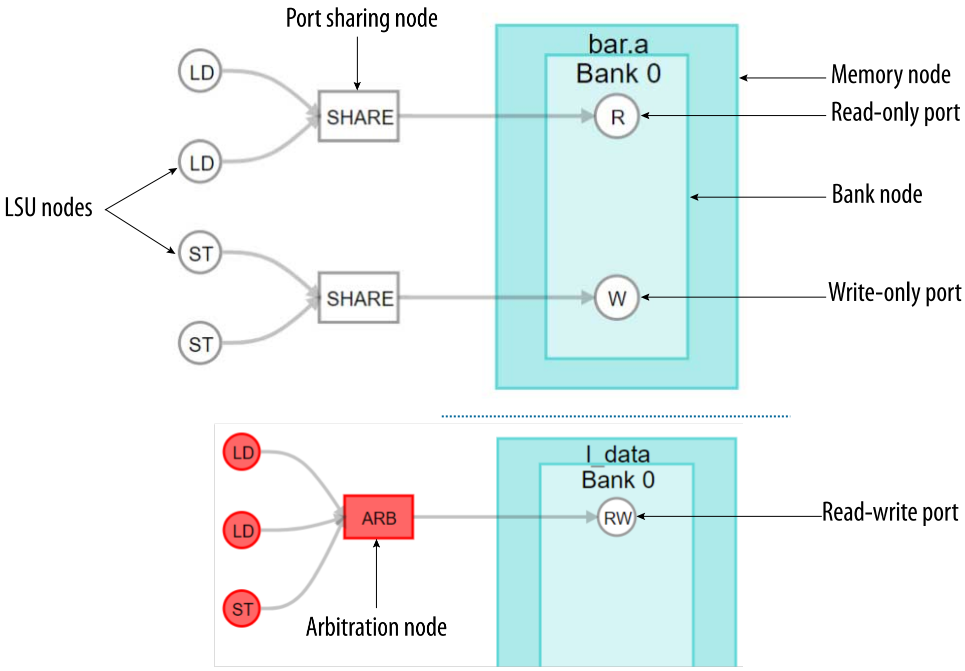
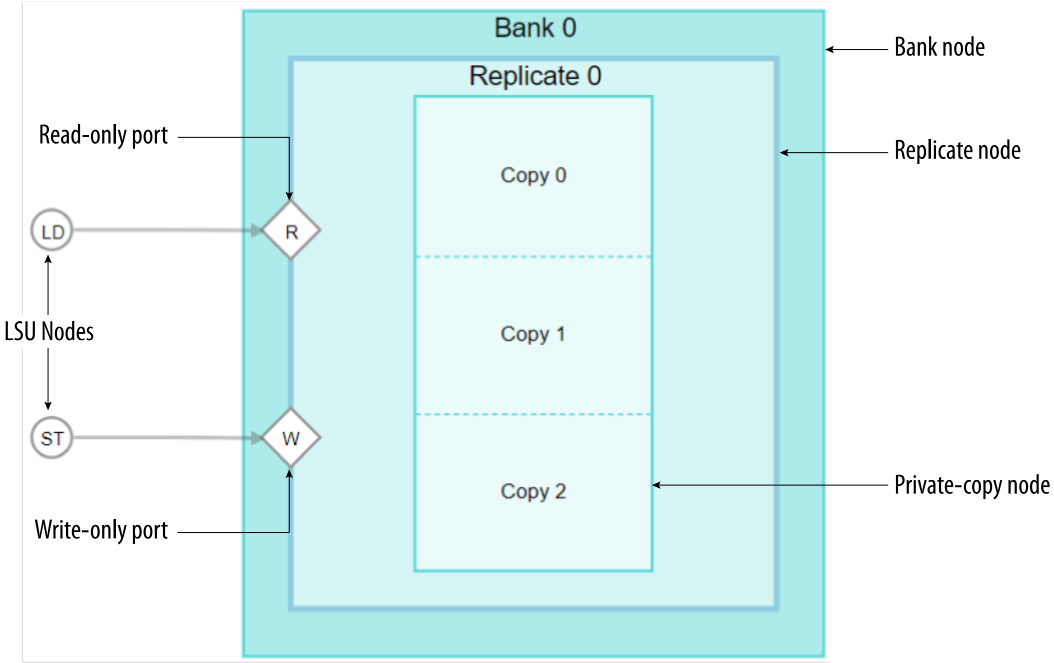


Figure 22. Bank View of a Memory Bank in Kernel Memory Viewer Pane



Code View

The code view pane displays your source code. When you select a memory or a bank in the Kernel Memory Viewer pane, the code view pane highlights the line of your code where you declared the memory.

Details

The Details pane shows the attributes of the node selected in the Kernel Memory Viewer pane. For example, when you select a memory in a kernel, the Details pane displays information such as:

- width and depths of the memory banks
- memory layout
- address-bit mapping
- memory attributes that you specified in your source code

The content of the Details pane persists until you select a different node in the Kernel Memory Viewer pane.

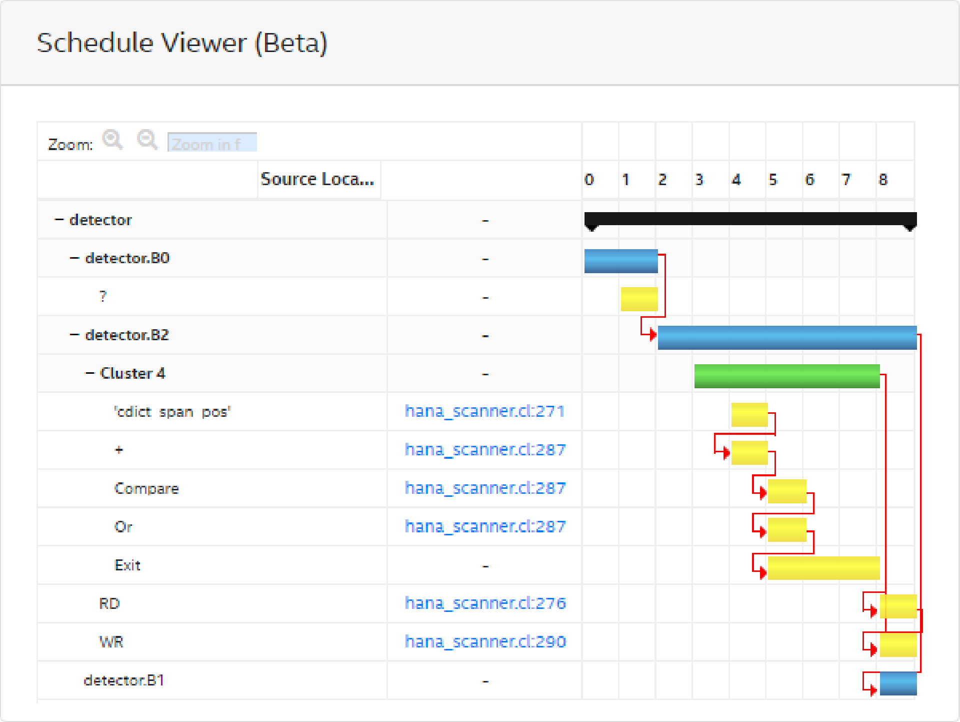
2.4.3. Features of the Schedule Viewer

The Schedule Viewer displays a static view of the scheduled cycle and latency of a clustered group of instructions in your design. Use this report to view loop bottlenecks such as f_{MAX}/II bottlenecks, memory dependency, and occupancy limiter.

In the Schedule Viewer:

- Columns depict the clock cycles.
- Rows display a list of kernels, blocks, clusters, and instructions ranked by the order of execution.
- The red arrows are dependency lines for each block, cluster, or instruction. The arrows show how each block, cluster, or instruction is dependent on other blocks, clusters, or instructions. Hovering over a node (bar) highlights its outgoing dependency lines.
- Each row represents a node and its start and end cycle.
- The bars are color-coded. Black indicates a kernel, blue indicates a block, green indicates a cluster, and yellow indicates an instruction.

Figure 23. Schedule Viewer



	Source Loca...
− detector	-
− detector.B0	-
?	-
− detector.B2	-
− Cluster 4	-
'cdict span pos'	hana_scanner.cl:271
+	hana_scanner.cl:287
Compare	hana_scanner.cl:287
Or	hana_scanner.cl:287
Exit	-
RD	hana_scanner.cl:276
WR	hana_scanner.cl:290
detector.B1	-

2.5. Analyzing Throughput

The *<your_kernel_filename>*`/reports/report.html` file contains information that helps you optimize your design based on the throughput knobs.

From the Reports menu's **Throughput Analysis** drop-down menu, you can analyze throughput Loop Analysis report. The purpose of this view is to show the bottleneck and loop hardware. For each loop, you can identify if it is pipelined and uses hyper-optimized loop structure, and whether user pragma is applied. You can also find out the loop's II and loop speculation value. For more information, refer to Reviewing Loop Information on page 33.

Note: Loop Analysis does not report anything about NDRange loops.

2.5.1. Reviewing Loop Information

The Loops Analysis report contains information about all the loops (coalesced, unrolled, and fused loops) in your design and their unroll statuses. This report helps you examine whether the Intel FPGA SDK for OpenCL Offline Compiler can maximize the throughput of your kernel.

Note: The f_{MAX} II report is now deprecated and its information is merged with the Loop Analysis report.

To view detailed information about the throughput bottlenecks, use the Bottlenecks viewer.

To access the report, click **Throughput Analysis ➤ Loop Analysis**. The left-hand **Loops List** pane displays the following types of loops:

- Fused loops
- Fused subloops
- Coalesced loops
- Fully unrolled loops
- Partial unrolled loops
- Regular loops

The Loops Analysis report captures the following key performance metrics on all blocks:

- **Source Location**: Indicates the loop location in the source code.
- **Pipelined**: Indicates whether the body of a loop is pipelined. Pipelining allows for many data items to get processed concurrently (in the same clock cycle) while making efficient use of the hardware in the datapath by keeping it occupied.
- **II**: Shows the sustainable initiation interval (II) of the loop. Processing data in loops is an additional source of pipeline parallelism. When you pipeline a loop, the next iteration of the loop begins before previous iterations complete.

  You can determine the number of clock cycles between iterations by the number of clock cycles you require to resolve any dependencies between iterations. You can refer to this number as the initiation interval (II) of the loop.

  The Intel FPGA SDK for OpenCL Offline Compiler automatically identifies these dependencies and builds hardware to resolve these dependencies while minimizing the II. For additional information, refer to Specifying a loop initiation interval (II).

- **Scheduled f_{MAX}**: Shows the scheduled maximum clock frequency at which the loop operates. The f_{MAX} is the maximum rate at which the outputs of registers are updated.

  The physical propagation delay of the signal between two consecutive registers limits the clock speed. This propagation delay is a function of the complexity of the Boolean logic in the path. The path with the most logic (and the highest delay) limits the speed of the entire circuit, and you can refer to this path as the critical path.

  The f_{MAX} is calculated as the inverse of the critical path delay. High f_{MAX} is desirable because it correlates directly with high performance in the absence of other bottlenecks. The offline compiler attempts to optimize the kernel for different objectives for the scheduled f_{MAX} depending on whether the f_{MAX} target is set and whether the `#pragma II` is set for each of the loops. The f_{MAX} target is a strong suggestion and the compiler does not error out if it is not able to achieve this f_{MAX}, whereas the `#pragma II` triggers an error if the compiler cannot achieve the requested II. The f_{MAX} achieved for each block of code is shown in the Loops report.

  The following table outlines the behavior of the scheduler in the Intel FPGA SDK for OpenCL Offline Compiler:

Explicitly Specify f_{MAX}?	Explicitly Specify II?	Compiler Behavior
No	No	Use heuristic to achieve best f_{MAX}/II trade-off.
No	Yes	Best effort to achieve the II for the corresponding loop (may not achieve the best possible f_{MAX}).
Yes	No	Best effort to achieve f_{MAX} specified (may not achieve the best possible II).
Yes	Yes	Best effort to achieve the f_{MAX} specified at the given II. The compiler errors out if it cannot achieve the requested II.

  Note: If you are using an f_{MAX} target in the command line or for a kernel, use `#pragma II = <N>` for performance-critical loops in your design.

- **Latency**: Shows the number of clock cycles a loop takes to complete one or more instructions. Typically, you want to have low latency. However, lowering latency often results in decreased f_{MAX}.

- **Speculated Iterations**: Shows the loop speculation. Loop speculation is an optimization technique that enables more efficient loop pipelining by allowing future iterations to get initiated before determining whether the loop exited already. For more information, refer to Loop Speculation on page 78.

- **Max Interleaving Iterations**: Indicates the number of interleaved invocations of an inner loop that can be executed simultaneously. For more information, refer to Loop Interleaving Control.

You can use the Loops Analysis report to determine where to deploy one or more pragmas on your loops. Refer to the following pragma documentation in the *Intel FPGA SDK for OpenCL Programming Guide*:

Table 3. Loop Pragmas

Pragma	Reference
#pragma unroll	Unrolling a Loop
#pragma loop_coalesce	Coalescing Nested Loops
#pragma ii	Specifying a loop initiation interval (II)
#pragma speculated_iterations	Loop Speculation
#pragma max_concurrency	Loop Concurrency
#pragma max_interleaving	Loop Interleaving Control
#pragma disable_loop_pipelining	Disabling Pipelining of a Loop
#pragma loop_fuse	Fusing Adjacent Loops
#pragma nofusion	Marking Loops to Prevent Automatic Fusion

Example 1. OpenCL Kernel Example

The following is an OpenCL kernel example that includes four loops:

```
// ND-Range kernel with unrolled loops
__attribute((reqd_work_group_size(1024,1,1)))
kernel void t (global int * out, int N) {
  int i = get_global_id(0);
  int j = 1;
  for (int k = 0; k < 4; k++) {
    #pragma unroll
    for (int n = 0; n < 4; n++) {
      j += out[k+n];
    }
  }
  out[i] = j;

  int m = 0;
  #pragma unroll 1
  for (int k = 0; k < N; k++) {
    m += out[k/3];
  }
  #pragma unroll
  for (int k = 0; k < 6; k++) {
    m += out[k];
  }
  #pragma unroll 2
  for (int k = 0; k < 6; k++) {
    m += out[k];
  }
  out[2] = m;
}
```

The loop analysis report of this design example highlights the unrolling strategy for the different kinds of loops defined in the code.

The Intel FPGA SDK for OpenCL Offline Compiler executes the following loop unrolling strategies based on the source code:

- Fully unrolls the inner loop (line 8) within the first loop because of the `#pragma unroll` specification
- Does not unroll the second outer loop, Block4 (line 16), because of the `#pragma unroll 1` specification
- Fully unrolls the third outer loop (line 20) because of the `#pragma unroll` specification
- Unrolls the fourth outer loop, Block5 (line 24), twice because of the `#pragma unroll 2` specification

2.6. Reviewing Area Information

The <*your_kernel_filename*>`/reports/report.html` file contains information about area usage of your OpenCL system. You can view the area usage information of the system.

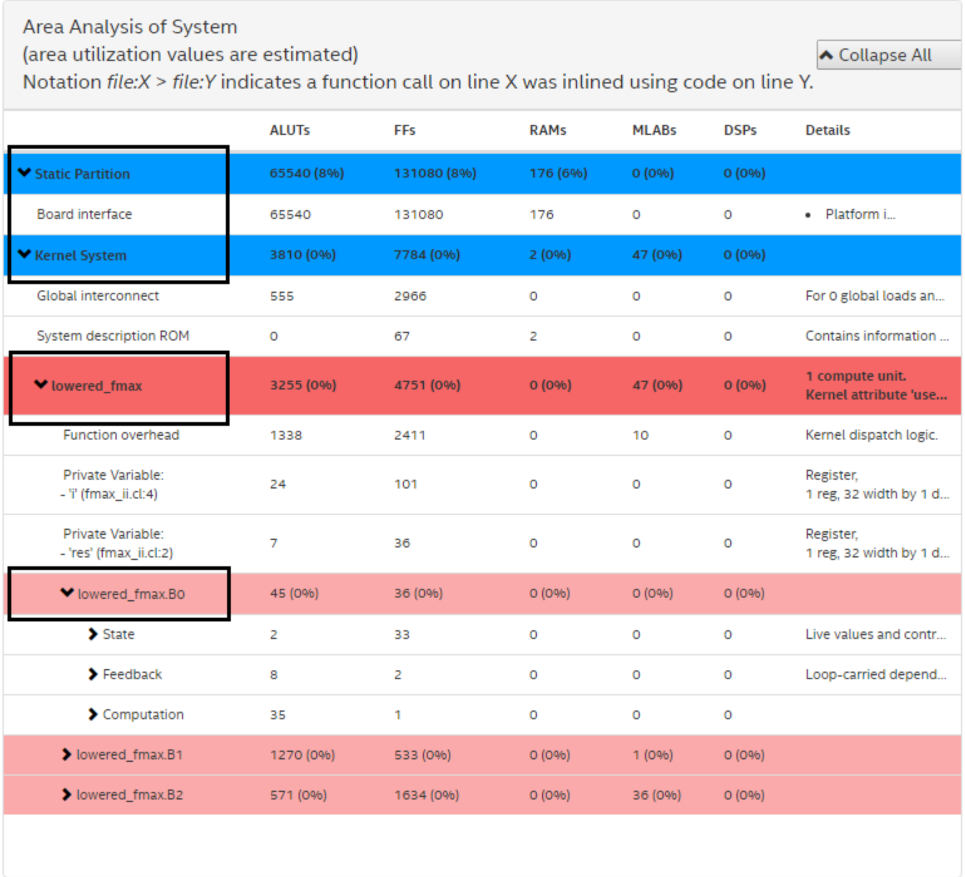
The area report serves the following purposes:

- Provides detailed area breakdown of the whole OpenCL system. The breakdown is related to the source code.
- Provides architectural details to give insight into the generated hardware and offers actionable suggestions to resolve potential inefficiencies.

As observed in the following figure, the area report is divided into three levels of hierarchy:

- **System area**: It is used by all kernels, channels, interconnects, and board logic.
- **Kernel area**: It is used by a specific kernel, including overheads, for example, dispatch logic.
- **Block area**: It is used by a specific basic block within a kernel. A basic block area represents a branch-free section of your source code, for example, a loop body.

Figure 24. Area Report Hierarchy



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

	ALUTs	FFs	RAMs	MLABs	DSPs	Details
Static Partition	65540 (8%)	131080 (8%)	176 (6%)	0 (0%)	0 (0%)	
Board interface	65540	131080	176	0	0	• Platform i...
Kernel System	3810 (0%)	7784 (0%)	2 (0%)	47 (0%)	0 (0%)	
Global interconnect	555	2966	0	0	0	For 0 global loads an...
System description ROM	0	67	2	0	0	Contains information ...
lowered_fmax	3255 (0%)	4751 (0%)	0 (0%)	47 (0%)	0 (0%)	1 compute unit. Kernel attribute 'use...
Function overhead	1338	2411	0	10	0	Kernel dispatch logic.
Private Variable: - 'i' (fmax_ii.cl:4)	24	101	0	0	0	Register, 1 reg, 32 width by 1 d...
Private Variable: - 'res' (fmax_ii.cl:2)	7	36	0	0	0	Register, 1 reg, 32 width by 1 d...
lowered_fmax.B0	45 (0%)	36 (0%)	0 (0%)	0 (0%)	0 (0%)	
State	2	33	0	0	0	Live values and contr...
Feedback	8	2	0	0	0	Loop-carried depend...
Computation	35	1	0	0	0	
lowered_fmax.B1	1270 (0%)	533 (0%)	0 (0%)	1 (0%)	0 (0%)	
lowered_fmax.B2	571 (0%)	1634 (0%)	0 (0%)	36 (0%)	0 (0%)	

Note: The area usage data are estimates that the Intel FPGA SDK for OpenCL Offline Compiler generates. These estimates might differ from the final area utilization results.

In the Reports pane's **Area Analysis** drop-down menu, select **Area Analysis of System**.

In the system view, the kernel is divided into logic blocks. To view the area usage information for the code lines associated with a block, simply expand the report entry for that block.

Note: The `analyze-area` Intel FPGA SDK for OpenCL utility option has been deprecated. For reference information on the deprecated area report, refer to the *Review Your Kernel's Area Report to Identify Inefficiencies in Resource Usage* section in version 16.0 of the *Altera SDK for OpenCL Best Practices Guide*.

Related Information

Altera SDK for OpenCL Best Practices Guide version 16.0

2.6.1. Area Report Message for Board Interface

The area report identifies the amount of logic that the Intel FPGA SDK for OpenCL Offline Compiler generates for the Custom Platform, or board interface. The board interface is the static region of the device that facilitates communication with external interfaces such as PCIe®. The Custom Platform specifies the size of the board interface.

Table 4. Additional Information on Area Report Message

Message	Notes
Platform interface logic.	—

2.6.2. Area Report Message for Function Overhead

The area report identifies the amount of logic that the Intel FPGA SDK for OpenCL Offline Compiler generates for tasks such as dispatching kernels.

Table 5. Additional Information on Area Report Message

Message	Notes
Kernel dispatch logic.	A kernel that includes the `max_global_work_dim(0)` kernel attribute contains no overhead. As a result, this row is not present in the corresponding area report.

2.6.3. Area Report Message for State

The area report identifies the amount of resources that your design uses for live values and control logic.

To reduce the reported area consumption under State, modify your design as follows:

- Decrease the size of local variables
- Decrease the scope of local variables by localizing them whenever possible
- Decrease the number of nested loops in the kernel

2.6.4. Area Report Message for Feedback

The area report specifies the resources that your design uses for loop-carried dependencies.

To reduce the reported area consumption under Feedback, decrease the number and size of loop-carried variables in your design.

2.6.5. Area Report Messages for Private Variable Storage

The area report provides information on the implementation of private memory based on your OpenCL design. For single work-item kernels, the Intel FPGA SDK for OpenCL Offline Compiler implements private memory differently, depending on the types of variable. The offline compiler implements scalars and small arrays in registers of various configurations (for example, plain registers, shift registers, and barrel shifter). The offline compiler implements larger arrays in block RAM.

Table 6. Additional Information on Area Report Messages

Message	Notes
Implementation of Private Memory Using On-Chip Block RAM	
Private memory implemented in on-chip block RAM.	The block RAM implementation creates a system that is similar to local memory for NDRange kernels.
Implementation of Private Memory Using On-Chip Block ROM	
—	For each usage of an on-chip block ROM, the offline compiler creates another instance of the same ROM. There is no explicit annotation for private variables that the offline compiler implements in on-chip block ROM.
Implementation of Private Memory Using Registers	
Implemented using registers of the following size: - <*X*> registers of width <*Y*> bits and depth <*Z*>. • Depth was increased by a factor of <*N*> due to a loop initiation interval of <*M*>. • Each register is implemented in a RAM-based FIFO and consumes <*U*> RAMs. - ...	Reports that the offline compiler implements a private variable in registers. The offline compiler might implement a private variable in many registers. This message provides a list of the registers with their specific widths and depths.
Implementation of Private Memory Using Shift Registers	
Implemented as a shift register with <*N*> or fewer tap points. This is a very efficient storage type. Implemented using registers of the following sizes: - <*X*> registers of width <*Y*> bits and depth <*Z*>. • Depth was increased by a factor of <*N*> due to a loop initiation interval of <*M*>. • Each register is implemented in a RAM-based FIFO and consumes <*U*> RAMs. - ...	Reports that the offline compiler implements a private variable in shift registers. This message provides a list of shift registers with their specific widths and depths. The offline compiler might break a single array into several smaller shift registers depending on its tap points. *Note:* The offline compiler might overestimate the number of tap points.
Implementation of Private Memory Using Barrel Shifters with Registers	
Implemented as a barrel shifter with registers due to dynamic indexing. This is a high overhead storage type. If possible, change to compile-time known indexing. The area cost of accessing this variable is shown on the lines where the accesses occur. Implemented using registers of the following size: - <*X*> registers of width <*Y*> bits and depth <*Z*>. • Depth was increased by a factor of <*N*> due to a loop initiation interval of <*M*>. • Each register is implemented in a RAM-based FIFO and consumes <*U*> RAMs. - ...	Reports that the offline compiler implements a private variable in a barrel shifter with registers because of dynamic indexing. This row in the report does not specify the full area use of the private variable. The report shows additional area use information on the lines where the variable is accessed.

Note:

- The area report annotates memory information on the line of code that declares or uses private memory, depending on its implementation.
- When the offline compiler implements private memory in on-chip block RAM, the area report displays relevant local-memory-specific messages to private memory systems.

2.6.5.1. Area Report Message for Constant Memory

The area report specifies the size of the constant cache memory. It also provides information such as data replication and the number of read operations.

Table 7. Additional Information on Area Report Message

Message	Notes
<N> bytes constant cache accessible to all kernels and is persistent across kernel invocations. Data inside the cache is replicated *<X>* times to support *<Y>* reads. Cache optimized 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 Intel FPGA dynamic profiler for OpenCL to check stalls on accesses to the cache to assess the cache's effectiveness. Profiling actual cache hit rate is currently not supported.	—

2.7. Optimizing an OpenCL Design Example Based on Information in the HTML Report

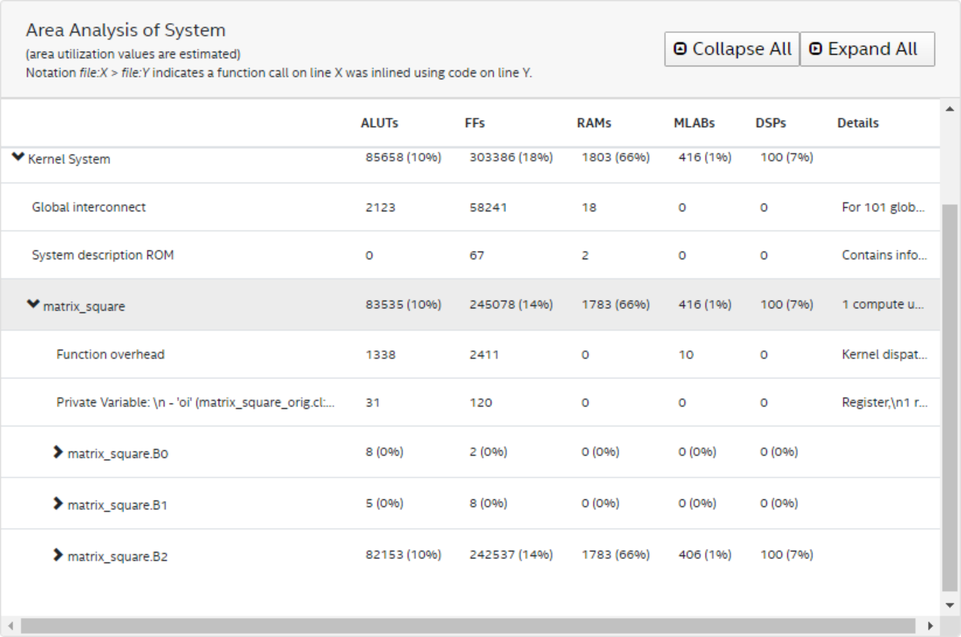
A guide on how to use the information in the HTML report to optimize an OpenCL kernel.

OpenCL design example that performs matrix square AxA:

```
// performs matrix square A*A
// A is a square LEN*LEN matrix
// A = [ r0       : [ c[0], c[1], ... c[LEN-1] ],
//       r1       : [ c[0], ...                ],
//       ...                                   ],
//       r[LEN-1] : [                          ] ]

// LEN = 100

kernel void matrix_square (global float* restrict A, global float* restrict out)
{
    for( unsigned oi = 0 ; oi < LEN*LEN ; oi++ )
    {
        float sum = 0.f;
        int row = oi / LEN;
        int col = oi % LEN;

        #pragma unroll
        for ( int stride = 0 ; stride < LEN ; stride++ )
        {
            unsigned i = (row * LEN) + stride;
            unsigned j = (stride * LEN) + col;
            sum += A[i] * A[j];
        }

        out[oi] = sum;
    }
}
```

The area analysis of the kernel `matrix_square` indicates that the estimated usages of flipflops (FF) and RAMs are high.

Figure 25. Area Report of the Unoptimized Kernel `matrix_square`

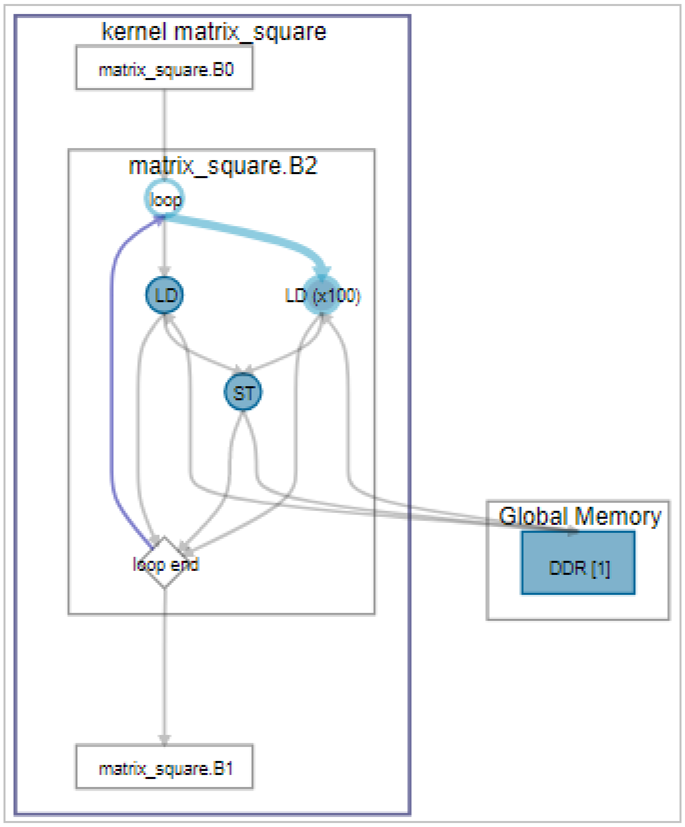


Area Analysis of System
(area utilization values are estimated)
Notation *file:X* > *file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	MLABs	DSPs	Details
Kernel System	85658 (10%)	303386 (18%)	1803 (66%)	416 (1%)	100 (7%)	
Global interconnect	2123	58241	18	0	0	For 101 glob...
System description ROM	0	67	2	0	0	Contains info...
matrix_square	83535 (10%)	245078 (14%)	1783 (66%)	416 (1%)	100 (7%)	1 compute u...
Function overhead	1338	2411	0	10	0	Kernel dispat...
Private Variable: \n - 'oi' (matrix_square_orig.cl:...	31	120	0	0	0	Register,\n1 r...
matrix_square.B0	8 (0%)	2 (0%)	0 (0%)	0 (0%)	0 (0%)	
matrix_square.B1	5 (0%)	8 (0%)	0 (0%)	0 (0%)	0 (0%)	
matrix_square.B2	82153 (10%)	242537 (14%)	1783 (66%)	406 (1%)	100 (7%)	

Further examination of block 2 in the System viewer shows that Block2 also has a high latency value.

Figure 26. System Viewer Results for the Unoptimized Kernel `matrix_square`

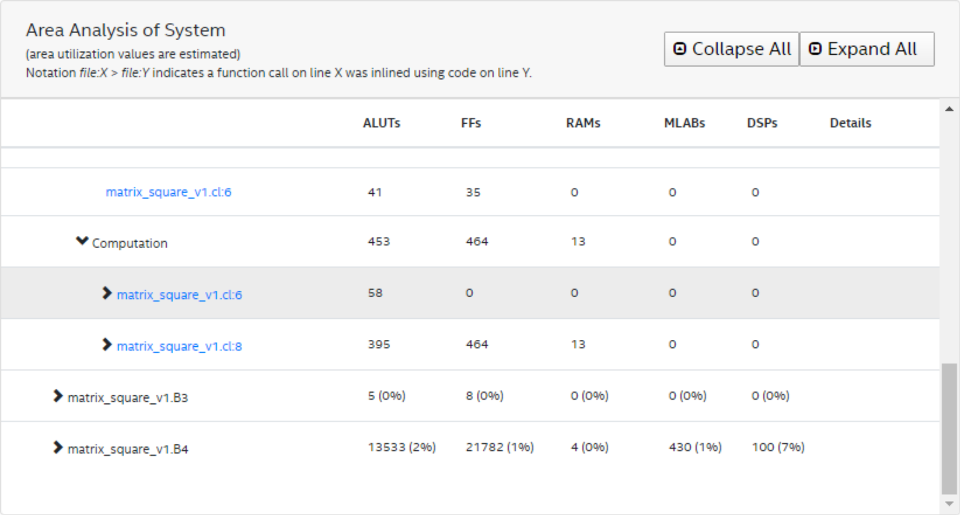


The cause for these performance bottlenecks is the system is loading data from global memory from inside a loop. Therefore, the first optimization step you can take is to preload the data into local memory, as shown in the following modified code:

```
 // 1. preload the data into local memory

 kernel void matrix_square_v1 (global float* restrict A, global float* restrict
out)
 {
     local float cache_a[LEN*LEN];
     for( unsigned k = 0 ; k < LEN*LEN ; k++ )
     {
         cache_a[k] = A[k];
     }

     for( unsigned oi = 0 ; oi < LEN*LEN ; oi++ )
     {
         float sum = 0.f;
         int row = oi / LEN;
         int col = oi % LEN;

     #pragma unroll
     for( unsigned stride = 0 ; stride < LEN ; stride++ )
         {
             unsigned i = (row * LEN) + stride;
             unsigned j = (stride * LEN) + col;
             sum += cache_a[i] * cache_a[j];
         }
```

```
            out[oi] = sum;
        }
    }
```

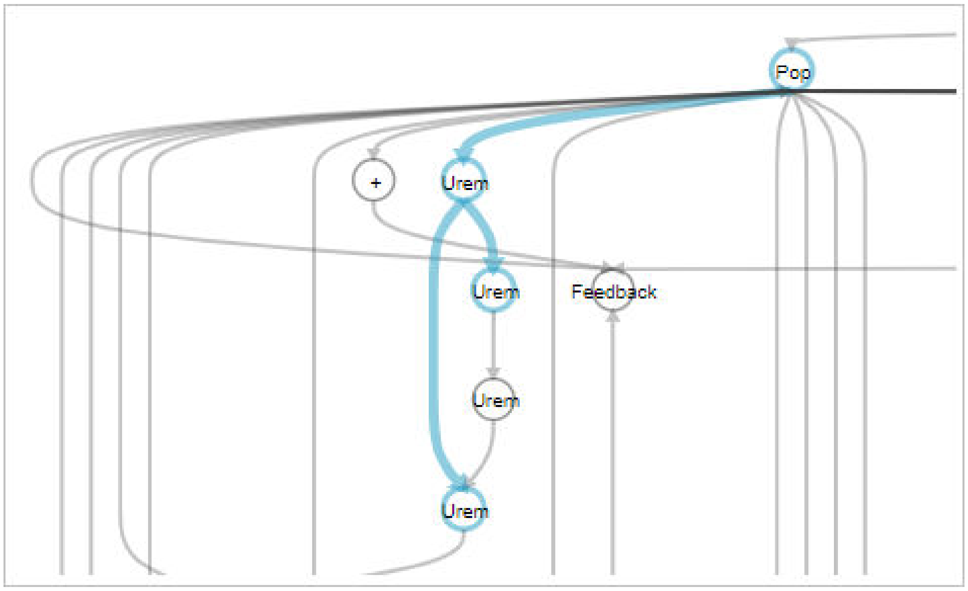
Figure 27. Area Report Results for the Modified Kernel matrix_square_v1



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

	ALUTs	FFs	RAMs	MLABs	DSPs	Details
matrix_square_v1.cl:6	41	35	0	0	0	
❯ Computation	453	464	13	0	0	
❯ matrix_square_v1.cl:6	58	0	0	0	0	
❯ matrix_square_v1.cl:8	395	464	13	0	0	
❯ matrix_square_v1.B3	5 (0%)	8 (0%)	0 (0%)	0 (0%)	0 (0%)	
❯ matrix_square_v1.B4	13533 (2%)	21782 (1%)	4 (0%)	430 (1%)	100 (7%)	

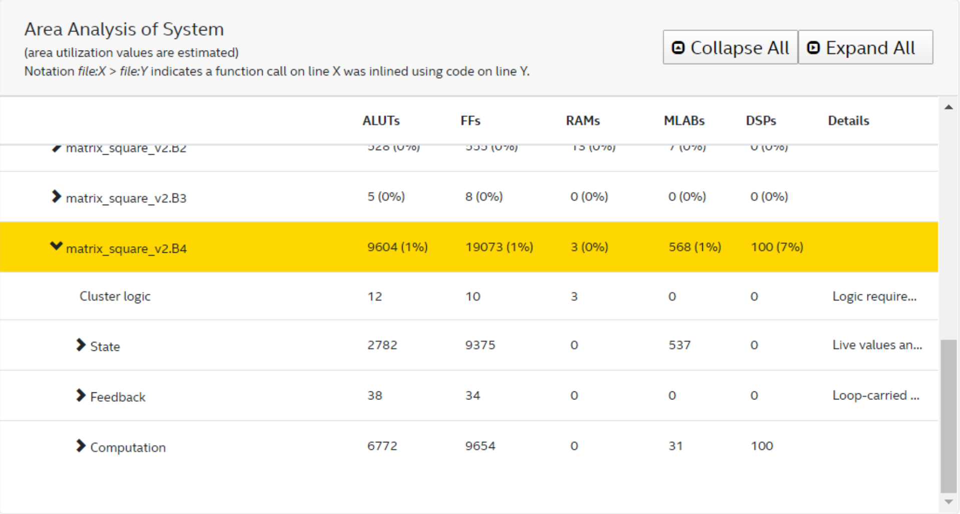
Figure 28. Cluster View of the Modified Kernel matrix_square_v1



If you remove the modulus computation and replace it with a column counter, as shown in the modified kernel `matrix_square_v2`, you can reduce the amount of adaptive look-up table (ALUT) and FF use.

```
 // 1. preload the data into local memory
 // 2. remove the modulus computation

 kernel void matrix_square_v2 (global float* restrict A, global float* restrict
out)
 {
     local float cache_a[LEN*LEN];
     for( unsigned k = 0 ; k < LEN*LEN ; k++ )
     {
         cache_a[k] = A[k];
     }

     unsigned row = 0;
     unsigned col = 0;

     for( unsigned oi = 0 ; oi < LEN*LEN ; oi++ )
     {
         float sum = 0.f;

         // keep a column counter to know when to increment row
         if( col == LEN - 1 )
         {
             col = 0;
             row += 1;
         }
         else
         {
             col += 1;
         }

         #pragma unroll
         for( unsigned stride = 0 ; stride < LEN ; stride++ )
         {
             unsigned i = (row * LEN) + stride;
             unsigned j = (stride * LEN) + col;
             sum += cache_a[i] * cache_a[j];
         }

         out[oi] = sum;
     }
 }
```

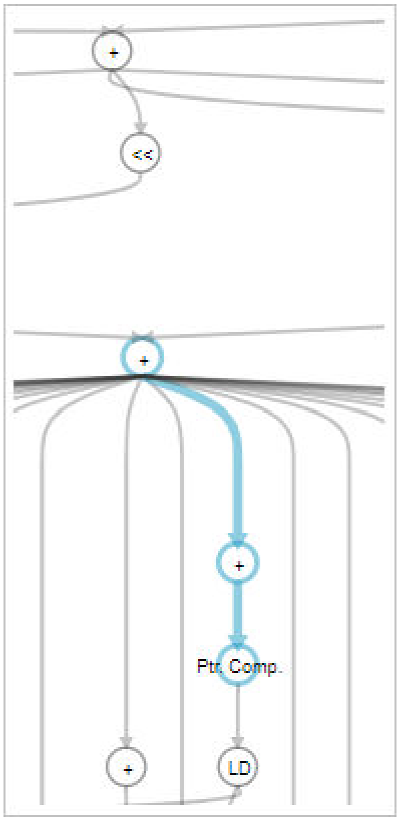
Figure 29. Area Report of Kernel `matrix_square_v2`



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

	ALUTs	FFs	RAMs	MLABs	DSPs	Details
matrix_square_v2.B2	328 (0%)	353 (0%)	13 (0%)	7 (0%)	0 (0%)	
› matrix_square_v2.B3	5 (0%)	8 (0%)	0 (0%)	0 (0%)	0 (0%)	
⌄ matrix_square_v2.B4	9604 (1%)	19073 (1%)	3 (0%)	568 (1%)	100 (7%)	
Cluster logic	12	10	3	0	0	Logic require...
› State	2782	9375	0	537	0	Live values an...
› Feedback	38	34	0	0	0	Loop-carried ...
› Computation	6772	9654	0	31	100	

Figure 30. Cluster View of Kernel `matrix_square_v2`



Further examination of the area report of `matrix_square_v2` reveals that the computations for indexes `i` and `j` (that is, `unsigned i = (row * LEN) + stride` and `unsigned j = (stride * LEN) + col`, respectively) have very different ALUT usage estimations.

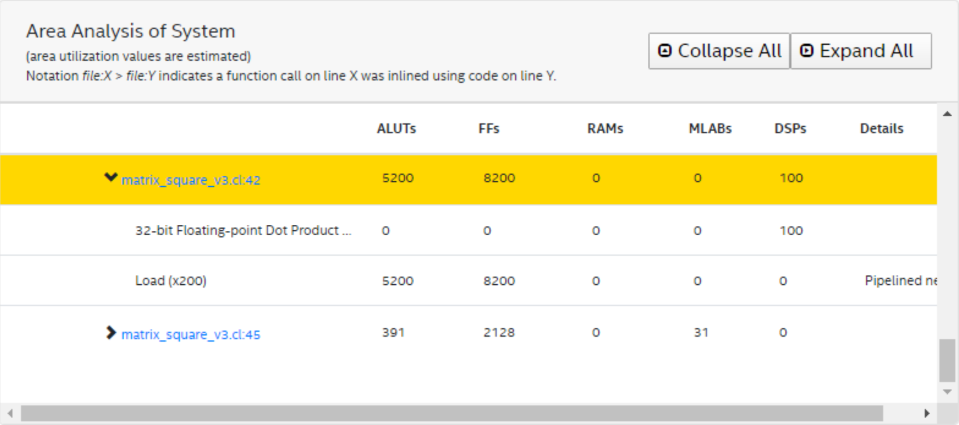
A way to optimize DSP and RAM block usages for index calculation is to remove the multiplication computation and simply keep track of the addition, as shown in the modified kernel `matrix_square_v3` below.

```
// 1. preload the data into local memory
 // 2. remove the modulus computation
 // 3. remove DSP and RAM blocks for index calculation helps reduce the latency

 kernel void matrix_square_v3 (global float* restrict A, global float* restrict 
out)
 {

   local float cache_a[LEN*LEN];
     for( unsigned k = 0 ; k < LEN*LEN ; k++ )
     {
         cache_a[k] = A[k];
     }

     unsigned row_i = 0;
     unsigned row_j = 0;

     for( unsigned oi = 0 ; oi < LEN*LEN ; oi++ )
     {
         unsigned i, j;
```

```
        // keep a column base counter to know when to increment the row base
        if( row_j == LEN - 1 )
        {
            row_i += LEN;
            row_j = 0;
        }
        else
        {
            row_j += 1;
        }

        // initialize i and j
        i = row_i;
        j = row_j;

        float sum = 0.f;
        #pragma unroll
        for( unsigned stride = 0 ; stride < LEN ; stride++ )
        {
            i += 1;       // 0, 1, 2, 3, 0,...
            j += LEN;     // 0, 3, 6, 9, 1,...
            sum += cache_a[i] * cache_a[j];
        }

        out[oi] = sum;
    }
}
```

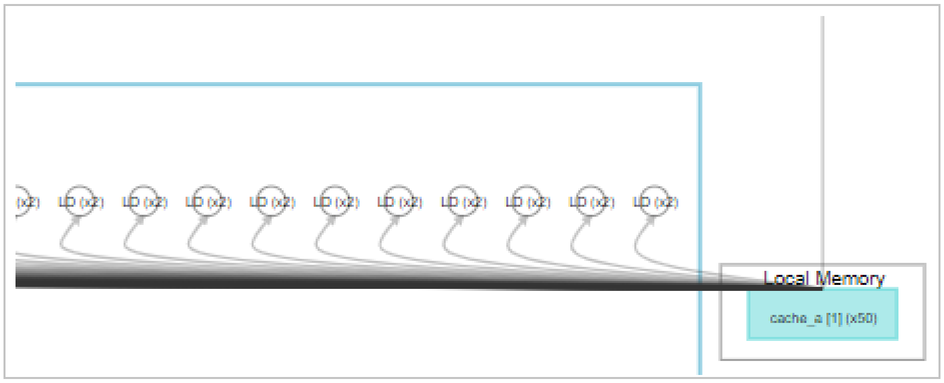
By removing the multiplication step, you can reduce DSP usage as shown in the area report below. In addition, the modification helps reduce latency.

Figure 31. Area Report of Kernels `matrix_square_v3`


Area Analysis of System
(area utilization values are estimated)
Notation *file:X* > *file:Y* indicates a function call on line X was inlined using code on line Y.
Collapse All
Expand All

	ALUTs	FFs	RAMs	MLABs	DSPs	Details
matrix_square_v3.cl:42	5200	8200	0	0	100	
32-bit Floating-point Dot Product ...	0	0	0	0	100	
Load (x200)	5200	8200	0	0	0	Pipelined ne
matrix_square_v3.cl:45	391	2128	0	31	0	

Figure 32. System Viewer of the Kernel `matrix_square_v3`



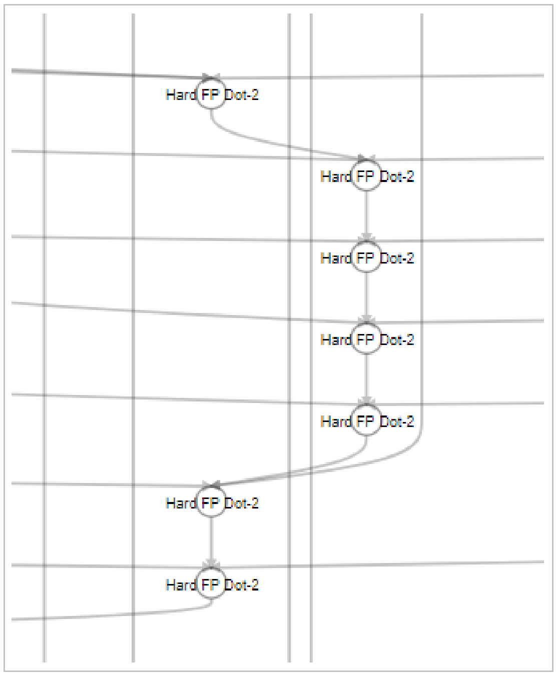
To resolve the loop-carried dependency, unroll the sum-product for complete parallelism and create registers to avoid multiple copies of `cache_a`, as shown in the code below in the modified kernel `matrix_square_v4`.

```
// 1. preload the data into local memory
 // 2. remove the modulus computation
 // 3. remove DSP and RAM blocks for index calculation helps reduce the latency
 // 4. unroll the sum-product for full parallelism, create registers to avoid
many copies of cache_a

 kernel void matrix_square_v4 (global float* restrict A, global float* restrict
out)
 {

   local float cache_a[LEN*LEN];
     for( unsigned k = 0 ; k < LEN*LEN ; k++ )
     {
         cache_a[k] = A[k];
     }

     unsigned row_i = 0;
     unsigned row_j = 0;

     for( unsigned oi = 0 ; oi < LEN*LEN ; oi++ )
     {
         unsigned i, j;
         // keep a column base counter to know when to increment the row base
         if( row_j == LEN - 1 )
         {
             row_i += LEN;
             row_j = 0;
         }
         else
         {
             row_j += 1;
         }

         // initialize i and j
         i = row_i;
         j = row_j;

         float r_buf[LEN];
         float c_buf[LEN];
         for( int stride = 0 ; stride < LEN ; stride++ )
         {
             i += 1;       // 0, 1, 2, 3, 0,...
             j += LEN;     // 0, 3, 6, 9, 1,...
             r_buf[stride] = cache_a[i];
             c_buf[stride] = cache_a[j];
         }
```

```
        // uses harder floating point when -fp-relaxed is used
        float sum = 0.f;
        #pragma unroll
        for(unsigned idx = 0; idx &lt; LEN; idx++)
        {
            sum += r_buf[idx] * c_buf[idx];
        }

        out[oi] = sum;
    }
}
```

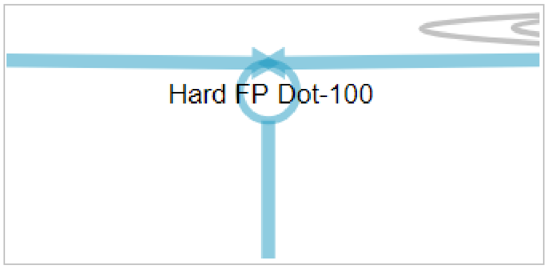
As shown in the cluster view results below, by breaking up the computation steps, you can achieve higher throughput at the expense of increased area usage. The modification also reduces the latency by 50%.

