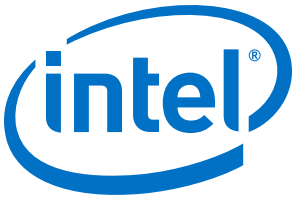


Intel® FPGA SDK for OpenCL™ Standard Edition

Best Practices Guide

Updated for Intel® Quartus® Prime Design Suite: **18.1**

Subscribe
Send Feedback

UG-20152 | 2018.09.24
Latest document on the web: PDF | HTML

Contents

UG-20152 | 2018.09.24

Send Feedback

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

The *Intel® FPGA SDK for OpenCL™ Standard Edition Best Practices Guide* provides guidance on leveraging the functionalities of the Intel FPGA Software Development Kit (SDK) for OpenCL(1) Standard Edition 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 on 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*.

Related Information

- OpenCL Reference Pages
- OpenCL Specification version 1.0

1.1. FPGA Overview

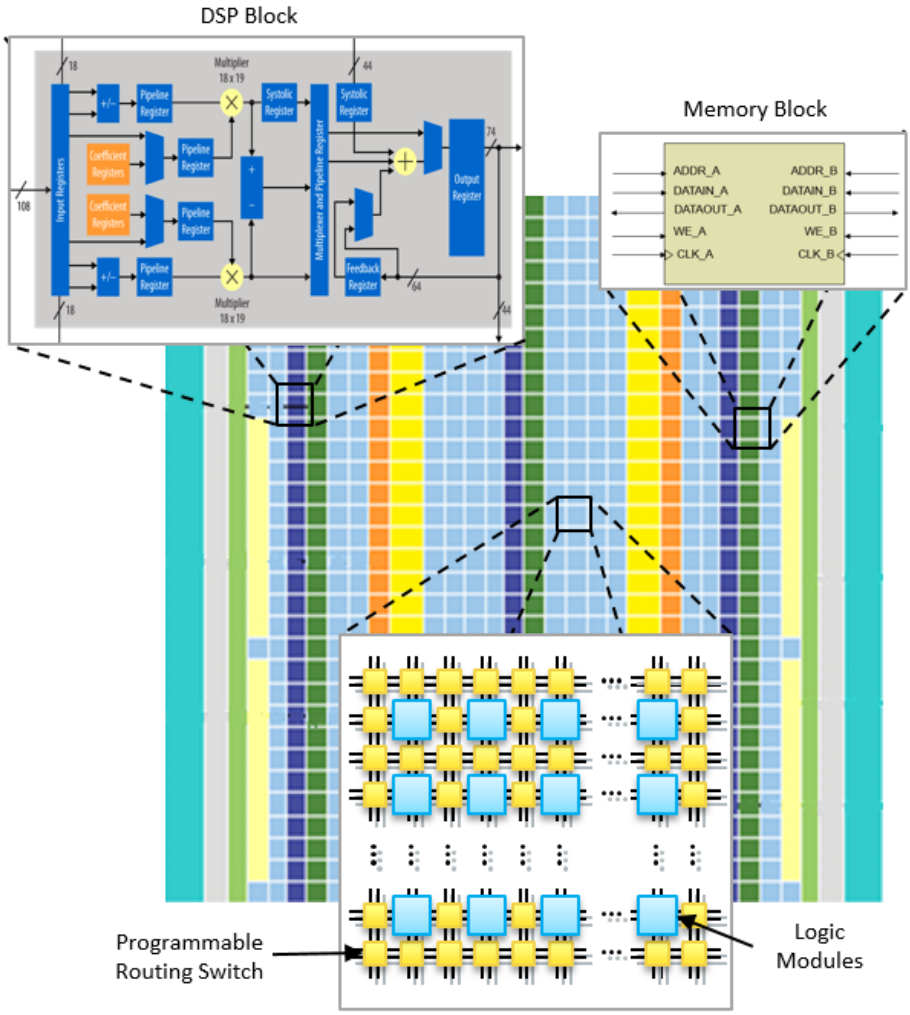
Field-programmable gate arrays (FPGAs) are integrated circuits that you can configure repeatedly to perform an infinite number of functions.

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 leverages 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

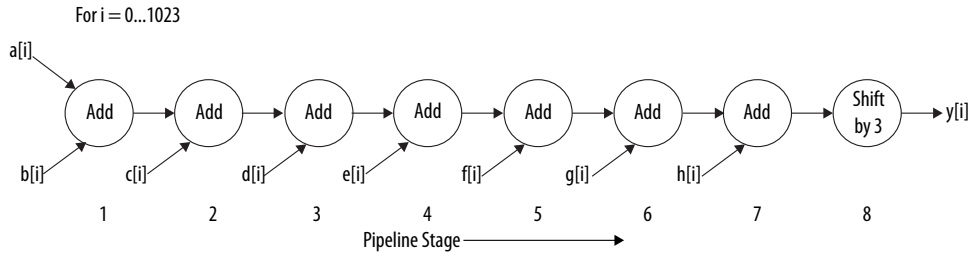
In a pipelined architecture, input data passes through a sequence of stages. Each stage performs an operation that contributes to the final result, such as memory operation or calculation.

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

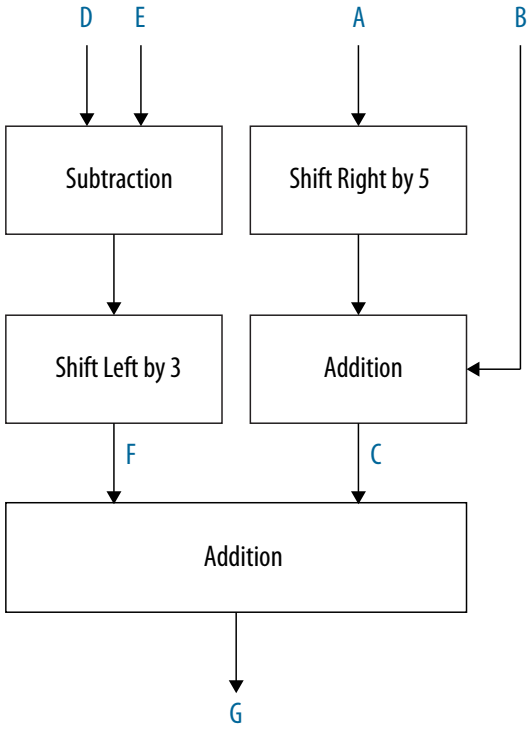
A new pipeline is constructed based on your design. As a result, it can accommodate the highly configurable nature of FPGAs.