Figure 33. Cluster View of the Modified Kernel `matrix_square_v4`



The following cluster view provides an alternative with `-fp-relaxed` to show dot product instead of chain:

Figure 34. Cluster View of the Modified Kernel `matrix_square_v4`



The following table provides the throughput comparison for all five versions of the kernel:

Table 8. Throughput Comparison

Kernel	ALUTs	FFs	RAMs	MLABs	DSPs	Dot Product Loop Latency
`matrix_square`	81806 (10%)	302792 (18%)	1989 (73%)	408 (1%)	100 (7%)	637
`matrix_square_v1`	20094 (2%)	38814 (2%)	1619 (60%)	248 (1%)	100 (7%)	380
`matrix_square_v2`	15487 (2%)	51813 (3%)	1110 (41%)	298 (1%)	100 (7%)	364
`matrix_square_v3`	18279 (2%)	37554 (2%)	1618 (60%)	244 (1%)	100 (7%)	362

`matrix_square_v4`
`(-fp-relaxed)`
	9681 (1%)	22409 (1%)	257 (9%)	67 (0%)	103 (7%)	37

2.8. Accessing HLD FPGA Reports in JSON Format

In addition to the `report.html` file, the Intel FPGA SDK for OpenCL also provides the HLD FPGA Report data in JSON files.

The JSON files containing the HLD FPGA Reportreport data are available in the *<your_kernel_filename>*`/reports/lib/json` directory. The directory provides the following `.json` files:

Table 9. JSON Files in the <your_kernel_filename>/reports/lib/json Directory

File	Description
`area.json`	Area Analysis of System
`block.json`	Block View of System Viewer
`bottleneck.json`	Bottleneck View of Loop Analysis Report
`gmv.json`	Global Memory View of the System Viewer
`info.json`	Summary of project name, compilation command, versions, and timestamps
`loops.json`	Navigation tree of Loop Analysis report
`loops_attr.json`	Loop Analysis report
`mav.json`	System View of System Viewer
`new_lmv.json`	Kernel Memory Viewer
`pipeline.json`	Cluster View of System Viewer
`quartus.json`	Quartus Prime compilation summary
`schedule.json`	Schedule Viewer

continued...

File	Description
summary.json	Kernel compilation name mapping
tree.json	Navigation tree of System Viewer
warnings.json	Compilation warning messages

Important: The structure of these JSON files might change from release to release without notice.

You can read the following `.json` files without a special parser:

- `area.json`
- `area_src.json`
- `loops.json`
- `quartus.json`
- `summary.json`

For example, if you want to identify all of the values and bottlenecks for the initiation interval (II) of a loop, you can find the information in the `children` section in the `loops.json` file, as shown below:

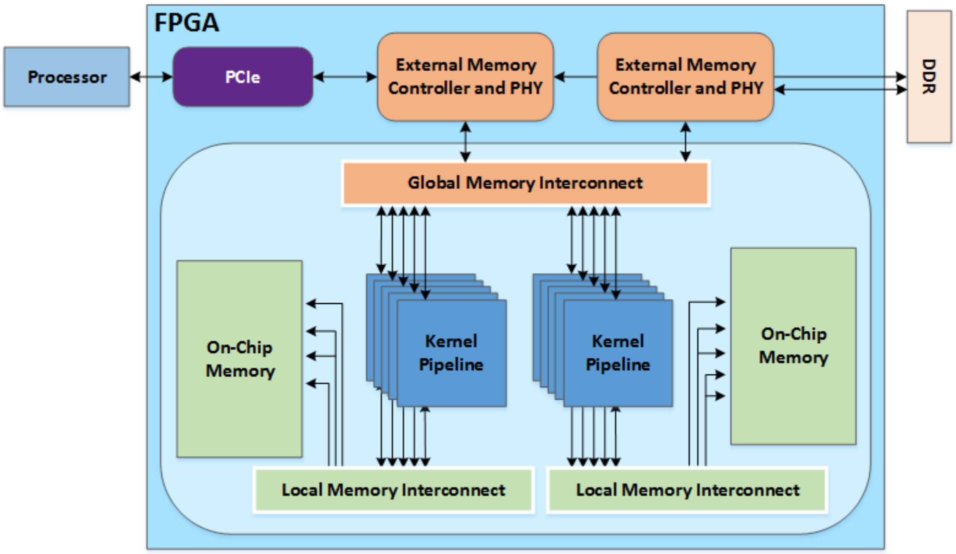
```
“name”:”<block name|Kernel: kernel name>  # Find the loops which does not begin
with “Kernel:”

“data”:[<Yes|No>, <#|n/a>, <II|n/a>]      # The data field corresponds to
“Pipelined”, “II”, “Bottleneck”
```

3. OpenCL Kernel Design Concepts

Familiarizing yourself on how the Intel FPGA SDK for OpenCL implements OpenCL design components such as kernels, global memory interconnect, local memory, loops, and channels can help you optimize your OpenCL design.

Tip: For more detailed explanation of FPGA design concepts, refer to Introduction to FPGA Design Concepts chapter in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

Figure 35. OpenCL Design Components


FPGA
Processor
PCIe
External Memory Controller and PHY
External Memory Controller and PHY
DDR
Global Memory Interconnect
On-Chip Memory
Kernel Pipeline
Kernel Pipeline
On-Chip Memory
Local Memory Interconnect
Local Memory Interconnect

Kernels on page 51

Global Memory Interconnect on page 52

Local Memory on page 54

Loops in a Single Work-Item Kernel on page 66

Channels on page 84

Load-Store Units on page 85

3.1. Kernels

Each kernel in your OpenCL system is represented by a set of blocks. Inside each block is a set of non-branching instructions that implement your algorithm and the offline compiler's loop orchestration logic. The block shows the execution flow of your kernel. When there are loops, there is a back edge to the block or its previous blocks, depending on the loop structure, for example, nested loops. Loops usually impose II bottlenecks and are a main focus for optimization.

A block has three main parts — an input or loop input node, a set of instructions and a branch node. The input and branch nodes may not be present depending on if there is branching in or out of the block. The input or loop input node determines the initial value for variables depending on where the branch into this block originated. The rest of the block contains stallable and non-stallable instructions, and clusters. A well-optimized design should contain a minimal number of stallable instructions, such as stallable I/O or memory accesses.

The non-stallable instructions within in block are grouped into clusters to reduce handshaking overheads with stallable instructions. A cluster has an entry and an exit node. There is only a stall-free cluster. You can find the exit FIFO information in the cluster's exit node. Finally, the branch node informs the next block to go to, under which condition.

In the HLD report, you can find different views of your kernel, under the **Views** drop-down menu. For more information, refer to Using Views on page 19.

The Intel FPGA SDK for OpenCL Offline Compiler compiles a kernel that does not use any built-in work-item functions, such as `get_global_id()` and `get_local_id()`, as a single work-item kernel. Otherwise, the offline compiler compiles the kernel as an NDRange kernel. For more information about built-in work-item functions, refer to section *6.11.1: Work-Item Functions* of the OpenCL Specification version 1.0.

For single work-item kernels, the offline compiler attempts to pipeline every loop in the kernel to allow multiple loop iterations to execute concurrently. Kernel performance might degrade if the compiler cannot pipeline some of the loops effectively, or if it cannot pipeline the loops at all.

The offline compiler cannot pipeline loops in NDRange kernels. However, these loops can accept multiple work-items simultaneously. A kernel might have multiple loops, each with nested loops. If you tabulate the total number of iterations of nested loops for each outer loop, kernel throughput is usually reduced by the largest total iterations value that you have tabulated. To execute an NDRange kernel efficiently, there must a large number of threads.

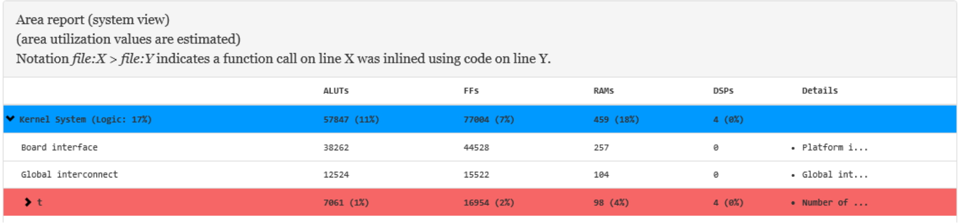
3.2. Global Memory Interconnect

The ability to maximize memory bandwidth for read and write accesses is crucial for high performance computing. Various types of modules for reading from and writing to global memory can exist in an OpenCL system. These modules are called *load-store units* (*LSUs*).

Unlike a GPU, an FPGA can build any custom LSU that is best suited for the memory access pattern that the compiler infers for your application. As a result, your ability to write OpenCL code that selects the ideal LSU types for your application might help improve the performance of your design significantly.

When reviewing the HTML area report of your design, the values in the **Global interconnect** entry at the system level represents the size of the global memory interconnect.

Figure 36. HTML Area Report Showing the Size of the Global Memory Interconnect in an OpenCL Design

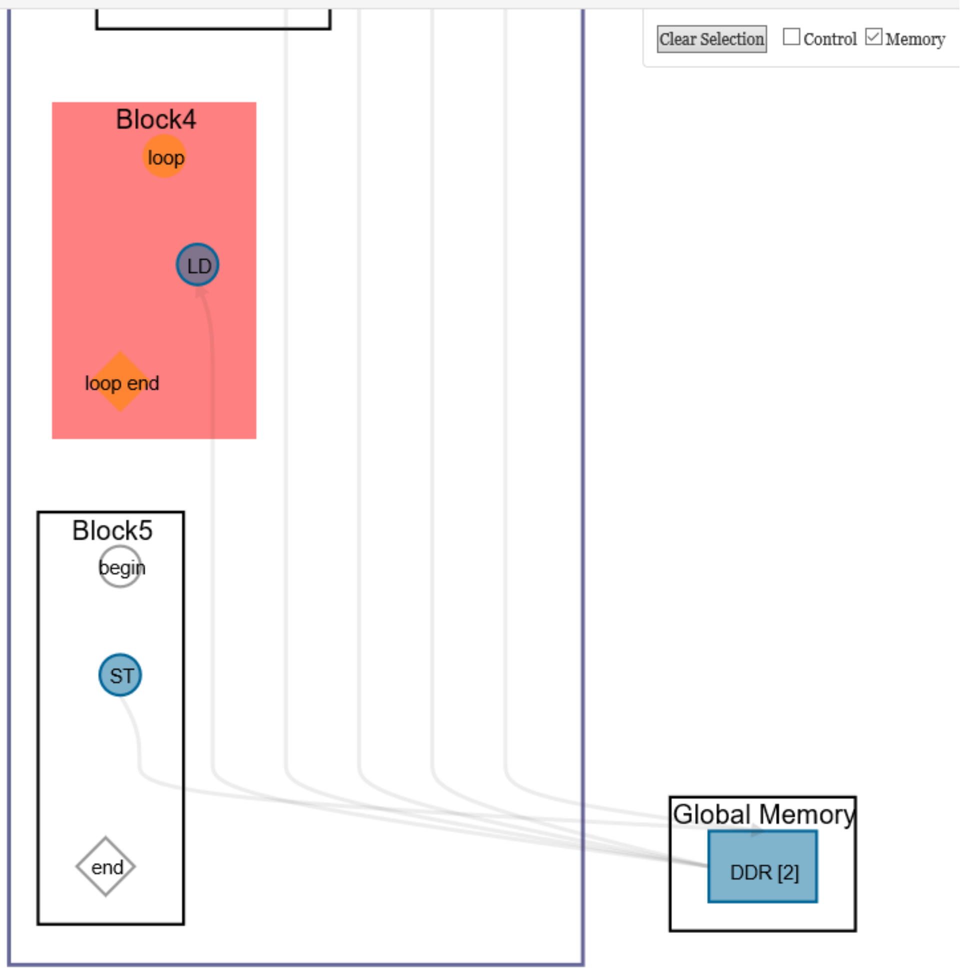


Area report (system view)
(area utilization values are estimated)
Notation *file:X* > *file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 17%)	57847 (11%)	77004 (7%)	459 (18%)	4 (0%)	
Board interface	38262	44528	257	0	• Platform i...
Global interconnect	12524	15522	104	0	• Global int...
t	7061 (1%)	16954 (2%)	98 (4%)	4 (0%)	• Number of ...

In the HTML report, the system view of the System Viewer depicts global memory interconnects as loads (LD), stores (ST), and connections (gray lines).

Figure 37. System Viewer (System View) Result of Global Memory Interconnects in an OpenCL Design



The Intel FPGA SDK for OpenCL Offline Compiler selects the appropriate type of LSU for your OpenCL system based on the memory access pattern of your design. Example LSU types include contiguous access (or consecutive access) and burst-interleaved access. Contiguous Memory Access and Global Memory Partitions illustrate the difference in access patterns between contiguous and burst-interleaved memory accesses, respectively.

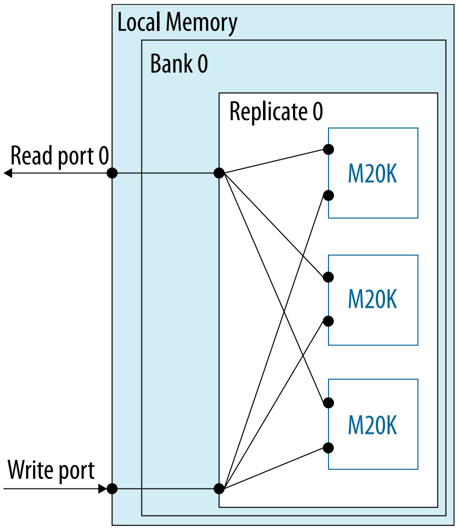
3.3. Local Memory

Local memory is a complex system. Unlike the typical GPU architecture where there are different levels of caches, FPGA implements local memory in dedicated memory blocks inside the FPGA.

Local Memory Characteristics

- **Ports**: Each read or write access to a local memory is mapped to a port.
- **Banks**: The contents of a local memory can be partitioned into one or more banks, such that each bank contains a subset of data contained in a local memory.
- **Replicate**: A bank consists of one or more replicates. Each replicate in a bank has the same data as the other replicates. Replicates are created to efficiently support multiple accesses to a local memory. Each replicate has one write port and one read port that your design can access simultaneously. If your local memory is double pumped, each replicate has four physical ports, of which up to three can be read ports. Refer to the Double Pumping on page 60 section for more information.
- **Private copies**: A replicate can contain one or more private copies to allow pipelined execution of multiple workgroups. Refer to the Local Memory Banks and Private Copies on page 56 section for more information.

Figure 38. Implementation of Local Memory in One or Multiple M20K Blocks



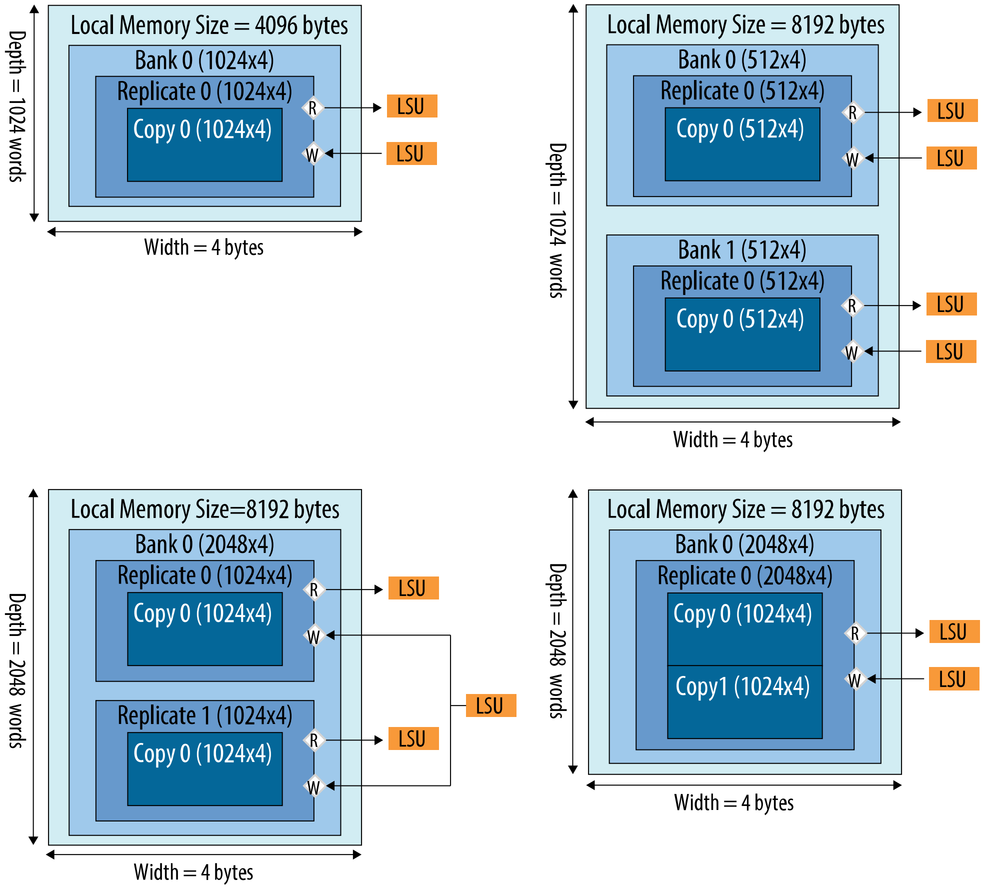
In your kernel code, declare local memory as a variable with type `local`:

```
local int lmem[1024];
```

The Intel FPGA SDK for OpenCL Offline Compiler customizes the local memory properties such as width, depth, banks, replication, number of private copies, and interconnect. The offline compiler analyzes the access pattern based on your code and then optimizes the local memory to minimize access contention.

The following diagrams illustrate these basic local memory properties (size, width, depth, banks, replication, and number of private copies):

Figure 39. Local Memory Examples Explaining Local Memory Properties

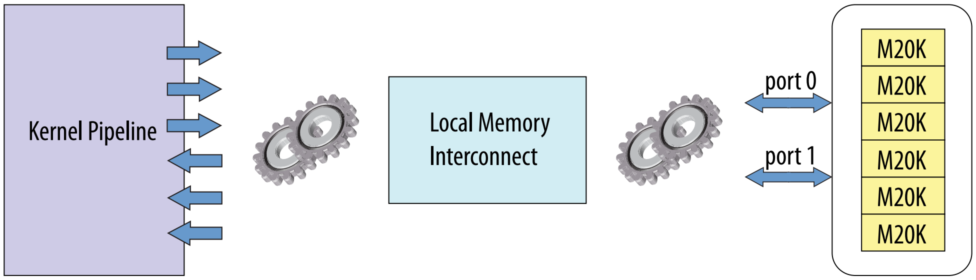


In the HTML report, the overall state of the local memory is reported as stall-free, stall-free with replication, and potentially inefficient.

The key to designing a highly efficient kernel is to have memory accesses that never stall. For a stall-free memory configuration, all possible concurrent memory access sites in the data path are guaranteed to access memory without contention.

The offline compiler always attempts to find a stall-free configuration for all local memories in your kernel code. However, in a complex kernel, the offline compiler might not have enough information to infer whether a memory access has any conflict. As a result, the offline compiler infers local interconnect arbitration to arbitrate the memory access. However, inferring arbitration might cause degradation in performance. Refer to Load-Store Units on page 85 for more information.

Figure 40. Complex Local Memory Systems



The offline compiler does not always implement local memory with the exact size that you specified. Since FPGA RAM blocks have specific dimensions, the offline compiler implements a local memory size that rounds up to the next supported RAM block dimension. Refer to device-specific information for more details on RAM blocks.

Local Memory Banks and Private Copies

Local memory banking works only on the lowest dimension by default. Having multiple banks allow simultaneous writes to take place. In the following code example, each local memory access in a loop has a separate address. The offline compiler can infer the access pattern to create four separate banks for `lmem`. Four separate banks allow four simultaneous accesses to `lmem[][]`, which achieves the stall-free configuration. In addition, the offline compiler creates two private copies for `lmem` to allow pipelined execution of two simultaneous workgroups.

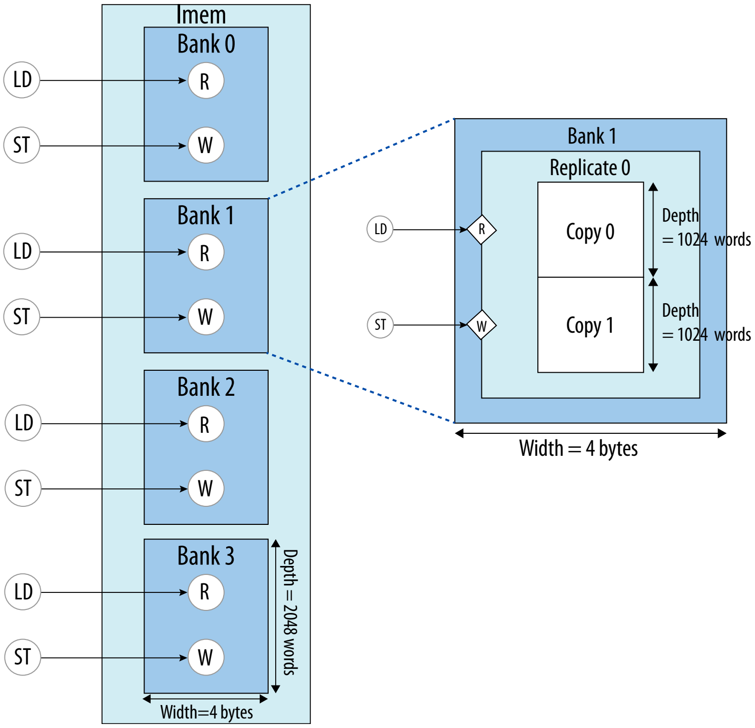
```
#define BANK_SIZE 4
__attribute__((reqd_work_group_size(8, 1, 1)))
kernel void bank_arb_consecutive_multidim (global int* restrict in,
                                           global int* restrict out) {
  local int lmem[1024][BANK_SIZE];
  int gi = get_global_id(0);
  int gs = get_global_size(0);
  int li = get_local_id(0);
  int ls = get_local_size(0);
  int res = in[gi];
  #pragma unroll
  for (int i = 0; i < BANK_SIZE; i++) {
    lmem[((li+i) & 0x7f)][i] = res + i;
    res = res >> 1;
  }
  int rdata = 0;
  barrier(CLK_GLOBAL_MEM_FENCE);
  #pragma unroll
  for (int i = 0; i < BANK_SIZE; i++) {
    rdata ^= lmem[((li+i) & 0x7f)][i];
  }
  out[gi] = rdata;
  return;
}
```

The following figure illustrates the implementation (as shown in the Kernel memory Viewer) of the following local variable:

```
local int lmem[1024][4];
```

Figure 41. Implementation of lmem[1024][4]

Local memory size = 32768 bytes = 2 private copies x (1024 x 4) x 4 bytes. The size of each bank is 8192 bytes.



If the number of private copies increase your design area significantly, consider reducing the number of barriers in the kernel or increasing the `max_work_group_size` value to reduce the inferred number of private copies.

You can specify the number of banks for your memory system by using `__attribute__((numbanks(N))`. For more information, refer to Improving Kernel Performance by Banking the Local Memory on page 154.

If you do not want to bank on the lowest dimension, use the `bank_bits` attribute to specify bits from a memory address to use as bank-select bits. By using the `bank_bits` attribute, you can separate memory data into multiple banks while specifying which address bits to use to select the bank. In the following example, the banking is done on seventh and eighth bits instead of the lowest two dimensions:

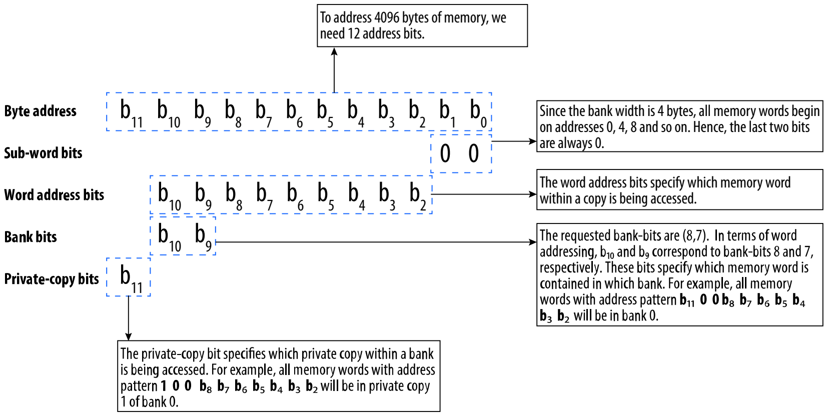
```
#define BANK_SIZE 4
kernel void bank_arb_consecutive_multidim_origin (global int* restrict in,
                                                  global int* restrict out) {
  local int a[BANK_SIZE][128] __attribute__((bank_bits(8,7),bankwidth(4)));
  int gi = get_global_id(0);
  int li = get_local_id(0);
  int res = in[gi];
  #pragma unroll
  for (int i = 0; i < BANK_SIZE; i++) {
    a[i][((li+i) & 0x7f)] = res + i;
    res = res >> 1;
  }
  int rdata = 0;
  barrier(CLK_GLOBAL_MEM_FENCE);
  #pragma unroll
  for (int i = 0; i < BANK_SIZE; i++) {
    rdata ^= a[i][((li+i) & 0x7f)];
  }
  out[gi] = rdata;
  return;
}
```

The view of the resulting memory is the same as the initial view from the first example, except that the size of the memory is now 4096 bytes = 2 private copies x (4 x 128) x 4 bytes. The Details pane of the Kernel Memory Viewer shows the address bit information, which also contains the `bank_bits` information.

The following figure illustrates the address bit information, as shown in the local memory report, for the following local variable declaration:

```
local int a[4][128] __attribute__((bank_bits(8,7),bankwidth(4)));
```

Figure 42. **Address Bit Information for `a[4][128]` with Requested `bank_bits(8,7)`**


Byte address
b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Sub-word bits
0 0
Word address bits
b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2
Bank bits
b_{10} b_9
Private-copy bits
b_{11}

To address 4096 bytes of memory, we need 12 address bits.

Since the bank width is 4 bytes, all memory words begin on addresses 0, 4, 8 and so on. Hence, the last two bits are always 0.

The word address bits specify which memory word within a copy is being accessed.

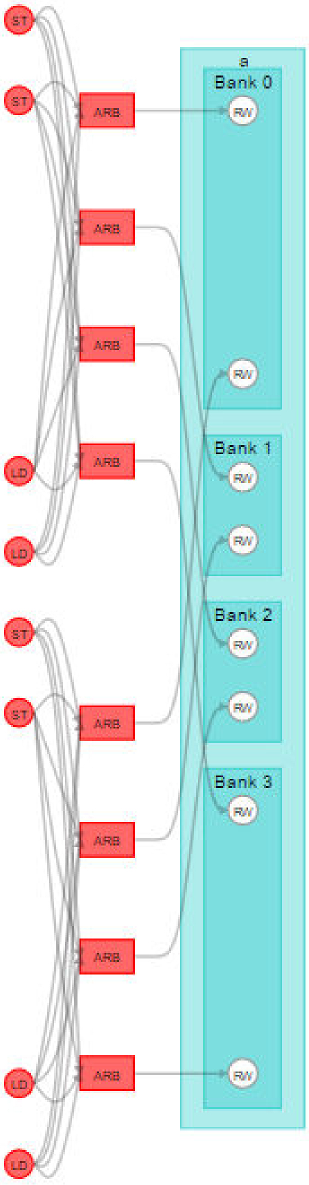
The requested bank-bits are (8,7). In terms of word addressing, b_{10} and b_9 correspond to bank-bits 8 and 7, respectively. These bits specify which memory word is contained in which bank. For example, all memory words with address pattern **b_{11} 0 0 b_8 b_7 b_6 b_5 b_4 b_3 b_2** will be in bank 0.

The private-copy bit specifies which private copy within a bank is being accessed. For example, all memory words with address pattern **1 0 0 b_8 b_7 b_6 b_5 b_4 b_3 b_2** will be in private copy 1 of bank 0.

The choice of bank-bits can alter the structure of the memory. If bank-bits (4,3) are specified in the previous example, it results in an arbitrated memory system. In this banking configuration, the local memory accesses no longer target separate banks. The compiler must build a local memory interconnect to arbitrate these accesses, which degrades performance.

```
local int a[4][128] __attribute__((bank_bits(4,3),bankwidth(4)));
```

Figure 43. Local Memory System for a[4][128] With Requested bank_bits (4,3)



Local Memory Replication

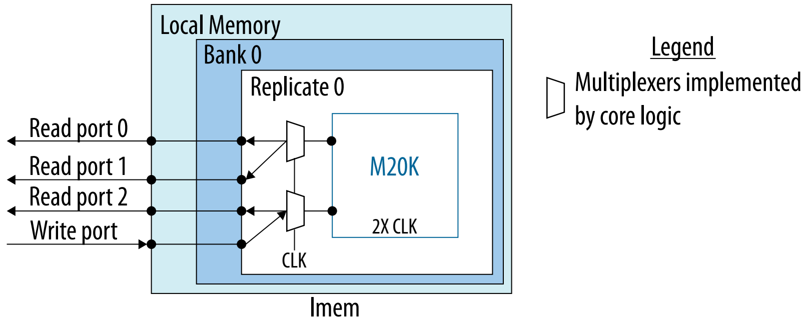
To achieve a stall-free configuration, the compiler may decide to replicate a local memory system to increase the number of available read ports. Each store operation to a local memory system is performed simultaneously on every replicate, so each replicate contains identical data. Each replicate can be independently read from. This increases the number of simultaneous read operations the local memory system can support.

Double Pumping

By default, each local memory replicate has two physical ports. The double pumping feature allows each local memory replicate to support up to four physical ports.

The underlying mechanism that enables double pumping is running the underlying M20K at double the frequency of the kernel logic. This enables two read or write operations to take place every clock cycle. From the perspective of kernel logic, a double-pumped memory has four effective physical ports.

Figure 44. Hardware Architecture of Double Pumping in Local Memory



By enabling the double pumping feature, the offline compiler trades off area versus maximum frequency.

Advantages of double pumping:

- Increases the number of available physical ports
- May reduce RAM usage by reducing replication

Disadvantages of double pumping:

- Higher logic and latency as compared to single pumped configuration
- Might reduce kernel clock frequency

You can control the pump configuration of your local memory system by using `__attribute__((singlepump))` and `__attribute__((doublepump))`. For more information, refer to Kernel Attributes for Configuring Local and Private Memory Systems.

The following code example illustrates the implementation of local memory with three read ports and three write ports. The offline compiler enables double pumping and replicates the local memory three times to implement a stall-free memory configuration.

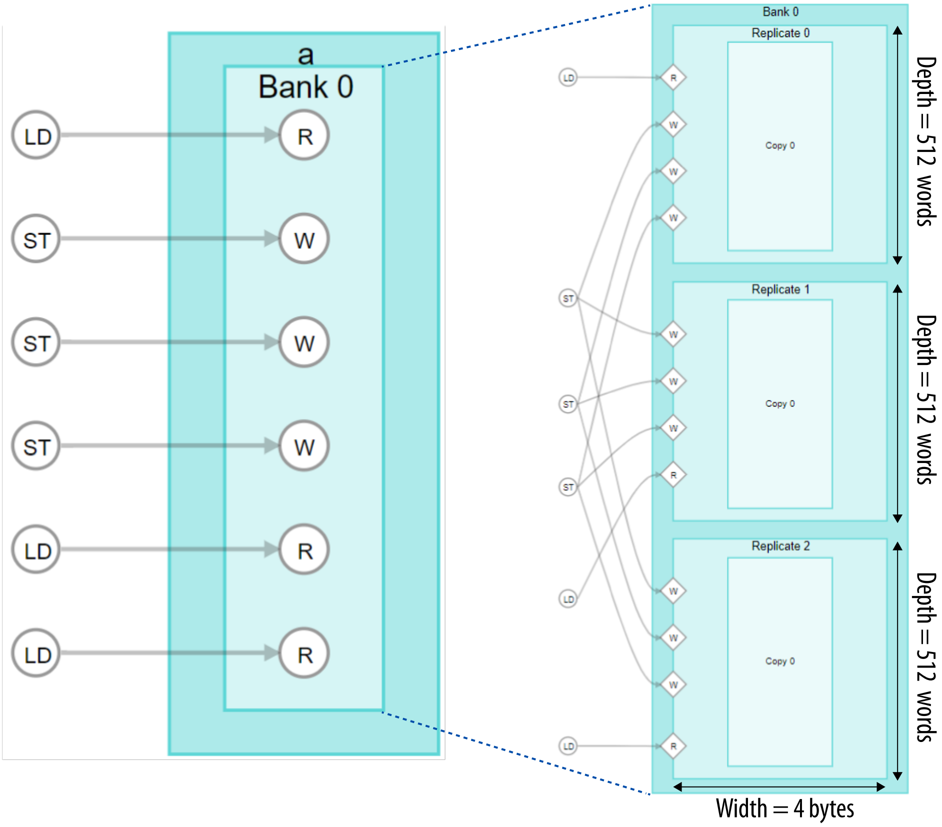
```
#define BANK_SIZE 4
kernel void bank_arb_consecutive_multidim_origin (global int* restrict in,
                                                  global int* restrict out) {
  local int a[BANK_SIZE][128];
  int gi = get_global_id(0);
  int li = get_local_id(0);
  int res = in[gi];
  #pragma unroll 1
  for (int i = 0; i < BANK_SIZE; i++) {
    a[i][li+i] = res + i;
    a[gi][li+i] = res + i;
    a[gi+i][li] = res + i;
    res = res >> 1;
  }
  int rdata = 0;
  barrier(CLK_GLOBAL_MEM_FENCE);
  #pragma unroll 1
  for (int i = 0; i < BANK_SIZE; i++) {
    rdata ^= a[i][li+i];
    rdata += a[gi+i][li+i];
    rdata += a[gi][li];
  }
  out[gi] = rdata;
  return;
}
```

The following figure illustrates the implementation (as shown in the Kernel Memory Viewer) for the following local variable declaration:

```
local int a[4][128];
```

Figure 45. Local Memory System for a[4][128]

Local memory size = 6144 bytes = 3 replicates x 512 words x 4 bytes. Each replicate has identical memory contents.



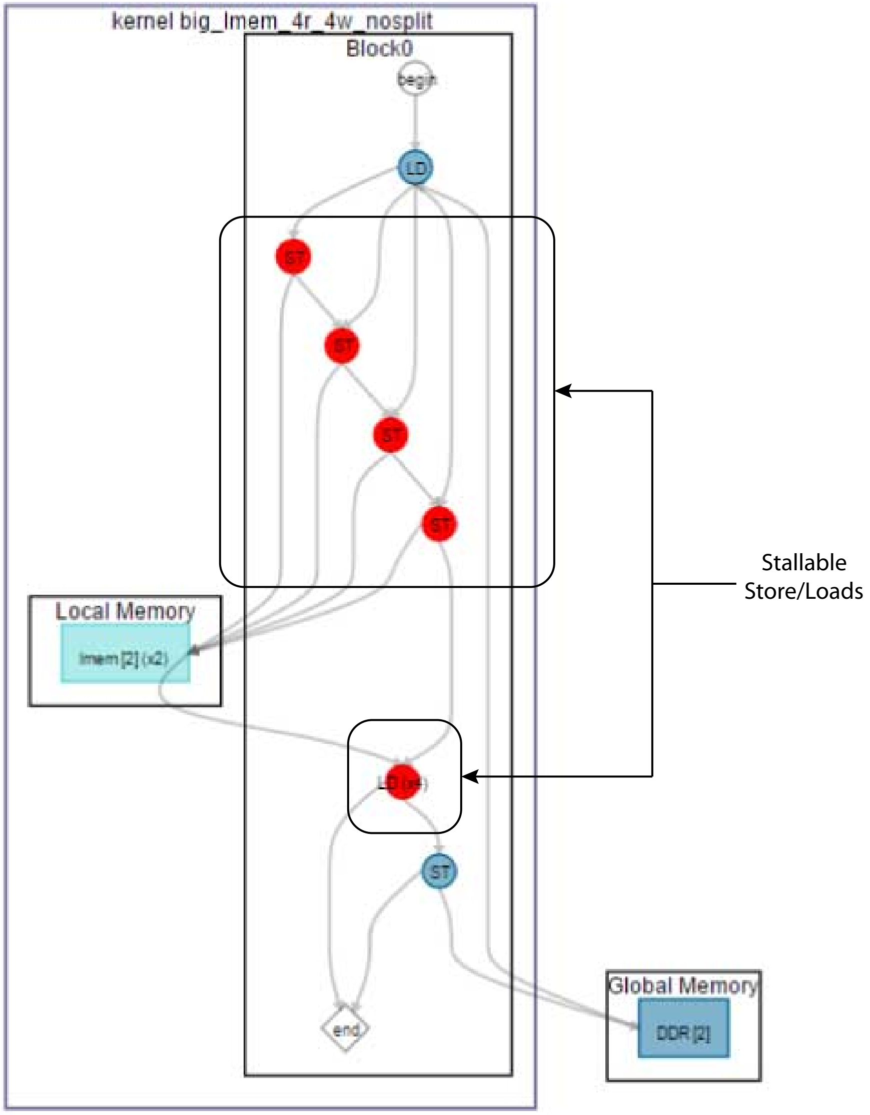
3.3.1. Changing the Memory Access Pattern Example

The following is an example code of a simple OpenCL kernel:

```
kernel void big_lmem_4r_4w_nosplit (global int* restrict in,
                                    global int* restrict out) {
    local int lmem[4][1024];
    int gi = get_global_id(0);
    int gs = get_global_size(0);
    int li = get_local_id(0);
    int ls = get_local_size(0);
    int res = in[gi];

    #pragma unroll
    for (int i = 0; i < 4; i++) {
         lmem[i][(li*i) % ls] = res;
         res >>= 1;  }

    // Global memory barrier
    barrier(CLK_GLOBAL_MEM_FENCE);

    res = 0;
    #pragma unroll
    for (int i = 0; i < 4; i++) {
        res ^= lmem[i][((ls-li)*i) % ls];  }
    out[gi] = res;
}
```

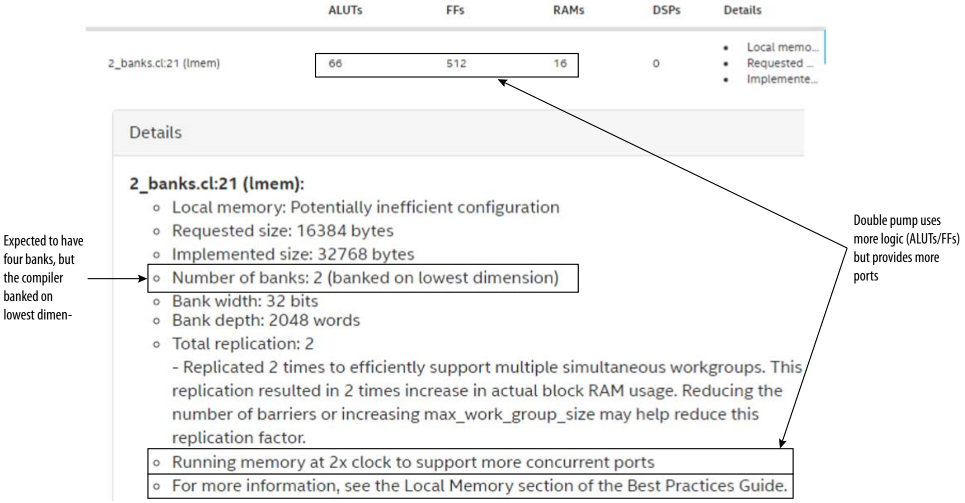
In the System Viewer report, the system view of this example highlights the stallable loads and stores.

Figure 46. System View of the Example



Stallable Store/Loads

Figure 47. Area Report of the Example



	ALUTs	FFs	RAMs	DSPs	Details
2_banks.cl:21 (lmem)	66	512	16	0	• Local memo... • Requested ... • Implemente...

Details

2_banks.cl:21 (lmem):

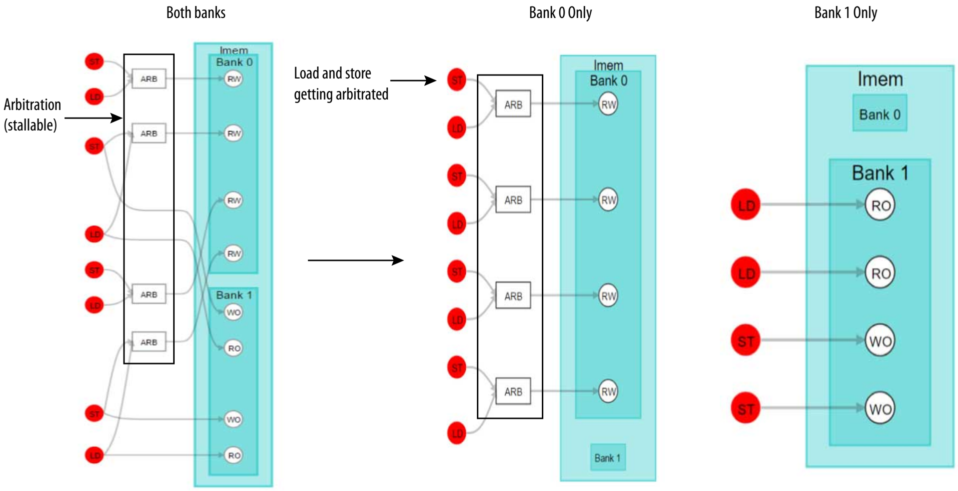
- Local memory: Potentially inefficient configuration
- Requested size: 16384 bytes
- Implemented size: 32768 bytes
- Number of banks: 2 (banked on lowest dimension)
- Bank width: 32 bits
- Bank depth: 2048 words
- Total replication: 2

  - Replicated 2 times to efficiently support multiple simultaneous workgroups. This replication resulted in 2 times increase in actual block RAM usage. Reducing the number of barriers or increasing max_work_group_size may help reduce this replication factor.
- Running memory at 2x clock to support more concurrent ports
- For more information, see the Local Memory section of the Best Practices Guide.

Expected to have four banks, but the compiler banked on lowest dimen-

Double pump uses more logic (ALUTs/FFs) but provides more ports

Figure 48. Kernel Memory Viewer of the Example



Observe that only two memory banks are created, with high arbitration on the first bank between load and store operations. Now, switch the banking indices to the second dimension, as shown in the following example code, :

```
kernel void big_lmem_4r_4w_nosplit (global int* restrict in,
                                    global int* restrict out) {
  local int lmem[1024][4];

  int gi = get_global_id(0);
  int gs = get_global_size(0);
  int li = get_local_id(0);
  int ls = get_local_size(0);
  int res = in[gi];

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    lmem[(li*i) % ls][i] = res;
    res >>= 1;
  }

  // Global memory barrier
  barrier(CLK_GLOBAL_MEM_FENCE);

  res = 0;
  #pragma unroll
  for (int i = 0; i < 4; i++) {
    res ^= lmem[((ls-li)*i) % ls][i];
  }
  out[gi] = res;
}
```

In the kernel memory viewer, you can observe that now four memory banks are created, with separate load store units. All load store instructions are stall-free.

Figure 49. Kernel Memory Viewer of the Example After Changing the Banking Indices

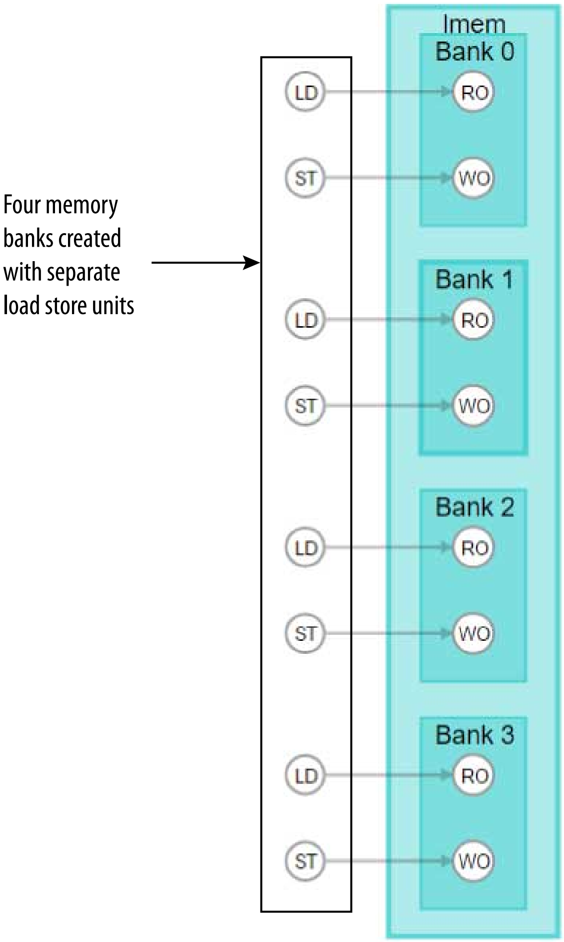
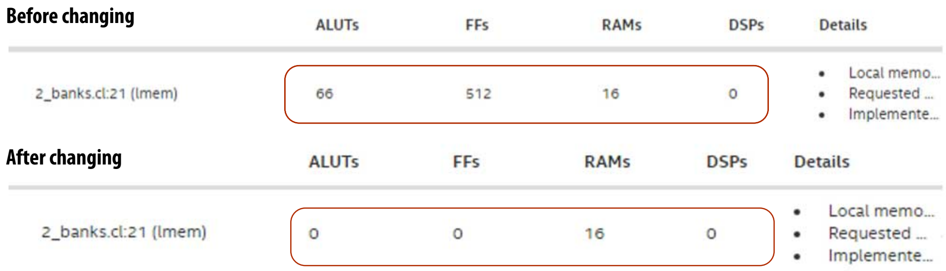


Figure 50. Area Report of the Example After Changing the Banking Indices



Before changing

	ALUTs	FFs	RAMs	DSPs	Details
2_banks.cl:21 (lmem)	66	512	16	0	• Local memo... • Requested ... • Implemente...

After changing

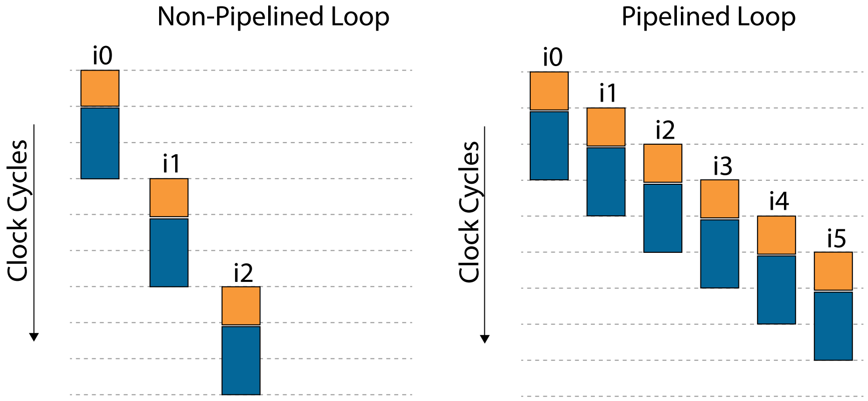
	ALUTs	FFs	RAMs	DSPs	Details
2_banks.cl:21 (lmem)	0	0	16	0	• Local memo... • Requested ... • Implemente...

3.4. Loops in a Single Work-Item Kernel

The Intel FPGA SDK for OpenCL Offline Compiler optimizes performance of single work-item kernels by pipelining loops.

The datapath of a loop within a single work-item kernel can contain multiple iterations in flight. This behavior is different from a loop within an NDRange kernel in that an NDRange kernel's loop contains multiple work-items (rather than loop iterations) in flight. In an optimally pipelined loop, a new loop iteration is launched every clock cycle. Launching one loop iteration per clock cycle maximizes pipeline efficiency and yields the best performance. As shown in the figure below, launching one loop per clock cycle allows a kernel to finish faster.

Figure 51. Comparison of the Launch Frequency of Loop Iterations Between a Non-Pipelined Loop and a Pipelined Loop



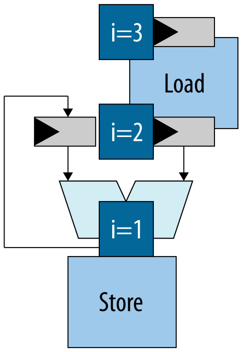
The number of clock cycles between the launch of one loop iteration and the next is called the loop's initiation interval (II). An optimally pipelined loop has an II value of 1 because a new loop iteration is launched every clock cycle.

The Intel FPGA SDK for OpenCL Offline Compiler may not pipeline every loop in the kernel. If a loop is not pipelined, a loop iteration can not begin until the previous iteration finishes executing. In this case, only one loop iteration is active in the loop's datapath at a time. View the HTML report to find out which loops are pipelined, and for pipelined loops, what is their II.

Consider the following example:

```
kernel void simple_loop (unsigned N,
                         global unsigned* restrict b,
                         global unsigned* restrict c,
                         global unsigned* restrict out)
{
    for (unsigned i = 1; i < N; i++) {
        c[i] = c[i-1] + b[i];
    }
    out[0] = c[N-1];
}
```

Figure 52. Hardware Datapath of the Kernel `simple_loop`



The figure depicts how the offline compiler uses loop pipelining to execute `simple_loop` efficiently. The figure shows that the loop's datapath contains three loop iterations at the same time. Therefore, this loop is pipelined. The figure also shows that a new loop iteration enters the datapath every clock cycle. Therefore, the loop has II=1.

Related Information

Single Work-Item Kernel versus NDRange Kernel on page 9

3.4.1. Trade-Off Between Initiation Interval and Maximum Frequency

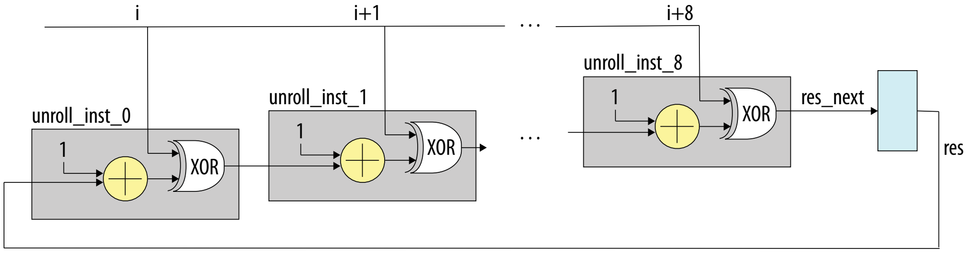
The offline compiler attempts to achieve an II value of 1 for a given loop whenever possible. In some cases, the offline compiler might strive for an II of 1 at the expense of a reduced f_{MAX}.

Consider the following example:

```
kernel void lowered_fmax (global int *dst, int N) {
     int res = N;
     #pragma unroll  9
     for (int i = 0; i < N; i++) {
         res += 1;
         res ^= i;
     }
     dst[0] = res;
 }
```

The following figure shows the datapath of the loop in kernel lowered_fmax. The loop is partially unrolled by a factor of 9, so the datapath contains nine copies of the original loop's body. To save space, only three of these copies are depicted in the following figure:

Figure 53. Datapath of the Partially Unrolled Loop in Kernel lowered_fmax



The loop in kernel lowered_fmax has a loop-carried dependence involving the res variable. This loop carried dependence forms a cycle in the loop's datapath, as shown in Datapath of the Partially Unrolled Loop in Kernel lowered_fmax.

Note: The value of res from one iteration must be available when the next iteration is launched. Therefore, if the loop is to achieve II=1, this cycle must contain at most one register. This cycle contains a chain of nine additions and XORs, so f_{MAX} must be lowered in order for this chain of operations to complete within one clock cycle. The offline compiler may lower the kernel's f_{MAX} to achieve II=1, since II is an important factor to achieving good performance. Consult the HTML report to find loops whose loop carried dependencies limit f_{MAX}.

3.4.2. Loop-Carried Dependencies that Affect the Initiation Interval of a Loop

There are cases where a loop is pipelined but it does not achieve an II value of 1. These cases are usually caused by data dependencies or memory dependencies within a loop.

Data Dependencies

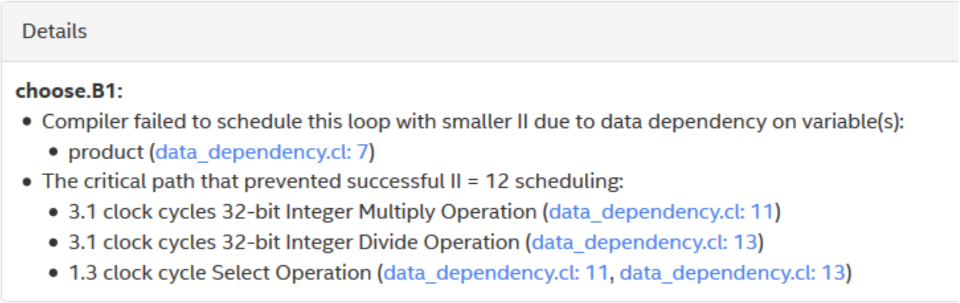
Data dependency refers to a situation where a loop iteration uses variables that rely on the previous iteration. In this case, a loop can be pipelined, but its II value may be greater than 1. Consider the following example:

```
// An example that shows data dependency
// choose(n, k) = n! / (k! * (n-k)!)

kernel void choose( unsigned n, unsigned k,
                    global unsigned* restrict result )
{
    unsigned product = 1;
    unsigned j = 1;

    for( unsigned i = k; i <= n; i++ ) {
        product *= i;
        if( j <= n-k ) {
            product /= j;
        }
        j++;
    }
17
18      *result = product;
19  }
```

For every loop iteration, the value for the `product` variable in the kernel `choose` is calculated by multiplying the current value of index `i` by the value of `product` from the previous iteration. As a result, a new iteration of the loop cannot launch until the current iteration finishes processing.

The loop in kernel `choose` has an II value of 12. This information can be found in the Loop Analysis report. In addition, the details pane in the following figure shows that the high II value is caused by a data dependency on `product`, and the largest contributor to the critical path is the integer division operation on line 13.

Figure 56. Details Pane of the Loop Analysis Report for the Kernel `choose`



Details

choose.B1:

- Compiler failed to schedule this loop with smaller II due to data dependency on variable(s):
  - product (data_dependency.cl: 7)
- The critical path that prevented successful II = 12 scheduling:
  - 3.1 clock cycles 32-bit Integer Multiply Operation (data_dependency.cl: 11)
  - 3.1 clock cycles 32-bit Integer Divide Operation (data_dependency.cl: 13)
  - 1.3 clock cycle Select Operation (data_dependency.cl: 11, data_dependency.cl: 13)

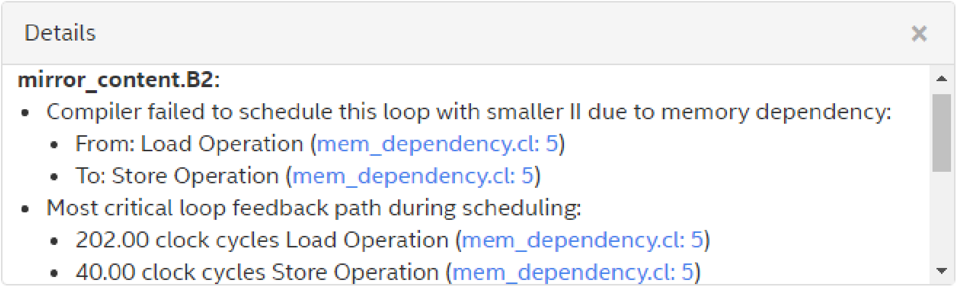
Memory Dependency

Memory dependency refers to a situation where memory access in a loop iteration cannot proceed until memory access from the previous loop iteration is completed. Consider the following example:

```
1  kernel void mirror_content( unsigned max_i,
2                              global int* restrict out)
3  {
4    for (int i = 1; i < max_i; i++) {
5      out[max_i*2-i] = out[i];
6    }
7  }
```

In the loop analysis report, the details pane shows that the memory dependency is between two load and store operations on line 5, and that the load operation takes 202 clock cycles.

Figure 58. Details Pane of the Loop Analysis for the Kernel `mirror_content`



Details

mirror_content.B2:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation (mem_dependency.cl: 5)
  - To: Store Operation (mem_dependency.cl: 5)
- Most critical loop feedback path during scheduling:
  - 202.00 clock cycles Load Operation (mem_dependency.cl: 5)
  - 40.00 clock cycles Store Operation (mem_dependency.cl: 5)

3.4.3. Nested Loops

The Intel FPGA SDK for OpenCL Offline Compiler does not infer pipelined execution because of the ordering of loop iterations. As a result, outer loop iterations might be out of order with respect to the ensuing inner loops because the number of iterations of the inner loops might differ for different out loop iterations.

To solve the problem of out-of-order outer loop iterations, design inner loops with lower and upper bounds that do not change between outer loop iterations.

Single Work-Item Execution

To ensure high-throughput single work-item-based kernel execution on the FPGA, the Intel FPGA SDK for OpenCL Offline Compiler must process multiple pipeline stages in parallel at any given time. This parallelism is realized by pipelining the iterations of loops.

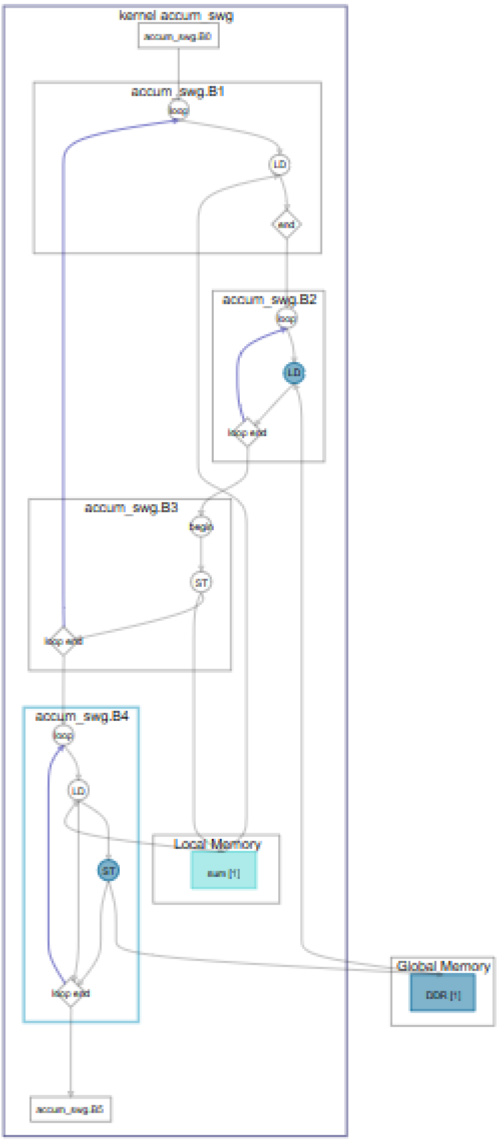
Consider the following simple example code that shows accumulating with a single-work item:

```
1 kernel void accum_swg (global int* a,
                         global int* c,
                         int size,
                         int k_size) {
2     int sum[1024];
3     for (int k = 0; k < k_size; ++k) {
4        for (int i = 0; i < size; ++i) {
5            int j = k * size + i;
6            sum[k] += a[j];
7        }
8     }
9     for (int k = 0; k < k_size; ++k) {
10       c[k] = sum[k];
11    }
12 }
```

During each loop iteration, data values from the global memory a is accumulated to `sum[k]`. In this example, the inner loop on line 4 has an initiation interval value of 1 with a latency of 11. The outer loop also has an initiation interval value greater than or equal to 1 with a latency of 8.

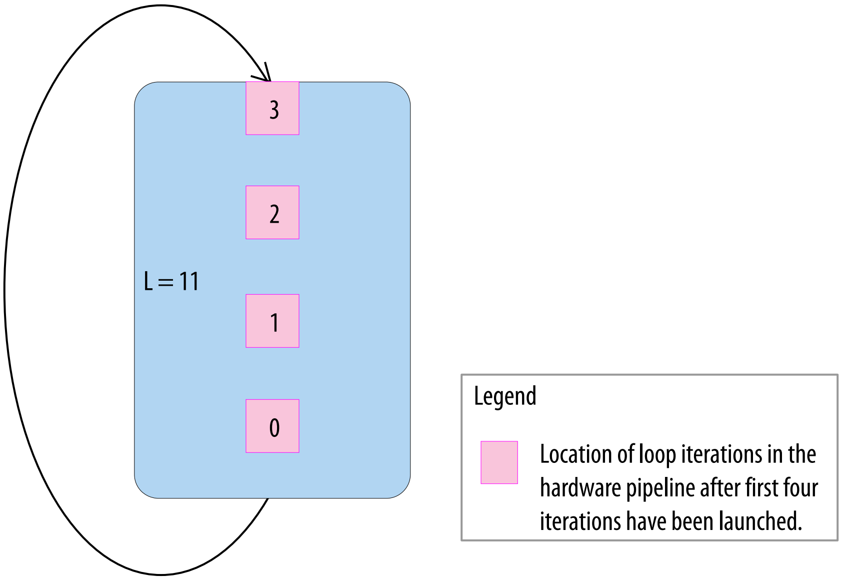
Note: The launch frequency of a new loop iteration is called the initiation interval (II). II refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. An optimally unrolled loop has an II value of 1 because one loop iteration is processed every clock cycle.

Figure 60. System View of Single-Work Item Kernel



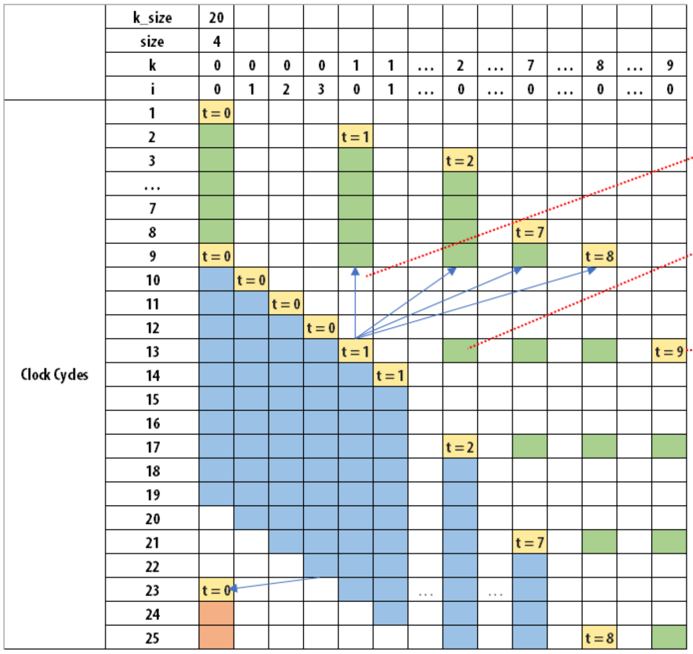
The following figure illustrates how each iteration of i enters into the block:

Figure 61. Inner Loop `accum_swg.B2` Execution


3
2
L = 11
1
0
Legend
Location of loop iterations in the hardware pipeline after first four iterations have been launched.

When you observe the outer loop, having an II value of 1 also means that each iteration of the thread can enter at every clock cycle. In the example, `k_size` of 20 and `size` of 4 is considered. This is true for the first eight clock cycles as outer loop iterations 0 to 7 can enter without any downstream stalling it. Once thread 0 enters into the inner loop, it takes four iterations to finish. Threads 1 to 8 cannot enter into the inner loop and they are stalled for four cycles by thread 0. Thread 1 enters into the inner loop after thread 0's iterations are completed. As a result, thread 9 enters into the outer loop on clock cycle 13. Threads 9 to 20 enters into the loop at every four clock cycles, which is the value of `size`. Through this example, you can observe that the dynamic initiation interval of the outer loop is greater than the statically predicted initiation interval of 1 and it is a function of the trip count of the inner loop.

Figure 62. Single Work-Item Execution



Stalled until the inner loop has finished its computations.

Thread 2 can continue to pipeline through outer loop and stall in the next cycle until thread 1 is done.

Thread 9 enters into outer loop on cycle 13.

Nonlinear Execution

Loop structure does not support linear execution. The following code example shows that the outer loop i contains two divergent inner loops. Each iteration of the outer loop may execute one inner loop or the other, which is a nonlinear execution pattern.

```
__kernel void structure (__global unsigned* restrict output1,
                         __global unsigned* restrict output2,
                         int N) {
  for (unsigned i = 0; i < N; i++) {
    if ((i & 3) == 0) {
      for (unsigned j = 0; j < N; j++) {
        output1[i+j] = i * j;
      }
    }
    else
    {
      for (unsigned j = 0; j < N; j++) {
        output2[i+j] = i * j;
      }
    }
  }
}
```

Serial Regions

Serial region might occur in nested loops when an inner loop access causes an outer loop dependency. The inner loop becomes a serial region in the outer loop iteration due to data or memory dependencies.

At steady state, the II of outer loop = II of inner loop * trip count of inner loop. For inner loops with II greater than 1 and outer loop that has no serially executed regions, it is possible to interleave threads from the outer loop.

Consider the following code example:

```
kernel void serially_execute (global int * restrict A,
                              global int * restrict B,
                              global int * restrict result,
                              unsigned N) {
  int sum = 0;
  for (unsigned i = 0; i < N; i++) {
    int res;
    for (int j = 0; j < N; j++) {
      sum += A[i*N+j];
    }
    sum += B[i];
  }
  *result = sum;
}
```

In the example, the dependence in the outer loop resulted in the serial execution of the inner loop. The main difference in performance is the steady state II of outer loop = II of inner loop * (trip count of inner loop - 1) + latency. In this example, II of inner loop is 1 with latency of 4 and II of outer loop is 1 with latency of 7. If N is large, such as 400, when compared to latency, then serial execution has little impact from the outer loop II.

Figure 63. System View of the Kernel

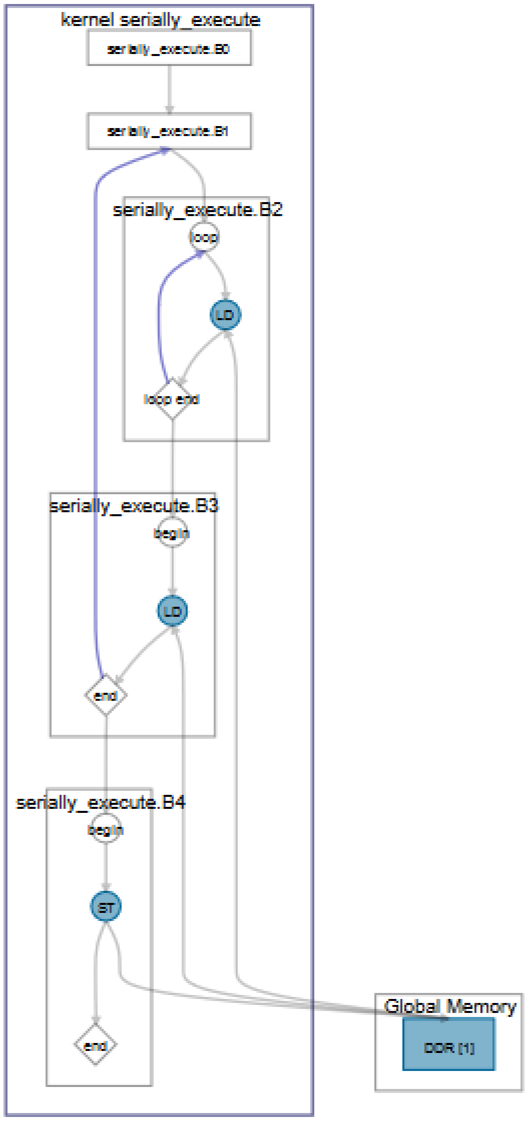
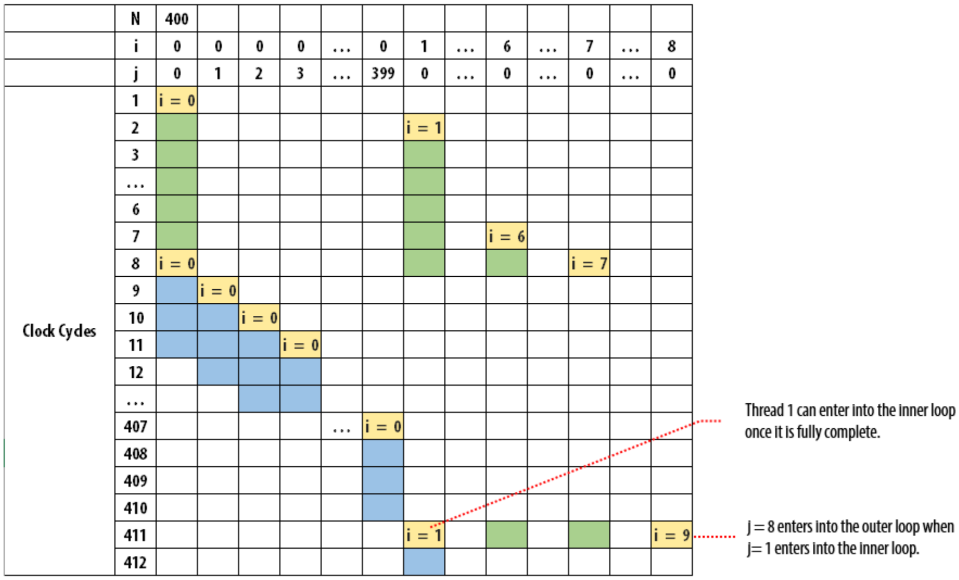


Figure 64. Serial Execution


N
400
i
j
Clock Cycles
i = 0
i = 1
i = 6
i = 7
i = 9
Thread 1 can enter into the inner loop once it is fully complete.
J = 8 enters into the outer loop when j= 1 enters into the inner loop.

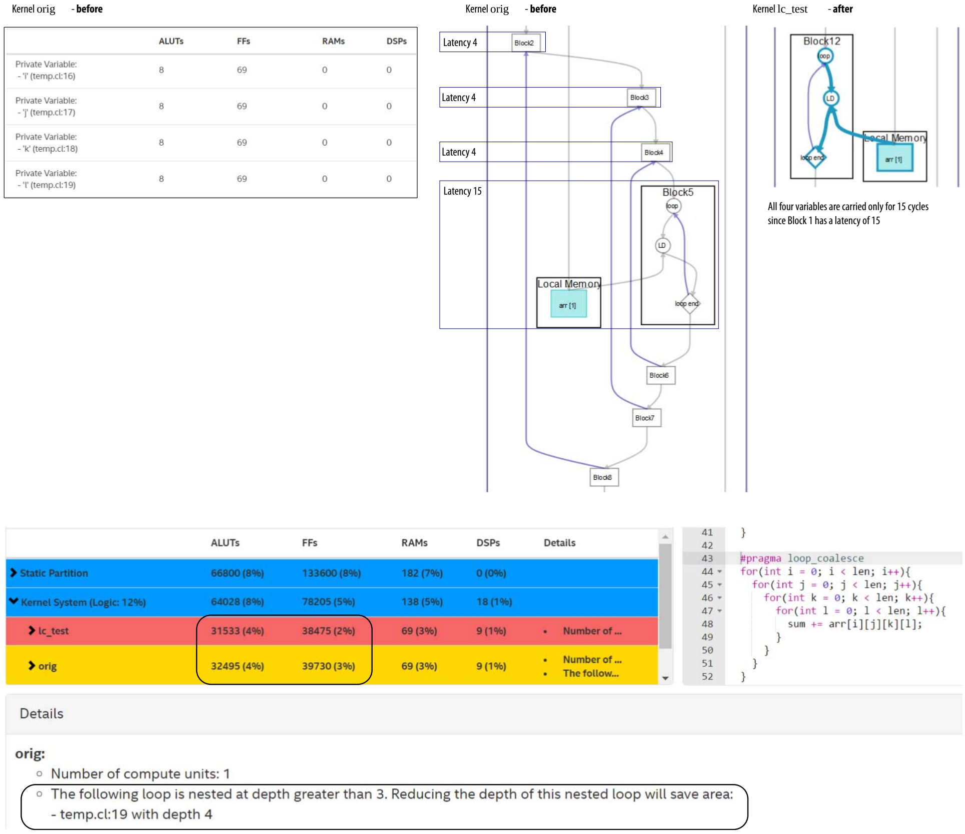
3.4.3.1. Reducing the Area Consumed by Nested Loops Using `loop_coalesce`

When loops are nested to a depth greater than three, more area is consumed.

Consider the following example where `orig` and `lc_test` kernels are used to illustrate how to reduce latency in nested loops.

The orig kernel has nested loops to a depth of four. The nested loops created extra blocks (Block 2, 3, 4, 6, 7 and 8) that consume area due to the variables being carried, as shown in the following reports:

Figure 65. Area Report and System Viewer (System View) Before and After Loop Coalescing



All four variables are carried only for 15 cycles since Block 1 has a latency of 15

	ALUTs	FFs	RAMs	DSPs
Private Variable: - 'i' (temp.cl:16)	8	69	0	0
Private Variable: - 'j' (temp.cl:17)	8	69	0	0
Private Variable: - 'k' (temp.cl:18)	8	69	0	0
Private Variable: - 'l' (temp.cl:19)	8	69	0	0

	ALUTs	FFs	RAMs	DSPs	Details
Static Partition	66800 (8%)	133600 (8%)	182 (7%)	0 (0%)	
Kernel System (Logic: 12%)	64028 (8%)	78205 (5%)	138 (5%)	18 (1%)	
lc_test	31533 (4%)	38475 (2%)	69 (3%)	9 (1%)	• Number of ...
orig	32495 (4%)	39730 (3%)	69 (3%)	9 (1%)	• Number of ... • The follow...

```
}

#pragma loop_coalesce
for(int i = 0; i < len; i++){
  for(int j = 0; j < len; j++){
    for(int k = 0; k < len; k++){
      for(int l = 0; l < len; l++){
        sum += arr[i][j][k][l];
      }
    }
  }
}
```

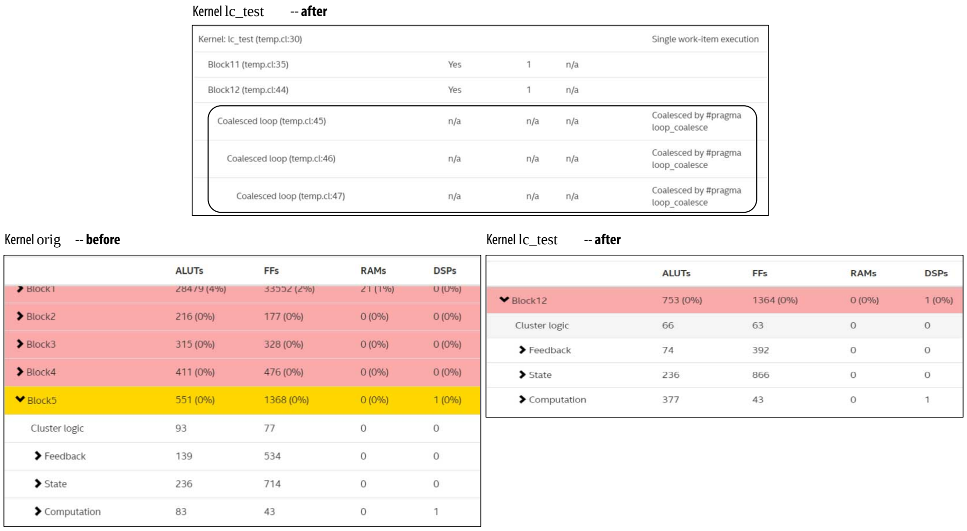
Details

orig:

- Number of compute units: 1
- The following loop is nested at depth greater than 3. Reducing the depth of this nested loop will save area:
  - temp.cl:19 with depth 4

Due to loop coalescing, you can see the reduced latency in the lc_test. The Block 5 of orig kernel and Block 12 of lc_test kernel are the inner most loops.

Figure 66. Area Report of `lc_test` and `orig` Kernels



Kernel lc_test -- **after**

Kernel: lc_test (temp.cl:30)				Single work-item execution
Block11 (temp.cl:35)	Yes	1	n/a	
Block12 (temp.cl:44)	Yes	1	n/a	
Coalesced loop (temp.cl:45)	n/a	n/a	n/a	Coalesced by #pragma loop_coalesce
Coalesced loop (temp.cl:46)	n/a	n/a	n/a	Coalesced by #pragma loop_coalesce
Coalesced loop (temp.cl:47)	n/a	n/a	n/a	Coalesced by #pragma loop_coalesce

Kernel orig -- **before**

	ALUTs	FFs	RAMs	DSPs
Block1	28479 (4%)	33552 (2%)	21 (1%)	0 (0%)
Block2	216 (0%)	177 (0%)	0 (0%)	0 (0%)
Block3	315 (0%)	328 (0%)	0 (0%)	0 (0%)
Block4	411 (0%)	476 (0%)	0 (0%)	0 (0%)
Block5	551 (0%)	1368 (0%)	0 (0%)	1 (0%)
Cluster logic	93	77	0	0
Feedback	139	534	0	0
State	236	714	0	0
Computation	83	43	0	1

Kernel lc_test -- **after**

	ALUTs	FFs	RAMs	DSPs
Block12	753 (0%)	1364 (0%)	0 (0%)	1 (0%)
Cluster logic	66	63	0	0
Feedback	74	392	0	0
State	236	866	0	0
Computation	377	43	0	1

Related Information

Coalescing Nested Loops

3.4.4. Loop Speculation

Loop speculation is an optimization technique that enables more efficient loop pipelining by allowing future iterations to be initiated before determining whether the loop was exited already. Consider the following simple loop example:

```
while (m*m*m < N) {
    m += 1;
}
```

Logically, the exit condition (`m*m*m < N`) for an iteration must be evaluated before determining whether you need to initiate another iteration or not. This means that, in the absence of speculation, the loop II cannot be lower than the number of cycles it takes to compute this exit condition. Speculated iterations are iterations that launch before the exit condition computation has completed. However, all operations with side-effects, such as stores to memory, are predicated by the exit condition. This means that operations with side-effects still waits for the exit condition to be computed. Loop speculation is beneficial when the exit condition is the bottleneck preventing from achieving a lower II. In the loop shown above, the exit condition contains two multiplications that cannot complete within a single clock cycle. However, loop speculation allows this loop to achieve II=1.

For example, for a given iteration `i` with exit condition `Ei`, the number of speculated iterations `s` is the number of iterations after `i` has been initiated but before `Ei` has been evaluated. By default, this number of speculated iterations is determined by the compiler on a per-loop basis, and can be found in the per-loop details of the Loop Analysis report.

The `#pragma speculated_iterations` pragma allows you to directly control the number of speculated iterations for a loop. If the exit condition calculation is the bottleneck to lowering II (as shown in the Loop Analysis report), increasing the number of speculated iterations may improve the II (this is not guaranteed as other bottlenecks may be uncovered). For details about `#pragma speculated_iterations`, refer to Loop Speculation in the *Intel FPGA SDK for OpenCL Programming Guide*.

Speculated iterations introduce some overhead in nested loops, since a new invocation of a loop may not begin until all speculated iterations of its previous invocation have completed. In cases where a loop body with low latency is expected to be frequently invoked, (for example, an inner loop with a short trip count), use the `#pragma speculated_iterations` pragma to reduce the number of speculated iterations. You can estimate the amount of this overhead by multiplying the number of speculated iterations with the II of the loop (as shown in the Loop Analysis report). Using the `#pragma speculated_iterations` pragma can reduce this overhead, but be aware that choosing a pragma value that is too low may increase the II (due to not having enough time to evaluate the exit condition).

Consider the following example:

```
kernel void unopt_int_cube_root (global int *dst, int N) {
   int m = 0;
   while (m*m*m < N) {
       m += 1;
   }
   dst[0] = m;
}

kernel void opt_int_cube_root (global int *dst, int N) {
   int m = 0;
   #pragma speculated_iterations 7
   while (m*m*m < N) {
        m += 1;
   }
   dst[0] = m;
}

kernel void unopt2_int_cube_root (global int *dst, int N) {
   int m = 0;
   #pragma speculated_iterations 0
   while (m*m*m < N) {
       m += 1;
   }
   dst[0] = m;
}
```

In this example, the exit condition that has two multiplies and a compare is the bottleneck preventing II=1. The compiler's choice of four speculated iterations result in II=2 since the exit condition takes seven cycles (each multiply takes three cycles and the compare takes one cycle) and four speculated iterations times two-cycle II gives eight cycles to cover this evaluation. Then, the speculated iterations are increased to seven to cover the seven-cycle exit condition calculation allows us to achieve II=1. By setting the `speculated_iterations` pragma to 0, you can verify that the II has increased to 7, which matches the exit condition bottleneck.

Related Information

Loop Speculation

3.4.5. Loop Fusion

Loop fusion is a compiler transformation in which two adjacent loops are merged into a single loop over the same index range. This transformation is typically applied to reduce loop overhead and improve run-time performance.

The following example shows the effects of fusing loops in a simple case:

Unfused Loops

```
for (i = 0; i < 300; i++)
    a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
    b[i] = b[i] + 4;
```

Fused Loops

```
for (i = 0; i < 300; i++) {
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
}
```

Loop control structures represent a significant overhead. By fusing two loops, the number of control structures needed for the loops is reduced from two to one, reducing this overhead. The main goal of reducing the number of control structures is to save FPGA area for your design while still maintaining (ideally increasing) component throughput.

Fusing outer loops introduces concurrency where there was previously none. Combining bodies of two adjacent loops (L_j and L_k) forms a single loop (L_f) with a loop body that spans the bodies of L_j and L_k. This combined loop body creates an opportunity for operations that are serialized across a given iteration of L_j and L_k to execute concurrently. In effect, the two loops now execute as one, reducing latency.

If inner loops are fused, concurrency is already achieved by pipelined execution of the outer loop iteration. In these cases, the concurrency effect of loop fusion is diminished.

Fusion Criteria

The compiler considers the fusion of two loops (L_j and L_k) to be valid if the loops meet the following criteria:

- The loops must be adjacent.

  That is, you cannot have a statement S_i with side-effects such that S_i executes after L_j and before L_k.
- Each loop must have a single-entry point and a single exit point. For example, loops that contain `break` statements are not considered for fusion.
- The loops must have no negative-distance dependencies.

  That is, for loops L_j and L_k where L_j is defined before L_k, iteration *m* of loop L_k does not depend on values calculated in iteration $m+n$ (where $n>0$) of loop L_j.

Automatic Loop Fusion

The Intel FPGA SDK for OpenCL Offline Compiler fuses loops with the same trip counts automatically if the compiler analysis of your component determines that fusing the loops is profitable.

Examples of where fusing loops is a valid transformation (based on the earlier criteria) but are not considered profitable by the compiler include the following situations:

- One of the two loops, but not both, is annotated with the `ivdep` pragma.
- One of the two loops, but not both, contains stall-free logic.

The Loop Analysis Report in the High-Level Design Reports indicates when loops are fused.

In addition to automatic loop fusion, the Intel FPGA SDK for OpenCL Offline Compiler provides two pragmas to help you control when loops are fused:

- `loop_fuse` pragma

  Override the compiler profitability analysis and fuse adjacent loops if it is safe.

- `nofusion` pragma

  Annotate loops with this pragma to request that the compiler not fuse the annotated loop.

3.4.6. Loop Bottlenecks

Bottlenecks in a loop means one or more loop carried dependencies caused the compiler to reduce the number of data items to be processed concurrently (in the same clock cycle) or f_{MAX} is reduced. Bottlenecks occur only on single work-item kernels and are always created for loops.

Before analyzing the throughput of a simple loop, it is important to understand the concept of dynamic initiation interval. The initiation interval (II) is the statically determined number of cycles between successive iteration launches of a given loop invocation. However, the statically scheduled II may differ from the actual realized dynamic II when considering interleaving.

Note: Interleaving allows the iterations of more than one invocation of a loop to execute in parallel, provided that the static II of that loop is greater than 1. By default, the maximum amount of interleaving for a loop is equal to the static II of that loop.

In the presence of interleaving, the dynamic II of a loop can be approximated by the static II of the loop divided by the degree of interleaving, that is, by the number of concurrent invocations of the loop that are in flight.

Simple Loop Example

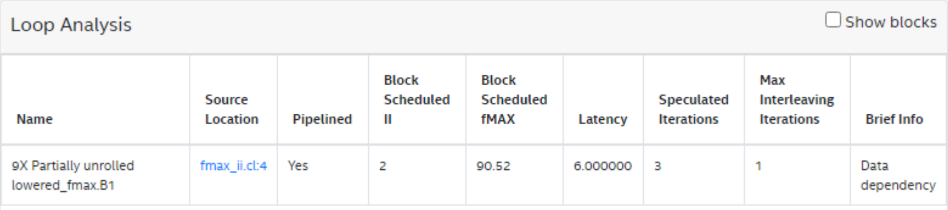
In a simple loop, the maximum number of data items to be processed concurrently (also known as maximum concurrency) can be expressed as:

Concurrency_{MAX} = (Block latency × Maximum interleaving iterations) / Initiation Interval

Consider the following simple loop:

```
kernel void lowered_fmax (global int *dst, int N) {
  int res = N;
  #pragma unroll  9
  for (int i = 0; i < N; i++) {
    res += 1;
    res ^= i;
  }
  dst[0] = res;
}
```

The Loop Analysis report displays the following for the simple loop:

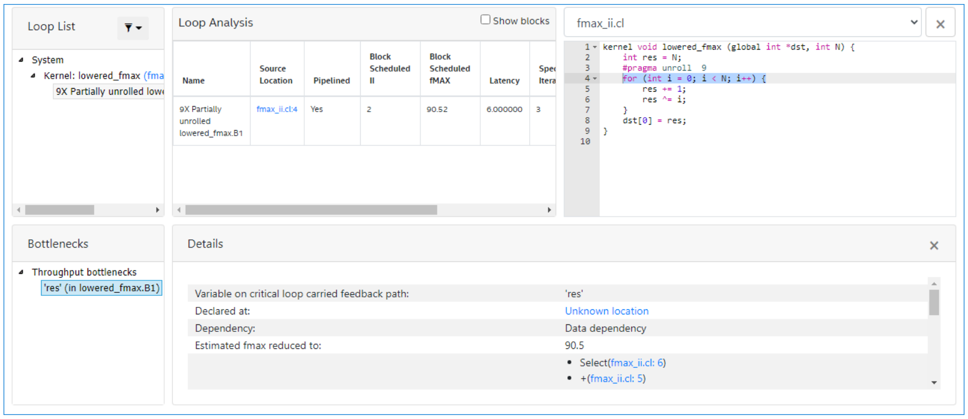


Loop Analysis

□ Show blocks

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
9X Partially unrolled lowered_fmax.B1	fmax_ii.cl:4	Yes	2	90.52	6.000000	3	1	Data dependency

The for loop in line:4 has a latency of 6, maximum interleaving iterations of 1, and initiation interval of 2. So, the maximum concurrency is 3 (latency of 6 / II of 2). The bottleneck results from loop carried dependency caused by a data dependency on the variable res. This is reported in the Bottlenecks viewer as shown in the following image:



Loop List
System
Kernel: lowered_fmax (fma
9X Partially unrolled lowe

Loop Analysis

□ Show blocks

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Spec Itera
9X Partially unrolled lowered_fmax.B1	fmax_ii.cl:4	Yes	2	90.52	6.000000	3

fmax_ii.cl

```
1   kernel void lowered_fmax (global int *dst, int N) {
2       int res = N;
3       #pragma unroll  9
4       for (int i = 0; i < N; i++) {
5           res += 1;
6           res ^= i;
7       }
8       dst[0] = res;
9   }
10
```

Bottlenecks
Throughput bottlenecks
'res' (in lowered_fmax.B1)
Details

Variable on critical loop carried feedback path:	'res'
Declared at:	Unknown location
Dependency:	Data dependency
Estimated fmax reduced to:	90.5
	• Select(fmax_ii.cl: 6) • +(fmax_ii.cl: 5)

Another type of loop carried dependency is memory dependency, as shown in the following example:

```
for (…)
  for (…)
     … = mem[x];
  mem[y] = …;
```

Nested Loop Example

In a nested loop, the maximum concurrency is more difficult to compute. For example, the loop carried dependency in a nested loop does not necessarily affect the initiation interval of the outer loop. Additionally, a nested loop often requires the knowledge of the inner loop's trip count. Consider the following example:

```
1  __kernel void serial_exe_sample( __global unsigned* restrict input,
2                                   __global unsigned* restrict output,
3                                    int N ) {
4    unsigned offsets[1024];
5    unsigned size[1024];
6    unsigned results[4];
7    for (unsigned i = 0; i < N; i++) {
8      offsets[i] = input[i];
9    }
10
11    for (unsigned i = 1; i < (N-1); i++) {
12      unsigned num =  offsets[i];
13      unsigned sum = 0;
14      // There's a memory dependency, so the inner loops are executed
15      // serially, i.e. the both loops finish before the next iteration
16      // of i in the outer loop can start.
17      for (unsigned j = 0; j < num; j++) {
18        size[j] = offsets[i+j] - offsets[i+j-1];
19      }
20      for (unsigned j = 0; j < 4; j++) {
21        results[j] = size[j];
22      }
23    }
24
25    // store it back
26    #pragma unroll 1
27    for (unsigned i = 0; i < 4; i++) {
28      output[i] = results[i];
29    }
30  }
```

In this example, the bottleneck is resulted from loop carried dependency caused by a memory dependency on the variable `size`. The `size` variable must finish loading in the loop in line:20 before the next outer loop (line:11) iteration can be launched. Therefore, the maximum concurrency of the outer loop is 1. This information is reported in the details sections of the Loop Analysis and Schedule Viewer reports.

Addressing Bottlenecks

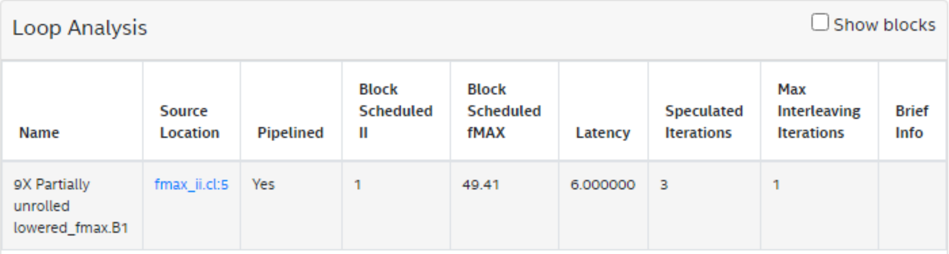
To address the bottlenecks, primarily ***consider restructuring your design code***.

After restructuring, consider applying the following loop pragmas or attributes on arrays:

- `#pragma II`. See Specifying a loop initiation interval (II) in the *Intel FPGA SDK for OpenCL Programming Guide*
- `#pragma ivdep safelen`. See Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays on page 132
- `#pragma max_concurrency`. See Loop Interleaving Control in the *Intel FPGA SDK for OpenCL Programming Guide*
- `attribute private_copies`. See Specifying the `private_copies` Memory Attribute in the *Intel FPGA SDK for OpenCL Programming Guide*

Consider the previous Simple Loop Example on page 81 where the concurrency is 3 as the initiation interval is 2. Applying `#pragma II 1`, as shown in the following code snippet, comes at the expense of lowered predicted f_{MAX} from 90MHz to 50MHz:

```
kernel void lowered_fmax (global int *dst, int N) {
    int res = N;
    #pragma unroll  9
    #pragma ii 1
    for (int i = 0; i < N; i++) {
        res += 1;
        res ^= i;
    }
    dst[0] = res;
 }
```



Loop Analysis

☐ Show blocks

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
9X Partially unrolled lowered_fmax.B1	fmax_ii.cl:5	Yes	1	49.41	6.000000	3	1	

Related Information

- Reviewing Loop Information on page 33
- Features of the Schedule Viewer on page 31

3.5. Channels

The Intel FPGA SDK for OpenCL's channel implementation provides a flexible way to pass data from one kernel to another kernel to improve performance.

When declaring a channel in your kernel code, precede the declaration with the keyword `channel`.

For example:

```
channel long16 myCh __attribute__((depth(16)));
```

In the HTML report, the area report maps the channel area to the declaration line in the source code. Channels and channel arrays are reported with their width and depth.

Note: The implemented channel depth can differ from the depth that you specify in the channel declaration. The Intel FPGA SDK for OpenCL Offline Compiler can implement the channel in shift registers or RAM blocks. The offline compiler decides on the type of channel implementation based on the channel depth.

The `depth` attribute is treated as the minimum depth specification. The offline compiler may increase the depth for the following reasons:

- Instruction scheduling requirements. This may happen due to the following reasons:
  - — To balance reconverging paths through multiple kernels.

    When multiple paths exist from one kernel to another via channels and other kernels, the depths on these channels may be increased to balance latencies among all these paths. This is a throughput optimization, as unbalanced paths are likely to lead to pipeline stalls.
  - — To achieve a lower II for a loop containing a non-blocking write to a channel.

    The offline compiler may increase the depth of the channel in order to achieve a lower loop II.

    Consider the following loop that reads `next_val` from the global memory only if the non-blocking write to `my_channel` succeeded in the previous iteration.

    ```
    bool write_valid = true;
    int next_val = 0;
    while (not_done) {
    ```

```
   if (write_valid) {
      next_val = *global_mem_ptr;
      global_mem_ptr++;
   }
   write_valid = write_channel_nb_intel(my_channel, next_val);
   not_done = some_fn(next_val);
}
```

With a naive implementation, this loop has a very high II because the high-latency global memory read must complete before the write into the channel can begin and the next value of `write_valid` is computed. To remove the global read from the II-critical path, the compiler can instead check if `my_channel` has space to accept whatever value is read from the global memory before doing the actual global read. The check for channel fullness takes one clock cycle and hence, the next loop iteration can start as soon as the channel fullness check is complete, giving II=1 for this loop. To make the resulting hardware functionally correct, the channel must be deepened by the latency of the global read or, to be precise, schedule distance between the channel fullness check and the actual write, which may be slightly greater than the global read. If you do not want to have your channel deepened in this situation, identify and remove the loop-carried dependency involving a valid return valid from the `write_channel_nb_intel()` call.

- The nature of the underlying FIFO implementation. This happens if the chosen underlying implementation cannot support the exact depth required and must be increased to the next supported size.

3.6. Load-Store Units

The Intel FPGA SDK for OpenCL Offline Compiler generates a number of different types of load-store units (LSUs). For some types of LSU, the compiler might modify the LSU behavior and properties depending on the memory access pattern and other memory attributes.