Consider the following OpenCL code fragment:

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

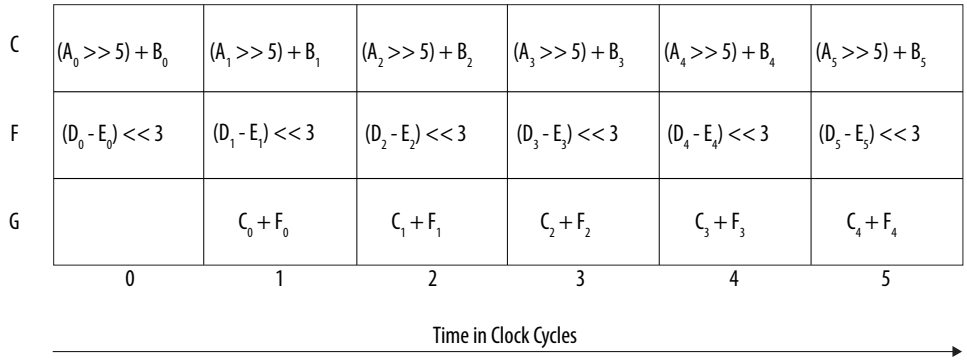
You can configure an FPGA to instantiate a complex pipeline structure that executes the entire code simultaneously. In this case, the SDK implements the code as two independent pipelined entities that feed into a pipelined adder, as shown in the figure below.

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.

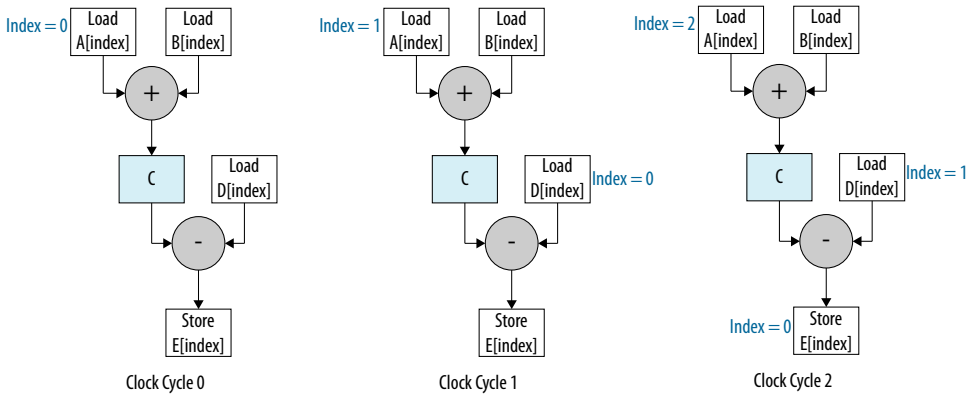
Figure 4. An FPGA Pipeline with Three Operations Per Clock Cycle



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 does not have loop and memory dependencies, you may structure your application as an NDRange kernel because the kernel can execute multiple work-items in parallel efficiently.

The Intel FPGA SDK for OpenCL host can execute a kernel as a single work-item, which is equivalent to launching a kernel with an NDRange size of (1, 1, 1).

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] += 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.

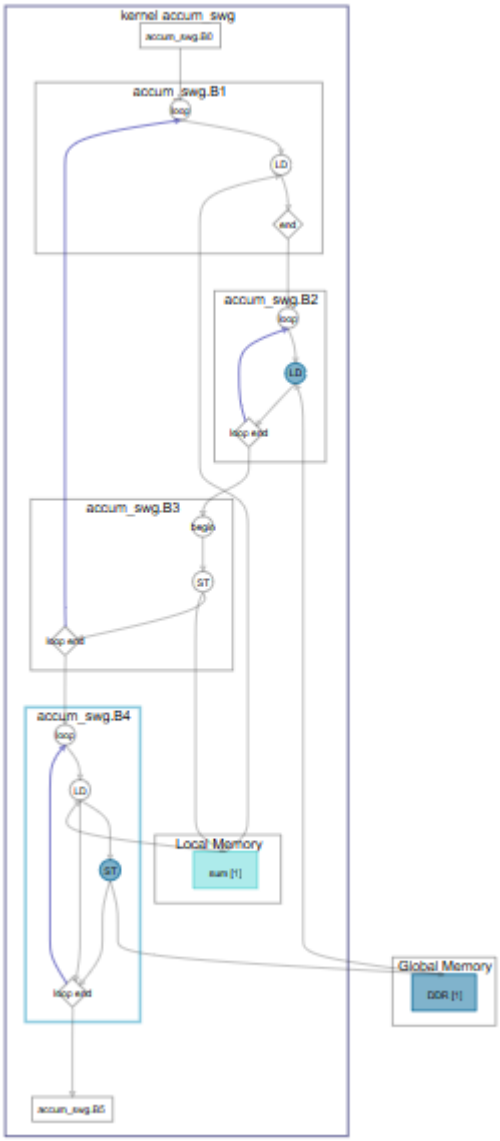
Note: The launch frequency of a new loop iteration is called the initiation interval (II). II refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. An optimally unrolled loop has an II value of 1 because one loop iteration is processed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

Figure 6. Loop Analysis Report

Loops analysis — ☑ Show fully unrolled loops

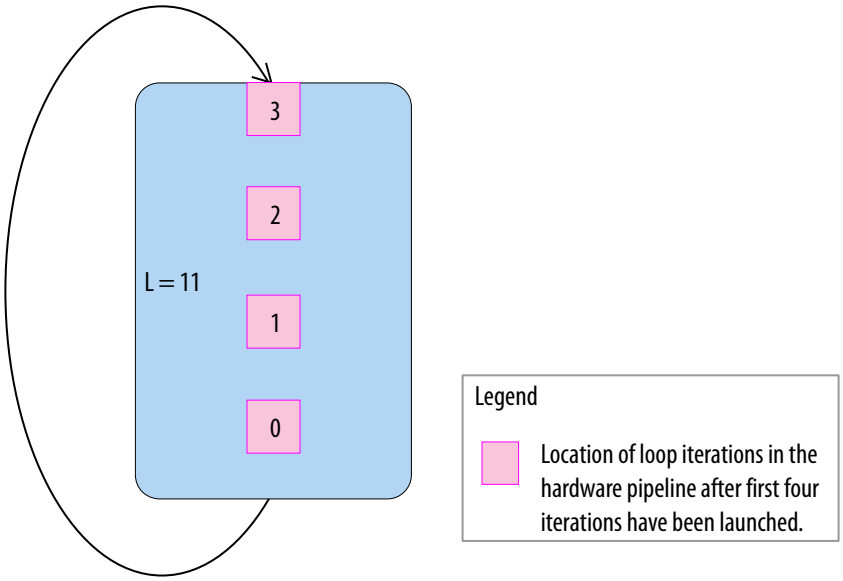
	Pipelined	II	Bottleneck	Details
Kernel: accum_swg (accum_swg.cl:3)				Single work-item execution
accum_swg.B1 (accum_swg.cl:5)	Yes	>=1	n/a	
accum_swg.B2 (accum_swg.cl:6)	Yes	~1	n/a	II is an approximation.
accum_swg.B4 (accum_swg.cl:12)	Yes	~1	n/a	II is an approximation.