Tip: For Intel oneAPI DPC++/C++ Compiler-specific details, refer to Load-Store Units section in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

3.6.1. Load-Store Unit Types

The compiler can generate several different types of load-store units (LSUs) based on the inferred memory access pattern, the types of memory available on the target platform, and whether the memory accesses are to local or global memory.The Intel FPGA SDK for OpenCL Offline Compiler can generate the following types of LSU:

- Burst-Coalesced Load-Store Units on page 86
- Prefetching Load-Store Units on page 86
- Pipelined Load-Store Units on page 86
- Constant-Pipelined Load-Store Units on page 87
- Atomic-Pipelined Load-Store Units on page 87

Burst-Coalesced Load-Store Units

A burst-coalesced LSU is the default LSU type instantiated by the compiler for accessing global memory. It buffers requests until the largest possible burst can be made. The burst-coalesced LSU can provide efficient access to global memory, but it requires a considerable amount of FPGA resources.

```
kernel void burst_coalesced (global int * restrict in,
                             global int * restrict out) {
  int i = get_global_id(0);
  int value = in[i/2];      // Burst-coalesced LSU
  out[i] = value;
}
```

Depending on the memory access pattern and other attributes, the compiler might modify a burst-coalesced LSU in the following ways:

- Cached
- Write-Acknowledge (write-ack)
- Nonaligned

Prefetching Load-Store Units

A prefetching LSU instantiates a FIFO which burst reads large blocks from memory to keep the FIFO full of valid data based on the previous address and assuming contiguous reads. Non-contiguous reads are supported, but a penalty is incurred to flush and refill the FIFO. A prefetching LSU is inferred only for non-volatile global pointers.

```
kernel void prefetching (global int * restrict in,
                         global int * restrict out,
                         int N) {
  int res = 1;
  for (int i = 0; i < N; i++) {
    int v = in[i];             // Prefetching LSU
    res ^= v;
  }
  out[0] = res;
}
```

Pipelined Load-Store Units

A pipelined LSU is used for accessing local memory. Requests are submitted as soon as they are received. Memory accesses are pipelined, so multiple requests can be in flight at a time. If there is no arbitration between the LSU and the local memory, a pipelined never-stall LSU is created.

```
__attribute((reqd_work_group_size(1024,1,1)))
kernel void local_pipelined (global int* restrict in,
                             global int* restrict out) {
  local int lmem[1024];
  int gi = get_global_id(0);
  int li = get_local_id(0);

  int res = in[gi];
  for (int i = 0; i < 4; i++) {
    lmem[li - i] = res;                 // pipelined LSU
    res >>= 1;
  }

  barrier(CLK_GLOBAL_MEM_FENCE);

  res = 0;
  for (int i = 0; i < 4; i++) {
    res ^= lmem[li - i];                // pipelined LSU
  }

  out[gi] = res;
}
```

The compiler might modify a local-pipelined LSU in the following way:

- Never-stall

The compiler may also infer a pipelined LSU for global memory accesses that can be proven to be infrequent. The compiler uses a pipelined LSU for such accesses because a pipelined LSU is smaller than other LSU types. While a pipelined LSU might have lower throughput, this throughput tradeoff is acceptable because memory accesses are infrequent.

```
kernel void global_infrequent (global int * restrict in,
                               global int * restrict out,
                               int N) {
  int a = 0;
  if (get_global_id(0) == 0)
      a = in[0];                   // Pipelined LSU
  for (int i = 0; i < N; i++) {
    out[i] = in[i] + a;
  }
}
```

Constant-Pipelined Load-Store Units

A constant-pipelined LSU is a pipelined LSU that is used mainly to read from the constant cache. The constant-pipelined LSU consumes less area than a burst-coalesced LSU. The throughput of a constant-pipelined LSU depends greatly on whether the reads hit in the constant cache. Cache misses are expensive.

```
 kernel void constant_pipelined (constant int *src,
                                 global int *dst) {
  int i = get_global_id(0);
  dst[i] = src[i];              // Constant pipelined LSU
}
```

For information about the constant cache, see Constant Cache Memory on page 151.

Atomic-Pipelined Load-Store Units

An atomic-pipelined LSU is used for all atomic operations. Using atomic operations can significantly reduce kernel performance.

```
kernel void atomic_pipelined (global int* restrict out) {
  atomic_add(&out[0], 1);  // Atomic LSU
}
```

3.6.2. Load-Store Unit Modifiers

Depending on the memory access pattern in your kernel, the compiler modifies some LSUs.

Cached

Burst-coalesced LSUs might sometimes include a cache. A cache is created when the memory access pattern is data-dependent or appears to be repetitive. The cache cannot be shared with other loads even if the loads want the same data. The cache is flushed on kernel start and consumes more hardware resources than an equivalent LSU without a cache. The cache is inferred only for non-volatile global pointers.

```
kernel void cached (global int * restrict in,
                    global int * restrict out,
                    int N) {
  int gid = get_global_id(0);
  for (int i = 0; i < N; i++) {
    out[N*gid + i] = in[i];
  }
}
```

Write-Acknowledge (write-ack)

Burst-coalesced store LSUs sometimes require a write-acknowledgment signal when data dependencies exist. LSUs with a write-acknowledge signal require additional hardware resources. Throughput might be reduced if multiple write-acknowledge LSUs access the same memory.

```
kernel void write_ack (global int * restrict in,
                       global int * restrict out,
                       int N) {
  for (int i = 0; i < N; i++) {
    if (i < 2)
      out[i] = 0;            // Burst-coalesced write-ack LSU
    out[i] = in[i];
  }
}
```

Nonaligned

When a burst-coalesced LSU can access memory that is not aligned to the external memory word size, a nonaligned LSU is created. Additional hardware resources are required to implement a nonaligned LSU. The throughput of a nonaligned LSU might be reduced if it receives many unaligned requests.

```
kernel void non_aligned (global int * restrict in,
                         global int * restrict out) {
  int i = get_global_id(0);

  // Three loads are statically coalesced into one,
  // creating a burst-coalesced non-aligned LSU.
  int a1 = in[3*i+0];
  int a2 = in[3*i+1];
  int a3 = in[3*i+2];

  // Three stores statically coalesced into one,
  // creating a burst-coalesced non-aligned LSU.
  out[3*i+0] = a3;
  out[3*i+1] = a2;
  out[3*i+2] = a1;
}
```

Never-stall

If a pipelined LSU is connected to a local memory without arbitration, a never-stall LSU is created because all accesses to the memory take a fixed number of cycles that are known to the compiler.

```
__attribute((reqd_work_group_size(1024,1,1)))
kernel void never_stall (global int* restrict in,
                         global int* restrict out,
                         int N) {
  local int lmem[1024];
  int gi = get_global_id(0);
  int li = get_local_id(0);

  lmem[li] = in[gi];                   // Pipelined never-stall LSU
  barrier(CLK_GLOBAL_MEM_FENCE);
  out[gi] = lmem[li] ^ lmem[li + 1];
}
```

3.6.3. Controlling the Load-Store Units

The Intel FPGA SDK for OpenCL Offline Compiler allows you to control the type of LSU that is being generated for global memory accesses via a set of built-in calls that you can use for loading from and storing to global memory.

Load Built-ins

The variations of the load built-in are summarized in the following table:

Table 10. Load Built-ins

Built-in	LSU Type Implemented
`__pipelined_load()`	Pipelined if possible
`__prefetching_load()`	Prefetching if possible
`__burst_coalesced_load()`	Burst-coalesced
`__burst_coalesced_cached_load()`	Burst-coalesced cached if possible

All variations expect the following arguments:

Table 11. Load Built-in Arguments

Built-in	Type	Description
Argument #1	Pointer	Memory location to load from.
Argument #2	Integer	• Available only for `__burst_coalesced_cached_load()` function. • Describes the LSU cache size in bytes. • Non-negative compile-time constant integer.
Return value	Object	• Data that the pointer argument points to. • Same type as the base type of the pointer argument.

Store Built-ins

The variations of the store built-in are summarized in the following table:

Table 12. Store Built-ins

Built-in	LSU Type Implemented
`__pipelined_store()`	Pipelined if possible
`__burst_coalesced_store()`	Burst-coalesced

All variations expect the following arguments:

Table 13. Store Built-in Arguments

Built-in	Type	Description
Argument #1	Pointer	Memory location to store to.
Argument #2	Same as the pointer's base type	Value to be stored.

Note: All variations of the store built-in are non-value-returning.

Example

Following is an OpenCL example depicting different variations of the load and the store built-ins:

```
kernel void oclTest(global int * restrict in,
                    global int * restrict out) {
    int i = get_global_id(0);

    int a1 = __pipelined_load(in + 3*i+0); // Uses a pipelined LSU
    // Uses a burst-coalesced LSU with a cache of size 1024 bytes
    int a2 = __burst_coalesced_cached_load(&in[3*i+1], 1024);
    int a3 = __prefetching_load(&in[3*i+2]); // Uses a prefetching LSU

    __burst_coalesced_store(&out[3*i+0], a3); // Uses a burst-coalesced LSU
}
```

Note:

- The compiler does not allow you to select an LSU that may cause functionally incorrect results in the context in which it is being requested. For example, if you request a prefetching LSU on a volatile pointer, the compiler errors out. The compiler also errors out if caching is requested in a situation where the cache (which is local to the LSU) may become incoherent due to other LSUs writing to memory.
- The prefetching LSU is not available on the Intel Stratix® 10 device.

3.6.4. When to Use Each LSU

You can decide between different LSUs to use either based on what you know about the access patterns of your load/store site or on your silicon area requirements.The following are the LSU styles in an increasing order of their area requirements:

1. **Pipelined LSU (load/store)**: It is area efficient but it can be slower than other LSUs. You should use this LSU if you are constricted on area or if your access patterns are not necessarily consecutive.
2. **Prefetching LSU (only for loads)**: It is also area efficient but it is perfect for fully consecutive access patterns. There is a throughput penalty for using it for non-consecutive access patterns, so, use it only if you know that the addresses accessed are strictly consecutive.
3. **Burst-coalesced LSU (load/store)**: It is expensive in area but can process consecutive access patterns very efficiently. There is an area penalty for checking whether the access patterns are consecutive or not. The LSU dynamically attempts to combine several kernel requests into one big burst spanning multiple memory words, if possible.
4. **Burst-coalesced cached LSU (only for loads)**: It is the most expensive in area because it contains an extra cache that is local to the LSU. It can help the throughput in cases where you intend to read the same location in memory multiple times, especially across multiple ND-range threads.

4. OpenCL Kernel Design Best Practices

With the Intel FPGA SDK for OpenCL Offline Compiler technology, you do not need to change your kernel to fit it optimally into a fixed hardware architecture. Instead, the offline compiler customizes the hardware architecture automatically to accommodate your kernel requirements.

In general, you should optimize a kernel that targets a single compute unit first. After you optimize this compute unit, increase performance by scaling the hardware to fill the remainder of the FPGA by increasing the number of compute units. For more information, refer to Multiple Compute Units on page 140. The area use of the kernel correlates with the time it takes for hardware compilation. Therefore, to avoid waiting for long hardware compiles, focus on optimizing the performance of your kernel on a single compute unit first.

For important best practices for optimizing kernel performance, including data processing and memory access optimizations, read through the remaining chapters of this guide. The remainder of this chapter covers the following list of additional best practices. Consider implementing the following design practices, if applicable, when your create your kernels.

Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes on page 92

Unrolling Loops on page 97

Optimizing Floating-Point Operations on page 99

Allocating Aligned Memory on page 102

Aligning a Struct with or without Padding on page 103

Maintaining Similar Structures for Vector Type Elements on page 105

Avoiding Pointer Aliasing on page 105

Avoid Expensive Functions on page 106

Avoiding Work-Item ID-Dependent Backward Branching on page 107

4.1. Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes

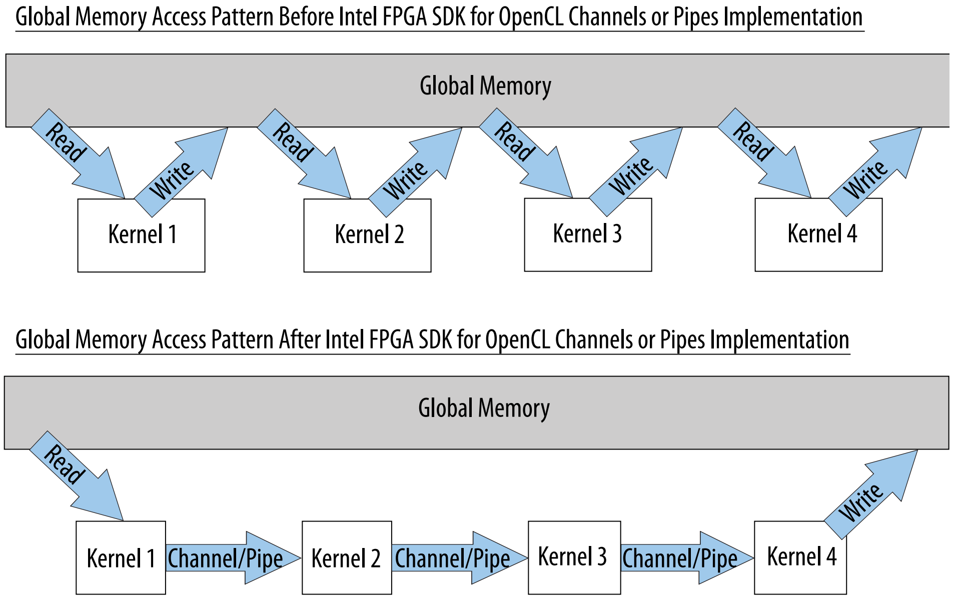
To increase data transfer efficiency between kernels, implement the Intel FPGA SDK for OpenCL channels extension in your kernel programs. If you want to use the capabilities of channels but also have the ability to run your kernel program using other SDKs, implement OpenCL pipes.

Sometimes, FPGA-to-global memory bandwidth constrains the data transfer efficiency between kernels. The theoretical maximum FPGA-to-global memory bandwidth varies depending on the number of global memory banks available in the targeted Custom Platform and board. To determine the theoretical maximum bandwidth for your board, refer to your board vendor's documentation.

In practice, a kernel does not achieve 100% utilization of the maximum global memory bandwidth available. The level of utilization depends on the access pattern of the algorithm.

If global memory bandwidth is a performance constraint for your OpenCL kernel, first try to break down the algorithm into multiple smaller kernels. Secondly, as shown in the figure below, eliminate some of the global memory accesses by implementing the SDK's channels or OpenCL pipes for data transfer between kernels.

Figure 68. Difference in Global Memory Access Pattern as a Result of Channels or Pipes Implementation


Global Memory Access Pattern Before Intel FPGA SDK for OpenCL Channels or Pipes Implementation
Global Memory
Read
Write
Read
Write
Read
Write
Read
Write
Kernel 1
Kernel 2
Kernel 3
Kernel 4
Global Memory Access Pattern After Intel FPGA SDK for OpenCL Channels or Pipes Implementation
Global Memory
Read
Write
Kernel 1
Channel/Pipe
Kernel 2
Channel/Pipe
Kernel 3
Channel/Pipe
Kernel 4

For more information about the usage of channels, refer to the *Implementing Intel FPGA SDK for OpenCL Channels Extension* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

For more information about the usage of pipes, refer to the *Implementing OpenCL Pipes* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

- Implementing Intel FPGA SDK for OpenCL Channels Extension
- Implementing OpenCL Pipes

4.1.1. Characteristics of Channels and Pipes

To implement channels or pipes in your OpenCL kernel program, keep in mind their respective characteristics that are specific to the Intel FPGA SDK for OpenCL.

Default Behavior

The default behavior of channels is blocking. The default behavior of pipes is nonblocking.

Concurrent Execution of Multiple OpenCL Kernels

You can execute multiple OpenCL kernels concurrently. To enable concurrent execution, modify the host code to instantiate multiple command queues. Each concurrently executing kernel is associated with a separate command queue.

Important: Pipe-specific considerations:

The OpenCL pipe modifications outlined in *Ensuring Compatibility with Other OpenCL SDKs* in the *Intel FPGA SDK for OpenCL Programming Guide* allow you to run your kernel on the SDK. However, they do not maximize the kernel throughput. The OpenCL Specification version 2.0 requires that pipe writes occur before pipe reads so that the kernel is not reading from an empty pipe. As a result, the kernels cannot execute concurrently. Because the Intel FPGA SDK for OpenCL supports concurrent execution, you can modify your host application and kernel program to take advantage of this capability. The modifications increase the throughput of your application; however, you can no longer port your kernel to another SDK. Despite this limitation, the modifications are minimal, and it does not require much effort to maintain both types of code.

To enable concurrent execution of kernels containing pipes, replace the `depth` attribute in your kernel code with the `blocking` attribute (that is, `__attribute__((blocking))`). The `blocking` attribute introduces a blocking behavior in the `read_pipe` and `write_pipe` function calls. The call site blocks kernel execution until the other end of the pipe is ready.

If you add both the `blocking` attribute and the `depth` attribute to your kernel, the `read_pipe` calls only a block when the pipe is empty, and the `write_pipe` calls only a block when the pipe is full. Blocking behavior causes an implicit synchronization between the kernels, which forces the kernels to run in lock step with each other.

Implicit Kernel Synchronization

Synchronize the kernels implicitly via blocking channel calls or blocking pipe calls. Consider the following examples:

Table 14. Blocking Channel and Pipe Calls for Kernel Synchronization

Kernels with Blocking Channel Call	Kernels with Blocking Pipe Call

`channel int c0;`
 	
`__kernel`
`void producer (__global int * in_buf)`
`{`
`  for (int i = 0; i < 10; i++)`
`  {`
`    write_channel_intel (c0, in_buf[i]);`
`  }`
`}`
 	
`__kernel`
`void consumer (__global int * ret_buf)`
`{`
`  for (int i = 0; i < 10; i++)`
`  {`
	`__kernel`
	`void producer (__global int * in_buf,`
	`  write_only pipe int __attribute__`
	`  ((blocking)) c0)`
	`{`
	`  for (int i = 0; i < 10; i++)`
	`{`
	`    write_pipe (c0, &in_buf[i]);`
	`  }`
	`}`
	
	`__kernel`
	`void consumer (__global int * ret_buf,`
	`  read_only pipe int __attribute__`
	`  ((blocking)) c0)`
	`{`
	`  for (int i = 0; i < 10; i++)`
	`  {`

Kernels with Blocking Channel Call	Kernels with Blocking Pipe Call

`    ret_buf[i] = read_channel_intel(c0);`
`  }`
`}`
	`    int x;`
	`    read_pipe (c0, &x);`
	`    ret_buf[i] = x;`
	`  }`
	`}`

You can synchronize the kernels such that a `producer` kernel writes data and a `consumer` kernel reads the data during each loop iteration. If the `write_channel_intel` or `write_pipe` call in `producer` does not write any data, `consumer` blocks and waits at the `read_channel_intel` or `read_pipe` call until `producer` sends valid data, and vice versa.

Data Persistence Across Invocations

After the `write_channel_intel` call writes data to a channel or the `write_pipe` call writes data to a pipe, the data is persistent across work-groups and NDRange invocations. Data that a work-item writes to a channel or a pipe remains in that channel or pipe until another work-item reads from it. In addition, the order of data in a channel or a pipe is equivalent to the sequence of write operations to that channel or pipe, and the order is independent of the work-item that performs the write operation.

For example, if multiple work-items try to access a channel or a pipe simultaneously, only one work-item can access it. The `write_channel_intel` call or `write_pipe` call writes the particular work-item data, called *DATAX*, to the channel or pipe, respectively. Similarly, the first work-item to access the channel or pipe reads DATAX from it. This sequential order of read and write operations makes channels and pipes an effective way to share data between kernels.

Imposed Work-Item Order

The SDK imposes a work-item order to maintain the consistency of the read and write operations for a channel or a pipe.

Related Information

Ensuring Compatibility with Other OpenCL SDKs

4.1.2. Execution Order for Channels and Pipes

Each channel or pipe call in a kernel program translates into an instruction executed in the FPGA pipeline. The execution of a channel call or a pipe call occurs if a valid work-item executes through the pipeline. However, even if there is no control or data dependence between channel or pipe calls, their execution might not achieve perfect instruction-level parallelism in the kernel pipeline.

Consider the following code examples:

Table 15. Kernel with Two Read Channel or Pipe Calls

Kernel with Two Read Channel Calls	Kernel with Two Read Pipe Calls

`__kernel void`
`consumer (__global uint*restrict dst) {`
`  for (int i = 0; i < 5; i++) {`
`    dst[2*i] = read_channel_intel(c0);`
`    dst[2*i+2] = read_channel_intel(c1);`
`  }`
`}`
	`__kernel void`
	`consumer (__global uint*restrict dst,`
	`  read_only pipe uint`
	`    __attribute__((blocking)) c0,`
	`  read_only pipe uint`
	`    __attribute__((blocking)) c1)`
	`{`
	`  for (int i = 0; i < 5; i++) {`

<table>
<tr><th>Kernel with Two Read Channel Calls</th><th>Kernel with Two Read Pipe Calls</th></tr>
<tr><td></td><td><pre>
    read_pipe (c0, &dst[2*i]);
    read_pipe (c1, &dst[2*i+2]);
  }
}
</pre></td></tr>
</table>

The code example on the left makes two read channel calls. The code example on the right makes two read pipe calls. In most cases, the kernel executes these channel or pipe calls in parallel; however, channel and pipe call executions might occur out of sequence. Out-of-sequence execution means that the read operation from `c1` can occur and complete before the read operation from `c0`.

4.1.3. Optimizing Buffer Inference for Channels or Pipes

In addition to the manual addition of buffered channels or pipes, the Intel FPGA SDK for OpenCL Offline Compiler improves kernel throughput by adjusting buffer sizes whenever possible.

During compilation, the offline compiler computes scheduling mismatches between interacting channels or pipes. These mismatches might cause imbalances between read and write operations. The offline compiler performs buffer inference optimization automatically to correct the imbalance.

Consider the following examples:

Table 16. Buffer Inference Optimization for Channels and Pipes

<table>
<tr><th>Kernels with Channels</th><th>Kernels with Pipes</th></tr>
<tr><td><pre>
__kernel void producer (
  __global const uint * restrict src,
  const uint iterations)
{
  for(int i = 0; i < iteration; i++)
  {
    write_channel_intel(c0,src[2*i]);
    write_channel_intel(c1,src[2*i+1]);
  }
}

__kernel void consumer (
  __global uint * restrict dst,
  const uint iterations)
{
  for(int i = 0; i < iterations; i++)
  {
    dst[2*i] = read_channel_intel(c0);
    dst[2*i+1] = read_channel_intel(c1);
  }
}
</pre></td><td><pre>
__kernel void producer (
  __global const uint * restrict src,
  const uint iterations,
  write_only pipe uint
    __attribute__((blocking)) c0,
  write_only pipe uint
    __attribute__((blocking)) c1)
{
  for(int i = 0; i < iteration; i++)
  {
    write_pipe(c0,&src[2*i]);
    write_pipe(c1,&src[2*i+1]);
  }
}

__kernel void consumer (
  __global uint * restrict dst,
  const uint iterations,
  read_only pipe uint
    __attribute__((blocking)) c0,
  read_only pipe uint
    __attribute__((blocking)) c1)
{
  for(int i = 0; i < iterations; i++)
  {
    read_pipe(c0,&dst[2*i]);
    read_pipe(c1,&dst[2*i+1]);
  }
}
</pre></td></tr>
</table>

The offline compiler performs buffer inference optimization if channels or pipes between kernels cannot form a cycle. A *cycle* between kernels is a path that originates from a kernel, through a write channel or a write pipe call, and returns to the original kernel. For the example, assume that the write channel or write pipe calls in the kernel `producer` are scheduled 10 cycles apart and the read channel or read pipe calls are scheduled 15 cycles apart. There exists a temporary mismatch in the read and write operations to `c1` because five extra write operations might occur before a read operation to c1 occurs. To correct this imbalance, the offline compiler assigns a buffer size of five cycles to c1 to avoid stalls. The extra buffer capacity decouples the c1 write operations in the producer kernel and the c1 read operations in the consumer kernel.

4.1.4. Best Practices for Channels and Pipes

Consider the following best practices when designing channels and pipes:

- Use single-threaded kernels over multi-threaded kernels.
- Consider how the design model can be represented with a feed forward datapath, for example, back-to-back loops or discrete processing steps. Determine whether you should split the design into multiple kernels connected by channels.
- Aggregate data on channels only when the entire data is used at the same point of kernel.
- Attempt to keep the number of channels per kernel reasonable.
- Do not use non-blocking channels or pipes if you are using a looping structure waiting for the data. Non-blocking channels consume more resources than the blocking channels.

4.2. Unrolling Loops

You can control the way the Intel FPGA SDK for OpenCL Offline Compiler translates OpenCL kernel descriptions to hardware resources. If your OpenCL kernel contains loop iterations, increase performance by unrolling the loop. Loop unrolling decreases the number of iterations that the offline compiler executes at the expense of increased hardware resource consumption.

Tip: For Intel oneAPI DPC++/C++ Compiler-specific details, refer to Unroll Loops topic in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

Consider the OpenCL code for a parallel application in which each work-item is responsible for computing the accumulation of four elements in an array:

```
__kernel void example ( __global const int * restrict x,
                        __global int * restrict sum ) {
   int accum = 0;

   for (size_t i = 0; i < 4; i++) {
      accum += x[i + get_global_id(0) * 4];
   }

   sum[get_global_id(0)] = accum;
}
```

Notice the following three main operations that occur in this kernel:

- Load operations from input x
- Accumulation
- Store operations to output sum

The offline compiler arranges these operations in a pipeline according to the data flow semantics of the OpenCL kernel code. For example, the offline compiler implements loops by forwarding the results from the end of the pipeline to the top of the pipeline, depending on the loop exit condition.

The OpenCL kernel performs one loop iteration of each work-item per clock cycle. With sufficient hardware resources, you can increase kernel performance by unrolling the loop, which decreases the number of iterations that the kernel executes. To unroll a loop, add a `#pragma unroll` directive to the main loop, as shown in the code example below. Keep in mind loop unrolling significantly changes the structure of the compute unit that the offline compiler creates.

```
__kernel void example ( __global const int * restrict x,
                        __global int * restrict sum ) {
  int accum = 0;

  #pragma unroll
  for (size_t i = 0; i < 4; i++) {
    accum += x[i + get_global_id(0) * 4];
  }

  sum[get_global_id(0)] = accum;
}
```

In this example, the `#pragma unroll` directive causes the offline compiler to unroll the four iterations of the loop completely. To accomplish the unrolling, the offline compiler expands the pipeline by tripling the number of addition operations and loading four times more data. With the removal of the loop, the compute unit assumes a feed-forward structure. As a result, the compute unit can store the `sum` elements every clock cycle after the completion of the initial load operations and additions. The offline compiler further optimizes this kernel by coalescing the four load operations so that the compute unit can load all the necessary input data to calculate a result in one load operation.

Caution: Avoid nested looping structures. Instead, implement a large single loop or unroll inner loops by adding the `#pragma unroll` directive whenever possible.

For example, if you compile a kernel that has a heavily-nested loop structure, wherein each loop includes a `#pragma unroll` directive, you might experience a long compilation time. The Intel FPGA SDK for OpenCL Offline Compiler might fail to meet scheduling because it cannot unroll this nested loop structure easily, resulting in a high II. In this case, the offline compiler issues the following error message along with the line number of the outermost loop:

```
Kernel <function> exceeded the Max II. The Kernel's resource
usage is estimated to be much larger than FPGA capacity. It
performs poorly even if it fits. Reduce resource utilization of
the kernel by reducing loop unroll factors within it (if any) or
otherwise reduce amount of computation within the kernel.
```

Unrolling the loop and coalescing the load operations from global memory allow the hardware implementation of the kernel to perform more operations per clock cycle. In general, the methods you use to improve the performance of your OpenCL kernels should achieve the following results:

- Increase the number of parallel operations
- Increase the memory bandwidth of the implementation
- Increase the number of operations per clock cycle that the kernels can perform in hardware

The offline compiler might not be able to unroll a loop completely under the following circumstances:

- You specify complete unrolling of a data-dependent loop with a very large number of iterations. Consequently, the hardware implementation of your kernel might not fit into the FPGA.
- You specify complete unrolling and the loop bounds are not constants.
- The loop consists of complex control flows (for example, a loop containing complex array indexes or exit conditions that are unknown at compilation time).

For the last two cases listed above, the offline compiler issues the following warning:

```
Full unrolling of the loop is requested but the loop bounds
cannot be determined. The loop is not unrolled.
```

To enable loop unrolling in these situations, specify the `#pragma unroll` *<N>* directive, where *<N>* is the unroll factor. The unroll factor limits the number of iterations that the offline compiler unrolls. For example, to prevent a loop in your kernel from unrolling, add the directive `#pragma unroll 1` to that loop.

Refer to *Good Design Practices for Single Work-Item Kernel* for tips on constructing well-structured loops.

Related Information

Good Design Practices for Single Work-Item Kernel on page 134

4.3. Optimizing Floating-Point Operations

For floating-point operations, you can manually direct the Intel FPGA SDK for OpenCL Offline Compiler to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware usage. These optimizations can cause small differences in floating-point results.

Tip: For more oneAPI DPC++-specific details, refer to Optimize Floating-point Operation topic in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

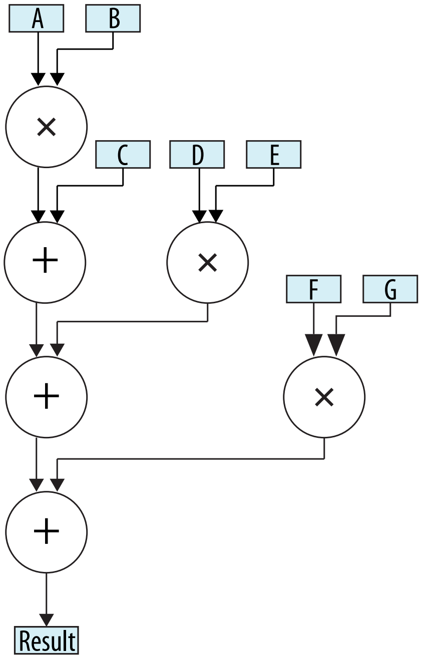
Tree Balancing

Order of operation rules apply in the OpenCL language. In the following example, the offline compiler performs multiplications and additions in a strict order, beginning with operations within the innermost parentheses:

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

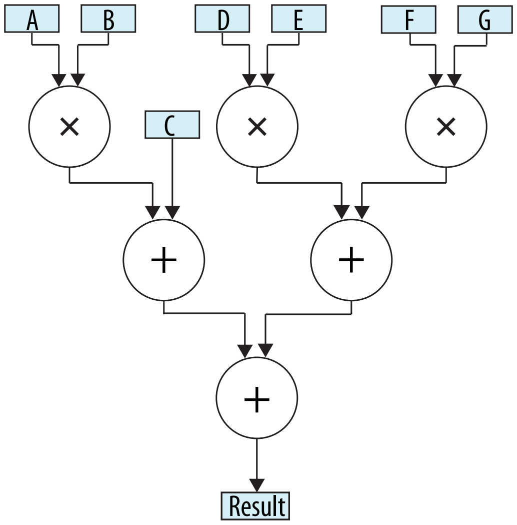
By default, the offline compiler creates an implementation that resembles a long vine for such computations:

Figure 69. Default Floating-Point Implementation



Long, unbalanced operations lead to more expensive hardware. A more efficient hardware implementation is a *balanced tree*, as shown below:

Figure 70. Balanced Tree Floating-Point Implementation



In a balanced tree implementation, the offline compiler converts the long vine of floating-point adders into a tree pipeline structure. The offline compiler does not perform tree balancing of floating-point operations automatically because the outcomes of the floating-point operations might differ. As a result, this optimization is inconsistent with the IEEE Standard 754-2008.

If you want the offline compiler to optimize floating-point operations using balanced trees and your program can tolerate small differences in floating-point results, include the `-fp-relaxed` option in the `aoc` command, as shown below:

```
aoc -fp-relaxed <your_kernel_filename>.cl
```

Rounding Operations

The balanced tree implementation of a floating-point operation includes multiple rounding operations. These rounding operations can require a significant amount of hardware resources in some applications. The offline compiler does not reduce the number of rounding operations automatically because doing so violates the results required by IEEE Standard 754-2008.

You can reduce the amount of hardware necessary to implement floating-point operations with the `-fpc` option of the `aoc` command. If your program can tolerate small differences in floating-point results, invoke the following command:

```
aoc -fpc <your_kernel_filename>.cl
```

The `-fpc` option directs the offline compiler to perform the following tasks:

- Remove floating-point rounding operations and conversions whenever possible.

  If possible, the `-fpc` argument directs the offline compiler to round a floating-point operation only once—at the end of the tree of the floating-point operations.

- Carry additional mantissa bits to maintain precision.

  The offline compiler carries additional precision bits through the floating-point calculations, and removes these precision bits at the end of the tree of floating-point operations.

This type of optimization results in hardware that performs a fused *floating-point operation*, and it is a feature of many new hardware processing systems. Fusing multiple floating-point operations minimizes the number of rounding steps, which leads to more accurate results. An example of this optimization is a fused multiply-accumulate (FMAC) instruction available in new processor architectures. The offline compiler can provide fused floating-point mathematical capabilities for many combinations of floating-point operators in your kernel.

4.3.1. Floating-Point versus Fixed-Point Representations

An FPGA contains a substantial amount of logic for implementing floating-point operations. However, you can increase the amount of hardware resources available by using a fixed-point representation of the data whenever possible. The hardware necessary to implement a fixed-point operation is typically smaller than the equivalent floating-point operation. As a result, you can fit more fixed-point operations into an FPGA than the floating-point equivalent.

The OpenCL standard does not support fixed-point representation; you must implement fixed-point representations using integer data types. Hardware developers commonly achieve hardware savings by using fixed-point data representations and only retain a data resolution required for performing calculations. You must use an 8, 16, 32, or 64-bit scalar data type because the OpenCL standard supports only these data resolutions. However, you can incorporate the appropriate masking operations in your source code so that the hardware compilation tools can perform optimizations to conserve hardware resources.

For example, if an algorithm uses a fixed-point representation of 17-bit data, you must use a 32-bit data type to store the value. If you then direct the Intel FPGA SDK for OpenCL Offline Compiler to add two 17-bit fixed-point values together, the offline compiler must create extra hardware to handle the addition of the excess upper 15 bits. To avoid having this additional hardware, you can use static bit masks to direct the hardware compilation tools to disregard the unnecessary bits during hardware compilation. The code below implements this masking operation:

```
__kernel 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);

       unsigned int temp;
       temp = 0x3_FFFF & ((0x1_FFFF & a[gid]) + ((0x1_FFFF & b[gid]));

       result[gid] = temp & 0x3_FFFF;
}
```

In this code example, the upper 15 bits of inputs a and b are masked away and added together. Because the result of adding two 17-bit values cannot exceed an 18-bit resolution, the offline compiler applies an additional mask to mask away the upper 14 bits of the result. The final hardware implementation is a 17-bit addition as opposed to a full 32-bit addition. The logic savings in this example are relatively minor compared to the sheer number of hardware resources available in the FPGA. However, these small savings, if applied often, can accumulate into a larger hardware saving across the entire FPGA.

4.4. Allocating Aligned Memory

When allocating host-side memories that are used to transfer data to and from the FPGA,the memory must be at least 64-byte aligned.

Aligning the host-side memories allows direct memory access (DMA) transfers to occur to and from the FPGA and improves buffer transfer efficiency.

Attention: Depending on how the host-side memory is used, Intel recommends to allocate more strict alignment. For example, if the allocated memory is used to create a buffer using the CL_MEM_USE_HOST_PTR flag, the memory should also be properly aligned to the data types used to access the buffer in kernels. For more information about the alignment requirements of host-side memory, refer to section C.3 of the *OpenCL Specification version 1.2*.

To set up aligned memory allocations, add the following source code to your host program:

- For Windows:

  ```
  #define AOCL_ALIGNMENT 64
  #include <malloc.h>
  void *ptr = _aligned_malloc (size, AOCL_ALIGNMENT);
  ```

  To free up an aligned memory block, include the function call `_aligned_free(ptr);`

- For Linux:

  ```
  #define AOCL_ALIGNMENT 64
  #include <stdlib.h>
  void *ptr = NULL;
  posix_memalign (&ptr, AOCL_ALIGNMENT, size);
  ```

  To free up an aligned memory block, include the function call `free(ptr);`

Related Information

OpenCL Specification version 1.2

4.5. Aligning a Struct with or without Padding

A properly aligned struct helps the Intel FPGA SDK for OpenCL Offline Compiler generate the most efficient hardware. A proper `struct` alignment means that the alignment can be evenly divided by the `struct` size.

Important: Ensure a 4-byte alignment for the data structures. `struct` alignments smaller than four bytes result in larger and slower hardware. Hardware efficiency increases with the increasing alignment. In the following example, the `Pixel_s` structure is only one-byte aligned but the `Pixel` structure is four-byte aligned due to the presence of a four-byte `not_used` integer:

```
typedef struct {
    char r,g,b,alpha;
} Pixel_s;

typedef union {
     Pixel_s p;
     int not_used;
} Pixel;
```

You can also use the `aligned` attribute to force a 4-byte alignment, as shown in the following example code:

```
typedef struct {
       char r,g,b,alpha;
} __attribute__((aligned(4))) Pixel;
```

The offline compiler conforms with the ISO C standard that requires the alignment of a `struct` to satisfy all of the following criteria:

- The alignment must be an integer multiple of the lowest common multiple between the alignments of all `struct` members.
- The alignment must be a power of two.

You may set the struct alignment by including the aligned(*N*) attribute in your kernel code. Without an aligned attribute, the offline compiler determines the alignment of each struct in an array of struct based on the size of the struct. Consider the following example:

```
__kernel void test (struct mystruct* A,
                    struct mystruct* B)
{
    A[get_global_id(0)] = B[get_global_id(0)];
}
```

If the size of mystruct is 101 bytes, each load or store access is 1-byte aligned. If the size of mystruct is 128 bytes, each load or store access is 128-byte aligned, which generates the most efficient hardware.

When the struct fields are not aligned within the struct, the offline compiler inserts padding to align them. Inserting padding between struct fields affects hardware efficiency in the following manner:

- Increases the size of the struct
- Might affect the alignment

To prevent the offline compiler from inserting padding, include the packed attribute in your kernel code. The aforementioned ISO C standard applies when determining the alignment of a packed or unpacked struct. Consider the following example:

```
struct mystruct1
{
    char a;
    int b;
};
```

The size of mystruct1 is 8 bytes. Therefore, the struct is 8-byte aligned, resulting in efficient accesses in the kernel. Now consider another example:

```
struct mystruct2
{
    char a;
    int b;
    int c;
};
```

The size of mystruct2 is 12 bytes and the struct is 4-byte aligned. Because the struct fields are padded and the struct is unaligned, accesses in the kernel are inefficient.

Following is an example of a struct that includes the packed attribute:

```
struct __attribute__((packed)) mystruct3
{
    char a;
    int b;
    int c;
};
```

The size of mystruct3 is 16 bytes. Because mystruct3 is aligned and there is no padding between struct fields, accesses in this kernel are more efficient than accesses in mystruct3.

To include both the `aligned(N)` and `packed` attributes in a struct, consider the following example:

```
struct __attribute__((packed)) __attribute__((aligned(16))) mystruct5
{
    char a;
    int b;
    int c;
};
```

The size of `mystruct5` is 9 bytes. Because of the `aligned(16)` attribute, the `struct` is stored at 16-byte aligned addresses in an array. Because `mystruct5` is 16-byte aligned and has no padding, accesses in this kernel is efficient.

For more information about `struct` alignment and the `aligned(N)` and `packed` attributes, refer to the following documents:

- Section 6.11.1 of the *OpenCL Specification version 1.2*
- *Disabling Insertion of Data Structure Padding* section of the *Intel FPGA SDK for OpenCL Programming Guide*
- *Specifying the Alignment of a Struct* section of the *Intel FPGA SDK for OpenCL Programming Guide*

Related Information

- OpenCL Specification version 1.2
- Disabling Insertion of Data Structure Padding
- Specifying the Alignment of a Struct

4.6. Maintaining Similar Structures for Vector Type Elements

If you update one element of a vector type, update all elements of the vector.

The following code example illustrates a scenario where you should update a vector element:

```
__kernel void update (__global const float4 * restrict in,
                      __global const float4 * restrict out)
{
      size_t gid = get_global_id(0);

      out[gid].x = process(in[gid].x);
      out[gid].y = process(in[gid].y);
      out[gid].z = process(in[gid].z);
      out[gid].w = 0; //Update w even if that variable is not required.
}
```

4.7. Avoiding Pointer Aliasing

Insert the `restrict` keyword in pointer arguments whenever possible. Including the `restrict` keyword in pointer arguments prevents the Intel FPGA SDK for OpenCL Offline Compiler from creating unnecessary memory dependencies between non-conflicting load and store operations.

The `restrict` keyword informs the offline compiler that the pointer does not alias other pointers. For example, if your kernel has two pointers to global memory, A and B, that never overlap each other, declare the kernel in the following manner:

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

Warning: Inserting the `restrict` keyword on a pointer that aliases other pointers might result in incorrect results.

4.8. Avoid Expensive Functions

Some functions are expensive to implement in FPGAs. Expensive functions might decrease kernel performance or require a large amount of hardware to implement.

The following functions are expensive:

- Integer division and modulo (remainder) operators
- Most floating-point operators except addition, multiplication, absolute value, and comparison

  Note: For more information about optimizing floating-point operations, refer to the *Optimize Floating-Point Operations* section.
- Atomic functions

In contrast, inexpensive functions have minimal effects on kernel performance, and their implementation consumes minimal hardware.

The following functions are inexpensive:

- Binary logic operations such as AND, NAND, OR, NOR, XOR, and XNOR
- Logical operations with one constant argument
- Shift by constant
- Integer multiplication and division by a constant that is a power of two

If an expensive function produces a new piece of data for every work-item in a work-group, it is beneficial to code it in a kernel. On the contrary, the code example below shows a case of an expensive floating-point operation (division) executed by every work-item in the NDRange:

```
__kernel void myKernel (__global const float * restrict a,
                        __global float * restrict b,
                        const float c, const float d)
{
   size_t gid = get_global_id(0);

   //inefficient since each work-item must calculate c divided by d
   b[gid] = a[gid] * (c / d);
}
```

The result of this calculation is always the same. To avoid this redundant and hardware resource-intensive operation, perform the calculation in the host application and then pass the result to the kernel as an argument for all work-items in the NDRange to use. The modified code is shown below:

```
__kernel void myKernel (__global const float * restrict a,
                        __global float * restrict b,
                        const float c_divided_by_d)
```

```
{
   size_t gid = get_global_id(0);

   /*host calculates c divided by d once and passes it into
   kernel to avoid redundant expensive calculations*/
   b[gid] = a[gid] * c_divided_by_d;
}
```

The Intel FPGA SDK for OpenCL Offline Compiler consolidates operations that are not work-item-dependent across the entire NDRange into a single operation. It then shares the result across all work-items. In the first code example, the offline compiler creates a single divider block shared by all work-items because division of *c* by *d* remains constant across all work-items. This optimization helps minimize the amount of redundant hardware. However, the implementation of an integer division requires a significant amount of hardware resources. Therefore, it is beneficial to off-load the division operation to the host processor and then pass the result as an argument to the kernel to conserve hardware resources.

Related Information

Optimizing Floating-Point Operations on page 99

4.9. Avoiding Work-Item ID-Dependent Backward Branching

The Intel FPGA SDK for OpenCL Offline Compiler collapses conditional statements into single bits that indicate when a particular functional unit becomes active. The offline compiler completely eliminates simple control flows that do not involve looping structures, resulting in a flat control structure and more efficient hardware usage. The offline compiler compiles kernels that include forward branches, such as conditional statements, efficiently.
Avoid including any work-item ID-dependent backward branching (that is, branching that occurs in a loop) in your kernel because it degrades performance.

For example, the following code fragment illustrates branching that involves work-item ID such as `get_global_id` or `get_local_id`:

```
for (size_t i = 0; i < get_global_id(0); i++)
{
      // statements
}
```

5. Profiling Your Kernel to Identify Performance Bottlenecks

The Intel FPGA dynamic profiler for OpenCL uses performance counters to collect kernel performance data during the design's execution. This data can be viewed using the Intel VTune Profiler.

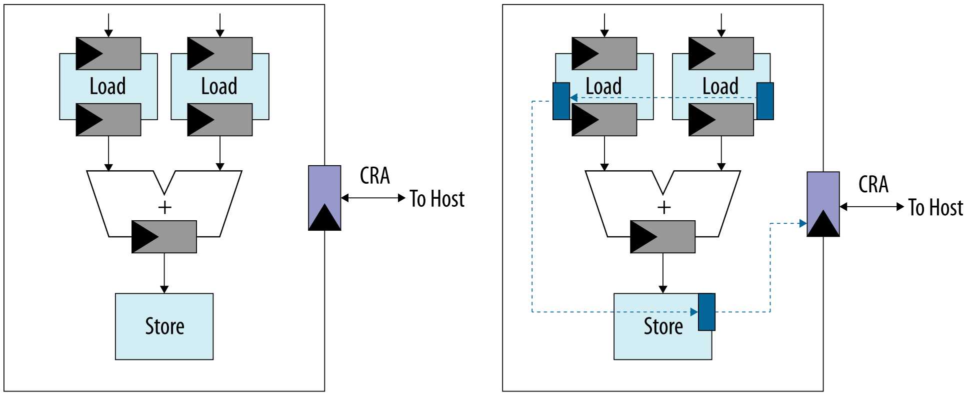
Tip: If you are looking for information about the Intel FPGA dynamic profiler for DPC++, then refer to the Intel FPGA Dynamic Profiler for DPC++ section in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

Consider the following OpenCL kernel program:

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

As shown in the figure below, the Profiler instruments and connects performance counters in a daisy chain throughout the pipeline generated for the kernel program. The host then reads the data collected by these counters. For example, in PCI Express® (PCIe)-based systems, the host reads the data via the PCIe control register access (CRA) or control and status register (CSR) port.

Figure 71. Intel FPGA Dynamic Profiler for OpenCL: Performance Counters Instrumentation



Work-item execution stalls might occur at various stages of an Intel FPGA SDK for OpenCL pipeline. Applications with large amounts of memory accesses or load and store operations might stall frequently to enable the completion of memory transfers. The Profiler helps identify the load and store operations or channel accesses that cause the majority of stalls within a kernel pipeline.

5.1. Best Practices for Profiling Your Kernel

Intel recommends that you follow these best practices when profiling your OpenCL kernel.

- Include the `-profile` Intel FPGA SDK for OpenCL Offline Compiler command option in your `aoc` command during development to insert performance counters into your kernel.
- Run the host application from a local folder to reduce profiler overhead. Avoid running your host from a remote or NAS folder.
- Ensure that kernel runtime is longer than 20 ms. Otherwise, the overhead of reading Profiler performance data to host takes over.
- Understand how all the load and store operations and channels are connected in the data flow.

5.2. Instrumenting the Kernel Pipeline with Performance Counters (-profile)

To instrument the OpenCL kernel pipeline with performance counters, include the `-profile=(all|autorun|enqueued)` option of the `aoc` command when you compile your kernel.

Note: Instrumenting the Verilog code with performance counters increases hardware resource utilization (that is, increases FPGA area usage) and typically decreases performance.

- To instrument the Verilog code in the <*your_kernel_filename*>`.aocx` file with performance counters, invoke the `aoc -profile=(all|autorun|enqueued)` <*your_kernel_filename*>`.cl` command, where:
  - `all` argument instruments all kernels in the <*your_kernel_filename*>`.cl` file with performance counters. This is the default option if no argument is provided.
  - `autorun` argument instruments only the autorun kernels with performance counters.
  - `enqueued` argument instruments only the non-autorun kernels with performance counters.

  Note:
  - When profiling multiple, different kernels, do not use the same kernel names across different `.aocx` files. If the kernel names are the same, the profile data is wrong for these kernels.
  - Regardless of the input to the `clGetProfileDataDeviceIntelFPGA` host library call, the Intel FPGA dynamic profiler for OpenCL only profiles kernel types that you indicate during compilation.
  - Instrumenting the OpenCL kernel pipeline with performance counters on all or enqueued kernels disables the use of hardware kernel invocation queue by the OpenCL runtime environment. This may result in different profile time.

  Caution: Profiling autorun kernels results in some hardware overhead for the counters. For large designs, the overhead can cause f_{MAX} and design frequency degradation. It can also lead to designs that cannot fit on the chip if the Intel FPGA dynamic profiler for OpenCL profiles every kernel.

5.3. Obtaining Profiling Data During Runtime

You can obtain profiling data during runtime in one of the following ways:

- Use the Profiler Runtime Wrapper from the command line to obtain the data. This data can later be imported into Intel VTune Profiler. For more information, refer to Invoking the Profiler Runtime Wrapper on page 110.
- Run your host application in Intel VTune Profiler using the CPU/FPGA Interaction view. For more information about how to configure and run your host application, refer to CPU/FPGA Interaction Analysis and Viewing Profiling Data Using Intel VTune™ Profiler on page 111.

5.3.1. Invoking the Profiler Runtime Wrapper

To profile your FPGA design using the Profiler Runtime Wrapper, first ensure that you have included the `-profile` option in your aoc command when you compiled your kernels.

The Profiler Runtime Wrapper ensures that data is collected from the performance counters, which are in the compiled design, during the host execution. Data is saved in a `profile.mon` monitor description file, which the Profiler Runtime Wrapper then post processes and converts into a readable `profile.json` file. While both the `profile.mon` and `profile.json` files are available after host execution completes, you are encouraged to use the `profile.json` file for further data processing.

To invoke the Profiler Runtime Wrapper, execute the following command:

```
aocl profile [options] /path/to/host-executable [executable options]
```

where

- `[options]` are any additional flags you want to pass to the wrapper. Use the `aocl profile –help` command to view a list of options and their uses.
- `/path/to/host-executable` is the path to the host executable.
- `[executable options]` are options or arguments that must be passed to the host executable.

Note: If you are executing from a different directory than your compilation directory, the wrapper also needs the compiled binary (`.aocx`) file, which you can pass using the option `-x <path/to/.aocx>`.

Caution: Because of slow network disk accesses, running the host application from a networked directory might introduce delays between kernel executions. These delays might increase the overall execution time of the host application. In addition, they might introduce delays during kernel executions while the runtime stores profile output data to disk.

5.3.1.1. Splitting Execution and Data Post Processing

By default, the Profiler Runtime Wrapper automatically runs a post-processing step on your `profile.mon` monitor file to produce a readable `profile.json` file. In some situations, the post-processing step may take longer time than expected. Because of this, you can choose to separate the execution and data post-processing step into two separate manual steps. To do this, use the `--no-json` and `--no-run <path to profile.mon file>` Profiler Runtime Wrapper options.

- The `--no-json` flag only runs your host application and produces a `profile.mon` monitor file, without post-processing.
- The `--no-run <path to profile.mon file>` flag does not invoke your host application, instead, just calls the post-processing step on the supplied `profile.mon` file.

  Caution: The Profiler Runtime Wrapper's `--no-run` flow is not backwards compatible for `profile.mon` files created using a runtime version earlier than Intel FPGA SDK for OpenCL version 20.3.

5.3.2. Viewing Profiling Data Using Intel VTune™ Profiler

Intel VTune Profiler is a performance analysis tool for developing serial and multithreaded applications. It helps you analyze the algorithm choices and identify where and how your application can benefit from available hardware resources.

To view performance data, upload your `profile.json` file to the **CPU/FPGA Interaction** view in the Intel VTune Profiler. For more information about how to upload the file and open the correct views, refer to CPU/FPGA Interaction Analysis (Preview) in the *Intel VTune Profiler User Guide*.

Use the **CPU/FPGA Interaction** view in the Intel VTune Profiler to determine the following performance information about your design in various graphical representations:

Table 17. Types of Information in CPU/FPGA Interaction view

Tab	Information
Summary	Contains a summarized or average data about your kernels' execution.
Bottom-up	Contains a graphical timeline view and an expandable summary of each kernel's execution including host and device-side events. Double-click on a kernel to view its source in the **Source** tab.
Platform	Contains information about the memory transfers and CPU-side information.
Source	Contains detailed statistics about memory and pipe accesses in source view format. For more information, refer to Performance Data Types on page 114.

5.4. Reducing Area Resource Use While Profiling

Due to various performance counters being added to the pipeline, introducing profiling into your design can result in a large amount of area resource use. This may be inconvenient for particularly large designs as adding profiling performance counters might result in *no fit* errors.

To reduce the amount of area resources that profiling takes up, you can choose to profile with *shared* performance counters. This profiling mode allows counters to be shared by various signals over multiple design runs to reduce the number of performance counters added to the design. During runtime, the Profiler Runtime Wrapper runs the host application four times, where, for each run, the counters *count* a different signal.

Note: You must invoke the Profiler Runtime Wrapper only once.

To turn on the shared performance counters profiling mode, perform these steps:

1. Include the `-profile-shared-counters` flag along with the `-profile` flag during your `aoc` compile.
2. Include the `-sc` flag when running your design with the Profiler Runtime Wrapper.

   Without the `-sc` flag, your design runs only once so, you lack data for everything after the first shared signal.

   Caution: The shared performance counters profiling mode works well only for kernels and designs that are deterministic. Because the host application and design are run multiple times to collect all of the data, non-deterministic designs result in shared data that is difficult to combine, and it may be difficult to determine where design problems occur temporally.

5.5. Temporal Performance Collection

During the run of your host application, the Profiler collects performance counter data at a given sample rate n. After n cycles, the Profiler collects performance counter data and outputs to the `profile.mon` monitor file.

- You can control the rate at which the Profiler counters are sampled by setting the Profiler Runtime Wrapper's `-period` flag. The specified period is the minimum number of kernel pipeline clock cycles between profiling samples. If you do not set a period, the default behavior is to profile as often as possible.

  Note: For particularly large or long running designs, the amount of data generated by the default temporal period might result in a very large `profile.mon` and `profile.json` file. To reduce this file size, either increase the sampling period or turn off temporal profiling.

- To turn off temporal profiling and instead collect performance data only after a kernel has finished executing, you can set the Profiler Runtime Wrapper's `-no-temporal` flag.
- The Profiler does not automatically collect the profiling information for autorun kernels if you disable temporal profiling, since autorun kernels never finish. You can use the host API call `clGetProfileDataDeviceIntelFPGA` to obtain profiling data from autorun kernels. For more information about triggering profiling using your host application, refer to Collecting Profile Data During Kernel Execution in the *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*.

  Note: If you collect the performance data only at the end of execution, the data is an average representation of the kernel's overall execution.

5.5.1. Profiling Autorun Kernels

Autorun kernel profiling feature allows you to profile autorun kernels.

Kernels that are marked with the `autorun` attribute are referred to as autorun kernels. An autorun kernel starts executing without being created and launched by the host, so it runs before the kernels that are explicitly enqueued, and restarts automatically on completion. For more information about the `autorun` attribute, refer to *Omit Communication Hardware between the Host and the Kernel* topic.

Attention: Autorun kernel profiling feature does not allow profiling individual kernels. The data for all autorun kernels in the design are read in a single attempt.

When temporal profiling is enabled, all autorun kernels in the design are profiled at the specified temporal period.

When temporal profiling is disabled, by default, the Profiler does not provide any profiling information since autorun kernels never finish. You can inform the Profiler when to profile by calling the host library function `clGetProfileDataDeviceIntelFPGA` to capture the autorun profiler data. This call can be made at any point during execution. For more about profiling using host API calls, refer to Profiling Enqueued and Autorun Kernels.

Note: The host API call works whether temporal is enabled or not, but it is required to get autorun profiling data when temporal is disabled.

Related Information

- Omit Communication Hardware between the Host and the Kernel
- Profiling Enqueued and Autorun Kernels
- Profile Data Acquisition

- Multiple Autorun Profiling Calls

5.6. Performance Data Types

The Intel FPGA dynamic profiler for OpenCL provides various types of performance data and information that you can view using the Intel VTune Profiler .

The following tables describe these information types:

Table 18. Types of Performance Data

Column	Description	Access Type
Attributes	Memory or channel attributes information such as memory type (local or global), corresponding memory system (DDR or quad data rate (QDR)), and read or write access.	All memory and channel accesses
Stall%	Percentage of time the memory or channel access is causing pipeline stalls. It is a measure of the ability of the memory or channel access to fulfill an access request.	All memory and channel accesses
Occupancy%	Percentage of the overall profiled time frame when a valid work-item executes the memory or channel instruction.	All memory and channel accesses
Bandwidth	Average memory bandwidth that the memory access uses and its overall efficiency. For each global memory access, FPGA resources are assigned to acquire data from the global memory system. However, the amount of data a kernel program uses might be less than the acquired data. The overall efficiency is the percentage of total bytes, acquired from the global memory system, that the kernel program uses.	Global memory accesses
Channel Depth[1]	Occupancy of the channel FIFO (in bytes) when the channel is not idling. This is measured in the following ways: • Average Channel Depth measures the average occupancy of the channel in the measured sample time-slice. • Maximum Channel Depth measures the fill level of the channel, indicating the maximum occupancy of the channel in the sample time-slice.	All channel accesses
Idle[1]	Percentage of the overall profiled time frame when there are no valid work item executing or stalling the memory or channel instruction.	All memory and channel accesses

Note: If your kernel undergoes memory optimization that consolidates hardware resources and implements multiple memory operations, statistical data might not be available for each memory operation. One set of statistical data maps to the point of consolidation in hardware.

5.7. Interpreting the Profiling Information

Profiling information helps you identify poor memory or channel behaviors that lead to unsatisfactory kernel performance.

The following sections explain the Intel FPGA dynamic profiler for OpenCL metrics that are displayed in various tabs of the **CPU/FPGA Interaction** view in the Intel VTune Profiler.

[1] Intel VTune Profiler will show this information in a future release.

Important: Profiling information that relates to the Intel FPGA SDK for OpenCL channels also applies to OpenCL pipes.

Stall, Occupancy, Bandwidth on page 115

Stalling Channels on page 116

Channel Depths on page 117

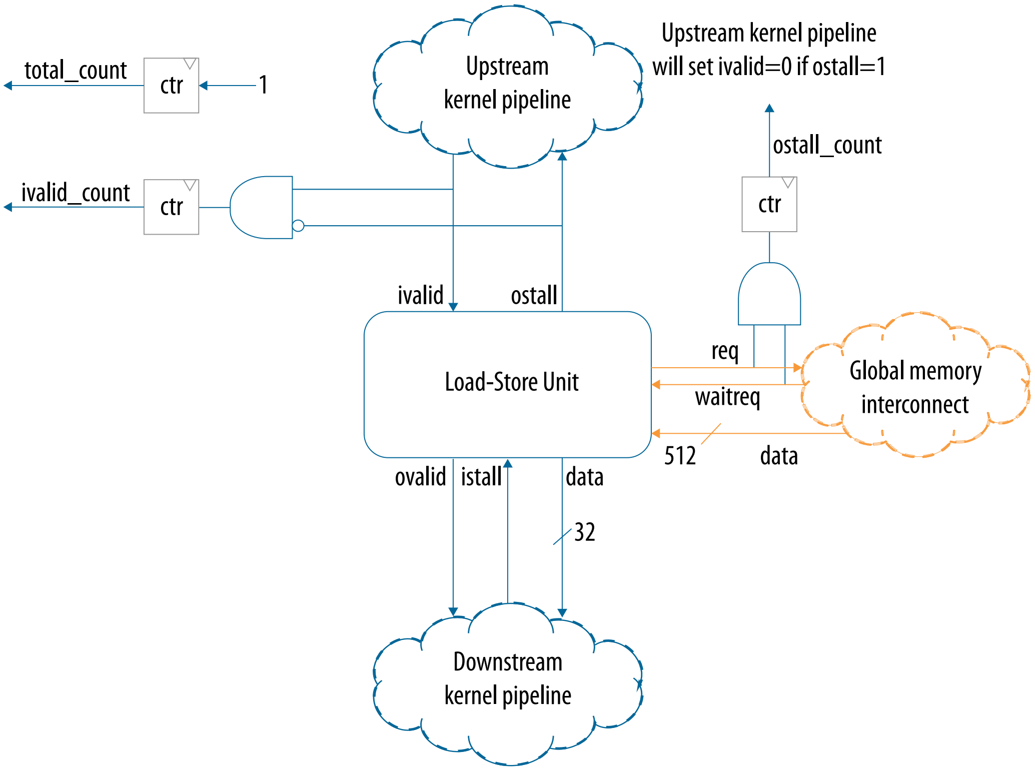
5.7.1. Stall, Occupancy, Bandwidth

For specific lines of kernel code, the **Source View** tab of the Intel VTune Profiler GUI shows stall percentage, occupancy percentage, data transfer size, and average memory bandwidth.

For definitions of stall, occupancy, and bandwidth, refer to Types of Performance Data.

The Intel FPGA SDK for OpenCL generates a pipeline architecture where work-items traverse through the pipeline stages sequentially (that is, in a pipeline-parallel manner). As soon as a pipeline stage becomes empty, a work-item enters and occupies the stage. Pipeline parallelism also applies to iterations of pipelined loops, where iterations enter a pipelined loop sequentially.

Figure 72. Simplified Representation of a Kernel Pipeline Instrumented with Performance Counters



The following are simplified equations that describe the Profiler calculates stall, occupancy, and bandwidth:

$$\text{Stall} = \frac{\text{ostall\_count}}{\text{total\_count}} \text{ x } 100\%$$

$$\text{Occupancy} = \frac{\text{ivalid\_count}}{\text{total\_count}} \text{ x } 100\%$$

$$\text{Bandwidth} = \frac{\text{data\_width x ivalid\_count}}{\text{kernel\_time}}$$

Note: `ivalid_count` in the bandwidth equation also includes the `predicate=true` input to the load-store unit.

Ideal kernel pipeline conditions:

- Stall percentage equals 0%
- Occupancy percentage equals 100%
- Bandwidth equals the board's bandwidth

For a given location in the kernel pipeline if the sum of the stall percentage and the occupancy percentage approximately equals 100%, the Profiler identifies the location as the stall source. If the stall percentage is low, the Profiler identifies the location as the victim of the stall.

The Profiler reports a high occupancy percentage if the offline compiler generates a highly efficient pipeline from your kernel, where work-items or iterations are moving through the pipeline stages without stalling.

If all LSUs are accessed the same number of times, they have the same occupancy value.

- If work-items cannot enter the pipeline consecutively, they insert bubbles into the pipeline.
- In loop pipelining, loop-carried dependencies also form bubbles in the pipeline because of bubbles that exist between iterations.
- If an LSU is accessed less frequently than other LSUs, such as the case when an LSU is outside a loop that contains other LSUs, this LSU has a lower occupancy value than the other LSUs.

The same rule regarding occupancy value applies to channels.

5.7.2. Stalling Channels

Channels provide a point-to-point communication link between either two kernels, or between a kernel and an I/O channel. If an I/O channel stalls, it implies that the I/O channel cannot keep up with the kernel.

For example, if a kernel has a read channel call to an Ethernet I/O and the Profiler identifies a stall, it implies that the write channel is not writing data to the Ethernet I/O at the same rate as the read rate of the kernel.

For kernel-to-kernel channels, stalls occur if there is an imbalance between the read and write sides of the channel, or if the read and write kernels are not running concurrently.

For example, if the kernel that reads is not launched concurrently with the kernel that writes, or if the read operations occur much slower than the write operations, the Profiler identifies a stall for the `write_channel_intel` call in the write kernel.

Related Information

Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes on page 92

5.7.3. Channel Depths

As mentioned in Stalling Channels on page 116, channels provide a communication link either between two kernels or between a kernel and an I/O channel. The channel depth counters complement the stall counts in explaining the issues that are causing the channel to stall.

- If a channel is continuously nearly empty, the read side of the channel is likely working faster than the write side, so the write side must be sped up. The channel depth can probably be reduced.
- If the channel is full, the write side is likely faster. The channel depth may need to be increased.
- In more complicated patterns, for example, if the average depth of the channel is far lower than the maximum depth, the write side might be writing a lot of a data in a single attempt and overwhelming the read side but writing slowly the rest of the time. A repetition of this pattern can create bubbles in the pipeline without creating a long stall, so it can be useful to track down using the channel depth counters.

Note: The depth of the channel cannot be directly controlled since the compiler optimizes the channel for better use of area resources and always rounded up based on the requested channel size. So, the final channel depth is approximately 32 bytes, 512 bytes, or a multiple of 1024 bytes.

5.8. Profiler Analyses of Example OpenCL Design Scenarios

Understanding the problems and solutions presented in example OpenCL design scenarios might help you use the Profiler metrics of your design to optimize its performance.

5.8.1. High Stall Percentage

A high stall percentage implies that the memory or channel instruction is unable to fulfill the access request because of contention for memory bandwidth or channel buffer space.

Memory instructions stall often whenever bandwidth usage is inefficient or if a large amount of data transfer is necessary during the execution of your application. Inefficient memory accesses lead to suboptimal bandwidth utilization. In such cases, analyze your kernel memory accesses for possible improvements.

Channel instructions stall whenever there is a strong imbalance between read and write accesses to the channel. Imbalances might be caused by channel reads or writes operating at different rates.

For example, if you find that the stall percentage of a write channel call is high, check to see if the occupancy and activity of the read channel call are low. If they are, the performing speed of the kernel controlling the read channel call is too slow for the kernel controlling the write channel call, leading to a performance bottleneck.

Related Information

Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes on page 92

5.8.2. Low Occupancy Percentage

A low occupancy percentage implies that a work-item is accessing the load and store operations or the channel infrequently. This behavior is expected for load and store operations or channels that are in non-critical loops. However, if the memory or channel instruction is in critical portions of the kernel code and the occupancy or activity percentage is low, it implies that a performance bottleneck exists because work-items or loop iterations are not being issued in the hardware.

Consider the following code example:

```
__kernel void proc (__global int * a, ...) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < 1000; j++) {
      write_channel_intel (c0, data0);
    }
    for (int k = 0; k < 3; k++) {
      write_channel_intel (c1, data1);
    }
  }
}
```

Assuming all the loops are pipelined, the first inner loop with a trip count of 1000 is the critical loop. The second inner loop with a trip count of three is executed infrequently. As a result, you can expect that the occupancy and activity percentages for channel `c0` are high and for channel `c1` are low.

Also, occupancy percentage might be low if you define a small work-group size, the kernel might not receive sufficient work-items. This is problematic because the pipeline is empty generally for the duration of kernel execution, which leads to poor performance.

5.8.3. High Stall and High Occupancy Percentages

A load and store operation or channel with a high stall percentage is the cause of the kernel pipeline stall.

Remember: An ideal kernel pipeline condition has a stall percentage of 0% and an occupancy percentage of 100%.

Usually, the sum of the stall and occupancy percentages approximately equals 100%. If a load and store operation or channel has a high stall percentage, it means that the load and store operation or channel has the ability to execute every cycle but is generating stalls.

Solutions for stalling global load and store operations:

- Use local memory to cache data.
- Reduce the number of times you read the data.
- Improve global memory accesses.
  - Change the access pattern for more global-memory-friendly addressing (for example, change from stride accessing to sequential accessing).
  - Compile your kernel with the `-no-interleaving=default` Intel FPGA SDK for OpenCL Offline Compiler command option, and separate the read and write buffers into different DDR banks.
  - Have fewer but wider global memory accesses.
- Acquire an accelerator board that has more bandwidth (for example, a board with three DDRs instead of 2 DDRs).

Solution for stalling local load and store operations:

- Review the HTML area report to verify the local memory configuration and modify the configuration to make it stall-free.

Solutions for stalling channels:

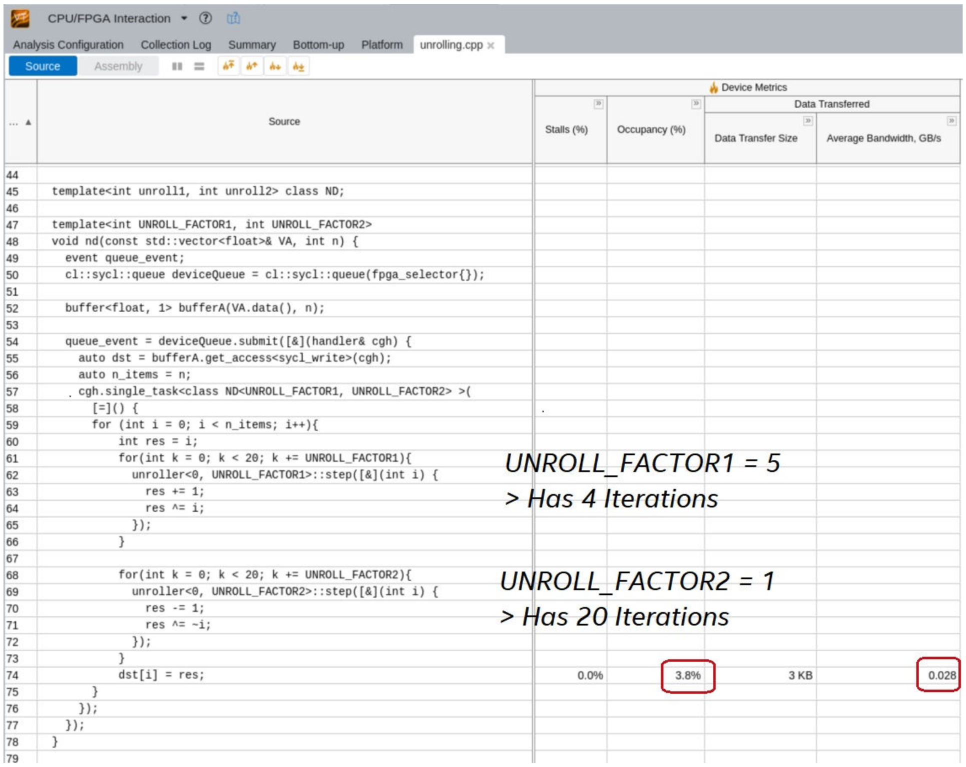
- Fix stalls on the other side of the channel. For example, if channel read stalls, it means that the writer to the channel is not writing data into the channel fast enough and needs to be adjusted.
- If there are channel loops in your design, specify the channel depth.

5.8.4. No Stalls, Low Occupancy Percentage, and Low Bandwidth

Loop-carried dependencies might create a bottleneck in your design that causes a low occupancy percentage and a low bandwidth.

Remember: An ideal kernel pipeline condition has a stall percentage of 0%, an occupancy percentage of 100%, and a bandwidth that equals the board's available bandwidth.

Figure 73. Example OpenCL Kernel and Profiler Analysis



```
template<int unroll1, int unroll2> class ND;

template<int UNROLL_FACTOR1, int UNROLL_FACTOR2>
void nd(const std::vector<float>& VA, int n) {
  event queue_event;
  cl::sycl::queue deviceQueue = cl::sycl::queue(fpga_selector{});

  buffer<float, 1> bufferA(VA.data(), n);

  queue_event = deviceQueue.submit([&](handler& cgh) {
    auto dst = bufferA.get_access<sycl_write>(cgh);
    auto n_items = n;
    cgh.single_task<class ND<UNROLL_FACTOR1, UNROLL_FACTOR2> >(
      [=]() {
      for (int i = 0; i < n_items; i++){
          int res = i;
          for(int k = 0; k < 20; k += UNROLL_FACTOR1){
            unroller<0, UNROLL_FACTOR1>::step([&](int i) {
              res += 1;
              res ^= i;
            });
          }

          for(int k = 0; k < 20; k += UNROLL_FACTOR2){
            unroller<0, UNROLL_FACTOR2>::step([&](int i) {
              res -= 1;
              res ^= ~i;
            });
          }
          dst[i] = res;
      }
    });
  });
}
```

UNROLL_FACTOR1 = 5
> Has 4 Iterations

UNROLL_FACTOR2 = 1
> Has 20 Iterations

Source	Stalls (%)	Occupancy (%)	Data Transfer Size	Average Bandwidth, GB/s
dst[i] = res;	0.0%	3.8%	3 KB	0.028

In this example, `dst[]` is executed once every 20 iterations of the FACTOR2 loop and once every four iterations of the FACTOR1 loop. Therefore, FACTOR2 loop is the source of the bottleneck.

Solutions for resolving loop bottlenecks:

- Unroll the FACTOR1 and FACTOR2 loops evenly. Simply unrolling FACTOR2 loop further does not resolve the bottleneck.
- Vectorize your kernel to allow multiple work-items to execute during each loop iteration.

Related Information

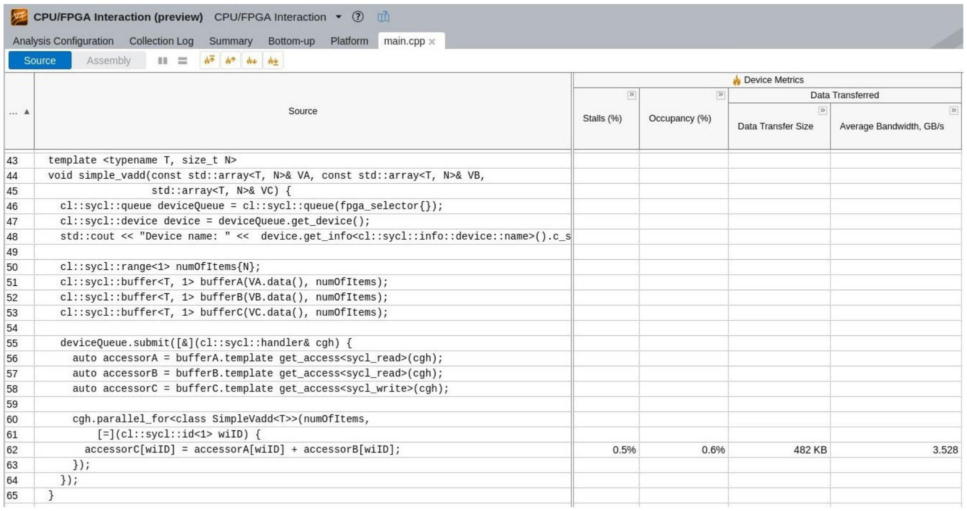
Kernel Vectorization on page 139

5.8.5. No Stalls, High Occupancy Percentage, and Low Bandwidth

The structure of a kernel design might prevent it from leveraging all the available bandwidth that the accelerator board can offer.

Remember: An ideal kernel pipeline condition has a stall percentage of 0%, an occupancy percentage of 100%, and a bandwidth that equals the board's available bandwidth.

Figure 74. Example OpenCL Kernel and Profiler Analysis



		Device Metrics			
				Data Transferred	
... ▲	Source	Stalls (%)	Occupancy (%)	Data Transfer Size	Average Bandwidth, GB/s
43	template <typename T, size_t N>				
44	void simple_vadd(const std::array<T, N>& VA, const std::array<T, N>& VB,				
45	std::array<T, N>& VC) {				
46	cl::sycl::queue deviceQueue = cl::sycl::queue(fpga_selector{});				
47	cl::sycl::device device = deviceQueue.get_device();				
48	std::cout << "Device name: " << device.get_info<cl::sycl::info::device::name>().c_s				
49					
50	cl::sycl::range<1> numOfItems{N};				
51	cl::sycl::buffer<T, 1> bufferA(VA.data(), numOfItems);				
52	cl::sycl::buffer<T, 1> bufferB(VB.data(), numOfItems);				
53	cl::sycl::buffer<T, 1> bufferC(VC.data(), numOfItems);				
54					
55	deviceQueue.submit([&](cl::sycl::handler& cgh) {				
56	auto accessorA = bufferA.template get_access<sycl_read>(cgh);				
57	auto accessorB = bufferB.template get_access<sycl_read>(cgh);				
58	auto accessorC = bufferC.template get_access<sycl_write>(cgh);				
59					
60	cgh.parallel_for<class SimpleVadd<T>>(numOfItems,				
61	[=](cl::sycl::id<1> wiID) {				
62	accessorC[wiID] = accessorA[wiID] + accessorB[wiID];	0.5%	0.6%	482 KB	3.528
63	});				
64	});				
65	}				

In this example, the accelerator board can provide a bandwidth of 25600 megabytes per second (MB/s). However, the `vector_add` kernel is requesting (2 reads + 1 write) x 4 bytes x 294 MHz = 12 bytes/cycle x 294 MHz = 3528 GB/s, which is 14% of the available bandwidth. To increase the bandwidth, increase the number of tasks performed in each clock cycle.

Solutions for low bandwidth:

- Automatically or manually vectorize the kernel to make *wider* requests
- Unroll the innermost loop to make more requests per clock cycle
- Delegate some of the tasks to another kernel

5.8.6. High Stall and Low Occupancy Percentages

There might be situations where a global store operation might have a high stall percentage (for example, 30%) and a very low occupancy percentage (for example, 0.01%). If such a store operation happens once every 10000 cycles of computation, the efficiency of this store is not a cause for concern.

5.9. Intel FPGA Dynamic Profiler for OpenCL Limitations

The Intel FPGA dynamic profiler for OpenCL has some limitations.

- Profile data is not persistent across OpenCL programs or multiple devices.

  You can request profile data from a single OpenCL program and on a single device only. If your host swaps a new kernel program in and out of the FPGA, the Profiler does not save the profile data.

- All profiling data is read to the host during execution and is only stored on the device long enough to be read on the next readback. Any reprogram of new designs or restarting the same design results in new profiling data, erasing any previous data that may have existed.

- Instrumenting the Verilog code with performance counters increases hardware resource utilization (that is, FPGA area usage) and typically decreases performance.

  For information on instrumenting the Verilog code with performance counters, refer to the *Instrumenting the Kernel Pipeline with Performance Counters* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

- Collecting Profile Data During Kernel Execution
- Instrumenting the Kernel Pipeline with Performance Counters (-profile) on page 109

6. Strategies for Improving Single Work-Item Kernel Performance

Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 123

Good Design Practices for Single Work-Item Kernel on page 134

6.1. Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback

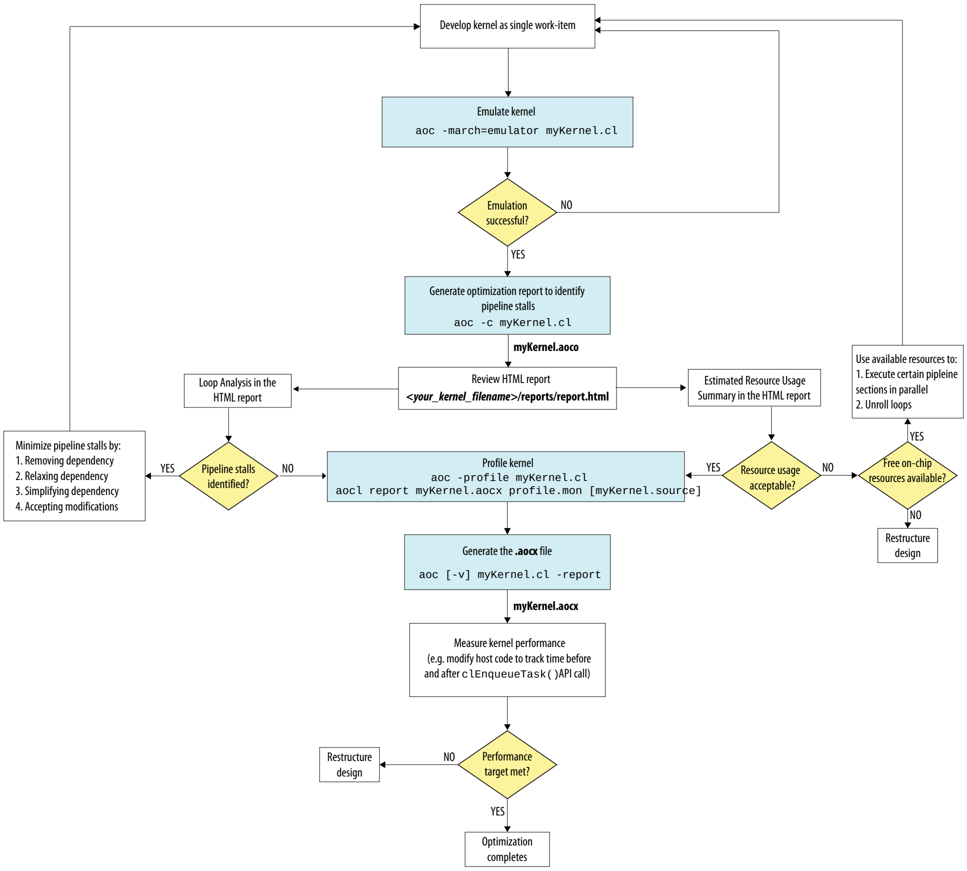
In many cases, designing your OpenCL application as a single work-item kernel is sufficient to maximize performance without performing additional optimization steps. To further improve the performance of your single work-item kernel, you can optimize it by addressing dependencies that the optimization report identifies.

Tip: If you are looking for Intel oneAPI DPC++/C++ Compiler-specific details, refer to Single Work-item Kernels section in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

The following flowchart outlines the approach you can take to iterate on your design and optimize your single work-item kernel. For usage information on the Intel FPGA SDK for OpenCL Emulator and the Profiler, refer to the *Emulating and Debugging Your OpenCL Kernel* and *Profiling Your OpenCL Kernel* sections of the *Intel FPGA SDK for OpenCL Programming Guide*, respectively. For information on the Intel FPGA dynamic profiler for OpenCL GUI and profiling information, refer to the *Profile Your Kernel to Identify Performance Bottlenecks* section.

Intel recommends the following optimization options to address single work-item kernel loop-carried dependencies, in order of applicability: removal, relaxation, simplification, and transfer to local memory.

Figure 75. Optimization Work Flow of a Single Work-Item Kernel


Develop kernel as single work-item
Emulate kernel
aoc -march=emulator myKernel.cl
Emulation successful?
NO
YES
Generate optimization report to identify pipeline stalls
aoc -c myKernel.cl
myKernel.aoco
Review HTML report
<your_kernel_filename>/reports/report.html
Loop Analysis in the HTML report
Estimated Resource Usage Summary in the HTML report
Use available resources to:
1. Execute certain pipleine sections in parallel
2. Unroll loops
YES
Minimize pipeline stalls by:
1. Removing dependency
2. Relaxing dependency
3. Simplifying dependency
4. Accepting modifications
YES
Pipeline stalls identified?
NO
Profile kernel
aoc -profile myKernel.cl
aocl report myKernel.aocx profile.mon [myKernel.source]
YES
Resource usage acceptable?
NO
Free on-chip resources available?
NO
Restructure design
Generate the .aocx file
aoc [-v] myKernel.cl -report
myKernel.aocx
Measure kernel performance
(e.g. modify host code to track time before and after clEnqueueTask( )API call)
Restructure design
NO
Performance target met?
YES
Optimization completes

1. Removing Loop-Carried Dependency on page 124
2. Relaxing Loop-Carried Dependency on page 127
3. Transferring Loop-Carried Dependency to Local Memory on page 129
4. Relaxing Loop-Carried Dependency by Inferring Shift Registers on page 130
5. Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays on page 132

Related Information

- Emulating and Debugging Your OpenCL Kernel
- Profiling Your Kernel to Identify Performance Bottlenecks on page 108

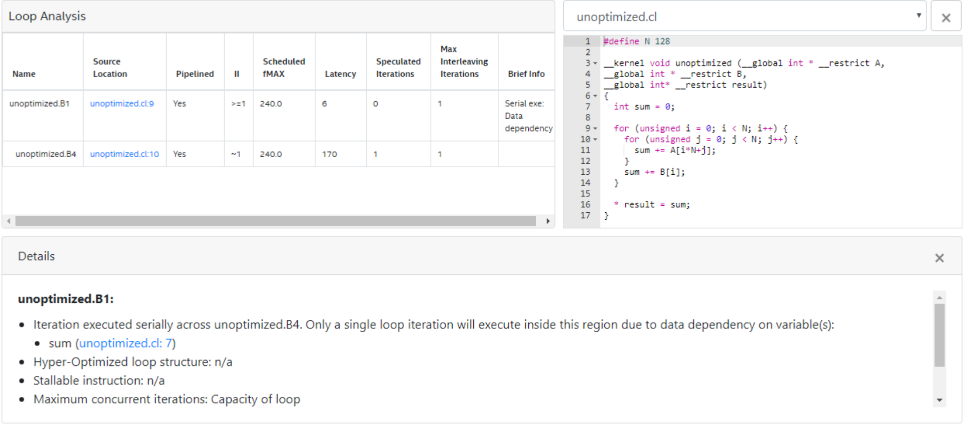
6.1.1. Removing Loop-Carried Dependency

Based on the feedback from the optimization report, you can remove a loop-carried dependency by implementing a simpler memory access pattern.

Consider the following kernel:

```
#define N 128

__kernel void unoptimized (__global int * restrict A,
                             __global int * restrict B,
                             __global int* restrict result)
{
  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];
  }

  * result = sum;
}
```

The optimization report for kernel `unoptimized` resembles the following:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
unoptimized.B1	unoptimized.cl:9	Yes	>=1	240.0	6	0	1	Serial exe: Data dependency
unoptimized.B4	unoptimized.cl:10	Yes	~1	240.0	170	1	1	

unoptimized.cl

```
#define N 128

__kernel void unoptimized (__global int * __restrict A,
__global int * __restrict B,
__global int* __restrict result)
{
  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];
  }

  * result = sum;
}
```

Details

unoptimized.B1:

- Iteration executed serially across unoptimized.B4. Only a single loop iteration will execute inside this region due to data dependency on variable(s):
    - sum (unoptimized.cl: 7)
- Hyper-Optimized loop structure: n/a
- Stallable instruction: n/a
- Maximum concurrent iterations: Capacity of loop

- The first row of the report indicates that the Intel FPGA SDK for OpenCL Offline Compiler successfully infers pipelined execution for the outer loop, and a new loop iteration launches every other cycle.
- The message `due to Pipeline structure` indicates that the offline compiler creates a pipeline structure that causes an outer loop iteration to launch every two cycles. The behavior is not a result of how you structure your kernel code.

  Note: For recommendations on how to structure your single work-item kernel, refer to the *Good Design Practices for Single Work-Item Kernel* section.
- The remaining messages in the first row of report indicate that the loop executes a single iteration at a time across the subloop because of data dependency on the variable *sum*. This data dependency exists because each outer loop iteration requires the value of *sum* from the previous iteration to return before the inner loop can start executing.
- The second row of the report notifies you that the inner loop executes in a pipelined fashion with no performance-limiting loop-carried dependencies.

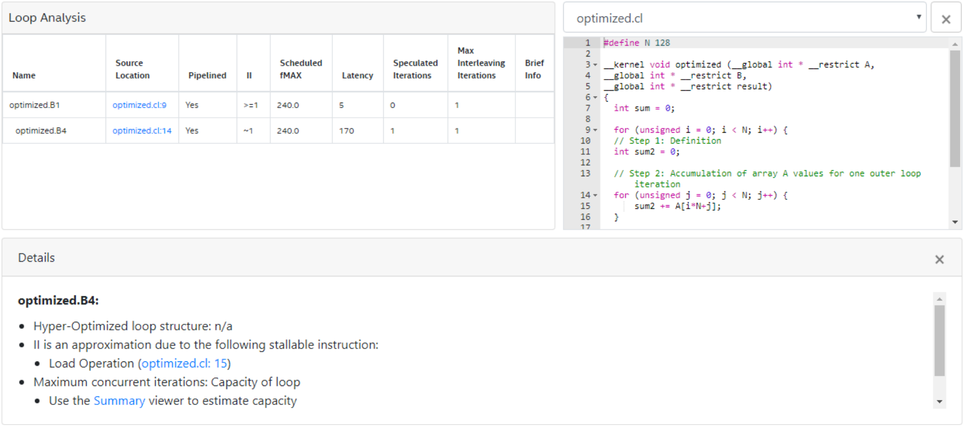
To optimize the performance of this kernel, remove the data dependency on variable *sum* so that the outer loop iterations do not execute serially across the subloop. Perform the following tasks to decouple the computations involving *sum* in the two loops:

1. Define a local variable (for example, *sum2*) for use in the inner loop only.
2. Use the local variable from Step 1 to store the cumulative values of `A[i*N + j]` as the inner loop iterates.
3. In the outer loop, store the variable *sum* to store the cumulative values of `B[i]` and the value stored in the local variable.

Below is the restructured kernel `optimized`:

```
#define N 128

__kernel void optimized (__global int * restrict A,
                         __global int * restrict B,
                         __global int * restrict result)
{
  int sum = 0;

  for (unsigned i = 0; i < N; i++) {
    // Step 1: Definition
    int sum2 = 0;

    // Step 2: Accumulation of array A values for one outer loop iteration
    for (unsigned j = 0; j < N; j++) {
      sum2 += A[i*N+j];
    }

    // Step 3: Addition of array B value for an outer loop iteration
    sum += sum2;
    sum += B[i];
  }

  * result = sum;
}
```

An optimization report similar to the one below indicates the successful removal of the loop-carried dependency on the variable `sum`:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
optimized.B1	optimized.cl:9	Yes	>=1	240.0	5	0	1	
optimized.B4	optimized.cl:14	Yes	~1	240.0	170	1	1	

optimized.cl

```
1   #define N 128
2
3   __kernel void optimized (__global int * __restrict A,
4   __global int * __restrict B,
5   __global int * __restrict result)
6   {
7     int sum = 0;
8
9     for (unsigned i = 0; i < N; i++) {
10    // Step 1: Definition
11    int sum2 = 0;
12
13    // Step 2: Accumulation of array A values for one outer loop
          iteration
14    for (unsigned j = 0; j < N; j++) {
15        sum2 += A[i*N+j];
16    }
17
```

Details

optimized.B4:

- Hyper-Optimized loop structure: n/a
- II is an approximation due to the following stallable instruction:
  - Load Operation (optimized.cl: 15)
- Maximum concurrent iterations: Capacity of loop
  - Use the Summary viewer to estimate capacity

You have addressed all the loop-carried dependence issues successfully when you see only the following messages in the optimization report:

- `Pipelined execution inferred` for innermost loops.
- `Pipelined execution inferred. Successive iterations launched every 2 cycles due to: Pipeline structure` for all other loops.

Related Information

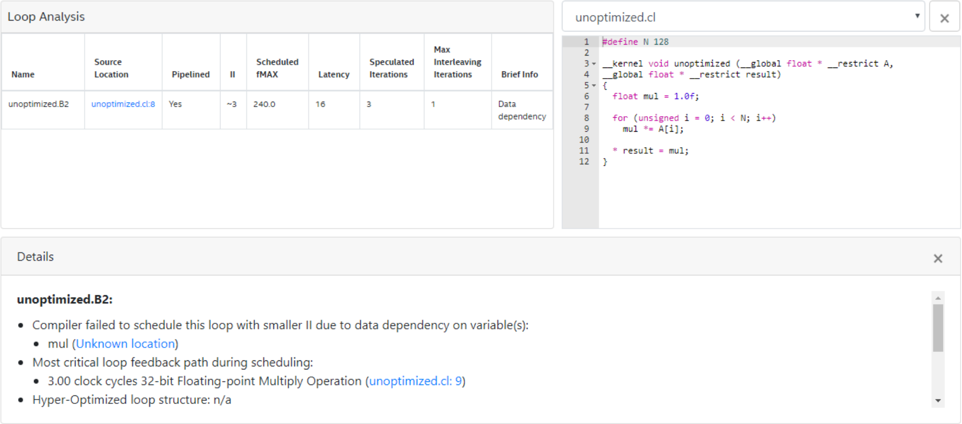
Good Design Practices for Single Work-Item Kernel on page 134

6.1.2. Relaxing Loop-Carried Dependency

Based on the feedback from the optimization report, you can relax a loop-carried dependency by increasing the dependence distance.Increase the dependence distance by increasing the number of loop iterations that occurs between the generation of a loop-carried value and its usage.

Consider the following code example:

```
#define N 128

__kernel void unoptimized (__global float * restrict A,
                           __global float * restrict result)
{
  float mul = 1.0f;

  for (unsigned i = 0; i < N; i++)
    mul *= A[i];

  * result = mul;
}
```



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
unoptimized.B2	unoptimized.cl:8	Yes	~3	240.0	16	3	1	Data dependency

unoptimized.cl

```
1  #define N 128
2
3  __kernel void unoptimized (__global float * __restrict A,
4  __global float * __restrict result)
5  {
6    float mul = 1.0f;
7
8    for (unsigned i = 0; i < N; i++)
9      mul *= A[i];
10
11   * result = mul;
12 }
```

Details

unoptimized.B2:

- Compiler failed to schedule this loop with smaller II due to data dependency on variable(s):
  - mul (Unknown location)
- Most critical loop feedback path during scheduling:
  - 3.00 clock cycles 32-bit Floating-point Multiply Operation (unoptimized.cl: 9)
- Hyper-Optimized loop structure: n/a

The optimization report above shows that the Intel FPGA SDK for OpenCL Offline Compiler infers pipelined execution for the loop successfully. However, the loop-carried dependency on the variable *mul* causes loop iterations to launch every six cycles. In this case, the floating-point multiplication operation on line 9 (that is, `mul *= A[i]`) contributes the largest delay to the computation of the variable *mul*.

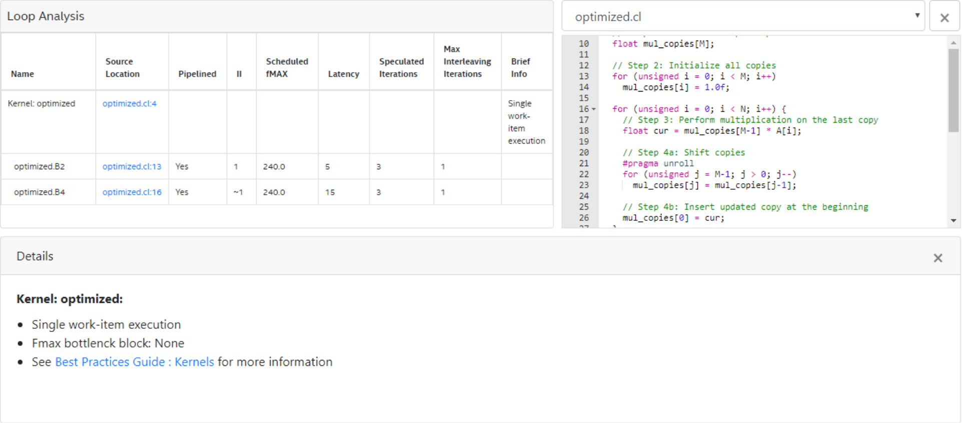
To relax the loop-carried data dependency, instead of using a single variable to store the multiplication results, operate on *M* copies of the variable and use one copy every *M* iterations:

1. Declare multiple copies of the variable *mul* (for example, in an array called *mul_copies*).
2. Initialize all the copies of *mul_copies*.
3. Use the last copy in the array in the multiplication operation.
4. Perform a shift operation to pass the last value of the array back to the beginning of the shift register.
5. Reduce all the copies to *mul* and write the final value to *result*.