Figure 7. System View of Single-Work Item Kernel



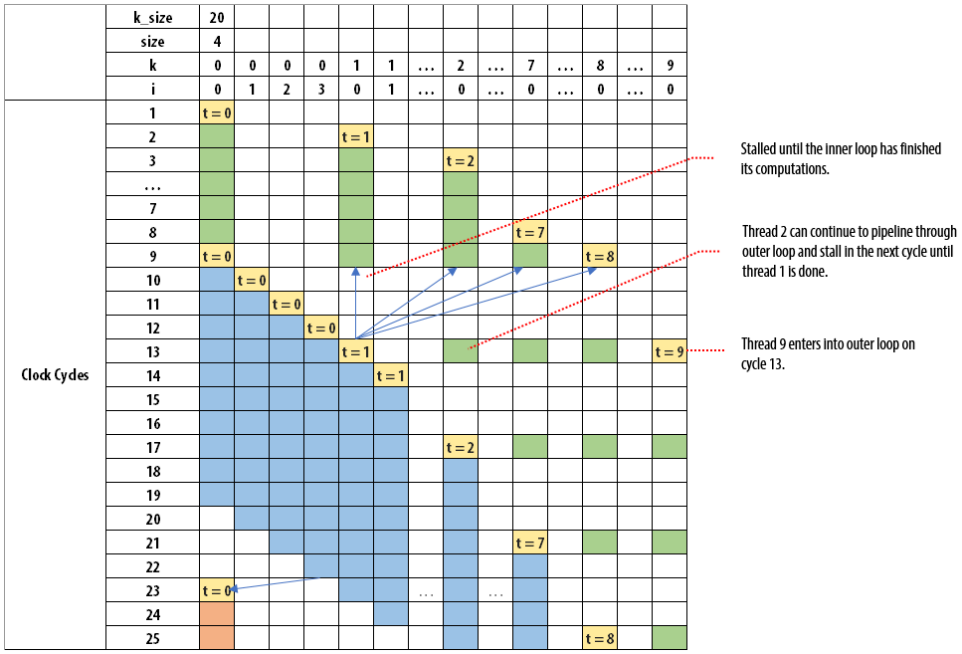
The following figure illustrates how each iteration of i enters into the block:

Figure 8. Inner Loop `accum_swg.B2` Execution



When we 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 will take 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 will enter into the loop at every four clock cycles, which is the value of `size`. Through this example, we 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



Important:

- Using any of the following functions will cause 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 will be interpreted as a single-work-item kernel.

Consider the same accumulate example written in NDRange:

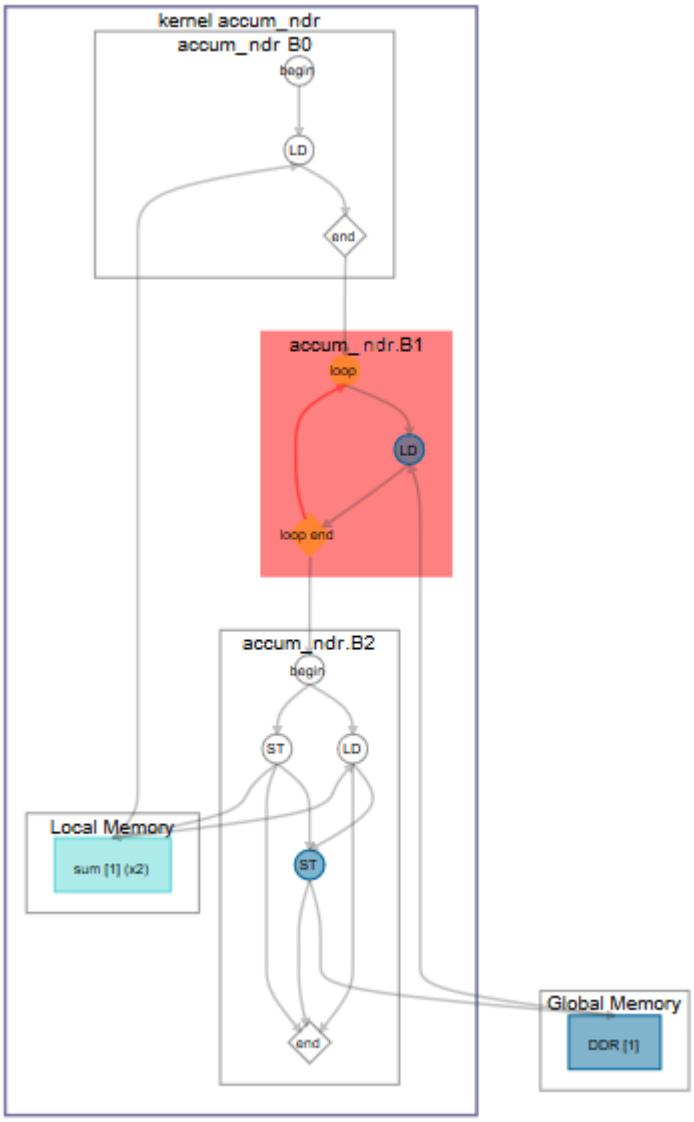
```
kernel void accum_ndr (global int* a,
                       global int* c,
                       int size) {
   int k = get_global_id(0);

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

Figure 10. Loop Analysis Report

Loops analysis ☑ Show fully unrolled loops

	Pipelined	II	Bottleneck	Details
Kernel: accum_ndr (accum_nd.cl:3)				ND-Range
accum_swg.B1 (accum_nd.cl:7)	No	n/a	n/a	Thread capacity = 147

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.

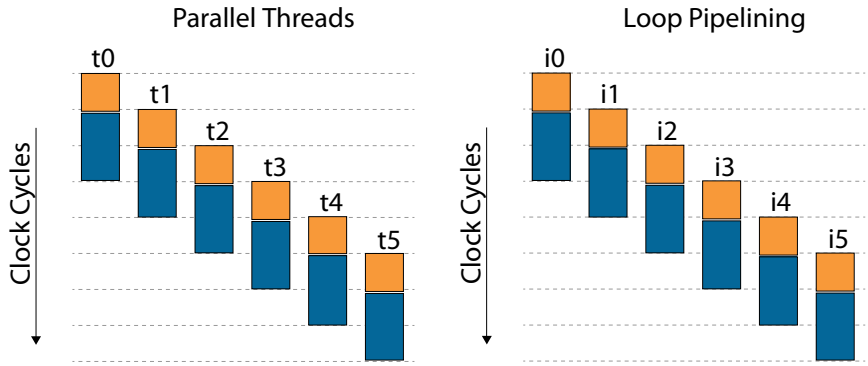
Related Information

- Multiple Work-Item Ordering for Channels
- Loops in a Single Work-Item Kernel on page 61
- Nested Loops on page 54

1.4. Multi-Threaded Host Application

When there are parallel, independent data paths and the host processes the data between kernel executions, consider using a multi-threaded host application.

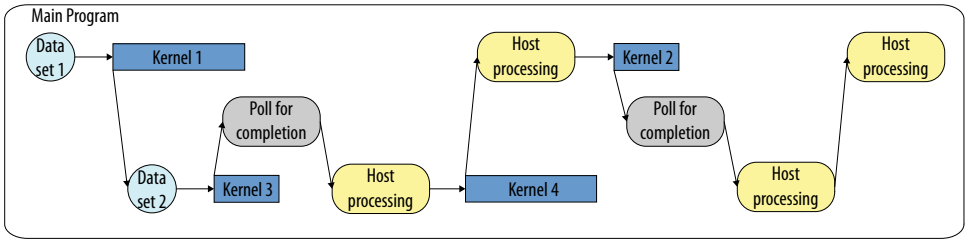
Figure 12. Comparison of the Launch Frequency of a Multi-Threaded Execution with a Pipelined Loop



Parallel threads are launched one thread every clock cycle in a pipelined manner where as loop pipelining enables pipeline parallelism and communication of state information between loop iterations. Loop dependencies may not be resolved in one clock cycle.

The figure below illustrates how a single-threaded host application processes parallel, independent data paths between kernel executions:

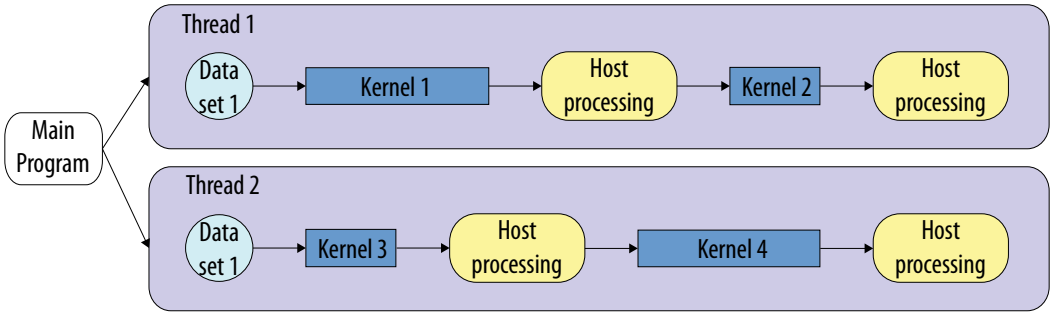
Figure 13. Kernel Execution by a Single-Threaded Host Application



With a single-threaded host application, you need to build an external synchronization mechanism around the OpenCL host function calls. Using a multi-threaded host application in a thread-safe runtime environment allows you to simplify the host code. In addition, processing multiple sets of data in the host simultaneously can speed up kernel execution.

The figure below illustrates how a multi-threaded host application processes parallel, independent data paths between kernel executions:

Figure 14. Kernel Execution by a Multi-Threaded Host Application in a Thread-Safe Runtime Environment



Related Information

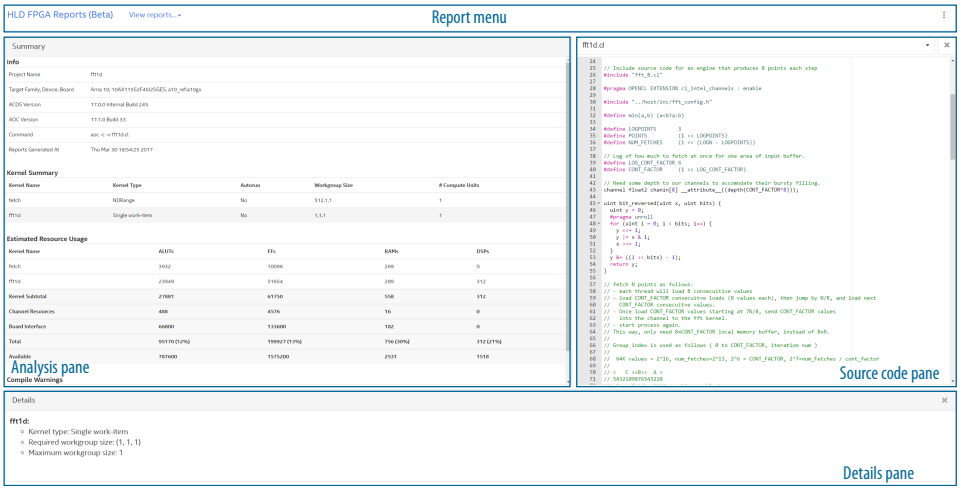
Multiple Host Threads

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.

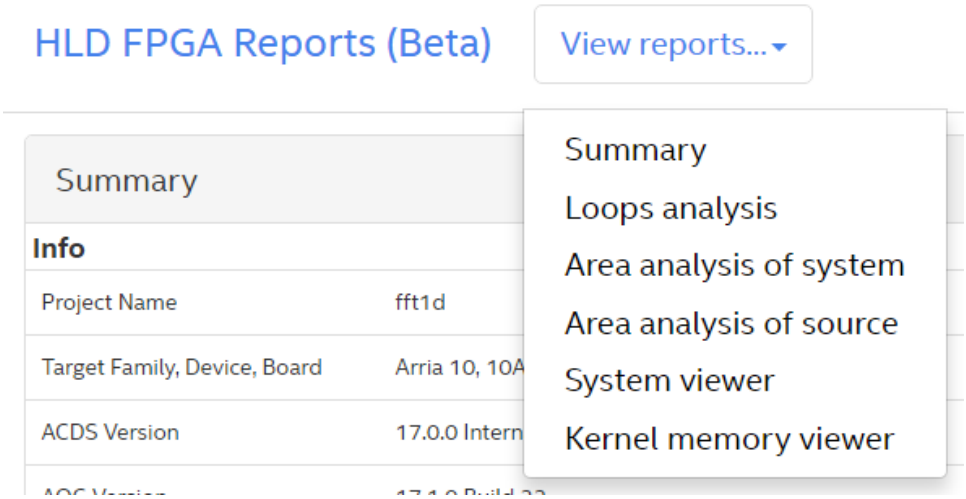
2.1. High Level Design Report Layout

The High Level Design Report (report.html) is divided into four main sections: report menu, analysis pane, source code pane, and details pane.



Report Menu

From the **View reports** pull-down menu, you can select a report to see an analysis of different parts of your kernel design.



Analysis Pane

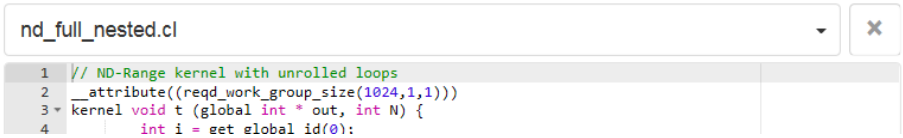
The analysis pane displays detailed information of the report that you selected from the **View reports** pull-down menu.

Source Code Pane

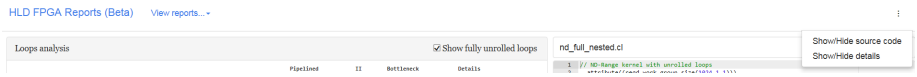
The source code pane displays the code for all the source files in your kernel.

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.



- Click the vertical ellipsis icon on the right-hand side of the report menu and then select **Show/Hide source code**.

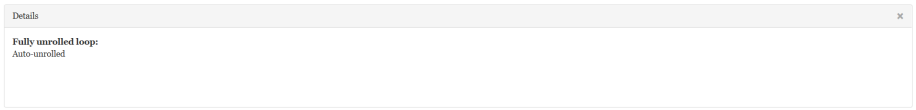


If you previously collapsed the source code pane and want to expand it, click the vertical ellipsis icon on the right-hand side of the report menu and then select **Show/Hide source code**.

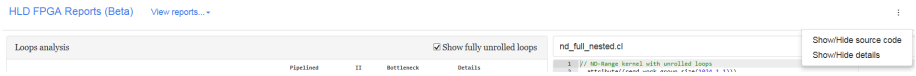
Details Pane

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 right-hand side of the details pane.