Below is the restructured kernel:

```
#define N 128
#define M 8

__kernel void optimized (__global float * restrict A,
                         __global float * restrict result)
{
  float mul = 1.0f;

  // Step 1: Declare multiple copies of variable mul
  float mul_copies[M];

  // Step 2: Initialize all copies
  for (unsigned i = 0; i < M; i++)
    mul_copies[i] = 1.0f;

  for (unsigned i = 0; i < N; i++) {
    // Step 3: Perform multiplication on the last copy
    float cur = mul_copies[M-1] * A[i];

    // Step 4a: Shift copies
    #pragma unroll
    for (unsigned j = M-1; j > 0; j--)
      mul_copies[j] = mul_copies[j-1];

    // Step 4b: Insert updated copy at the beginning
    mul_copies[0] = cur;
  }

  // Step 5: Perform reduction on copies
  #pragma unroll
  for (unsigned i = 0; i < M; i++)
    mul *= mul_copies[i];

  * result = mul;
}
```

An optimization report similar to the one below indicates the successful relaxation of the loop-carried dependency on the variable *mul*:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
Kernel: optimized	optimized.cl:4							Single work-item execution
optimized.B2	optimized.cl:13	Yes	1	240.0	5	3	1	
optimized.B4	optimized.cl:16	Yes	~1	240.0	15	3	1	

optimized.cl

```
  float mul_copies[M];

  // Step 2: Initialize all copies
  for (unsigned i = 0; i < M; i++)
    mul_copies[i] = 1.0f;

  for (unsigned i = 0; i < N; i++) {
    // Step 3: Perform multiplication on the last copy
    float cur = mul_copies[M-1] * A[i];

    // Step 4a: Shift copies
    #pragma unroll
    for (unsigned j = M-1; j > 0; j--)
      mul_copies[j] = mul_copies[j-1];

    // Step 4b: Insert updated copy at the beginning
    mul_copies[0] = cur;
```

Details

Kernel: optimized:

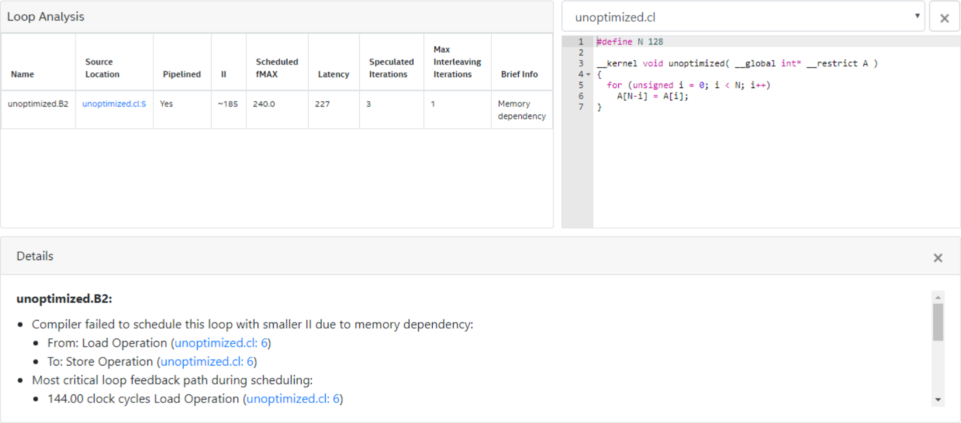
- Single work-item execution
- Fmax bottlenck block: None
- See Best Practices Guide : Kernels for more information

6.1.3. Transferring Loop-Carried Dependency to Local Memory

For a loop-carried dependency that you cannot remove, improve the II by moving the array with the loop-carried dependency from global memory to local memory.

Consider the following kernel example:

```
#define N 128

__kernel void unoptimized( __global int* restrict A )
{
    for (unsigned i = 0; i < N; i++)
          A[N-i] = A[i];
}
```



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
unoptimized.B2	unoptimized.cl:5	Yes	~185	240.0	227	3	1	Memory dependency

unoptimized.cl

```
#define N 128

__kernel void unoptimized( __global int* __restrict A )
{
  for (unsigned i = 0; i < N; i++)
    A[N-i] = A[i];
}
```

Details

unoptimized.B2:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation (unoptimized.cl: 6)
  - To: Store Operation (unoptimized.cl: 6)
- Most critical loop feedback path during scheduling:
  - 144.00 clock cycles Load Operation (unoptimized.cl: 6)

Global memory accesses have long latencies. In this example, the loop-carried dependency on the array `A[i]` causes the long latency. This latency is reflected by an II of 227 in the optimization report. To reduce the II value by transferring the loop-carried dependency from global memory to local memory, perform the following tasks:

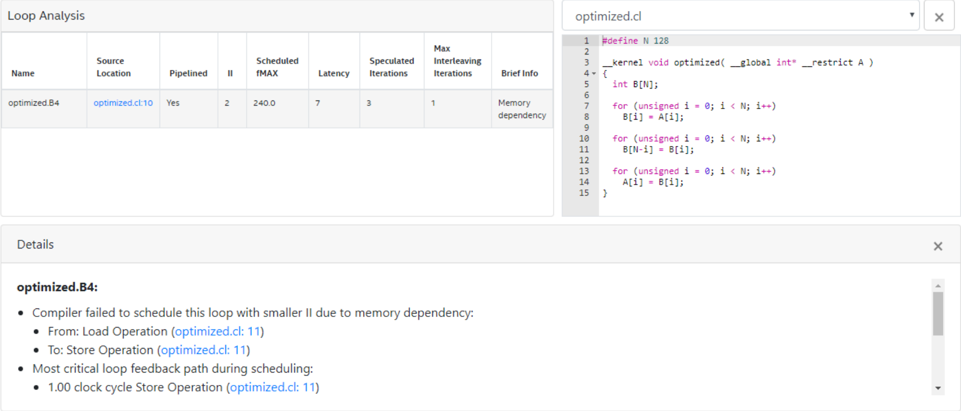
1. Copy the array with the loop-carried dependency to local memory. In this example, array `A[i]` becomes array `B[i]` in local memory.
2. Execute the loop with the loop-carried dependence on array `B[i]`.
3. Copy the array back to global memory.

When you transfer array `A[i]` to local memory and it becomes array `B[i]`, the loop-carried dependency is now on `B[i]`. Because local memory has a much lower latency than global memory, the II value improves.

Below is the restructured kernel optimized:

```
 1 #define N 128
 2
 3 __kernel void optimized( __global int* restrict A )
 4 {
 5     int B[N];
 6
 7     for (unsigned i = 0; i < N; i++)
 8         B[i] = A[i];
 9
10     for (unsigned i = 0; i < N; i++)
11         B[N-i] = B[i];
12
13     for (unsigned i = 0; i < N; i++)
14         A[i] = B[i];
15 }
```

An optimization report similar to the one below indicates the successful reduction of II from 227 to 2:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
optimized.B4	optimized.cl:10	Yes	2	240.0	7	3	1	Memory dependency

optimized.cl

```
 1 #define N 128
 2
 3 __kernel void optimized( __global int* __restrict A )
 4 {
 5   int B[N];
 6
 7   for (unsigned i = 0; i < N; i++)
 8     B[i] = A[i];
 9
10   for (unsigned i = 0; i < N; i++)
11     B[N-i] = B[i];
12
13   for (unsigned i = 0; i < N; i++)
14     A[i] = B[i];
15 }
```

Details

optimized.B4:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation (optimized.cl: 11)
  - To: Store Operation (optimized.cl: 11)
- Most critical loop feedback path during scheduling:
  - 1.00 clock cycle Store Operation (optimized.cl: 11)

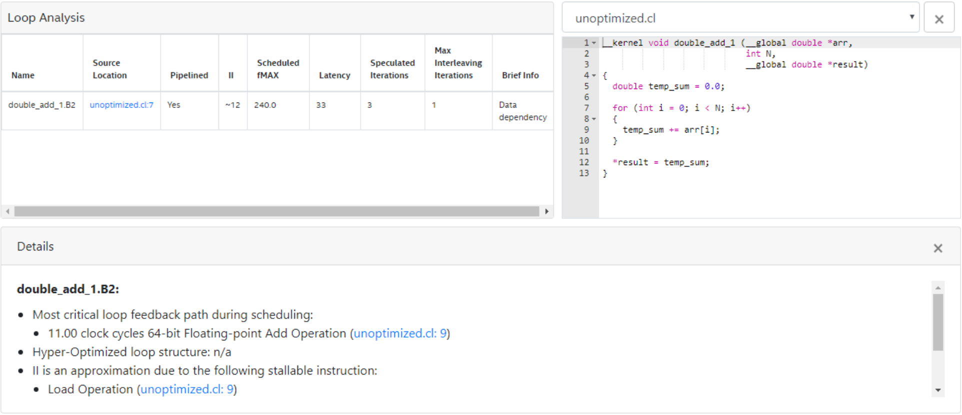
6.1.4. Relaxing Loop-Carried Dependency by Inferring Shift Registers

To enable the Intel FPGA SDK for OpenCL Offline Compiler to handle single work-item kernels that carry out double precision floating-point operations efficiently, remove loop-carried dependencies by inferring a shift register.

Consider the following kernel:

```
__kernel void double_add_1 (__global double *arr,
                            int N,
                            __global double *result)
{
  double temp_sum = 0;

  for (int i = 0; i < N; ++i)
  {
      temp_sum += arr[i];
  }

  *result = temp_sum;
}
```

The optimization report for kernel `unoptimized` resembles the following:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
double_add_1.B2	unoptimized.cl:7	Yes	~12	240.0	33	3	1	Data dependency

unoptimized.cl

```
__kernel void double_add_1 (__global double *arr,
                            int N,
                            __global double *result)
{
  double temp_sum = 0.0;

  for (int i = 0; i < N; i++)
  {
    temp_sum += arr[i];
  }

  *result = temp_sum;
}
```

Details

double_add_1.B2:

- Most critical loop feedback path during scheduling:
  - 11.00 clock cycles 64-bit Floating-point Add Operation (unoptimized.cl: 9)
- Hyper-Optimized loop structure: n/a
- II is an approximation due to the following stallable instruction:
  - Load Operation (unoptimized.cl: 9)

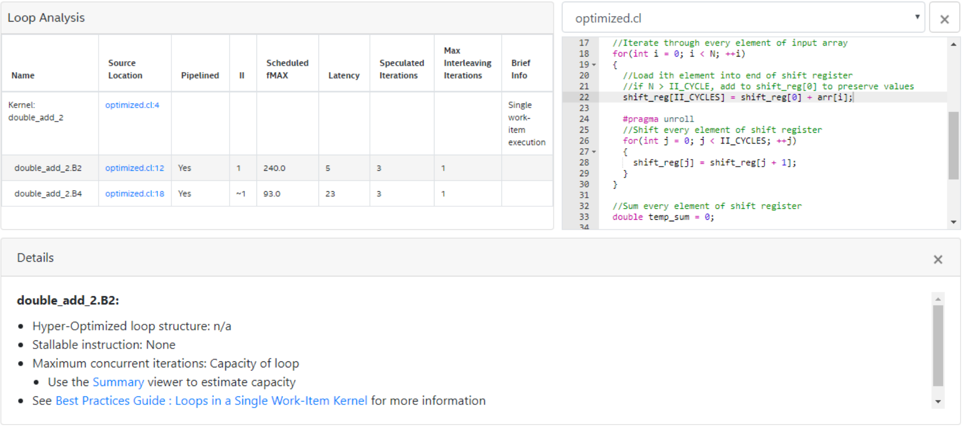
The kernel unoptimized is an accumulator that sums the elements of a double precision floating-point array `arr[i]`. For each loop iteration, the offline compiler takes 11 cycles to compute the result of the addition and then stores it in the variable *temp_sum*. Each loop iteration requires the value of *temp_sum* from the previous loop iteration, which creates a data dependency on *temp_sum*.

- To relax the data dependency, infer the array `arr[i]` as a shift register.

Below is the restructured kernel `optimized`:

```
//Shift register size must be statically determinable
#define II_CYCLES 12

__kernel void double_add_2 (__global double *arr,
                            int N,
                            __global double *result)
{
    //Create shift register with II_CYCLE+1 elements
    double shift_reg[II_CYCLES+1];

    //Initialize all elements of the register to 0
    for (int i = 0; i < II_CYCLES + 1; i++)
    {
        shift_reg[i] = 0;
    }

    //Iterate through every element of input array
    for(int i = 0; i < N; ++i)
    {
        //Load ith element into end of shift register
        //if N > II_CYCLE, add to shift_reg[0] to preserve values
        shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

        #pragma unroll
        //Shift every element of shift register
        for(int j = 0; j < II_CYCLES; ++j)
        {
            shift_reg[j] = shift_reg[j + 1];
        }
    }

    //Sum every element of shift register
    double temp_sum = 0;

    #pragma unroll
    for(int i = 0; i < II_CYCLES; ++i)
    {
        temp_sum += shift_reg[i];
    }

    *result = temp_sum;
}
```

The following optimization report indicates that the inference of the shift register `shift_reg[II_CYCLES]` successfully removes the data dependency on the variable *temp_sum*:



Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
Kernel: double_add_2	optimized.cl:4							Single work-item execution
double_add_2.B2	optimized.cl:12	Yes	1	240.0	5	3	1	
double_add_2.B4	optimized.cl:18	Yes	~1	93.0	23	3	1	

optimized.cl

```
  //Iterate through every element of input array
  for(int i = 0; i < N; ++i)
  {
    //Load ith element into end of shift register
    //if N > II_CYCLE, add to shift_reg[0] to preserve values
    shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

    #pragma unroll
    //Shift every element of shift register
    for(int j = 0; j < II_CYCLES; ++j)
    {
      shift_reg[j] = shift_reg[j + 1];
    }
  }

  //Sum every element of shift register
  double temp_sum = 0;
```

Details

double_add_2.B2:

- Hyper-Optimized loop structure: n/a
- Stallable instruction: None
- Maximum concurrent iterations: Capacity of loop
  - Use the Summary viewer to estimate capacity
- See Best Practices Guide : Loops in a Single Work-Item Kernel for more information

6.1.5. Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays

Include the `ivdep` pragma in your single work-item kernel to assert that accesses to memory arrays do not cause loop-carried dependencies.

During compilation, the Intel FPGA SDK for OpenCL Offline Compiler creates hardware that ensures load and store instructions operate within dependency constraints. An example of a dependency constraint is that dependent load and store instructions must execute in order. The presence of the `ivdep` pragma instructs the offline compiler to remove this extra hardware between load and store instructions in the loop that immediately follows the pragma declaration in the kernel code. Removing the extra hardware might reduce logic utilization and lower the II value in single work-item kernels.

You can provide more information about loop dependencies by adding the `safelen(N)` clause to the `ivdep` pragma. The `safelen(N)` clause specifies the maximum number of consecutive loop iterations without loop-carried dependencies.

For example, #pragma ivdep safelen(32) indicates to the compiler that there are a maximum of 32 iterations of the loop before loop-carried dependencies might be introduced. That is, while #pragma ivdep promises that there are no implicit memory dependency between any iteration of this loop, #pragma safelen(32) promises that the iteration that is 32 iterations away is the closest iteration that could be dependent on this iteration.

- If all accesses to memory arrays that are inside a loop do not cause loop-carried dependencies, add the line #pragma ivdep before the loop in your kernel code.

  Example kernel code:

  ```
  // no loop-carried dependencies for A and B array accesses
  #pragma ivdep
  for (int i = 0; i < N; i++) {
      A[i] = A[i - X[i]];
      B[i] = B[i - Y[i]];
  }
  ```

- To specify that accesses to a particular memory array inside a loop does not cause loop-carried dependencies, add the line #pragma ivdep array (*array_name*) before the loop in your kernel code.

  The array specified by the ivdep pragma must be a local or private memory array, or a pointer variable that points to a global, local, or private memory storage. If the specified array is a pointer, the ivdep pragma also applies to all arrays that may alias with specified pointer.

  The array specified by the ivdep pragma can also be an array or a pointer member of a struct.

  Example kernel code:

  ```
  // No loop-carried dependencies for A array accesses
  // The offline compiler will insert hardware that reinforces dependency
  constraints for B
  #pragma ivdep array(A)
  for (int i = 0; i < N; i++) {
      A[i] = A[i - X[i]];
      B[i] = B[i - Y[i]];
  }

  // No loop-carried dependencies for array A inside struct
  #pragma ivdep array(S.A)
  for (int i = 0; i < N; i++) {
      S.A[i] = S.A[i - X[i]];
  }

  // No loop-carried dependencies for array A inside the struct pointed by S
  #pragma ivdep array(S->X[2][3].A)
  for (int i = 0; i < N; i++) {
      S->X[2][3].A[i] = S.A[i - X[i]];
  }

  // No loop-carried dependencies for A and B because ptr aliases
  // with both arrays
  int *ptr = select ? A : B;
  #pragma ivdep array(ptr)
  for (int i = 0; i < N; i++) {
      A[i] = A[i - X[i]];
      B[i] = B[i - Y[i]];
  }

  // No loop-carried dependencies for A because ptr only aliases with A
  int *ptr = &A[10];
  #pragma ivdep array(ptr)
  for (int i = 0; i < N; i++) {
  ```

```
    A[i] = A[i - X[i]];
    B[i] = B[i - Y[i]];
}
```

6.2. Good Design Practices for Single Work-Item Kernel

If your OpenCL kernels contain loop structures, follow the Intel-recommended guidelines to construct the kernels in a way that allows the Intel FPGA SDK for OpenCL Offline Compiler to analyze them effectively. Well-structured loops are particularly important when you direct the offline compiler to perform pipeline parallelism execution in loops.

Avoid Pointer Aliasing

Insert the `restrict` keyword in pointer arguments whenever possible. Including the `restrict` keyword in pointer arguments prevents the offline compiler from creating unnecessary memory dependencies between non-conflicting read and write operations. Consider a loop where each iteration reads data from one array, and then it writes data to another array in the same physical memory. Without including the `restrict` keyword in these pointer arguments, the offline compiler might assume dependence between the two arrays, and extracts less pipeline parallelism as a result.

Construct "Well-Formed" Loops

A "well-formed" loop has an exit condition that compares against an integer bound, and has a simple induction increment of one per iteration. Including "well-formed" loops in your kernel improves performance because the offline compiler can analyze these loops efficiently.

The following example is a "well-formed" loop:

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

Important: "Well-formed" nested loops also contribute to maximizing kernel performance.

The following example is a "well-formed" nested loop structure:

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

Minimize Loop-Carried Dependencies

The loop structure below creates a loop-carried dependence because each loop iteration reads data written by the previous iteration. As a result, each read operation cannot proceed until the write operation from the previous iteration completes. The presence of loop-carried dependencies decreases the extent of pipeline parallelism that the offline compiler can achieve, which reduces kernel performance.

```
for (int i = 0; i < N; i++) {
    A[i] = A[i - 1] + i;
}
```

The offline compiler performs a static memory dependence analysis on loops to determine the extent of parallelism that it can achieve. In some cases, the offline compiler might assume dependence between two array accesses, and extracts less pipeline parallelism as a result. The offline compiler assumes loop-carried dependence if it cannot resolve the dependencies at compilation time because of unknown variables, or if the array accesses involve complex addressing.

To minimize loop-carried dependencies, following the guidelines below whenever possible:

- *Avoid pointer arithmetic.*

  Compiler output is suboptimal when the kernel accesses arrays by dereferencing pointer values derived from arithmetic operations. For example, avoid accessing an array in the following manner:

  ```
  for (int i = 0; i < N; i++) {
      int t = *(A++);
      *A = t;
  }
  ```

- *Introduce simple array indexes.*

  Avoid the following types of complex array indexes because the offline compiler cannot analyze them effectively, which might lead to suboptimal compiler output:

  — Nonconstants in array indexes.

  For example, `A[K + i]`, where `i` is the loop index variable and K is an unknown variable.

  — Multiple index variables in the same subscript location.

  For example, `A[i + 2 × j]`, where `i` and `j` are loop index variables for a double nested loop.

  Note: The offline compiler can analyze the array index `A[i][j]` effectively because the index variables are in different subscripts.

  — Nonlinear indexing.

  For example, `A[i & C]`, where `i` is a loop index variable and C is a constant or a nonconstant variable.

- *Use loops with constant bounds in your kernel whenever possible.*

  Loops with constant bounds allow the offline compiler to perform range analysis effectively.

Avoid Complex Loop Exit Conditions

The offline compiler evaluates exit conditions to determine if subsequent loop iterations can enter the loop pipeline. There are times when the offline compiler requires memory accesses or complex operations to evaluate the exit condition. In these cases, subsequent iterations cannot launch until the evaluation completes, decreasing overall loop performance.

Convert Nested Loops into a Single Loop

To maximize performance, combine nested loops into a single form whenever possible. Restructuring nested loops into a single loop reduces hardware footprint and computational overhead between loop iterations.

The following code examples illustrate the conversion of a nested loop into a single loop:

<table>
<tr><th>Nested Loop</th><th>Converted Single Loop</th></tr>
<tr><td><pre>
for (i = 0; i < N; i++) {
    //statements
    for (j = 0; j < M; j++) {
        //statements
    }
    //statements
}
</pre></td><td><pre>
for (i = 0; i < N*M; i++) {
    //statements
}
</pre></td></tr>
</table>

Avoid Conditional Loops

To maximize performance, avoid declaring conditional loops. Conditional loops are tuples of loops that are declared within conditional statements such that one and only one of the loops is expected to be reached. These loops cannot be efficiently parallelized and result in a serialized implementation.

The following code examples illustrate the conversion of conditional loops to the a more optimal implementation:

<table>
<tr><th>Conditional Loops</th><th>Converted Loop</th></tr>
<tr><td><pre>
if (condition) {
   for (int i = 0; i < m; i++) {
      // statements
   }
}
else {
   for (int i = 0; i < m; i++) {
      // statements
   }
}
</pre></td><td><pre>
for (int i = 0; i < m; i++) {
   if (condition) {
      // statements
   }
   else {
     // statements
   }
}
</pre></td></tr>
</table>

Declare Variables in the Deepest Scope Possible

To reduce the hardware resources necessary for implementing a variable, declare the variable prior to its use in a loop. Declaring variables in the deepest scope possible minimizes data dependencies and hardware usage because the offline compiler does not need to preserve the variable data across loops that do not use the variables.

Consider the following example:

```
int a[N];
for (int i = 0; i < m; ++i) {
    int b[N];
    for (int j = 0; j < n; ++j) {
        // statements
    }
}
```

The array a requires more resources to implement than the array b. To reduce hardware usage, declare array a outside the inner loop unless it is necessary to maintain the data through iterations of the outer loop.

Tip: Overwriting all values of a variable in the deepest scope possible also reduces the resources necessary to present the variable.

7. Strategies for Improving NDRange Kernel Data Processing Efficiency

Consider the following kernel code:

```
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid] = a[gid] + b[gid];
}
```

This kernel adds arrays a and b, one element at a time. Each work-item is responsible for adding two elements, one from each array, and storing the sum into the array `answer`. Without optimization, the kernel performs one addition per work-item. To maximize the performance of your OpenCL kernel, consider implementing the applicable optimization techniques to improve data processing efficiency.

1. Specifying a Maximum Work-group Size or a Required Work-Group Size on page 137
2. Kernel Vectorization on page 139
3. Multiple Compute Units on page 140
4. Combination of Compute Unit Replication and Kernel SIMD Vectorization on page 143
5. Reviewing Kernel Properties and Loop Unroll Status in the HTML Report on page 144

7.1. Specifying a Maximum Work-group Size or a Required Work-Group Size

Specify the `max_work_group_size` or `reqd_work_group_size` attribute for your kernels whenever possible. These attributes allow the Intel FPGA SDK for OpenCL Offline Compiler to perform aggressive optimizations to match the kernel to hardware resources without any excess logic.

Tip: For oneAPI DPC++-specific details, refer to Specify a Work-group Size topic in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

The offline compiler assumes a default work-group size for your kernel depending on certain constraints imposed during compilation time and runtime .

The offline compiler imposes the following constraints at compilation time:

- If you specify a value for the reqd_work_group_size attribute, the work-group size must match this value.
- If you specify a value for the max_work_group_size attribute, the work-group size must not exceed this value.
- If you do not specify values for reqd_work_group_size and max_work_group_size, and the kernel contains a barrier, the offline compiler defaults to a maximum work-group size of 256 work-items.
- If you do not specify values for both attributes and the kernel does not contain any barrier, the offline compiler does not impose any constraint on the work-group size at compilation time.

Tip: Use the CL_KERNEL_WORK_GROUP_SIZE and CL_KERNEL_COMPILE_WORK_GROUP_SIZE queries to the clGetKernelWorkGroupInfo API call to determine the work-group size constraints that the offline compiler imposes on a particular kernel at compilation time.

The OpenCL standard imposes the following constraints at runtime:

- The work-group size in each dimension must divide evenly into the requested NDRange size in each dimension.
- The work-group size must not exceed the device constraints specified by the CL_DEVICE_MAX_WORK_GROUP_SIZE and CL_DEVICE_MAX_WORK_ITEM_SIZES queries to the clGetDeviceInfo API call.

Caution: If the work-group size you specify for a requested NDRange kernel execution does not satisfy all of the constraints listed above, the clEnqueueNDRangeKernel API call fails with the error CL_INVALID_WORK_GROUP_SIZE.

If you do not specify values for both the reqd_work_group_size and max_work_group_size attributes, the runtime determines a default work-group size as follows:

- If the kernel contains a barrier or refers to the local work-item ID, or if you use the clGetKernelWorkGroupInfo and clGetDeviceInfo API calls in your host code to query the work-group size, the runtime defaults the work-group size to one work-item.
- If the kernel does not contain a barrier or refer to the local work-item ID, or if your host code does not query the work-group size, the default work-group size is the global NDRange size.

When queuing an NDRange kernel (that is, not a single work-item kernel), specify an explicit work-group size under the following conditions:

- If your kernel uses memory barriers, local memory, or local work-item IDs.
- If your host program queries the work-group size.

If your kernel uses memory barriers, perform one of the following tasks to minimize hardware resources:

- Specify a value for the reqd_work_group_size attribute.
- Assign to the max_work_group_size attribute the smallest work-group size that accommodates all your runtime work-group size requests.

Caution: Including a memory barrier at the end of your NDRange kernel causes compilation to fail.

Specifying a smaller work-group size than the default at runtime might lead to excessive hardware consumption. Therefore, if you require a work-group size other than the default, specify the `max_work_group_size` attribute to set a maximum work-group size. If the work-group size remains constant through all kernel invocations, specify a required work-group size by including the `reqd_work_group_size` attribute. The `reqd_work_group_size` attribute instructs the offline compiler to allocate exactly the correct amount of hardware to manage the number of work-items per work-group you specify. This allocation results in hardware resource savings and improved efficiency in the implementation of kernel compute units. By specifying the `reqd_work_group_size` attribute, you also prevent the offline compiler from implementing additional hardware to support work-groups of unknown sizes.

For example, the code fragment below assigns a fixed work-group size of 64 work-items to a kernel:

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
  size_t gid = get_global_id(0);

  answer[gid] = a[gid] + b[gid];
}
```

7.2. Kernel Vectorization

To achieve higher throughput, you can vectorize your kernel. Kernel vectorization allows multiple work-items to execute in a single instruction multiple data (SIMD) fashion. You can direct the Intel FPGA SDK for OpenCL Offline Compiler to translate each scalar operation in the kernel, such as addition or multiplication, to an SIMD operation.

Include the `num_simd_work_items` attribute in your kernel code to direct the offline compiler to perform more additions per work-item without modifying the body of the kernel. The following code fragment applies a vectorization factor of four to the original kernel code:

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);

   answer[gid] = a[gid] + b[gid];
}
```

To use the `num_simd_work_items` attribute, you must also specify a required work-group size of the kernel using the `reqd_work_group_size` attribute. The work-group size you specify for `reqd_work_group_size` must be divisible by the value you assign to `num_simd_work_items`. In the code example above, the kernel has a fixed work-group size of 64 work-items. Within each work-group, the work-items are distributed evenly among the four SIMD vector lanes. After the offline compiler implements the four SIMD vector lanes, each work-item now performs four times more work.

The offline compiler vectorizes the code and might coalesce memory accesses. You do not need to change any kernel code or host code because the offline compiler applies these optimizations automatically.

You can vectorize your kernel code manually, but you must adjust the NDRange in your host application to reflect the amount of vectorization you implement. The following example shows the changes in the code when you duplicate operations in the kernel manually:

```
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
   size_t gid = get_global_id(0);

   answer[gid * 4 + 0] = a[gid * 4 + 0] + b[gid * 4 + 0];
   answer[gid * 4 + 1] = a[gid * 4 + 1] + b[gid * 4 + 1];
   answer[gid * 4 + 2] = a[gid * 4 + 2] + b[gid * 4 + 2];
   answer[gid * 4 + 3] = a[gid * 4 + 3] + b[gid * 4 + 3];
}
```

In this form, the kernel loads four elements from arrays a and b, calculates the sums, and stores the results into the array `answer`. Because the FPGA pipeline loads and stores data to neighboring locations in memory, you can manually direct the offline compiler to coalesce each group of four load and store operations.

Attention: Each work-item handles four times as much work after you implement the manual optimizations. As a result, the host application must use an NDRange that is four times smaller than in the original example. On the contrary, you do not need to adjust the NDRange size when you exploit the automatic vectorization capabilities of the offline compiler. You can adjust the vector width with minimal code changes by using the `num_simd_work_items` attribute.

7.3. Multiple Compute Units

To achieve higher throughput, the Intel FPGA SDK for OpenCL Offline Compiler can generate multiple compute units for each kernel. The offline compiler implements each compute unit as a unique pipeline. Generally, each kernel compute unit can execute multiple work-groups simultaneously.

To increase overall kernel throughput, the hardware scheduler in the FPGA dispatches work-groups to additional available compute units. A compute unit is available for work-group assignments provided that it has not reached its full capacity.

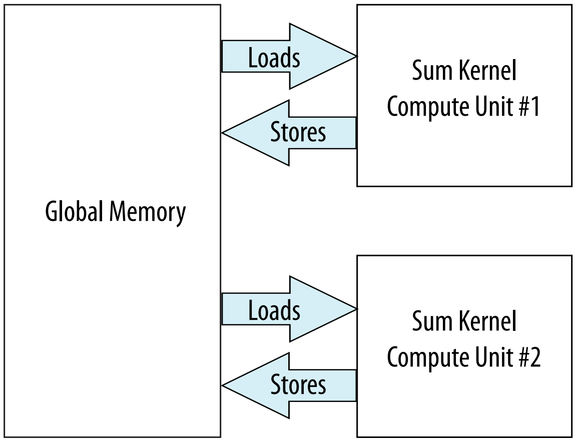
Assume each work-group takes the same amount of time to complete its execution. If the offline compiler implements two compute units, each compute unit executes half of the work-groups. Because the hardware scheduler dispatches the work-groups, you do not need to manage this process in your own code.

The offline compiler does not automatically determine the optimal number of compute units for a kernel. To increase the number of compute units for your kernel implementation, you must specify the number of compute units that the offline compiler should create using the `num_compute_units` attribute, as shown in the code sample below.

```
__attribute__((num_compute_units(2)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid] = a[gid] + b[gid];
}
```

Increasing the number of compute units achieves higher throughput. However, as shown in the figure below, you do so at the expense of increasing global memory bandwidth among the compute units. You also increase hardware resource utilization.

Figure 76. Data Flow with Multiple Compute Units



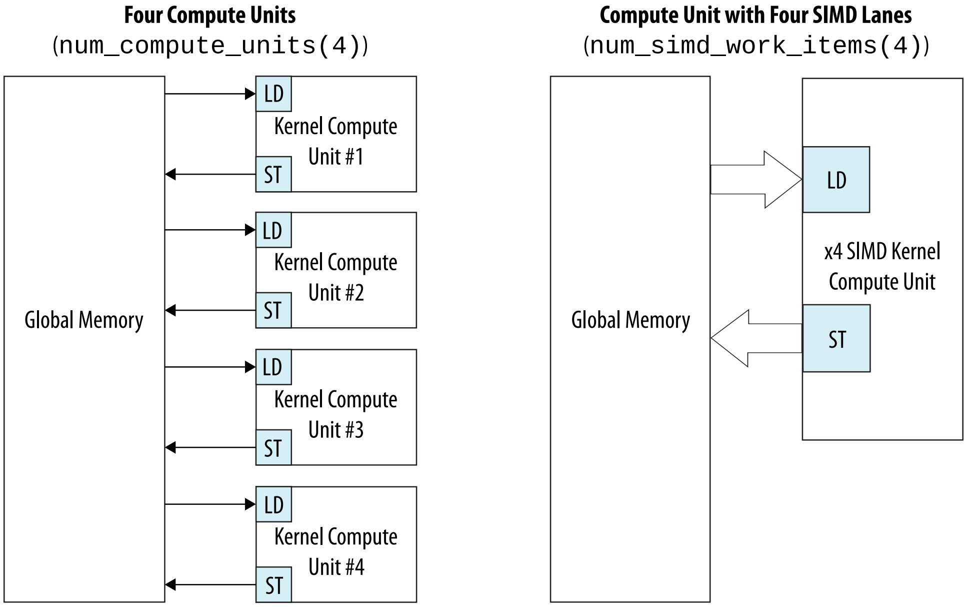
7.3.1. Compute Unit Replication versus Kernel SIMD Vectorization

In most cases, you should implement the `num_simd_work_items` attribute to increase data processing efficiency before using the `num_compute_units` attribute.

Both the `num_compute_units` and `num_simd_work_items` attributes increase throughput by increasing the amount of hardware that the Intel FPGA SDK for OpenCL Offline Compiler uses to implement your kernel. The `num_compute_units` attribute modifies the number of compute units to which work-groups can be scheduled, which also modifies the number of times a kernel accesses global memory. In contrast, the `num_simd_work_items` attribute modifies the amount of work a compute unit can perform in parallel on a single work-group. The `num_simd_work_items` attribute duplicates only the datapath of the compute unit by sharing the control logic across each SIMD vector lane.

Generally, using the `num_simd_work_items` attribute leads to more efficient hardware than using the `num_compute_units` attribute to achieve the same goal. The `num_simd_work_items` attribute also allows the offline compiler to coalesce your memory accesses.

Figure 77. Compute Unit Replication versus Kernel SIMD Vectorization



Multiple compute units competing for global memory might lead to undesired memory access patterns. You can alter the undesired memory access pattern by introducing the `num_simd_work_items` attribute instead of the `num_compute_units` attribute. In addition, the `num_simd_work_items` attribute potentially offers the same computational throughput as the equivalent kernel compute unit duplication that the `num_compute_units` attribute offers.

You cannot implement the `num_simd_work_items` attribute in your kernel under the following circumstances:

- The value you specify for `num_simd_work_items` is not 2, 4, 8 or 16.
- The value of `reqd_work_group_size` is not divisible by `num_simd_work_items`.

  For example, the following declaration is incorrect because 50 is not divisible by 4:

  ```
  __attribute__((num_simd_work_items(4)))
  __attribute__((reqd_work_group_size(50,0,0)))
  ```
- Kernels with complex control flows. You cannot vectorize lines of code within a kernel in which different work-items follow different control paths (for example, the control paths depend on `get_global_ID` or `get_local_ID`).

During kernel compilation, the offline compiler issues messages informing you whether the implementation of vectorization optimizations is successful. Kernel vectorization is successful if the reported vectorization factor matches the value you specify for the `num_simd_work_items` attribute.

7.4. Combination of Compute Unit Replication and Kernel SIMD Vectorization

If your replicated or vectorized OpenCL kernel does not fit in the FPGA, you can modify the kernel by both replicating the compute unit and vectorizing the kernel. Include the num_compute_units attribute to modify the number of compute units for the kernel, and include the num_simd_work_items attribute to take advantage of kernel vectorization.

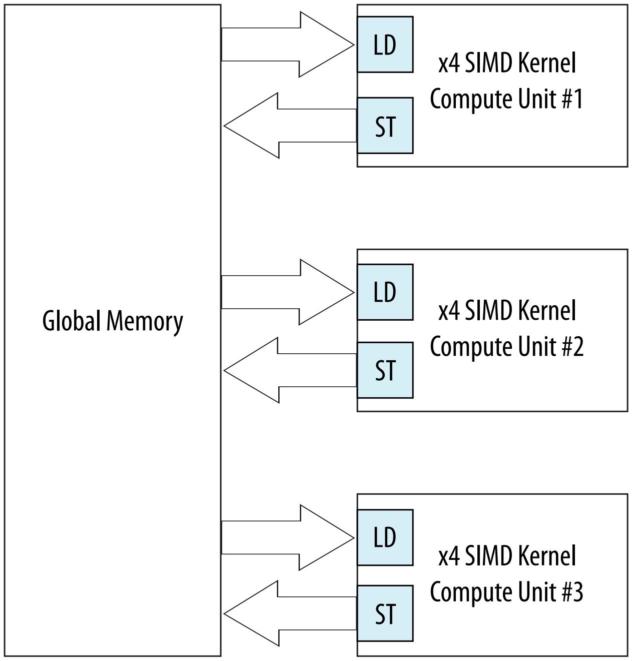
Consider a case where a kernel with a num_simd_work_items attribute set to 16 does not fit in the FPGA. The kernel might fit if you modify it by duplicating a narrower SIMD kernel compute unit. Determining the optimal balance between the number of compute units and the SIMD width might require some experimentation. For example, duplicating a four lane-wide SIMD kernel compute unit three times might achieve better throughput than duplicating an eight lane-wide SIMD kernel compute unit twice.

The following example code shows how you can combine the num_compute_units and num_simd_work_items attributes in your OpenCL code:

```
__attribute__((num_simd_work_items(4)))
__attribute__((num_compute_units(3)))
__attribute__((reqd_work_group_size(8,8,1)))
__kernel void matrixMult(__global float * restrict C,
                         __global float * restrict A,
. . .
```

The figure below illustrates the data flow of the kernel described above. The num_compute_units implements three replicated compute units. The num_simd_work_items implements four SIMD vector lanes.

Figure 78. Optimizing Throughput by Combining Compute Unit Replication and Kernel SIMD Vectorization


Global Memory
LD
ST
x4 SIMD Kernel Compute Unit #1
LD
ST
x4 SIMD Kernel Compute Unit #2
LD
ST
x4 SIMD Kernel Compute Unit #3

Attention: You can also enable the resource-driven optimizer to determine automatically the best combination of `num_compute_units` and `num_simd_work_items`.

Important: It is more time-consuming to compile a hardware design that fills the entire FPGA than smaller designs. When you adjust your kernel optimizations, remove the increased number of SIMD vector lanes and compute units prior to recompiling the kernel.

7.5. Reviewing Kernel Properties and Loop Unroll Status in the HTML Report

When you compile an NDRange kernel, the Intel FPGA SDK for OpenCL Offline Compiler generates a `<*your_kernel_filename*>/reports/report.html` file that provides information on select kernel properties and loop unroll status.

Related Information

Reviewing Your Kernel's report.html File on page 15

8. Strategies for Improving Memory Access Efficiency

Memory access efficiency often dictates the overall performance of your OpenCL kernel. When developing your OpenCL code, it is advantageous to minimize the number of global memory accesses. The *OpenCL Specification version 1.0* describes four memory types: *global*, *constant*, *local*, and *private* memories.

Tip: For Intel oneAPI DPC++/C++ Compiler-specific details, refer to Memory Accesses section in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

An interconnect topology connects shared global, constant, and local memory systems to their underlying memory. Interconnect includes access arbitration to memory ports.

Memory accesses compete for shared memory resources (that is, global, local, and constant memories). If your OpenCL kernel performs a large number of memory accesses, the Intel FPGA SDK for OpenCL Offline Compiler must generate complex arbitration logic to handle the memory access requests. The complex arbitration logic might cause a drop in the maximum operating frequency (f_{MAX}), which degrades kernel performance.

The following sections discuss memory access optimizations in detail. In summary, minimizing global memory accesses is beneficial for the following reasons:

- Typically, increases in OpenCL kernel performance lead to increases in global memory bandwidth requirements.
- The maximum global memory bandwidth is much smaller than the maximum local memory bandwidth.
- The maximum computational bandwidth of the FPGA is much larger than the global memory bandwidth.

Attention: Use local, private or constant memory whenever possible to increase the memory bandwidth of the kernel.

1. General Guidelines on Optimizing Memory Accesses on page 146
2. Optimize Global Memory Accesses on page 146
3. Performing Kernel Computations Using Constant, Local or Private Memory on page 151
4. Improving Kernel Performance by Banking the Local Memory on page 154
5. Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor on page 157
6. Minimizing the Memory Dependencies for Loop Pipelining on page 159
7. Static Memory Coalescing on page 160

8.1. General Guidelines on Optimizing Memory Accesses

Optimizing the memory accesses in your OpenCL kernels can improve overall kernel performance.

Consider implementing the following techniques for optimizing memory accesses, whenever possible:

- If your OpenCL program has a pair of kernels—one produces data and the other one consumes that data—convert them into a single kernel that performs both functions. Also, implement helper functions to logically separate the functions of the two original kernels.

  FPGA implementations favor one large kernel over separate smaller kernels. Kernel unification removes the need to write the results from one kernel into global memory temporarily before fetching the same data in the other kernel.

- The Intel FPGA SDK for OpenCL Offline Compiler implements local memory in FPGAs very differently than in GPUs. If your OpenCL kernel contains code to avoid GPU-specific local memory bank conflicts, remove that code because the offline compiler generates hardware that avoids local memory bank conflicts automatically whenever possible.

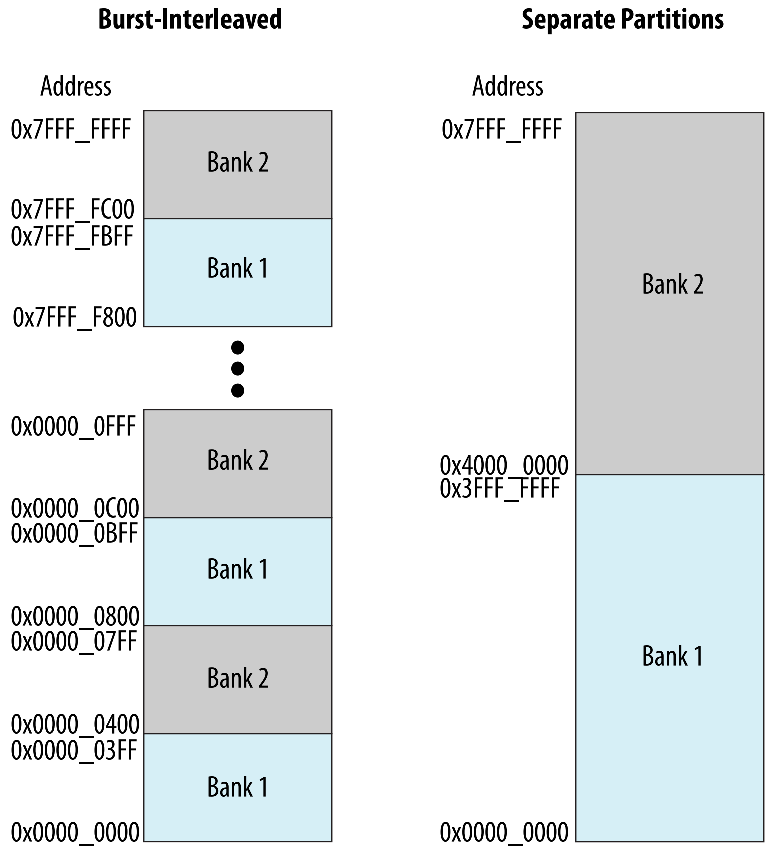
8.2. Optimize Global Memory Accesses

The Intel FPGA SDK for OpenCL Offline Compiler uses SDRAM as global memory. By default, the offline compiler configures global memory in a burst-interleaved configuration. The offline compiler interleaves global memory across each of the external memory banks.

In most circumstances, the default burst-interleaved configuration leads to the best load balancing between the memory banks. However, in some cases, you might want to partition the banks manually as two non-interleaved (and contiguous) memory regions to achieve better load balancing.

The figure below illustrates the differences in memory mapping patterns between burst-interleaved and non-interleaved memory partitions.

Figure 79. Global Memory Partitions



Global Memory Bandwidth Use

To ensure the global memory bandwidth listed in the board specification file is utilized completely, calculating the kernel bandwidth use is beneficial. The report.html file also displays the kernel bandwidth values in the global memory view of the System Viewer. The following formulas explain how you can calculate this value on a per-LSU basis:

Figure 80. Formulas for Calculating Kernel Bandwidth Use

$$\text{LSU bandwidth} = \min(BW_1, BW_2, BW_3)\ \text{MB/s}$$

$$BW_1 = \text{KWIDTH} * \text{FMAX}$$

$$BW_2 = \text{MWIDTH} * \text{FMAX}$$

$$BW_3 = \frac{\text{MaxBandwidth} * \text{NUM\_INTERLEAVING\_CHANNELS}}{\text{NUM\_CHANNELS}}$$

The LSU bandwidth equation is the minimum of three bottlenecks you need to calculate the use of global memory bandwidth. The remaining equations represent three bottlenecks that can limit the LSU bandwidth. These formulas represent the theoretical maximum bandwidth an LSU may consume, ignoring all other LSUs. The actual bandwidth depends on the LSU's access pattern and the interconnect's arbitration between all LSUs. To get an estimate of the overall bandwidth, a sum of the LSU bandwidths is available in the controller of the global memory view of the System Viewer.

The following table describes the variables used in the above equations:

Variable	Description
KWIDTH	Byte-width of the LSU on the kernel. In the report.html file, it is referred to as WIDTH.
MWIDTH	Byte-width of the LSU facing the external memory. In the report.html file, it is referred to as the <Memory Name>_Width.
FMAX	Clock speed of the kernel in MHz. In the report.html file, you can identify this as the design's clock speed.
MaxBandwidth	Maximum bandwidth (measured in MB/s) the global memory can achieve. You can find this in the board_spec.xml file for the specific global memory.
NUM_CHANNELS	Number of interfaces an external memory has. You can find this by counting the number of interfaces listed in the board_spec.xml file under that memory.
NUM_INTERLEAVING_CHANNELS	When interleaving is enabled, this is the number of channels. Otherwise, this value is 1.
BW_1	Bottleneck at the kernel boundary. Therefore, BW_1 uses only kernel values, which means, values you can change by optimizing the design. If this is limiting the overall bandwidth use than it indicates, changing your design can improve the bottleneck at the kernel boundary.
BW_2	Bottleneck at the memory interface to the kernel. Therefore, BW_2 uses the size of the memory interface and the FMAX, which means either improving FMAX of your design or switching to a board with a wider memory interface can improve the bandwidth use.
BW_3	Bottleneck in the external memory. Therefore, BW_3 uses external memory properties exclusively, and if this is limiting your design, you have utilized the board bandwidth completely.

8.2.1. Contiguous Memory Accesses

Contiguous memory access optimizations analyze statically the access patterns of global load and store operations in a kernel. For sequential load or store operations that occur for the entire kernel invocation, the Intel FPGA SDK for OpenCL Offline Compiler directs the kernel to access consecutive locations in global memory.

Consider the following code example:

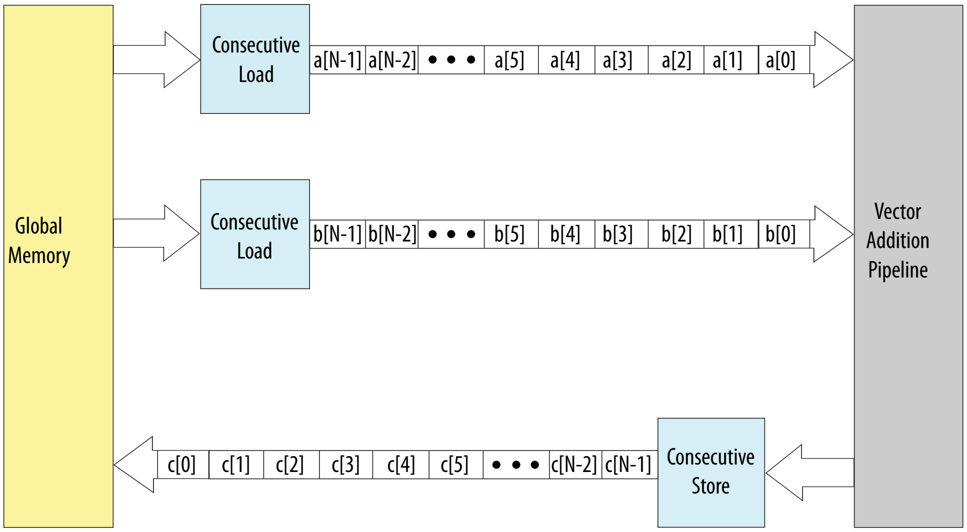
```
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict c )
{
    size_t gid = get_global_id(0);

    c[gid] = a[gid] + b[gid];
}
```

The load operation from array a uses an index that is a direct function of the work-item global ID. By basing the array index on the work-item global ID, the offline compiler can direct contiguous load operations. These load operations retrieve the data sequentially from the input array, and sends the read data to the pipeline as required. Contiguous store operations then store elements of the result that exits the computation pipeline in sequential locations within global memory.

Tip: Use the const qualifier for any read-only global buffer so that the offline compiler can perform more aggressive optimizations on the load operation.

The following figure illustrates an example of the contiguous memory access optimization:

Figure 81. Contiguous Memory Access


Global Memory
Consecutive Load
a[N-1] a[N-2] • • • a[5] a[4] a[3] a[2] a[1] a[0]
Consecutive Load
b[N-1] b[N-2] • • • b[5] b[4] b[3] b[2] b[1] b[0]
Vector Addition Pipeline
c[0] c[1] c[2] c[3] c[4] c[5] • • • c[N-2] c[N-1]
Consecutive Store

Contiguous load and store operations improve memory access efficiency because they lead to increased access speeds and reduced hardware resource needs. The data travels in and out of the computational portion of the pipeline concurrently, allowing overlaps between computation and memory accesses. If possible, use work-item IDs that index consecutive memory locations for load and store operations that access global memory. Sequential accesses to global memory increase memory efficiency because they provide an ideal access pattern.

8.2.2. Manual Partitioning of Global Memory

You can partition the memory manually so that each buffer occupies a different memory bank.

The default burst-interleaved configuration of the global memory prevents load imbalance by ensuring that memory accesses do not favor one external memory bank over another. However, you have the option to control the memory bandwidth across a group of buffers by partitioning your data manually.

- The Intel FPGA SDK for OpenCL Offline Compiler cannot burst-interleave across different memory types. To manually partition a specific type of global memory , compile your OpenCL kernels with the `-no-interleaving=<`*`global_memory_type`*`>` flag to configure each bank of a certain memory type as non-interleaved banks.

  If your kernel accesses two buffers of equal size in memory, you can distribute your data to both memory banks simultaneously regardless of dynamic scheduling between the loads. This optimization step might increase your apparent memory bandwidth.

  If your kernel accesses heterogeneous global memory types, include the `-no-interleaving=<`*`global_memory_type`*`>` option in the aoc command for each memory type that you want to partition manually.

For more information about the usage of the `-no-interleaving=<global_memory_type>` option, refer to the *Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>)* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>)

8.2.2.1. Heterogeneous Memory Buffers

You can execute your kernel on an FPGA board that includes multiple global memory types, such as DDR, QDR, and on-chip RAMs.

If your FPGA board offers heterogeneous global memory types, keep in mind that they handle different memory accesses with varying efficiencies.

For example:

- Use DDR SDRAM for long sequential accesses.
- Use QDR SDRAM for random accesses.
- Use on-chip RAM for random low latency accesses.

For more information about how to allocate buffers in global memory and how to modify your host application to use heterogeneous buffers, refer to the *Specifying Buffer Location in Global Memory* and *Allocating OpenCL Buffer for Manual Partitioning of Global Memory* sections of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

- Partitioning Buffers Across Different Memory Types (Heterogeneous Memory)
- Allocating OpenCL Buffer for Manual Partitioning of Global Memory

8.2.3. Optimizing for Two or More Banks of Global Memory

The Intel FPGA SDK for OpenCL Offline Compiler automatically creates a global memory interconnect designed to deliver most of the available global memory bandwidth from the BSP to the kernel system.

The throughput can be saturated using read-only, write-only, or mixed read/write traffic. By default, traffic is interleaved across all available banks. If you find that the throughput is insufficient, Intel recommends using the `-no-interleaving` option.

Related Information

- Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>)
- Manual Partitioning of Global Memory on page 149

8.3. Performing Kernel Computations Using Constant, Local or Private Memory

To optimize memory access efficiency, minimize the number for global memory accesses by performing your OpenCL kernel computations in constant, local, or private memory.

To minimize global memory accesses, you must first preload data from a group of computations from global memory to constant, local, or private memory. You perform the kernel computations on the preloaded data, and then write the results back to global memory.

8.3.1. Constant Cache Memory

Constant memory resides in global memory, but the kernel loads it into an on-chip cache shared by all work-groups at runtime. For example, if you have read-only data that all work-groups use, and the data size of the constant buffer fits into the constant cache, allocate the data to the constant memory. The constant cache is most appropriate for high-bandwidth table lookups that are constant across several invocations of a kernel. The constant cache is optimized for high cache hit performance.

By default, the constant cache size is 16 kB. You can specify the constant cache size by including the `-const-cache-bytes=`*<N>* option in your `aoc` command, where *<N>* is the constant cache size in bytes.

Unlike global memory accesses that have extra hardware for tolerating long memory latencies, the constant cache suffers large performance penalties for cache misses. If the `__constant` arguments in your OpenCL kernel code cannot fit in the cache, you might achieve better performance with `__global const` arguments instead. If the host application writes to constant memory that is already loaded into the constant cache, the cached data is discarded (that is, invalidated) from the constant cache.

For more information about the `-const-cache-bytes=`*<N>* option, refer to the *Configuring Constant Memory Cache Size (-const-cache-bytes=<N>)* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

Configuring Constant Memory Cache Size (-const-cache-bytes=<N>)

8.3.2. Preloading Data to Local Memory

Local memory is considerably smaller than global memory, but it has significantly higher throughput and much lower latency. Unlike global memory accesses, the kernel can access local memory randomly without any performance penalty. When you structure your kernel code, attempt to access the global memory sequentially, and buffer that data in on-chip local memory before your kernel uses the data for calculation purposes.

The Intel FPGA SDK for OpenCL Offline Compiler implements OpenCL local memory in on-chip memory blocks in the FPGA. On-chip memory blocks have two read and write ports, and they can be clocked at an operating frequency that is double the operating frequency of the OpenCL kernels. This doubling of the clock frequency allows the memory to be "double pumped," resulting in twice the bandwidth from the same memory. As a result, each on-chip memory block supports up to four simultaneous accesses.

Ideally, the accesses to each bank are distributed uniformly across the on-chip memory blocks of the bank. Because only four simultaneous accesses to an on-chip memory block are possible in a single clock cycle, distributing the accesses helps avoid bank contention.

This banking configuration is usually effective; however, the offline compiler must create a complex memory system to accommodate a large number of banks. A large number of banks might complicate the arbitration network and can reduce the overall system performance.

Because the offline compiler implements local memory that resides in on-chip memory blocks in the FPGA, the offline compiler must choose the size of local memory systems at compilation time. The method the offline compiler uses to determine the size of a local memory system depends on the local data types used in your OpenCL code.

Optimizing Local Memory Accesses

To optimize local memory access efficiency, consider the following guidelines:

- Implementing certain optimizations techniques, such as loop unrolling, might lead to more concurrent memory accesses.

  Caution: Increasing the number of memory accesses can complicate the memory systems and degrade performance.

- Simplify the local memory subsystem by limiting the number of unique local memory accesses in your kernel to four or less, whenever possible.

  You achieve maximum local memory performance when there are four or less memory accesses to a local memory system. If the number of accesses to a particular memory system is greater than four, the offline compiler arranges the on-chip memory blocks of the memory system into a banked configuration.

- If you have function scope local data, the offline compiler statically sizes the local data that you define within a function body at compilation time. You should define local memories by directing the offline compiler to set the memory to the required size, rounded up to the closest value that is a power of two.

- Avoid the use of the `local_mem_size` attribute. Use the `__local` kernel variable instead of `__local` kernel arguments.

  For more information, refer to the Programming Strategies for Optimizing Pointer-to-Local Memory Size section of the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*.

- When accessing local memory, use the simplest address calculations possible and avoid pointer math operations that are not mandatory.

  Intel recommends this coding style to reduce FPGA resource utilization and increase local memory efficiency by allowing the offline compiler to make better guarantees about access patterns through static code analysis. Complex address calculations and pointer math operations can prevent the offline compiler from creating independent memory systems representing different portions of your data, leading to increased area usage and decreased runtime performance.

- Avoid storing pointers to memory whenever possible. Stored pointers often prevent static compiler analysis from determining the data sets accessed, when the pointers are subsequently retrieved from memory. Storing pointers to memory almost always leads to suboptimal area and performance results.

- Create local array elements that are a power of 2 bytes to allow the offline compiler to provide an efficient memory configuration.

  Whenever possible, the offline compiler automatically pads the elements of the local memory to be a power of 2 to provide a more efficient memory configuration. For example, if you have a struct containing 3 chars, the offline compiler pads it to 4 bytes, instead of creating a narrower and deeper memory with multiple accesses (that is, a 1-byte wide memory configuration). However, there are cases where the offline compiler might not pad the memory, such as when the kernel accesses local memory indirectly through pointer arithmetic.

  To determine if the offline compiler has padded the local memory, review the memory dimensions in the Kernel Memory Viewer. If the offline compiler fails to pad the local memory, it prints the following message in the area report:

```
Memory system contains arrays whose element size is not a power of two.  This
may result in extra loads and stores, leading to stalls.  Try padding structs
to a power of two, or break them into multiple arrays of smaller elements.
```

Related Information

Programming Strategies for Optimizing Pointer-to-Local Memory Size

8.3.3. Storing Variables and Arrays in Private Memory

The Intel FPGA SDK for OpenCL Offline Compiler implements private memory using FPGA registers or block RAMs. The offline compiler analyzes the private memory accesses and promotes them to register accesses. Scalar variables, for example float, int and char, are mostly promoted. Aggregate data types are promoted, if accesses are compile-time constants. Typically, private memory is useful for storing single variables or small arrays. Registers are plentiful hardware resources in FPGAs, and it is almost always better to use private memory instead of other memory types whenever possible. The kernel can access private memories in parallel, allowing them to provide more bandwidth than any other memory type (that is, global, local, and constant memories).

For more information about the implementation of private memory using registers, refer to the *Inferring a Register* section of the *Intel FPGA SDK for OpenCL Programming Guide*.

Related Information

Inferring a Register

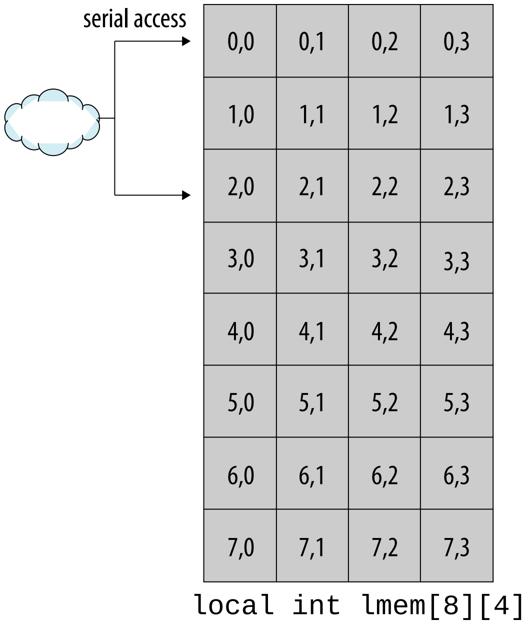
8.4. Improving Kernel Performance by Banking the Local Memory

Specifying the `numbanks(N)` and `bankwidth(M)` advanced kernel attributes allows you to configure the local memory banks for parallel memory accesses.The banking geometry described by these advanced kernel attributes determines which elements of the local memory system your kernel can access in parallel.

The following code example depicts an 8 x 4 local memory system that is implemented in a single bank. As a result, no two elements in the system can be accessed in parallel.

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

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

Figure 82. Serial Accesses to an 8 x 4 Local Memory System



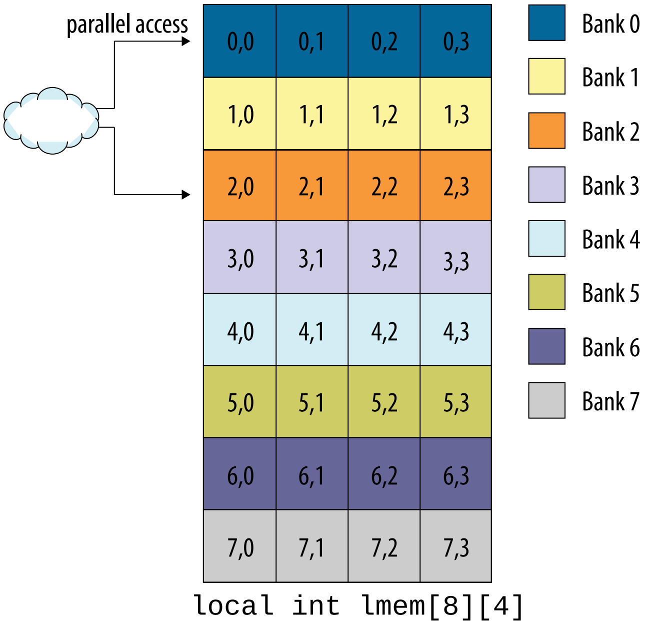
0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3
5,0	5,1	5,2	5,3
6,0	6,1	6,2	6,3
7,0	7,1	7,2	7,3

To improve performance, you can add `numbanks(N)` and `bankwidth(M)` in your code to define the number of memory banks and the bank widths in bytes. The following code implements eight memory banks, each 16-bytes wide. This memory bank configuration enables parallel memory accesses down the 8 x 4 array.

```
local int __attribute__((numbanks(8),
                        bankwidth(16)))
                        lmem[8][4];
#pragma unroll
for (int i = 0; i < 4; i+=2) {
    lmem[i][x & 0x3] = …;
}
```

Attention: To enable parallel access, you must mask the dynamic access on the lower array index. Masking the dynamic access on the lower array index informs the Intel FPGA SDK for OpenCL Offline Compiler that x does not exceed the lower index bounds.

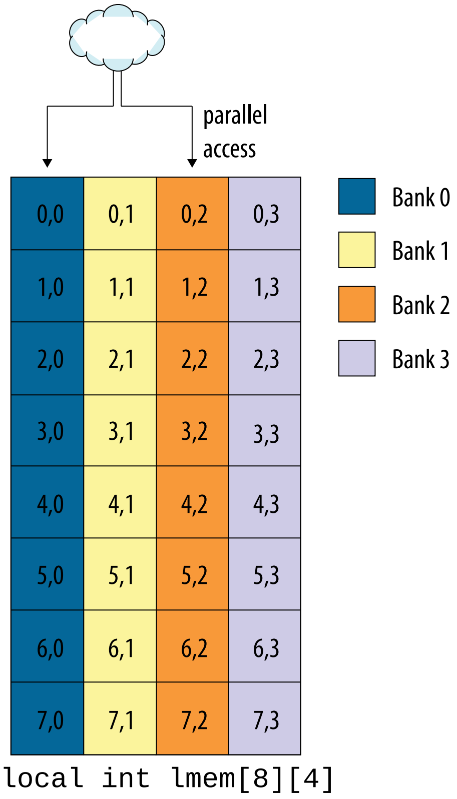
Figure 83. Parallel Access to an 8 x 4 Local Memory System with Eight 16-Byte-Wide Memory Banks



By specifying different values for the `numbanks(`*`N`*`)` and `bankwidth(`*`M`*`)` kernel attributes, you can change the parallel access pattern. The following code implements four memory banks, each 4-bytes wide. This memory bank configuration enables parallel memory accesses across the 8 x 4 array.

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

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

Figure 84. Parallel Access to an 8 x 4 Local Memory System with Four 4-Byte-Wide Memory Banks



8.4.1. Optimizing the Geometric Configuration of Local Memory Banks Based on Array Index

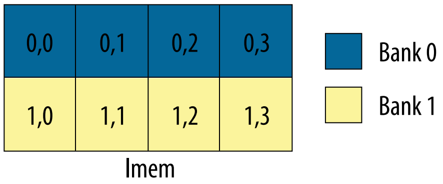
By default, the Intel FPGA SDK for OpenCL Offline Compiler might attempt to improve performance by automatically banking a local memory system. The Intel FPGA SDK for OpenCL includes advanced features that allow you to customize the banking geometry of your local memory system. To configure the geometry of local memory banks, include the `numbanks(N)` and `bankwidth(M)` kernel attributes in your OpenCL kernel .

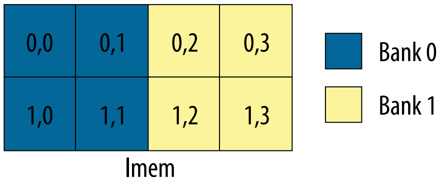
The following code examples illustrate how the bank geometry changes based on the values you assign to `numbanks` and `bankwidth`.

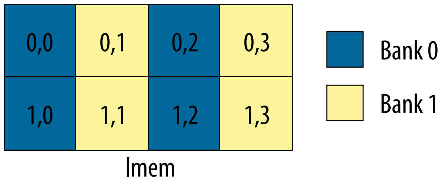
Table 20. Effects of numbanks and bankwidth on the Bank Geometry of 2 x 4 Local Memory System

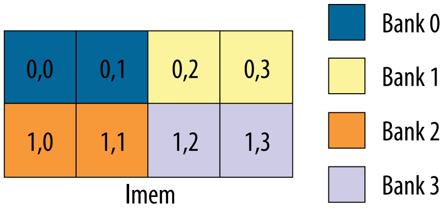
The first and last rows of this table illustrate how to bank memory on the upper and lower indexes of a 2D array, respectively.

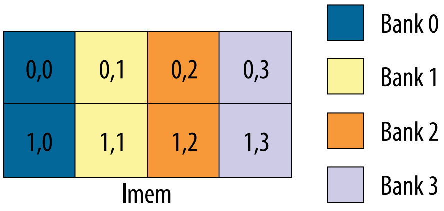
Code Example	Bank Geometry

`local int`
`__attribute__((numbanks(2),`
`bankwidth(16)))`
`lmem[2][4];`
	 0,0
	0,1
	0,2
	0,3
	1,0
	1,1
	1,2
	1,3
	lmem
	Bank 0
	Bank 1

`local int`
`__attribute__((numbanks(2),`
`bankwidth(8)))`
`lmem[2][4];`
	 0,0
	0,1
	0,2
	0,3
	1,0
	1,1
	1,2
	1,3
	lmem
	Bank 0
	Bank 1

`local int`
`__attribute__((numbanks(2),`
`bankwidth(4)))`
`lmem[2][4];`
	 0,0
	0,1
	0,2
	0,3
	1,0
	1,1
	1,2
	1,3
	lmem
	Bank 0
	Bank 1

`local int`
`__attribute__((numbanks(4),`
`bankwidth(8)))`
`lmem[2][4];`
	 0,0
	0,1
	0,2
	0,3
	1,0
	1,1
	1,2
	1,3
	lmem
	Bank 0
	Bank 1
	Bank 2
	Bank 3

`local int`
`__attribute__((numbanks(4),`
`bankwidth(4)))`
`lmem[2][4];`
	 0,0
	0,1
	0,2
	0,3
	1,0
	1,1
	1,2
	1,3
	lmem
	Bank 0
	Bank 1
	Bank 2
	Bank 3

Related Information

Kernel Attributes for Configuring Local and Private Memory Systems

8.5. Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor

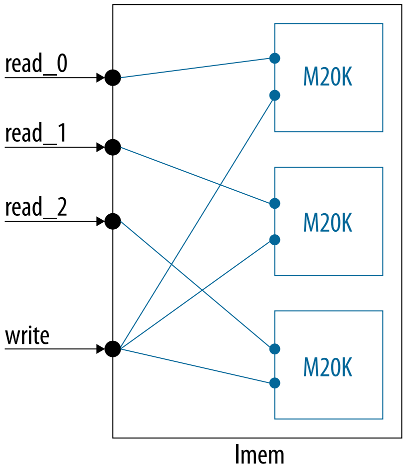
The memory replication factor is the number of M20K memory blocks that your design uses to implement the local memory system. To control the memory replication factor, use the `max_replicates` kernel attribute in your OpenCL kernel.

Intel's M20K memory blocks have two *physical* ports. The number of *logical* ports that are available in each M20K block depends on the degree of pumping. Pumping is a measure of the clock frequency of the M20K blocks relative to the rest of the design.

Consider the following code example where the `singlepump` attribute is applied to a local memory system, `lmem`, which has three read accesses and one write access. The `singlepump` attribute indicates that the M20K blocks runs at the same frequency as the rest of the design.

```
__kernel void three_copies(int raddr, int waddr) {
    int __attribute__((memory,
                       numbanks(1),
                       singlepump,
                       max_replicates(3)))
                       lmem[16];

    lmem[waddr] = lmem[raddr] + lmem[raddr + 1] + lmem[raddr + 2];
    // do something with lmem
}
```

Figure 85. Accesses to Single-Pumped M20K Memory Blocks



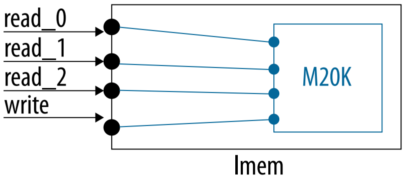
The compiler creates an arbitration-free network, as shown in Accesses to Single-Pumped M20K Memory Blocks. Each single-pumped M20K block has two logical ports available. Each write port in the local memory system must be connected to all M20K blocks that your design uses to implement the memory system. Each read port in the local memory system must be connected to one M20K block. Because of these connection constraints, there must be three M20K blocks to implement the specified number of ports in `lmem`.

Note: If you change `max_replicates(3)` to `max_replicates(1)`, you observes one M20K block with arbitration between the three reads.

If you include the `doublepump` kernel attribute in your local variable declaration, you specify that the M20K memory blocks runs at double the frequency as the rest of the design.

```
__kernel void three_copies(int raddr, int waddr) {
    int __attribute__((memory,
                       numbanks(1),
                       doublepump))
                       lmem[16];
    lmem[waddr] = lmem[raddr] + lmem[raddr + 1] + lmem[raddr + 2];
    // do something with lmem
}
```

Figure 86. Accesses to Double-Pumped M20K Memory Blocks



Each double-pumped M20K block has four logical ports available. As such, there only needs to be one M20K block to implement three read ports and one write port in `lmem`.

Attention:

- Double pumping the memory increases resource overhead. Use the `doublepump` kernel attribute only if it results in actual M20K savings, improves performance, or both.
- Stores must be connected to every replicate. Hence, if there are more than three stores, the memory is not replicated. Local memory replication works well with single store.
- Because the entire memory system is replicated, you might observe potentially large M20K memory blocks.

Related Information

Kernel Attributes for Configuring Local and Private Memory Systems

8.6. Minimizing the Memory Dependencies for Loop Pipelining

Intel FPGA SDK for OpenCL Offline Compiler ensures that the memory accesses from the same thread respects the program order. When you compile an NDRange kernel, use barriers to synchronize memory accesses across threads in the same work-group.

Loop dependencies might introduce bottlenecks for single work-item kernels due to latency associated with the memory accesses. The offline compiler defers a memory operation until a dependent memory operation completes. This can impact the loop initiation interval (II). The offline compiler indicates the memory dependencies in the optimization report.

To minimize the impact of memory dependencies for loop pipelining:

- Ensure that the offline compiler does not assume false dependencies.

  When the static memory dependence analysis fails to prove that dependency does not exist, the offline compiler assumes that a dependency exists and modifies the kernel execution to enforce the dependency. Impact of the dependency enforcement is lower if the memory system is stall-free.

  — Write after read operations with data dependency on a load-store unit can take just two clock cycles (II=2). Other stall-free scenarios can take up to seven clock cycles.

  — Read after write (control dependency) operation can be fully resolved by the offline compiler.

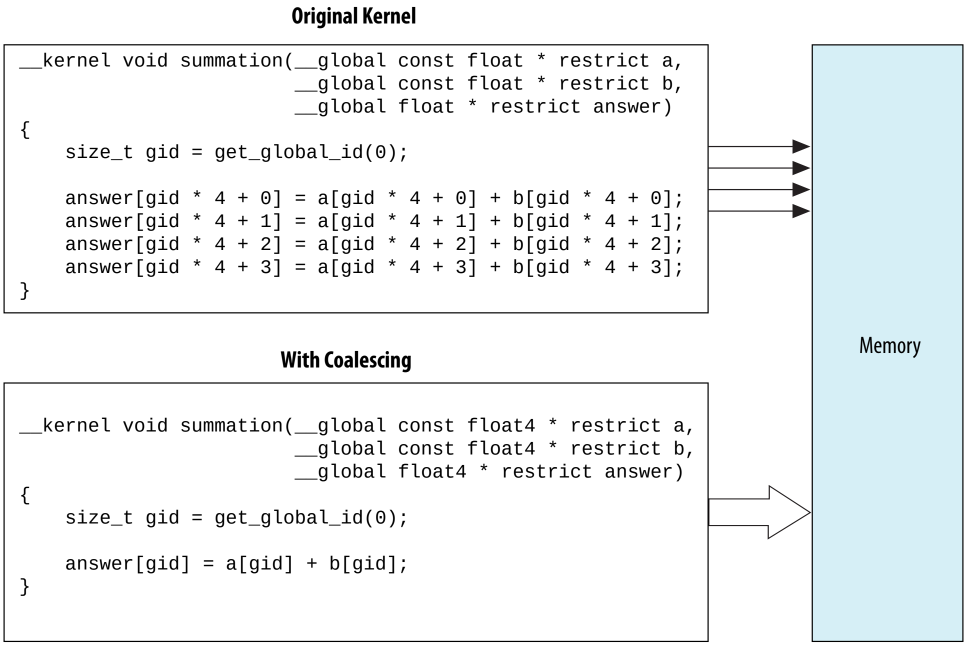
- Override the static memory dependence analysis by adding the line `#pragma ivdep` before the loop in your kernel code if you are sure that it carries no dependencies.

8.7. Static Memory Coalescing

Static memory coalescing is an Intel FPGA SDK for OpenCL Offline Compiler optimization step that attempts to reduce the number of times a kernel accesses non-private memory.

The figure below shows a common case where kernel performance might benefit from static memory coalescing:

Figure 87. Static Memory Coalescing



```
__kernel void summation(__global const float * restrict a,
                        __global const float * restrict b,
                        __global float * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid * 4 + 0] = a[gid * 4 + 0] + b[gid * 4 + 0];
    answer[gid * 4 + 1] = a[gid * 4 + 1] + b[gid * 4 + 1];
    answer[gid * 4 + 2] = a[gid * 4 + 2] + b[gid * 4 + 2];
    answer[gid * 4 + 3] = a[gid * 4 + 3] + b[gid * 4 + 3];
}
```

```
__kernel void summation(__global const float4 * restrict a,
                        __global const float4 * restrict b,
                        __global float4 * restrict answer)
{
    size_t gid = get_global_id(0);

    answer[gid] = a[gid] + b[gid];
}
```

Consider the following vectorized kernel:

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum (__global const float * restrict a,
                   __global const float * restrict b,
                   __global float * restrict answer)
```

```
{
   size_t gid = get_global_id(0);

   answer[gid] = a[gid] + b[gid];
}
```

The OpenCL kernel performs four load operations that access consecutive locations in memory. Instead of performing four memory accesses to competing locations, the offline compiler coalesces the four loads into a single wider vector load. This optimization reduces the number of accesses to a memory system and potentially leads to better memory access patterns.

Although the offline compiler performs static memory coalescing automatically when it vectorizes the kernel, you should use wide vector loads and stores in your OpenCL code whenever possible to ensure efficient memory accesses. To implement static memory coalescing manually, you must write your code in such a way that a sequential access pattern can be identified at compilation time. The original kernel code shown in the figure above can benefit from static memory coalescing because all the indexes into buffers a and b increment with offsets that are known at compilation time. In contrast, the following code does not allow static memory coalescing to occur:

```
__kernel void test (__global float * restrict a,
                  __global float * restrict b,
                    __global float * restrict answer;
                  __global int * restrict offsets)
{
 size_t gid = get_global_id(0);

 answer[gid*4 + 0] = a[gid*4 + 0 + offsets[gid]] + b[gid*4 + 0];
 answer[gid*4 + 1] = a[gid*4 + 1 + offsets[gid]] + b[gid*4 + 1];
 answer[gid*4 + 2] = a[gid*4 + 2 + offsets[gid]] + b[gid*4 + 2];
 answer[gid*4 + 3] = a[gid*4 + 3 + offsets[gid]] + b[gid*4 + 3];
}
```

The value `offsets[gid]` is unknown at compilation time. As a result, the offline compiler cannot statically coalesce the read accesses to buffer a.

9. Strategies for Optimizing FPGA Area Usage

Area usage is an important design consideration if your OpenCL kernels are executable on FPGAs of different sizes. When you design your OpenCL application, Intel recommends that you follow certain design strategies for optimizing hardware area usage.

Optimizing kernel performance generally requires additional FPGA resources. In contrast, area optimization often results in performance decreases. During kernel optimization, Intel recommends that you run multiple versions of the kernel on the FPGA board to determine the kernel programming strategy that generates the best size versus performance trade-off.

9.1. Compilation Considerations

You can direct the Intel FPGA SDK for OpenCL Offline Compiler to perform area usage analysis during kernel compilation.

1. To review the estimated resource usage summary on-screen, compile your kernel by including the `-report` flag in your aoc command. To review kernel-specific area usage information, refer to the `<your_kernel_filename>/reports/report.html` file.
2. If possible, perform floating-point computations by compiling your OpenCL kernel with the `-fpc` or `-fp-relaxed` option of the aoc command.

For more usage information about the `-report`, `-fp-relaxed` and `-fpc` options, refer to the *Displaying Estimated Resource Usage Summary (-report)*, *Relaxing Order of Floating-Point Operations (-fp-relaxed)*, and *Reducing Floating-Point Operations (-fpc)* sections of the *Intel FPGA SDK for OpenCL Programming Guide*.

For more information about floating-point operations, refer to *Optimize Floating-Point Operations*.

Related Information

- Reviewing Area Information on page 36
- Displaying the Estimated Resource Usage Summary On-Screen (-report)
- Relaxing the Order of Floating-Point Operations (-fp-relaxed)
- Reducing Floating-Point Rounding Operations (-fpc)
- Optimizing Floating-Point Operations on page 99

9.2. Board Variant Selection Considerations

Target a board variant in your Custom Platform that provides only the external connectivity resources you require.

For example, if your kernel requires one external memory bank, target a board variant that only supports a single external memory bank. Targeting a board with multiple external memory banks increases the area usage of your kernel unnecessarily.

If your Custom Platform does not provide a board variant that meets your needs, consider creating a board variant. Consult the *Intel FPGA SDK for OpenCL Custom Platform Toolkit User Guide* for more information.

Related Information

Intel FPGA SDK for OpenCL Pro Edition Custom Platform Toolkit User Guide

9.3. Memory Access Considerations

Intel recommends kernel programming strategies that can improve memory access efficiency and reduce area usage of your OpenCL kernel.

1. Minimize the number of access points to external memory.

   If possible, structure your kernel such that it reads its input from one location, processes the data internally, and then writes the output to another location.

2. Instead of relying on local or global memory accesses, structure your kernel as a single work-item with shift register inference whenever possible.
3. Instead of creating a kernel that writes data to external memory and a kernel that reads data from external memory, implement the Intel FPGA SDK for OpenCL channels extension between the kernels for direct data transfer.
4. If your OpenCL application includes many separate constant data accesses, declare the corresponding pointers using `__constant` instead of `__global const`. Declaration using `__global const` creates a private cache for each load or store operation. However, declaration using `__constant` creates a single constant cache on the chip only.

   Caution: If your kernel targets a Cyclone® V device (for example, Cyclone V SoC), declaring `__constant` pointer kernel arguments might degrade FPGA performance.

5. If your kernel passes a small number of constant arguments, pass them as values instead of pointers to global memory.

   For example, instead of passing `__constant int * coef` and then dereferencing `coef` with index 0 to 10, pass `coef` as a value (`int16 coef`). If `coef` was the only `__constant` pointer argument, passing it as a value eliminates the constant cache and the corresponding load and store operations completely.

6. Conditionally *shifting* large shift registers inside pipelined loops leads to the creation of inefficient hardware. For example, the following kernel consumes more resources when the `if (K > 5)` condition is present:

```
#define SHIFT_REG_LEN 1024
__kernel void bad_shift_reg (__global int * restrict src,
                             __global int * restrict dst,
                             int K)
{
    float shift_reg[SHIFT_REG_LEN];
    int sum = 0;

    for (unsigned i = 0; i < K; i++)
    {
        sum += shift_reg[0];
        shift_reg[SHIFT_REG_LEN-1] = src[i];

        // This condition will cause sever area bloat.
        if (K > 5)
        {
          #pragma unroll
          for (int m = 0; m < SHIFT_REG_LEN-1 ; m++)
          {
              shift_reg[m] = shift_reg[m + 1];
          }
        }
        dst[i] = sum;
    }
}
```

Attention: Conditionally *accessing* a shift register does not degrade hardware efficiency. If it is necessary to implement conditional shifting of a large shift register in your kernel, consider modifying your code so that it uses local memory.

9.4. Arithmetic Operation Considerations

Select the appropriate arithmetic operation for your OpenCL application to avoid excessive FPGA area usage.

1. Introduce floating-point arithmetic operations only when necessary.
2. The Intel FPGA SDK for OpenCL Offline Compiler defaults floating-point constants to double data type. Add an `f` designation to the constant to make it a single precision floating-point operation.

   For example, the arithmetic operation `sin(1.0)` represents a double precision floating-point sine function. The arithmetic operation `sin(1.0f)` represents a single precision floating-point sine function.

3. If you do not require full precision result for a complex function, compute simpler arithmetic operations to approximate the result. Consider the following example scenarios:

   a. Instead of computing the function `pow(x,n)` where *n* is a small value, approximate the result by performing repeated squaring operations because they require much less hardware resources and area.

   b. Ensure you are aware of the original and approximated area usages because in some cases, computing a result via approximation might result in excess area usage. For example, the `sqrt` function is not resource-intensive. Other than a rough approximation, replacing the `sqrt` function with arithmetic operations that the host has to compute at runtime might result in larger area usage.

   c. If you work with a small set of input values, consider using a LUT instead.

4. If your kernel performs a complex arithmetic operation with a constant that the offline compiler computes at compilation time (for example, `log(PI/2.0)`), perform the arithmetic operation on the host instead and pass the result as an argument to the kernel at runtime.

9.5. Data Type Selection Considerations

Select the appropriate data type to optimize the FPGA area usage by your OpenCL application.

1. Select the most appropriate data type for your application.

   For example, do not define your variable as `float` if the data type `short` is sufficient.

2. Ensure that both sides of an arithmetic expression belong to the same data type.

   Consider an example where one side of an arithmetic expression is a floating-point value and the other side is an integer. The mismatched data types cause the Intel FPGA SDK for OpenCL Offline Compiler to create implicit conversion operators, which can become expensive if they are present in large numbers.

3. Take advantage of padding if it exists in your data structures.

   For example, if you only need `float3` data type, which has the same size as `float4`, you may change the data type to `float4` to make use of the extra dimension to carry an unrelated value.

10. Strategies for Optimizing Intel Stratix 10 OpenCL Designs

This chapter of the *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide* discusses certain Intel-recommended optimization practices that you can apply to your Intel Stratix 10 OpenCL designs to use the optimization capabilities of the Intel FPGA SDK for OpenCL Offline Compiler to achieve best performance results.

Intel assumes that you are familiar with the information presented in the *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide* and know how to apply the general optimization strategies described in the document.

Reducing Channel Overhead on page 166

Optimizing Loop Control on page 169

Simplifying Memory Access to Local Memories on page 172

On-Chip Storage of Reused Data on page 173

Optimizing Data Path Control on page 174

Creating RTL Modules on page 177

10.1. Reducing Channel Overhead

Compared with earlier Intel FPGAs, channels in the Intel Stratix 10 FPGA have a higher resource overhead. This increase in resource overhead is more significant in blocking channels. Intel recommends that you reduce the number and type of channels in your design whenever possible.

The following topics outline strategies you can implement to reduce channel overhead.

Reducing the Number of Kernels on page 166

Using a Single Kernel to Describe Systolic Arrays on page 167

Using Non-Blocking Channels on page 168

10.1.1. Reducing the Number of Kernels

Instead of partitioning your design across multiple kernels, consider consolidating the design into fewer kernels. For Intel Stratix 10 designs, Intel recommends that you only use separate kernels for truly asynchronous execution.

The following example shows a `producer` kernel and a `consumer` kernel communicating via channels:

```
kernel producer(unsigned N) {
   int result;
   for (unsigned int i = 0; i < N; i++) {
      write_channel_intel(Produce(i));
   }
}

kernel consumer(unsigned N) {
   for (unsigned int i = 0; i < N; i++) {
      Consume(i, read_channel_intel());
   }
}
```

The optimized code below merges the two kernels in the example above into a single kernel, which uses the computation results directly without channel accesses:

```
kernel fused(unsigned N) {
   for (unsigned int i = 0; i < N; i++) {
      Consume(i, Produce(i));
   }
}
```

10.1.2. Using a Single Kernel to Describe Systolic Arrays

For an Intel Stratix 10 OpenCL design, Intel recommends that you describe a systolic array as a single kernel, using a function for the processing element (PE), and a fully-unrolled loop or nested loop to represent the array.

Unoptimized multi-kernel systolic array pseudocode:

```
// data distribution network over an array of channels

channel int c[ROWS][COLS];
channel int d[ROWS][COLS];

attribute((num_compute_units(ROWS,COLS))
kernel void PE() {
   // get data values from my neighbors
   while(1){
      x = read_channel_intel(c[ROWS-1][COLS]);
      y = read_channel_inel(d[ROWS][COLS-1]);


      // some code that uses x and y
      ...
      // send the same data values to the next neighbors
      write_channel_intel(c[ROWS][COLS], x);
      write_channel_intel(d[ROWS][COLS], y);
   }
}
```

Optimized single-kernel pseudocode:

```
kernel void allPEs() {
   while(1){
      int c[ROWS], d[COLS];

      #pragma unroll
      for (int i = 0; i < ROWS; i++)
         #pragma unroll
         for (int j = 0; j < COLS; j++) {
            PE(c[i], d[j]);
         }
      }
   }
}
```

Note: Instead of a kernel, the PE body becomes the function call `PE()`. Unrolling the loops results in an array of PEs, each of which uses a portion of the FPGA in a 2D array.

Depending on the size of the array, it can be challenging for the Intel FPGA SDK for OpenCL Offline Compiler to generate hardware that distributes the same values c and d to all PEs on a row or column of the array within a single clock cycle. Doing so might cause f_{MAX} to degrade. To remedy this problem, consider using the __fpga_reg() function to instruct the offline compiler to insert registers on c and d with every new PE. Intel recommends that you only use the __fpga_reg() function when you know that the PEs are spatially separate from one another on the FPGA.

Note: The __fpga_reg() built-in function is an advanced feature. The offline compiler does not provide guidance on where you should insert the __fpga_reg() function calls. To help determine whether it is appropriate to insert the __fpga_reg() function call, you can experimentally quantify the impact additional registers might have on f_{MAX}, and inspect the Intel Quartus Prime compilation reports.

Optimized pseudocode with the __fpga_reg() function:

```
kernel void allPEs() {
   int c[ROWS], d[COLS];

   while(1){
      #pragma unroll
      for (int i = 0; i < ROWS; i++)
         #pragma unroll
         for (int j = 0; j < COLS; j++) {
            // compute and store outputs
            PE(c[i], d[j]);
            c[i] = __fpga_reg(c[i]);
            d[j] = __fpga_reg(d[j]);
         }
      }
   }
}
```

After the offline compiler unrolls the loop, there is one more register before every PE on both c and d, allowing the Intel Quartus Prime Pro Edition software to place the PEs apart. You may add more than one register by inserting multiple __fpga_reg() calls in your code. For example, the call __fpga_reg(__fpga_reg(x)) adds two registers on the data path. However, having excessive __fpga_reg() calls in your kernel increases the design area, and the congestion might result in f_{MAX} degradation.

Related Information

Intra-Kernel Registered Assignment Built-In Function

10.1.3. Using Non-Blocking Channels

If you must implement channels in your Intel Stratix 10 OpenCL designs, consider using non-blocking channels. This may reduce area overhead in some cases.

The example code below has a blocking channel read:

```
while (cond) {
   val = read_channel_intel (my_ch);
   <do_compute (val)>
}
```

To switch to a non-blocking channel read that is functionally equivalent to the blocking channel read, modify the code in the following manner:

```
bool have_data = true;
while (cond) {
   val = read_channel_nb_intel (my_ch, &have_data);
   if (have_data) <do_compute (val)>
}
```

For this example, the downside of changing from a blocking channel read to a non-blocking channel read is that the loop control logic becomes more complex. If you transform multiple channel accesses this way, the loop control logic might limit your performance or actually increase area overhead.

10.2. Optimizing Loop Control

Intel Stratix 10 OpenCL designs leverage the the FPGA's Hyperflex™ architecture to achieve high performance. Because the Hyperflex architecture allows OpenCL designs to run faster, it becomes more critical to optimize loop structures in Intel Stratix 10 OpenCL designs; otherwise, they can cause notable performance limitations.

To achieve high performance, Intel recommends achieving a loop initiation interval (II) of 1. An II value of 1 indicates that a loop is able to start a new iteration of a loop data path every clock cycle. Doing so helps your design consume the available FPGA resources efficiently.

Intel has established a new loop control scheme specifically for the Intel Stratix 10 architecture.

Applying Loop Control Optimization in Intel Stratix 10 OpenCL Designs

Leveraging the Intel Stratix 10 Hyperflex architecture, you can now create deeply pipelined loops in your design to achieve higher f_{MAX}. Because the offline compiler might not be able to calculate the exit condition of such a complex loop structure in a single clock cycle, the offline compiler now defers the complete calculation of the exit condition. The compiler decouples the calculation from the loop body and splits the calculation across multiple clock cycles. Doing so allows loop iterations to launch each clock cycle before the compiler finishes calculating the exit condition; however, it takes a few clock cycles to flush the loop after the loop exit condition is signaled.

Refer to the Loop analysis report in the High Level Design Report (`report.html`) to find out to which loops the offline compiler has applied the new loop optimization strategy.

Effects of the new loop control optimization strategy:

1. Allows iterations of the current loop launch much faster.
2. Subsequent invocations of the loop starts after the current invocation flushes all the data.

Note: A loop iteration is one execution of the loop body. A loop invocation is one execution of an entire loop, from the initial value of the loop counter until the exit condition becomes TRUE.

The following code example illustrates the termination overhead associated with a nested loop:

```
kernel loop_overhead(unsigned N) {
   for (unsigned int i = 0; i < N; i++) {
      for (unsigned int j = 0; j < N; j++) {
         //do work
         //total iterations: i * (j + s)
      }
   }
}
```

The II value of this loop is 1; however, the number of clock cycles it takes to issue all the loop iterations in the nest is N × (N+s), where s is number of cycles it takes to flush the loop before the launch of the next few iterations. The loop overhead s is small; it does not have a notable effect on the design unless the design has very few iterations in the inner loop.

Types of Loops that Benefit from Loop Control Optimization

Most loops, even those with loop control that is deeply pipelined and with complex exit conditions, are able to achieve an II value of 1. This optimization strategy is primarily beneficial for high throughput designs with loops that have many iterations. The f_{MAX} increase in these designs adequately compensates for the comparatively small overhead on termination.

Note: The extent to which loop control is pipelined does not affect the loop's II value.

There are some loops to which the loop optimization strategy is not applicable:

- Loops in an NDRange kernel

  Because the offline compiler must be able to pipeline the loop, the loop must be part of a single work-item kernel.

- Loops with exit conditions that depend on instructions that can stall or have side effects outside the loop

The following are examples of loops that use the new loop control scheme versus those that do not:

Example 1: Loop can achieve optimal performance on Intel Stratix 10

```
kernel void good_loop(global int * restrict A,
                      global int * restrict result,
                      unsigned N) {

   unsigned int sum = 0;

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

Example 2: Loop can achieve optimal performance on Intel Stratix 10

In this example, the channel write has side effects outside the loop; however, the exit condition does not depend on the channel write.

```
channel unsigned int c0;

kernel void producer() {
   for (unsigned int i = 0; i < 10; i++) {
      write_channel_intel(c0, i);
   }
}
```

Example 3: Loop cannot fully benefit from the Intel Stratix 10 loop control scheme because the exit condition depends on the channel read (`read_channel_intel`) that might have side effects outside the loop. As a result, the computation for each iteration cannot proceed until the compiler determines the exit condition, otherwise the compiler does consume additional data from the channel.

```
kernel void consumer (global int * restrict A,
                      global int * restrict result,
                      unsigned N) {
   unsigned int sum = 0;
   for (unsigned int i = 0;
        i < N && read_channel_intel(c0) != 5; i++) {
      sum += A[i];
   }
   *result = sum;
}
```

If the offline compiler does not implement the new loop optimization, it also disables other f_{MAX} optimizations.

Warning: Disabling these optimizations might reduce the amount of logic usage at the expense of f_{MAX}. Check the offline compiler's HTML reports to verify the outcome of the compiler optimizations.

Related Information

Reviewing Your Kernel's report.html File on page 15

10.2.1. Simplifying Loop-Carried Dependencies in Intel Stratix 10 OpenCL Designs

To ensure that your Intel Stratix 10 OpenCL design achieves optimal performance, ensure that loop-carried computation is as simple as possible so that the Intel FPGA SDK for OpenCL Offline Compiler can compute in one clock cycle.

The offline compiler cannot pipeline computations used for loop-carried dependencies. Loops that contain many complex computations limit the amount of retiming optimizations that the compiler can perform because the compiler cannot make any functional changes to the loop path. Even if II=1, the HTML report identifies the f_{MAX} bottleneck. Use this information in conjunction with the information presented in the Loop Analysis report pane to assess the most critical paths in your design.

If a loop-carried dependency contains logic that the offline compiler cannot compute in one clock cycle, one mitigation approach is to lengthen the dependency distance. The dependency distance is the number of loop iterations that occur from when the compiler reads the value to when the next value becomes available. The Loop analysis report within the High Level Design Report identifies the most complex loop dependency.

Automated Loop-Carried Dependency Optimization

For Intel Stratix 10 designs, the Intel FPGA SDK for OpenCL Offline Compiler attempts to automate the incrementation of a value modulo *N* (mod *N*) on every iteration of a loop.

You can apply this optimization manually for any operation that is associative and communicative. If you refactor the code this way, the compiler can spread the computation across two or more loop-carried variables, and it can recombine the computation when the value is needed in a non-loop-carried computation. For more information, refer to Safari, Nima et al. "Methods for Implementation of Feedback Loops in High Speed FPGA Applications". *24th International Conference on Field Programmable Logic and Applications (FPL)* (2014) doi:10.1109/FPL.2014.6927434.

Intel recommends this optimization for operations that are on your design's critical path. Consider the following example:

```
int i = 0;
int N = 256;
while (!done) {
   i++;
   if (i == N) i = 0;
   <use i for some computation…>
}
```

On each loop iteration, the offline compiler must increment a value, compare it to a constant, and then reset the value if necessary. To optimize this code, the compiler effectively breaks down the expression and spreads the computation across two clock cycles to increase the dependence distance. The side effect of this optimization is a small increase in logic usage.

There are scenarios in which the offline compiler might not optimize the example code:

- If the initial value of `i` is non-zero, and the compiler cannot determine that the initial value is between 0 and *N*, the compiler cannot guarantee that the forms above are functionally equivalent.
- If any condition causes `i` to be modified or reset to 0, the offline compiler does not apply the optimization.