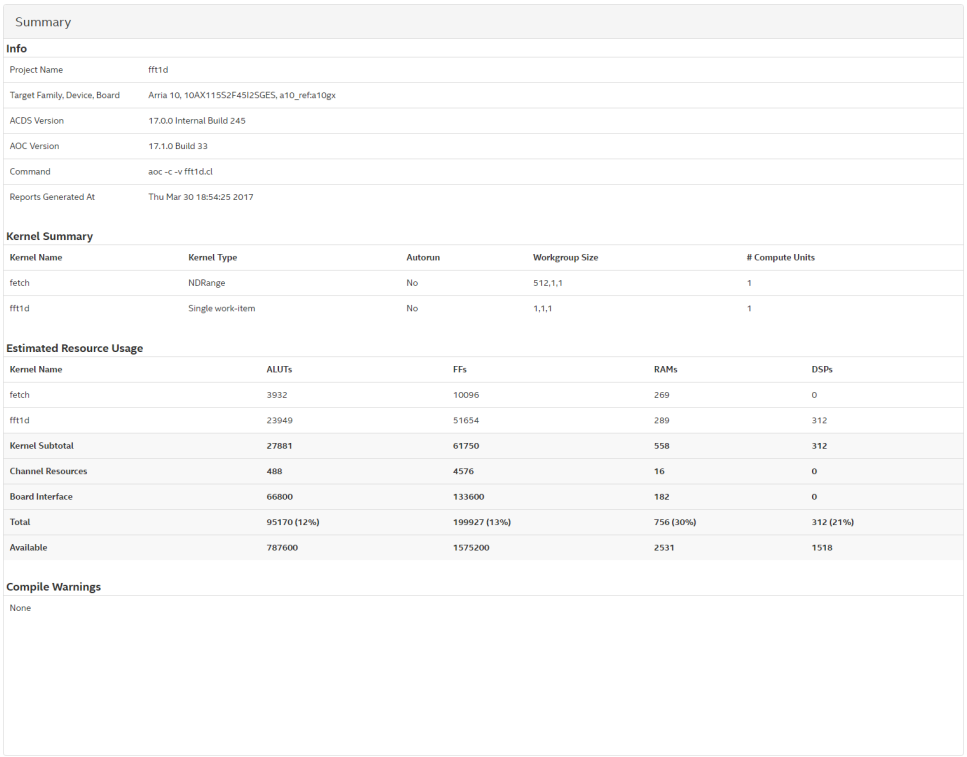
- Click the vertical ellipsis icon on the right-hand side of the report menu and then select **Show/Hide details**.



2.2. Reviewing the Report Summary

The report summary 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 report summary is divided into four sections: Info, Kernel Summary, Estimated Resource Usage, and Compile Warnings.



Summary

Info

Project Name	fft1d
Target Family, Device, Board	Arria 10, 10AX115S2F45I2SGES, a10_ref:a10gx
ACDS Version	17.0.0 Internal Build 245
AOC Version	17.1.0 Build 33
Command	aoc -c -v fft1d.cl
Reports Generated At	Thu Mar 30 18:54:25 2017

Kernel Summary

Kernel Name	Kernel Type	Autorun	Workgroup Size	# Compute Units
fetch	NDRange	No	512,1,1	1
fft1d	Single work-item	No	1,1,1	1

Estimated Resource Usage

Kernel Name	ALUTs	FFs	RAMs	DSPs
fetch	3932	10096	269	0
fft1d	23949	51654	289	312
Kernel Subtotal	**27881**	**61750**	**558**	**312**
Channel Resources	**488**	**4576**	**16**	**0**
Board Interface	**66800**	**133600**	**182**	**0**
Total	**95170 (12%)**	**199927 (13%)**	**756 (30%)**	**312 (21%)**
Available	**787600**	**1575200**	**2531**	**1518**

Compile Warnings

None

Info

The 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 Standard Edition software version
- Intel FPGA SDK for OpenCL Offline Compiler version
- The command that was used to compile the design
- The date and time at which the reports were generated

Kernel Summary

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

- 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:

- The vectorization of the kernel
- The maximum global work dimension
- The maximum workgroup size

Estimated Resource Usage

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.

Compile Warnings

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

2.3. Reviewing Loop Information

The High Level Design Report (`<your_kernel_filename>/reports/report.html`) file contains information about all the loops in your design and their unroll statuses. This loop analysis report helps you examine whether the Intel FPGA SDK for OpenCL Offline Compiler is able to maximize the throughput of your kernel.

You can use the loop analysis report to help determine where to deploy one or more of the following pragmas on your loops:

- `#pragma unroll`

  For details about `#pragma unroll`, see "Unrolling a Loop" in *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*

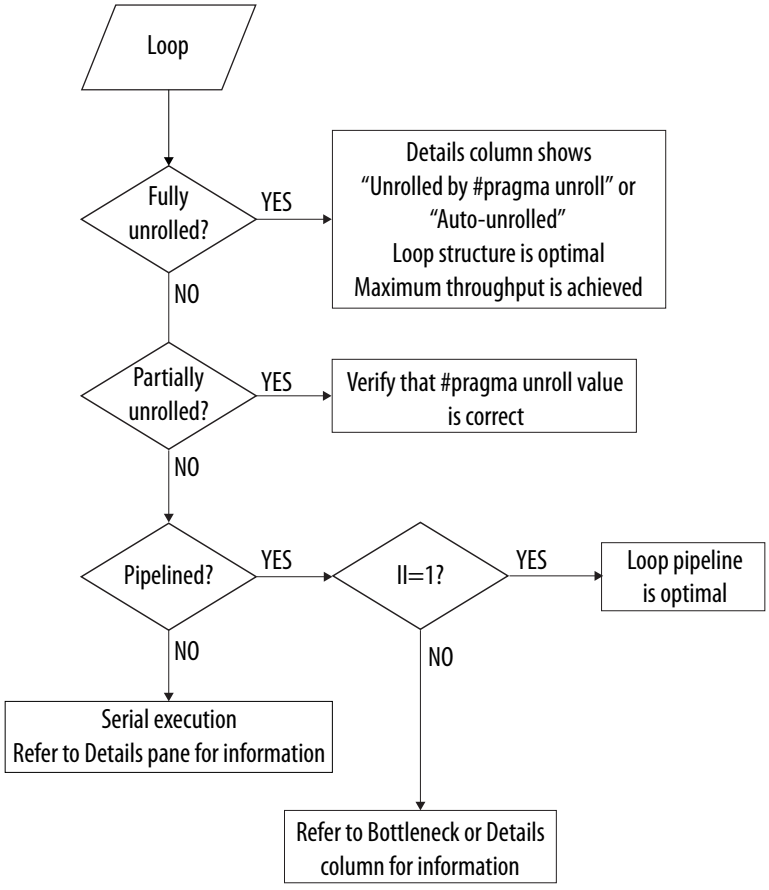
- `#pragma loop_coalesce`

  For details about `#pragma loop_coalesce`, see "Coalescing Nested Loops" in *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

- `#pragma ii`

  For details about `#pragma ii`, see "Specifying a loop initiation interval (II)" in *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

1. Click **View reports ➤ Loop Analysis** .
2. In the analysis pane, select **Show fully unrolled loops** to obtain information about the loops in your design.
3. Consult the flowchart below to identify actions you can take to improve the throughput of your design.



Remember: II refers to the initiation interval of a loop, which is the launch frequency of a new loop iteration. An II value of 1 is ideal; it indicates that the pipeline is functioning at maximum efficiency because the pipeline can process a new loop iteration every clock cycle.

2.3.1. Loop Analysis Report of an OpenCL Design Example

Loop analysis report of an OpenCL kernel example that includes four loops.