Related Information

Safari, Nima et al. "Methods for Implementation of Feedback Loops in High Speed FPGA Applications"

10.3. Simplifying Memory Access to Local Memories

The Intel Stratix 10 FPGA hardware has two ports per M20K memory. For other Intel FPGA device families, the Intel FPGA SDK for OpenCL Offline Compiler allows the Memory System Clock to run at 2x the main clock frequency, effectively providing four ports per M20K memory. For more information, refer to Double Pumping on page 60. However, for Intel Stratix 10 FPGAs, the offline compiler is currently discouraged from inferring a 2x Memory System Clock because transferring data between the 2x clock domain and the main clock at high speed generally leads to significant f_{MAX} degradation. You can still force double pumping for a given Memory System by applying the memory attribute `__attribute__((doublepump))`.

Due to the potential f_{MAX} implications of double pumping on Intel Stratix 10, Intel recommends limiting the number of concurrent stores.

Multiple concurrent stores often require more than two ports to implement. Multiple concurrent stores also cause the compiler to create stallable memories where memory accesses must be arbitrated on every clock cycle. To determine if a memory is arbitrated, examine the load and store units in the system or memory viewer of the High Level Design Report. In the viewer, load and store units that are highlighted red are stallable memories. The report presents this information when you hover over each highlighted unit.

Related Information

Strategies for Improving Memory Access Efficiency on page 145

10.4. On-Chip Storage of Reused Data

For Intel Stratix 10 designs, the presence of cached LSUs prevent certain optimizations. Intel recommends that you avoid inferring caching LSUs.

The unoptimized code below creates a cached burst-coalesced LSU that consumes more resources and disables other optimizations:

```
kernel void cached (global int * restrict in,
                    global int * restrict out) {
   int i = get_global_id(0);
   int idx = out[i];
   int cached_value = in[idx]; // Burst-coalesced cached LSU
   out[i] = cached_value;
}
```

To prevent the caching, mark the memory pointer as `volatile`, as shown in the optimized code below:

```
kernel void not_cached (global volatile int * restrict in,
                        global int * restrict out) {
   int i = get_global_id(0);
   int idx = out[i];
   int not_cached_value = in[idx];
   out[i] = not_cached_value;
}
```

For more information about optimizing load-store units, refer to the *Load-Store Units* section.

Intel also recommends that you use on-chip storage to achieve optimal results. Note that compared to non-Intel Stratix 10 devices,Intel Stratix 10 has a larger M20K to ALM ratio, allowing you to create larger local memory systems.

The unoptimized code below has a function that receives a pointer from the array in global memory. In this case, the offline compiler modifies the array and then stores it back to memory. Then, in a subsequent iteration of the outer loop, the array is reused.

```
void cached_array(int N, int M, int BUF_SIZE,
                  global float2 * global_buf)
{
   for (uint i = 0; i< N; i++) {
      float2 data_array[BUF_SIZE];
      for (uint j= 0; j < M; j++) {

         data_array[i] = global_buf [j]; //Load value
         ... //do work to modify data_array

         global_buf[j] = data_array[i];
      }
   }
}
```

To prevent unnecessary global memory accesses, define a private array in the kernel to declare the array on chip. The function accesses the private array instead of the one declared on chip. As a result, the array receives on-chip local memory storage. Accessing this on-chip local memory does not require accesses to global memory.

Optimized code:

```
void local_array(int N, int M, int BUF_SIZE,
                 global float2 * global_buf)
{
   float2 local_buf[BUF_SIZE];
   populate_buf(local_buf, global_buf);

   for (uint i = 0; i< N; i++) {
      float2    data_array[BUF_SIZE];
      for (uint j= 0; j < M; j++) {

         data_array[i] = local_buf[j];//Load value

         ... //do work to modify data_array

         local_buf[j] = data_array[i];

      }
   }
}
```

Related Information

Load-Store Units on page 85

10.5. Optimizing Data Path Control

To best use the Intel Stratix 10 design-specific new data path optimizations, modify your code to remove constructs or features that might prevent the implementation of these optimizations. If there are such constructs or features in your design, the Intel FPGA SDK for OpenCL Offline Compiler will revert to the legacy optimizations, which might result in a lower f_{MAX}. For example, if the offline compiler must instantiate cached LSUs for the memory access pattern, it will not enable the new optimizations.

The following constructs or features prevent Intel Stratix 10 data path optimization:

- NDRange designs with loops
- Stallable LSUs with the exception of burst-coalesced LSUs

  Burst-coalesced LSUs are the default type of LSUs that the offline compiler instantiates. Example of a burst coalesced LSU instantiation:

  ```
  kernel void burst_coalesced (global int * restrict in,
                               global int * restrict out){
     int i = get_global_id(0);
  ```

```
   int value = in[i/2];  //Burst-coalesced LSU
   out[i] = value;
}
```

You can view the LSU type of various instructions in the High Level Design Report by hovering over the load or store operation in the System Viewer. Refer to the *Load-Store Units* section for more information about the types of LSUs and how you can influence the compiler on which type of LSUs to instantiate.

- Channels with multiple call sites
- Stallable RTL library calls

  Refer to the *Create RTL Modules* section for more information.
- Reconvergent control flow in the optimized control flow graph, with the exception of loops that use the new control optimization

  The following pseudocode example of a simple reconvergent control flow shows that the flow of the code goes in one of two paths. The offline compiler implements different control logic for each path. It also implements logic to reconverge the control flow after the two paths are completed.

  ```
  while (some_some condition){
     if (some_other_condition){
        for(...){ }
     } else{
        for(...){ }
     }
  }
  ```
- Loops that do not use the new loop control scheme

  Refer to the *Loop Control Optimization* section for more information about what loops are affected by this restriction.
- Basic block structures with the exception of the following:
  - Basic block with only one predecessor, as shown in Basic Block Structure with One Predecessor
  - Basic block with exactly two predecessors, where one predecessor is the back-edge of a loop, as shown in Basic Block Structure with Two Predecessors

  Note: The majority of optimized designs belong to one of the two supported basic block structures. You may review images of these basic blocks in the System Viewer of the High Level Design Report.

  The following code example generates the two types of supported basic block structures:

  ```
  __attribute__((max_global_work_dim(0)))
  void kernel basic_block(global unsigned int *myvar,
                              unsigned int insize)
  {
     for(int i=0; i < insize; i++){
        myvar[i] += insize;
     }
  }
  ```

Figure 88. Basic Block Structure with One Predecessor

The basic block in question refers to gzip.B2, which is outlined in red. Its predecessor is the previous basic block, gzip.B1.

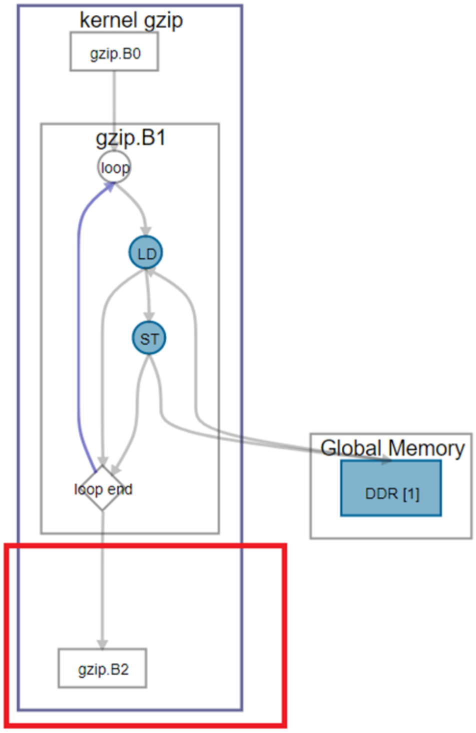
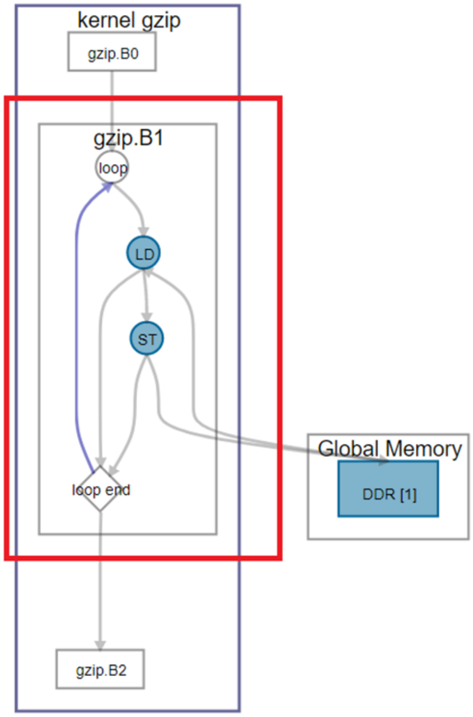


Figure 89. Basic Block Structure with Two Predecessors

The basic block in question is gzip.B1, which is outlined in red. This block refers to the body of the loop; its predecessors are itself and the previous basic block, gzip.B0. The purple line in gzip.B1 denotes the back end of the loop.



Related Information

- Optimizing Loop Control on page 169
- Creating RTL Modules on page 177
- Load-Store Units on page 85

10.6. Creating RTL Modules

You may embed RTL modules in an OpenCL kernel. For more information about how to integrate an RTL module into your OpenCL design, refer to the *Understanding RTL Modules and the OpenCL Pipeline* section. To create an Intel Stratix 10 OpenCL-compatible RTL module, you must understand the Intel Stratix 10-specific changes to the reset signal, and know how to write compatibly pipelined interfaces to the RTL module.

For more information about Intel Stratix 10-specific RTL design best practices, refer to the *Intel Stratix 10 High-Performance Design Handbook*.

There are stall-free and stallable RTL modules. A stall-free RTL module is a fixed-latency module for which the offline compiler can optimize away stall logic. Refer to the *Stall-Free RTL* section in the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide* for more information.

A stallable RTL module has variable latency and relies on backpressured input and output interfaces to function correctly. Implementing stallable interfaces in Intel Stratix 10 designs consumes a lot of FPGA resources because of the handshake logic, which limits retiming. Using stallable interfaces in Intel Stratix 10 designs also disables the data path control optimization scheme.

Intel strongly recommends that you use stall-free RTL modules because the offline compiler can incorporate them into your Intel Stratix 10 designs more effectively.

Note: There are important differences between the reset requirements for stall-free and stallable RTL modules. These requirements are necessary for the functional correctness of the RTL modules. For more information, refer to *Intel Stratix 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules* in the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide*.

Related Information

- Understanding RTL Modules and the OpenCL Pipeline
- Intel Stratix 10 High-Performance Design Handbook
- Stall-Free RTL
- Intel Stratix 10 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Modules

10.6.1. Reset Recommendations

Intel provides recommendations on how to handle resets in your Intel Stratix 10 OpenCL designs. The Intel FPGA SDK for OpenCL Offline Compiler applies these recommendations automatically. If you are developing an RTL library, you should implement these guidelines for best performance.

In traditional FPGA RTL for non-Intel Stratix 10 devices, it is common practice to reset every register indiscriminately for easy implementation without negative effects on performance. However, to improve the performance of your Intel Stratix 10 design, you must minimize the number of resets.

Note: If a register is not reset, the reset fanout signal is reduced by one.

Avoid unnecessary resets in your Intel Stratix 10 design for the following reasons:

- The resulting high-fanout signal prevents the Intel Quartus Prime Pro Edition software's retimer to find a satisfactory solution.

  For more in-depth explanation, refer to the *Avoid Broadcast Signals* section in the *Intel Stratix 10 High-Performance Design Handbook*. Note that the term "broadcast signals" refers to high-fanout signals.
- In some situations, simply having a reset on a register, regardless of whether it is a high-fanout reset signal, is enough to degrade the performance of your Intel Stratix 10 design.

  For more in-depth explanation, refer to the *Synchronous Resets and Limitations* section in the Intel Stratix 10 High-Performance Design Handbook.

Intel recommends the following design guidelines for resets:

- To improve Intel Stratix 10 design performance, do not reset registers that do not hold internal states to reduce reset fanout.
- The reset signal is guaranteed to remain asserted for at least 50 clock cycles. Use this guaranteed assertion by "flushing" chains of registers that have internal states, which further reduces the reset fanout.
- You have the option to pipeline the reset signal inside the RTL module for fanout management, to a depth of no more than 15 pipeline registers for stall-free RTL modules, or no more than 25 pipeline registers for stallable RTL modules. If the RTL module is sufficiently large, pipelining the reset signal might improve design performance.

Related Information

- Avoid Broadcast Signals
- Synchronous Resets and Limitations

11. Strategies for Improving Performance in Your Host Application

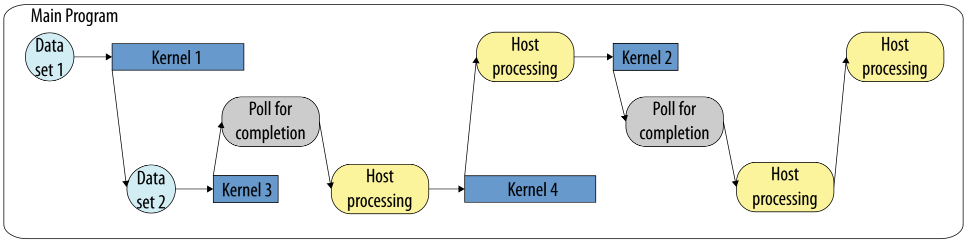
Multi-Threaded Host Application on page 180
Utilizing Hardware Kernel Invocation Queue on page 181

11.1. Multi-Threaded Host Application

When there are parallel, independent data paths and the host must process the data between kernel executions, consider using a multi-threaded host application.

The figure below illustrates how a single-threaded host application might process parallel, independent data paths between kernel executions:

Figure 90. Kernel Execution by a Single-Threaded Host Application



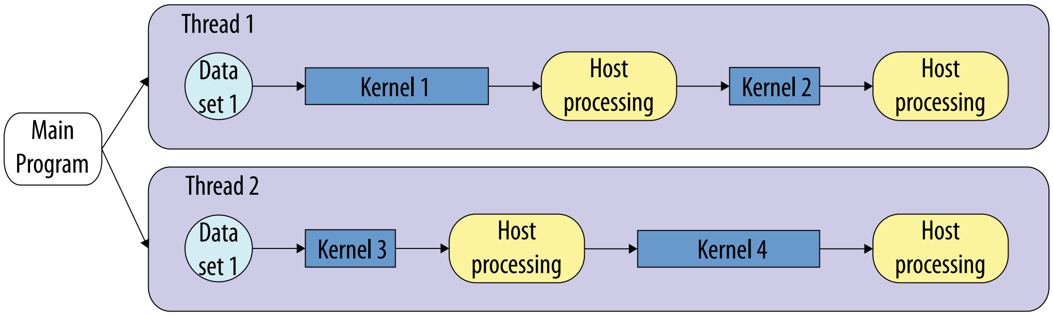
The OpenCL runtime is thread safe and supports multithreaded applications. Thus, you can perform work tasks on the host in parallel threads while still allowing those threads to access the OpenCL APIs in a thread-safe way.

However, thread safety is enforced at the OpenCL API boundary by synchronizing threads immediately after any OpenCL host API is called, which means, only one thread is ever active in the runtime while the other threads wait. As such, if one thread calls `clFlinish` on a queue full of work, another thread calling `clSetKernelArg` may have to wait for all of that work to complete before executing.

If possible, process data on CPU on multiple-threads, and use single dedicated thread to interact with the OpenCL API.

The figure below illustrates how a multi-threaded host application processes parallel, independent data paths between kernel executions:

Figure 91. Kernel Execution by a Multi-Threaded Host Application in a Thread-Safe Runtime Environment



Related Information

Multiple Host Threads

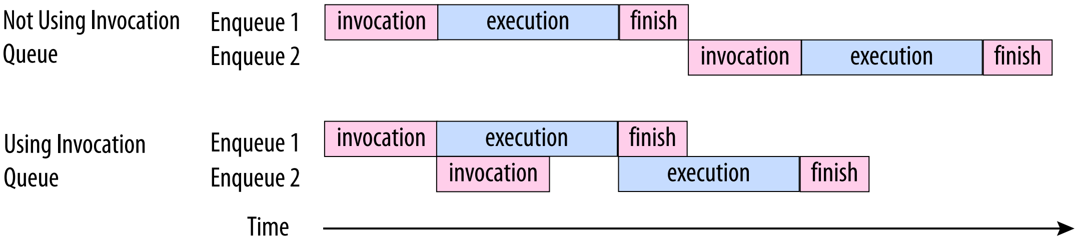
11.2. Utilizing Hardware Kernel Invocation Queue

OpenCL kernels are built with invocation queue to enable immediate launch of next invocation.

Tip: If you are looking for Intel oneAPI DPC++/C++ Compiler-specific details, refer to Utilizing Hardware Kernel Invocation Queue topic in the *FPGA Optimization Guide for Intel oneAPI Toolkits*.

As illustrated in the following figure, when the invocation queue is used, system and OpenCL runtime environment overheads (from responding to the finish and sending in the next set of invocation arguments) are overlapped with the kernel executions. This allows kernels to execute continuously, maximizing the system level throughput.

Figure 92. Kernel Execution With and Without Invocation Queue



Kernel invocations are queued in hardware when another enqueued kernel with the same kernel function name and program is already running on the device, and the following are true:

- OpenCL kernel is not compiled with hardware kernel invocation buffer disabled (`-no-hardware-kernel-invocation-buffer`).
- OpenCL kernel is not compiled with performance counters (`-profile`)
- Enqueued OpenCL kernel does not have `printf`.

- All event objects queued earlier in the command queue have execution status equal to `CL_COMPLETE`.

  If the status is `CL_SUBMITTED` or `CL_RUNNING`, then that status is for the enqueue kernel with the same kernel function name in the same program.
- All event objects in the event wait list have execution status equal to `CL_COMPLETE.`

  If the status is `CL_SUBMITTED` or `CL_RUNNING`, then that status is for the enqueue kernel on the same device with the same kernel function name in the same program.
- If the OpenCL kernel uses heterogeneous memory, kernel currently running on the device and the one getting enqueued did not set the same memory object on different memory types.

Consider the following two example host code snippets where the compiler can queue kernel invocation on hardware kernel invocation queue:

Example 1

```
int main()
{    …
  clEnqueueNDRangeKernel(queue, kernel, …, NULL);
  clEnqueueNDRangeKernel(queue, kernel, …, NULL);
  …
}
```

As soon as the first enqueue kernel is running, the compiler can queue the second enqueue kernel on hardware.

Example 2

```
int main()
{    …
  clEnqueueNDRangeKernel(queue0, kernel0, …, NULL);
  clEnqueueNDRangeKernel(queue1, kernel1, …, NULL);
  clEnqueueNDRangeKernel(queue0, kernel0, …, NULL);
  clEnqueueNDRangeKernel(queue1, kernel1, …, NULL);
  …
}
```

As soon as the first enqueue of `kernel0` is running, the compiler can queue the second enqueue of `kernel0` on the hardware irrespective of the `kernel1` status. Similarly, as soon as the first enqueue of `kernel1` is running, the compiler can queue the second enqueue of `kernel1` on the hardware irrespective of the `kernel0` status.

Now, consider the following two examples where the compiler cannot queue kernel invocation on hardware:

Example 1

```
int main()
{    …
  clEnqueueNDRangeKernel(queue, kernel0, …, NULL);
  clEnqueueNDRangeKernel(queue, kernel1, …, NULL);
  clEnqueueNDRangeKernel(queue, kernel0, …, NULL);
  …
}
```

Since the queue is in-order, enqueue `kernel1` prevents the second enqueue of `kernel0` from being queued on the hardware invocation queue.

Example 2

```
int main()
{    …
  clEnqueueNDRangeKernel(queue0, kernel0, …, NULL);
  clEnqueueNDRangeKernel(queue1, kernel1, …, &event);
  clFlush(queue1);
  clEnqueueNDRangeKernel(queue0, kernel0, …, 1, &event, NULL);
  …
}
```

Since the second enqueue of `kernel0` is waiting on enqueue of `kernel1` to complete, it only gets queued on the hardware kernel invocation queue if `kernel1` finishes execution before first enqueue of `kernel0` finishes.

Attention: If the difference in `clGetEventProfilingInfo()` time between `CL_PROFILING_COMMAND_END` and `CL_PROFILING_COMMAND_START` flags is used to calculate execution time of enqueue kernel commands, it is possible that the execution time is zero if it is queued on the invocation queue. Use the following formula to calculate average execution time of the kernel across multiple enqueues instead:

$$t_{avg} = \frac{end_n - start_1}{n} \quad \text{OR} \quad t_{avg} = \frac{\sum_{i=1}^{n} end_i - start_i}{n}$$

Related Information

- Disabling Hardware Kernel Invocation Queue (-no-hardware-kernel-invocation-queue)
- Instrumenting the Kernel Pipeline with Performance Counters (-profile) on page 109

11.2.1. Double Buffered Host Application Utilizing Kernel Invocation Queue

Double buffering in OpenCL host application allows OpenCL runtime environment to coalesce memory transfers and kernel execution.

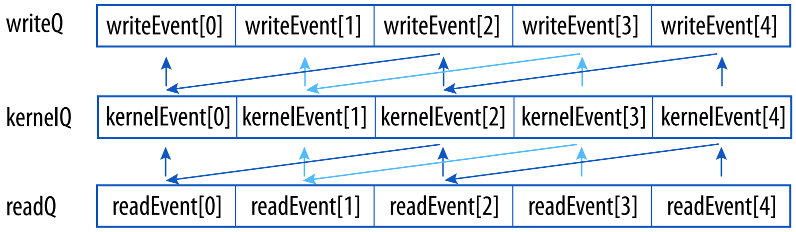
To utilize hardware kernel invocation queue while double buffering, write your host code as shown in the following code snippet:

```
int main()
{    …
  cl_event dependencies[2];
  for (int i=0; i<MAX_ITERATIONS; i++) {
    if (i < 2) {
      clEnqueueWriteBuffer(writeQ,  inputBufferD[i%2],  CL_FALSE,  …, 
inputBufferH[i],  0,  NULL,  &writeEvent[i]);
      clFlush(writeQ);
      clSetKernelArg(kernel,  0,  sizeof(cl_mem *),  &inputBufferD[i%2]);
      clSetKernelArg(kernel,  1,  sizeof(cl_mem *),  &outputBufferD[i%2]);
      clEnqueueNDRangeKernel(kernelQ,  kernel,  …,  1,  &writeEvent[i], 
&kernelEvent[i]);
      clFlush(kernelQ);
    } else {
      clEnqueueWriteBuffer(writeQ,  inputBufferD[i%2],  CL_FALSE,  …,
inputBufferH[i],  1,  &kernelEvent[i-2],  &writeEvent[i]);
      clFlush(writeQ);
      dependencies[0] = writeEvent[i];
      dependencies[1] = readEvent[i-2];
      clSetKernelArg(kernel,  0,  sizeof(cl_mem *),  &inputBufferD[i%2]);
      clSetKernelArg(kernel,  1,  sizeof(cl_mem *),  &outputBufferD[i%2]);
      clEnqueueNDRangeKernel(kernelQ,  kernel,  …,  2,  dependencies,
&kernelEvent[i]);
      clFlush(kernelQ);
    }
    clEnqueueReadBuffer(readQ,  output_device[i%2],  CL_FALSE,  …,
outputBufferH[i],  1,  &kernelEvent[i],  &readEvent[i]);
    clFlush(readQ);
  }
  …
}
```

The following diagram helps you in visualizing the event dependency:

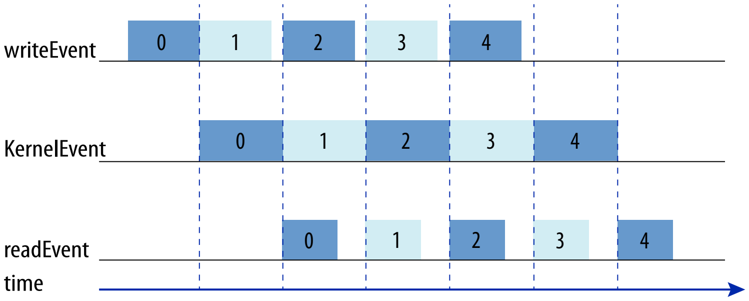
Note: Arrows represent the source of event in the event wait list.

Figure 93. Event Dependency Graph



The following figure illustrates the order the commands are executed on the device assuming kernel execution is longer than reads and writes, and the device supports concurrent reads and writes:

Figure 94. Order of Event Execution



12. Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide Archives

For the latest and previous versions of this release notes, refer to *Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide*. If a software version is not listed, the guide for the previous software version applies.

A. Document Revision History for the Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide

Document Version	Intel Quartus Prime Version	Changes
2022.12.19	22.4	• Maintenance release.
2022.09.26	22.3	• Maintenance release.
2022.06.21	22.2	• Maintenance release.
2022.03.28	22.1	• Maintenance release.
2021.12.13	21.4	• Maintenance release.
2021.10.04	21.3	• Maintenance release.
2021.06.21	21.2	• Removed some outdated images and improved some of the descriptions in the *Reviewing Your Kernel's report.html File* chapter.
2021.03.29	21.1	• Updated the messages in *Area Report Messages for Private Variable Storage*. • Added a new section in *Optimize Global Memory Accesses* about how to calculate the global memory bandwidth use. • Added a new topic *Reviewing Global Memory Information* to describe the global memory view of the System Viewer in the `report.html` file. • Updated the topic *Features of the Schedule Viewer* to include details about the dependency lines. • Changed the Graph Viewer report name to System Viewer.
2020.12.14	20.4	Maintenance release.
2020.09.28	20.3	• Minor update to *Loop Fusion* topic about trip count condition relaxation. • Added a topic *Viewing Throughput Bottlenecks in the Design*. • Added a new topic *Loop Bottlenecks*. • Updated *Accessing HLD FPGA Reports in JSON Format* and *High-level Design Report Layout* topics to include Bottlenecks viewer. • Made minor update in *Profiling Your Kernel to Identify Performance Bottlenecks* and *Best Practices for Profiling Your Kernel*. • In *Instrumenting the Kernel Pipeline with Performance Counters (-profile)*, removed a bullet point about running the host application from the local disk. • Updated the topic title and description of *Invoking the Profiler Runtime Wrapper* • Updated the *Profiling Autorun Kernels* topic completely. • Renamed the topic *Intel VTune™ Profiler* as *Viewing Profiling Data Using Intel VTune Profiler* and made minor update to the topic description. • In the *Performance Data Types* topic, updated the description, added two new information types in the Types of Performance Data table, removed the Types of Information of table, and added a note. • Made minor update in the *Interpreting the Profiling Information* topic description. • Made minor update in the *Stall, Occupancy, Bandwidth* topic description.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Removed information about the Intel FPGA dynamic profiler for OpenCL and the screenshot in *High Stall Percentage* topic. • Minor update to the topic titles of *No Stalls, Low Occupancy Percentage, and Low Bandwidth* and *No Stalls, High Occupancy Percentage, and Low Bandwidth*and updated their images. • In *Intel FPGA Dynamic Profiler for OpenCL Limitations*, removed a limitation and added a new limitation. • Removed the following topics: — Intel FPGA Dynamic Profiler for OpenCL GUI — Launching the Intel FPGA Dynamic Profiler for OpenCL GUI (report) — Source Code Tab — Tool Tip Options — Kernel Execution Tab — Autorun Captures Tab — Activity — Cache Hit — Low Bandwidth Efficiency — Autorun Profiler Data • Added the following new topics: — *Reducing Area Resource Usage While Profiling* — *Obtaining Profiling Data During Runtime* — *Splitting Execution and Data Post Processing* — *Temporal Performance Collection* — *Channel Depths*
2020.06.22	20.2	• Updated a guideline about the use of `local_mem_size` attribute in *Preloading Data to Local Memory*. • Added scheduler's behavior in different scenarios to *Reviewing Loop Information*. • Removed Out of Order Loop Iterations section in *Nested Loops* topic. • Made minor update regarding the support for double pumping in Intel Stratix 10 devices in *Simplifying Memory Access to Local Memories*
2020.04.13	20.1	• Updated the topic title and entire topic of *Optimizing for Two or More Banks of Global Memory*. • Updated the entire *Reviewing Your Kernel's report.html File* chapter. • Removed the *Reviewing* f_{MAX} *II Information* topic since F_{max} II report is deprecated. See Loop Analysis report. • Added f_{max} related information to the Loop Analysis report. • Added a new topic *Performance Data Types*. • Added a new topic *Intel VTune Profiler*. • Added a new topic *Invoking the Profiler Runtime Wrapper to Obtain Profiling* . • Made minor updates and reorganized the existing topics of *Profiling Your Kernel to Identify Performance Bottlenecks* chapter. • Added a new topic *Loop Fusion*. • Updated the *Loops in a Single Work-Item Kernel* topic completely. • Updated the *Loop-Carried Dependencies that Affect the Initiation Interval of a Loop* topic completely. • Updated the *Trade-Off Between Initiation Interval and Maximum Frequency* topic completely.

continued...

Document Version	Intel Quartus Prime Version	Changes
2019.09.30	19.3	- Updated the topic *Load-Store Units* completely. — Removed *Streaming Load-Store Units*, *Semi-Streaming Load-Store Units*, and *Global Infrequent Load-Store Units* sections. — Changed *Local-Pipelined Load-Store Units* as *Pipelined Load-Store Units* and added more information within this section. — Updated the code snippet in the *Cached* section. — Added new topics *Controlling the Load-Store Units* and *When to Use Each LSU*. - Updated the *Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor* topic completely and replaced the code snippets. - Updated the *Channels* topic to include more information about the `depth` attribute. - Added a new topic about *Schedule Viewer*. - Minor updates in *Reviewing Block Information* and *Reviewing Cluster Information* topics. - Added a new topic *Reviewing System Information* and moved some of the existing instructions to this page. - Removed system view related information and images from the *Features of the Graph Viewer* topic and moved it to the *Reviewing System Information* topic. - Updated images in *High Level Design Report Layout* and *Reviewing the Report Summary* topics. - Made minor updates in *Accessing HLD FPGA Reports in JSON Format* topic.
2019.07.01	19.2	- Added the following topics from the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide* in the *Profiling Your Kernel to Identify Performance Bottlenecks*: — *Launching the GUI (report)* — *Instrumenting the Kernel Pipeline with Performance Counters (-profile)* — *Profiling Autorun Kernels* - Removed the topic *HTML Report: Area Report Messages* and moved its subtopics under Reviewing Area Information on page 36. - In Reviewing Area Information on page 36, included a note about `analyze-area` from the *Intel FPGA SDK for OpenCL Pro Edition Programming Guide* - System Viewer, Block Viewer and Cluster Viewer topics merged into the Graph Viewer report. Relevant topics and images were updated accordingly. - In *Single Work-Item Kernel versus NDRange Kernel*, `accum_swg` kernel code line 6 was updated.
2019.05.08	19.1	- Updated *Kernel Execution Tab* since "Memory Copy (from device)" and "Memory Copy (to device)" are no longer supported. - Added document archives chapter.
2019.04.01	19.1	- In Nested Loops on page 70 topic, updated the code snippet and images in the *Out-of-Order Loop Iterations* section. - In Loops in a Single Work-Item Kernel on page 66 topic: — Updated the *Trade-Off Between Critical Path and Maximum Frequency* section to discuss kernel `lowered_fmax`. — Added *Loop Speculation* section. - Most of the topics under Reviewing Your Kernel's report.html File on page 15chapter were updated to map the content and images to GUI changes in the HTML report. - Removed the topic *Area Analysis by Source* since this view has been removed from the HTML report. - Removed the topic *Area Analysis*

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Added the following new topics: — *Reviewing f_{MAX} II Information* — Analyzing Throughput on page 32 — Reviewing Block Information on page 22 — Reviewing Cluster Information on page 25 • Added a new chapter *Strategies for Improving Performance in Your Host Application* and added the following new topics under it: — Utilizing Hardware Kernel Invocation Queue on page 181 — Double Buffered Host Application Utilizing Kernel Invocation Queue on page 183 • Moved Multi-Threaded Host Application on page 180 topic under *Strategies for Improving Performance in Your Host Application* chapter and made minor improvements in the description. • Updating the code snippets, text and images in Optimizing an OpenCL Design Example Based on Information in the HTML Report on page 40. • Removed step 3 along with flowchart in Reviewing Loop Information on page 33. • Removed *Loop Analysis Report of an OpenCL Design Example* topics and merged its content with Reviewing Loop Information on page 33. • Moved Changing the Memory Access Pattern Example on page 61 and Reducing the Area Consumed by Nested Loops Using loop_coalesce on page 76 to *HTML Report: Kernel Design Concepts* section. • Modified the topic title *Verifying Information on Memory Replication and Stalls* to Using Views on page 19. • Added a new topic Optimizing for Two or More Banks of Global Memory on page 150 to describe how to optimize global memory. • Removed the topic *Simplifying Loop-Carried Dependency*. • Updated Kernels on page 51 topic with more information about blocks and clusters. • Updated Local Memory on page 54 topic completely and added more images to explain the concept. • Rewrote the Features of the Kernel Memory Viewer on page 26 topic completely.
2018.09.24	18.1	• In Intel FPGA SDK for OpenCL Pro Edition, the Intel FPGA SDK for OpenCL Offline Compiler has a new front end. For a summary of changes introduced by this new front end, see *Improved Intel FPGA SDK for OpenCL Compiler Front End* in the Intel FPGA SDK for OpenCL Pro Edition Release Notes. • Moved Static Memory Coalescing on page 160 from Strategies for Improving NDRange Kernel Data Processing Efficiency on page 137 to Strategies for Improving Memory Access Efficiency on page 145. • Added information about the `ivdep` pragma `safelen(N)` clause to Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays on page 132. • Removed image that showed comparison between parallel threads and loop pipelining, along with explanation to Multi-Threaded Host Application on page 180. This image and its explanation did not apply to host applications.
2018.05.04	18.0	• Removed Intel FPGA SDK for OpenCL Standard Edition information. • Added a new Strategies for Optimizing Intel Stratix 10 OpenCL Designs on page 166 chapter. • In Preloading Data to Local Memory on page 151, added information on automatic padding of local memory elements. • Removed the topic *Resource-Driven Optimization* because it described an obsolete optimization behavior.

Table 21. Intel FPGA SDK for OpenCL Best Practices Guide Document Revision History

Date	Version	Changes
December 2017	2017.12.08	• Added the following new topics: — Autorun Captures Tab — Autorun Profiler Data
November 2017	2017.11.06	• Moved all topics into individual chapters. • Changed some of the topic titles to task-based titles. • Changed all occurrences of Fmax to f_{max}. • Rebranded Dynamic Profiler to Intel FPGA Dynamic Profiler for OpenCL. • Added a new short description to Stall, Occupancy, Bandwidth on page 115. • Added a new image to show comparison between parallel threads and loop pipelining, along with explanation to Multi-Threaded Host Application on page 180. • Added an FPGA architecture along with some explanation in FPGA Overview on page 5. • Added OpenCL Design Components image to OpenCL Design Components. • Added an important note to Aligning a Struct with or without Padding on page 103 about 4-byte alignment and remove information related to a struct that is aligned and not padded. • Added two bullet points to the last Attention section in Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor on page 157. • Added Minimizing the Memory Dependencies for Loop Pipelining on page 159. • Added area report hierarchy details to Reviewing Area Information on page 36. • Added Best Practices for Channels and Pipes on page 97. • Updated Allocating Aligned Memory on page 102. • Added Reducing the Area Consumed by Nested Loops Using loop_coalesce on page 76. • Added Changing the Memory Access Pattern Example on page 61. • Updated the image Optimization Work Flow of a Single Work-Item Kernel. • In the following topics, implemented single dash and `-option=<value>` conventions for `aoc` command. — Optimization Work Flow of a Single Work-Item Kernel — Optimizing Floating-Point Operations on page 99 — Manual Partitioning of Global Memory on page 149 — Constant Cache Memory on page 151 — Compilation Considerations on page 162 — High Stall and High Occupancy Percentages on page 118 • In *Source Code Tab* and *Tool Top Options*, updated the images to reflect Intel. • In *High Stall Percentage*, added a screenshot for high stall percentage identification along with relevant explanation. • In Local Memory on page 54, added a sentence about the overall state of the local memory as observed in the HTML report. • In Load-Store Units on page 85, updated the description of semi-streaming LSU to describe how data travels throughout the block. • New example codes and relevant explanation added to Nested Loops on page 70. • Updated the code fragment in Intel FPGA SDK for OpenCL Pipeline Approach on page 8 section by removing the `index` keyword updated Figure 4. • In Single Work-Item Kernel versus NDRange Kernel on page 9, — Removed the criteria for creating single work item kernels for your design. — Added new example codes and relevant explanation — Removed the subtopic on *Single Work-Item Execution* and merged its content with this topic.

continued...

Date	Version	Changes
May 2017	2017.05.08	• Rebranded some functions in code examples as follows: — Rebranded `read_channel_altera` to `read_channel_intel`. — Rebranded `write_channel_altera` to `write_channel_intel`. — Rebranded `read_channel_nb_altera` to `read_channel_nb_intel`. — Rebranded `write_channel_nb_altera` to `write_channel_nb_intel`. • Added Load-Store Units on page 85. • Added Reviewing the Summary Report on page 16. • Added Features of the Kernel Memory Viewer on page 26. • Revised the *Local Memory Banks* section of Local Memory on page 54 to include information about the `bank_bits` attribute. • Revised Optimization Work Flow of a Single Work-Item Kernel in Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 123 to reflect changes to the profiling commands.
December 2016	2016.12.02	Minor editorial modification.
October 2016	2016.10.31	• Rebranded the Altera SDK for OpenCL to Intel FPGA SDK for OpenCL. • Rebranded the Altera Offline Compiler to Intel FPGA SDK for OpenCL Offline Compiler. • In *Align a Struct with or without Padding*, modified code snippets to correct the placement of attributes with respect to the struct declaration. • Added the topic *Review Your Kernel's report.html File*, with subtopics describing the HTML GUI, the various reports the GUI provides, and a walkthrough on how to leverage the information in the HTML report to optimize an OpenCL design example. • Changed *Review Your Kernel's Area Report to Identify Inefficiencies in Resource Usage* to *HTML Report: Area Report Messages*, and removed the following subsections: — *Area Report Messages for Global Memory and Global Memory Interconnect* — *Area Report Messages for Local Memory* — *Area Report Messages for Channels* • Added the topic *HTML Report: Kernel Design Concepts*, which includes subtopics on kernels, global memory interconnect, local memory, nested loops, loops in single work-item kernels, and channels. • In *Interpreting the Profiling Information*, reorganized the content and added the following: — Additional explanations on stall, occupancy, bandwidth, activity, and cache hit. — Suggestions on addressing unsatisfactory Profiler metrics. • In *Addressing Single Work-Item Kernel Dependencies Based On Optimization Report Feedback*, modified the figure *Optimization Work Flow of a Single Work-Item Kernel* to replace area report with HTML report. • Removed the *Optimization Report* section along with the associated subsections because the information is now part of the HTML report. • Changed *Review Kernel Properties and Loop Unroll Status in the Optimization Report* to *Review Kernel Properties and Loop Unroll Status in the HTML Report* because the optimization report is now part of the `report.html` file.
May 2016	2016.05.02	• Added the topic *Removing Loop-Carried Dependencies Caused by Accesses to Memory Arrays* to introduce the `ivdep` pragma. • Under *Strategies for Improving Memory Access Efficiency*, added the following topics to explain how to use the `numbanks` and `bankwidth` kernel attributes to configure the geometry of local memory system: — *Improve Kernel Performance by Banking the Local Memory* — *Optimize the Geometric Configuration of Local Memory Banks Based on Array Index* • Under *Strategies for Improving Memory Access Efficiency*, added the topic *Optimize Accesses to Local Memory by Controlling the Memory Replication Factor* to explain the usage of the `singlepump` and `doublepump` kernel attributes.

continued...

Date	Version	Changes
		• Added information on the area report messages. Refer to the *Review Your Kernel's Area Report to Identify Inefficiencies in Resource Usage* section for more information. • Removed the *Kernel-Specific Area Report* section because it is replaced by the enhanced area report. Refer to the *Review Your Kernel's Area Report to Identify Inefficiencies in Resource Usage* section for more information. • Updated the subsections under *Optimization Report* to include the enhanced optimization report messages. — Added the *Optimization Report Message for Speed-Limiting Constructs* • Updated the subsections under *Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback* to include the enhanced optimization report messages. • Updated the figure *Optimization Work Flow for a Single Work-Item Kernel* to include steps on accessing the enhanced area report to review resource usage. • Under *Strategies for Improving NDRange Kernel Data Processing Efficiency*, added the *Review Kernel Properties and Loop Unroll Status in the Optimization Report* section.
November 2015	2015.11.02	• Added the topic *Multi-Threaded Host Application*. • Added Caution note regarding memory barrier in *Specify a Maximum Work-Group Size or a Required Work-Group Size*.
May 2015	15.0.0	• In *Memory Access Considerations*, added Caution note regarding performance degradation that might occur when declaring __constant pointer arguments in kernels targeting Cyclone V devices. • In *Good Design Practices for Single Work-Item Kernel*, removed the *Initialize Data Prior to Usage in a Loop* section and added a *Declare Variables in the Deepest Scope Possible* section. • Added *Removing Loop-Carried Dependency by Inferring Shift Registers*. The topic discusses how, in single work-item kernels, inferring double precision floating-point array as a shift register can remove loop-carried dependencies. • Added *Kernel-Specific Area Reports* to show examples of kernel-specific `.area` files that the Altera Offline Compiler generates during compilation. • Renamed *Transfer Data Via offline compiler Channels* to *Transfer Data Via offline compiler Channels or OpenCL Pipes* and added the following: — More information on how channels can help improve kernel performance. — Information on OpenCL pipes. • Renamed *Data Type Considerations* to *Data Type Selection Considerations*.
December 2014	14.1.0	• Reorganized the information flow in the *Optimization Report Messages* section to update report messages and the layout of the optimization report. • Included new optimization report messages detailing the reasons for unsuccessful and suboptimal pipelined executions. • Added the *Optimization Report Messages for Simplified Analysis of a Complex Design* subsection under *Optimization Report Messages* to describe new report message for simplified kernel analysis. • Renamed *Using Feedback from the Optimization Report to Address Single Work-Item Kernels Dependencies* to *Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback*. • Added the *Transferring Loop-Carried Dependency to Local Memory* subsection under *Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback* to describe new strategy for resolving loop-carried dependency. • Updated the Resource-Driven Optimization and Compilation Considerations sections to reflect the deprecation of the `-O3` and `--util` *<N>* Altera® Offline Compiler (offline compiler) command options. • Consolidated and simplified the *Heterogeneous Memory Buffers* and *Host Application Modifications for Heterogeneous Memory Accesses* sections. • Added the section *Align a Struct and Remove Padding between Struct Fields*. • Removed the section *Ensure 4-Byte Alignment to All Data Structures*. • Modified the figure *Single Work-Item Optimization Work Flow* to include emulation and profiling.

continued...

Date	Version	Changes
June 2014	14.0.0	• Renamed document as the *Intel FPGA SDK for OpenCL Best Practices Guide*. • Reorganized information flow. • Renamed *Good Design Practices* to *Good OpenCL Kernel Design Practices*. • Added channels information in *Transfer data via offline compiler Channels*. • Added profiler information in *Profile Your Kernel to Identify Performance Bottlenecks*. • Added the section *Single Work-Item Kernel Versus NDRange Kernel*. • Updated *Single Work-Item Execution* section. • Removed *Performance Warning Messages* section. • Renamed *Single Work-Item Kernel Programming Considerations* to *Good Design Practices for Single Work-Item Kernel*. • Added the section *Strategies for Improving Single Work-Item Kernel Performance*. • Renamed *Optimization of Data Processing Efficiency* to *Strategies for Improving NDRange Kernel Data Processing Efficiency*. • Removed *Resource Sharing* section. • Renamed *Floating-Point Operations* to *Optimize Floating-Point Operations*. • Renamed *Optimization of Memory Access Efficiency* to *Strategies for Improving Memory Access Efficiency*. • Updated *Manual Partitioning of Global Memory* section. • Added the section *Strategies for Optimizing FPGA Area Usage*.
December 2013	13.1.1	• Updated the section *Specify a Maximum Work-Group Size or a Required Work-Group Size*. • Added the section *Heterogeneous Memory Buffers*. • Updated the section *Single Work-Item Execution*. • Added the section *Performance Warning Messages*. • Updated the section *Single Work-Item Kernel Programming Considerations*.
November 2013	13.1.0	• Reorganized information flow. • Updated the section *Intel FPGA SDK for OpenCL Compilation Flow*. • Updated the section *Pipelines*; inserted the figure *Example Multistage Pipeline Diagram*. • Removed the following figures: — *Instruction Flow through a Five-Stage Pipeline Processor*. — *Vector Addition Kernel Compiled to an FPGA*. — *Effect of Kernel Vectorization on Array Summation*. — *Data Flow Implementation of a Four-Element Accumulation Kernel*. — *Data Flow Implementation of a Four-Element Accumulation Kernel with Loop Unrolled*. — *Complete Loop Unrolling*. — *Unrolling Two Loop Iterations*. — *Memory Master Interconnect*. — *Local Memory Read and Write Ports*. — *Local Memory Configuration*. • Updated the section *Good Design Practices*. • Removed the following sections: — *Predicated Execution*. — *Throughput Analysis*. — *Case Studies*. • Updated and renamed *Optimizing Data Processing Efficiency* to *Optimization of Data Processing Efficiency*. • Renamed *Replicating Compute Units versus Kernel SIMD Vectorization* to *Compute Unit Replication versus Kernel SIMD Vectorization*. • Renamed *Using num_compute_units and num_simd_work_items Together* to *Combination of Compute Unit Replication and Kernel SIMD Vectorization*. • Updated and renamed *Memory Streaming* to *Contiguous Memory Accesses*.

continued...

Date	Version	Changes
		• Updated and renamed *Optimizing Memory Access* to *General Guidelines on Optimizing Memory Accesses*. • Updated and renamed *Optimizing Memory Efficiency* to *Optimization of Memory Access Efficiency*. • Inserted the subsection *Single Work-Item Execution* under *Optimization of Memory Access Efficiency*.
June 2013	13.0 SP1.0	• Updated support status of OpenCL kernel source code containing complex exit paths. • Updated the figure *Effect of Kernel Vectorization on Array Summation* to correct the data flow between Store and Global Memory. • Updated content for the `unroll` pragma directive in the section *Loop Unrolling*. • Updated content of the *Local Memory* section. • Updated the figure *Local Memories Transferring Data Blocks within Matrices A and B* to correct the data transfer pattern in Matrix B. • Removed the figure *Loop Unrolling with Vectorization*. • Removed the section *Optimizing Local Memory Bandwidth*.
May 2013	13.0.1	• Updated terminology. For example, pipeline is replaced with compute unit; vector lane is replaced with SIMD vector lane. • Added the following sections under *Good Design Practices*: — *Preprocessor Macros*. — *Floating-Point versus Fixed-Point Representations*. — *Recommended Optimization Methodology*. — *Sequence of Optimization Techniques*. • Updated code fragments. • Updated the figure *Data Flow with Multiple Compute Units*. • Updated the figure *Compute Unit Replication versus Kernel SIMD Vectorization*. • Updated the figure *Optimizing Throughput Using Compute Unit Replication and SIMD Vectorization*. • Updated the figure *Memory Streaming*. • Inserted the figure *Local Memories Transferring Data Blocks within Matrices A and B*. • Reorganized the flow of information. Number of figures, tables, and examples have been updated. • Included information on new kernel attributes: `max_share_resources` and `num_share_resources`.
May 2013	13.0.0	• Updated pipeline discussion. • Updated case study code examples and results tables. • Updated figures.
November 2012	12.1.0	Initial release.