```
1  // ND-Range kernel with unrolled loops
2  __attribute((reqd_work_group_size(1024,1,1)))
3  kernel void t (global int * out, int N) {
4    int i = get_global_id(0);
5    int j = 1;
6    for (int k = 0; k < 4; k++) {
7      #pragma unroll
8      for (int n = 0; n < 4; n++) {
9        j += out[k+n];
10     }
11   }
12   out[i] = j;
13
14   int m = 0;
15   #pragma unroll 1
16   for (int k = 0; k < N; k++) {
17     m += out[k/3];
18   }
19   #pragma unroll
20   for (int k = 0; k < 6; k++) {
21     m += out[k];
22   }
23   #pragma unroll 2
24   for (int k = 0; k < 6; k++) {
25     m += out[k];
26   }
27   out[2] = m;
28 }
```

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

Figure 15. Loop Analysis Report of an OpenCL Design Example with Four Loops

Loops analysis — ☑ Show fully unrolled loops

	Pipelined	II	Bottleneck	Details
Fully unrolled loop (nd_full_nested.cl:6)	n/a	n/a	n/a	Auto-unrolled
Fully unrolled loop (nd_full_nested.cl:8)	n/a	n/a	n/a	Unrolled by #pragma unroll
Block2 (nd_full_nested.cl:16)	No	n/a	n/a	
Fully unrolled loop (nd_full_nested.cl:20)	n/a	n/a	n/a	Unrolled by #pragma unroll
2X Partially unrolled Block4 (nd_full_nested.cl:24)	No	n/a	n/a	

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

- Fully unrolls the first loop (line 6) automatically
- Fully unrolls the inner loop (line 8) within the first loop because of the `#pragma unroll` specification
- Does not unroll the second outer loop, Block2 (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, Block4 (line 24), twice because of the `#pragma unroll 2` specification

2.3.2. 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;
}
```

The system viewer report of this example highlights the stallable loads and stores.

Figure 16. System View of the Example

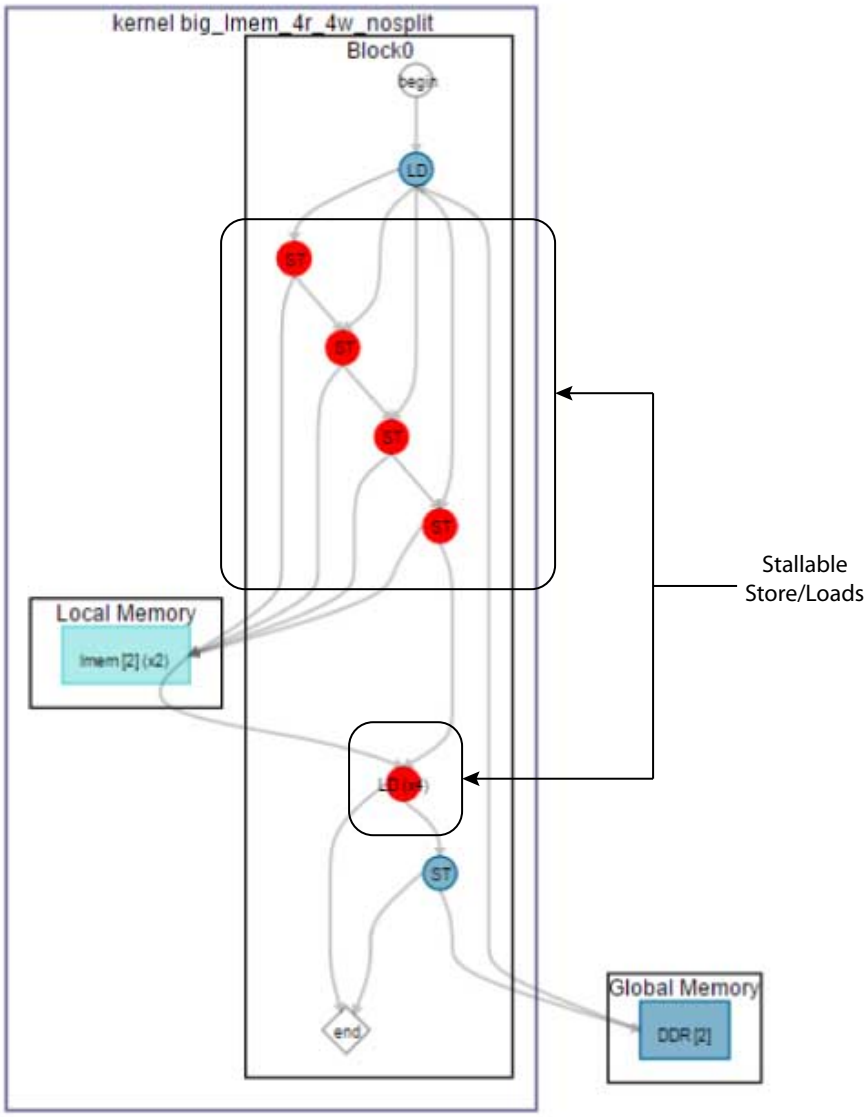


Figure 17. Area Report of the Example

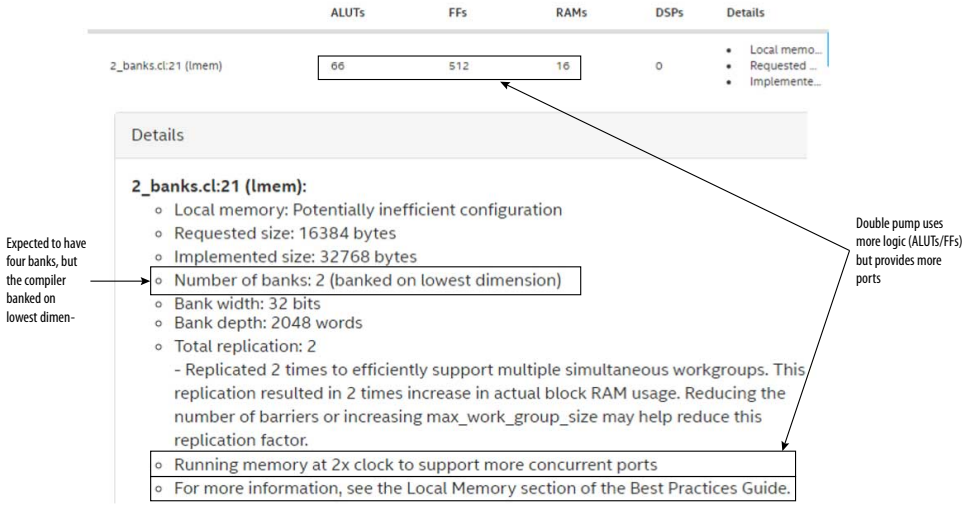
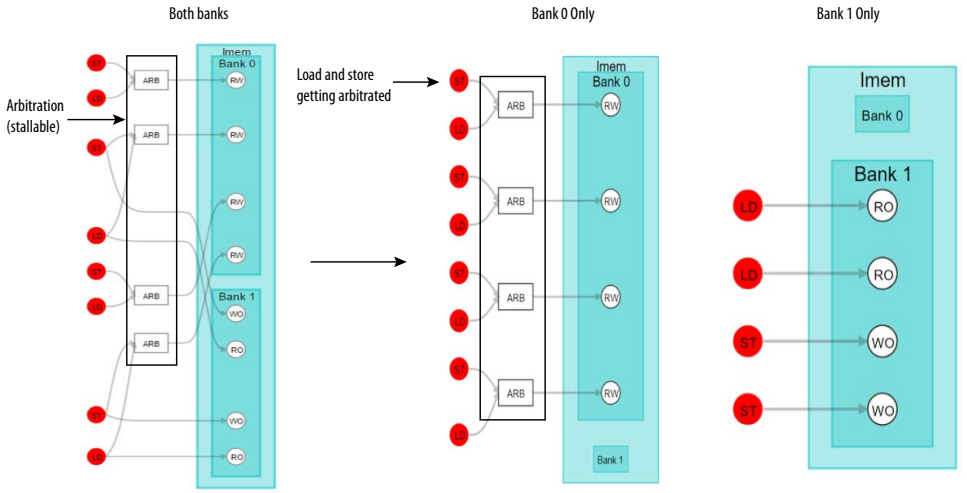


Figure 18. Kernel Memory Viewer of the Example



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 19. Kernel Memory Viewer of the Example After Changing the Banking Indices

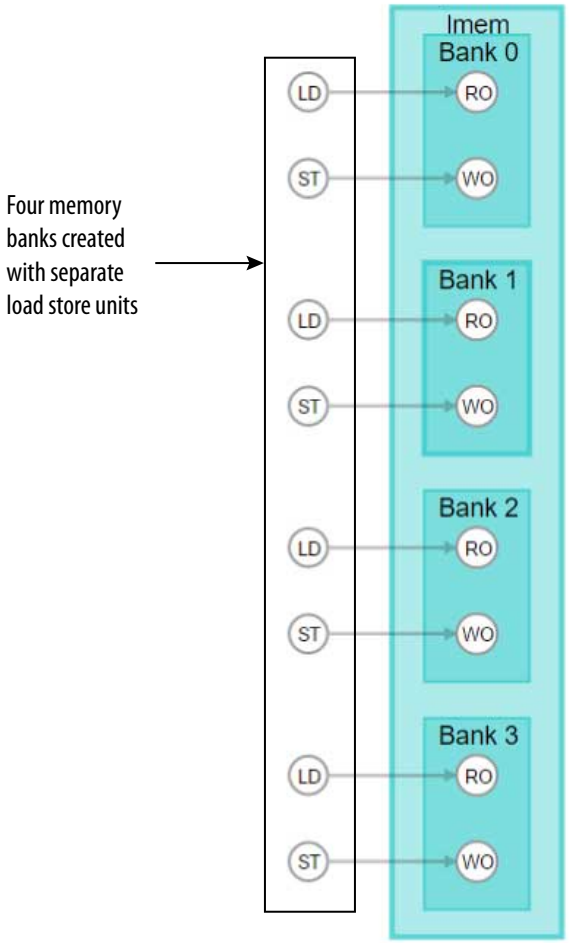
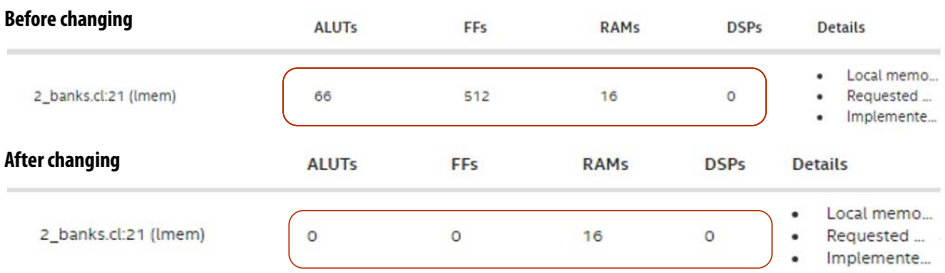


Figure 20. Area Report of the Example After Changing the Banking Indices



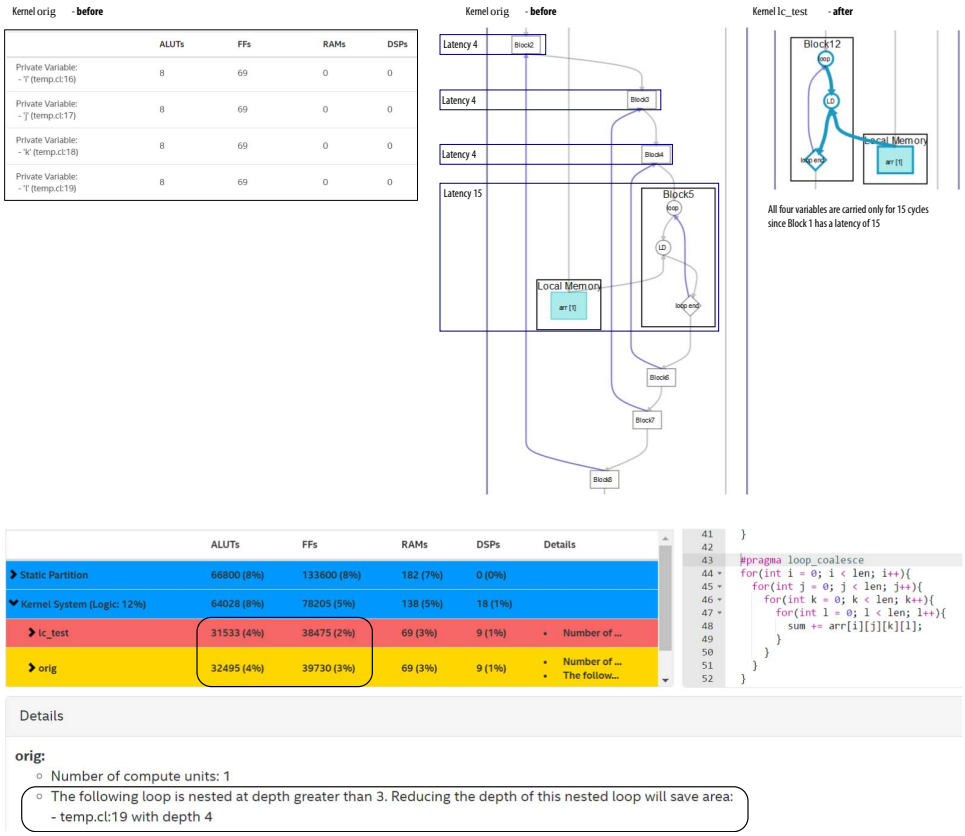
2.3.3. Reducing the Area Consumed by Nested Loops Using `loop_coalesce`

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.

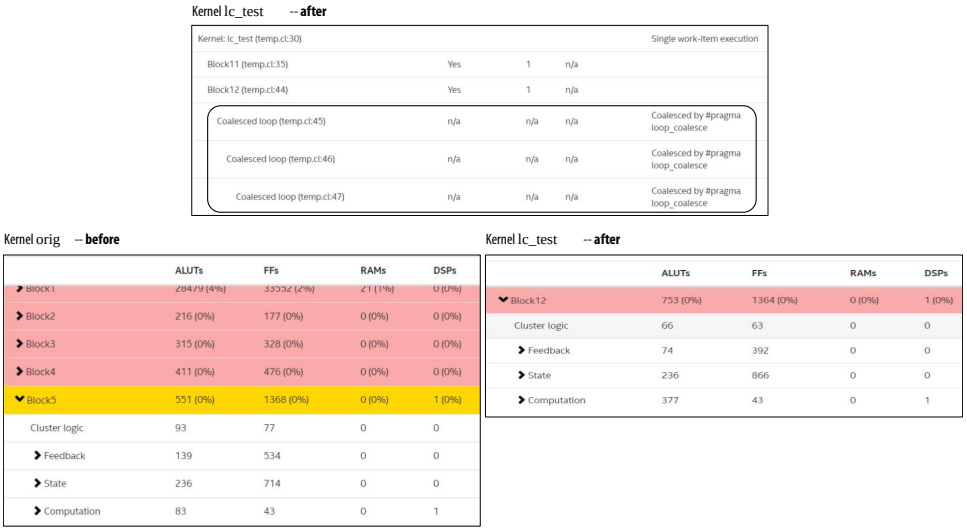
The `orig` kernel has nested loops to a depth of four. The nested loops created extra blocks (Block 2, 3, 4, 6, 7 and 8) that consume area due to the variables being carried, as shown in the following reports:

Figure 21. Area Report and System Viewer Before and After Loop Coalescing



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 22. Area Report of `lc_test` and `orig` Kernels



Related Information

Coalescing Nested Loops

2.4. Reviewing Area Information

The `<your_kernel_filename>/reports/report.html` file contains information about area usage of your OpenCL system. You may view the area usage information either by source (that is, code line) or by system.

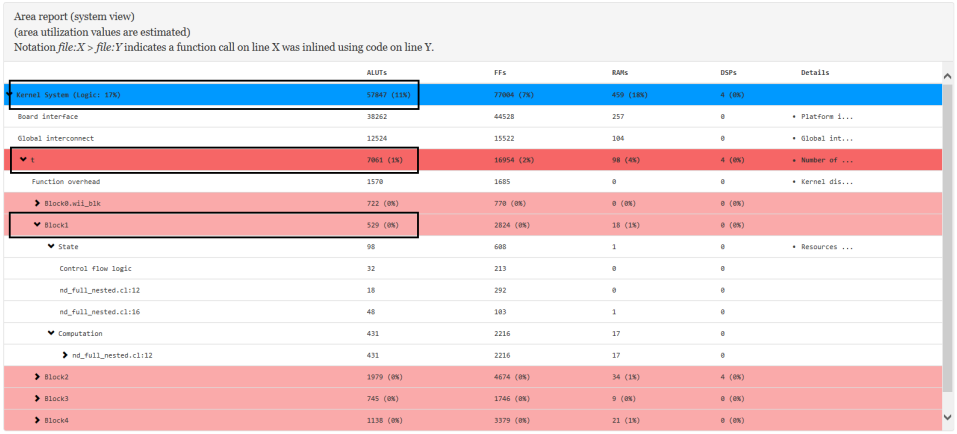
The area report serves the following purposes:

- 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.
- **Basic 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 23. Area Report Hierarchy



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 report menu's **View reports...** pull-down menu, select **Area analysis by source** or **Area analysis of system**.

2.4.1. Area Analysis by Source

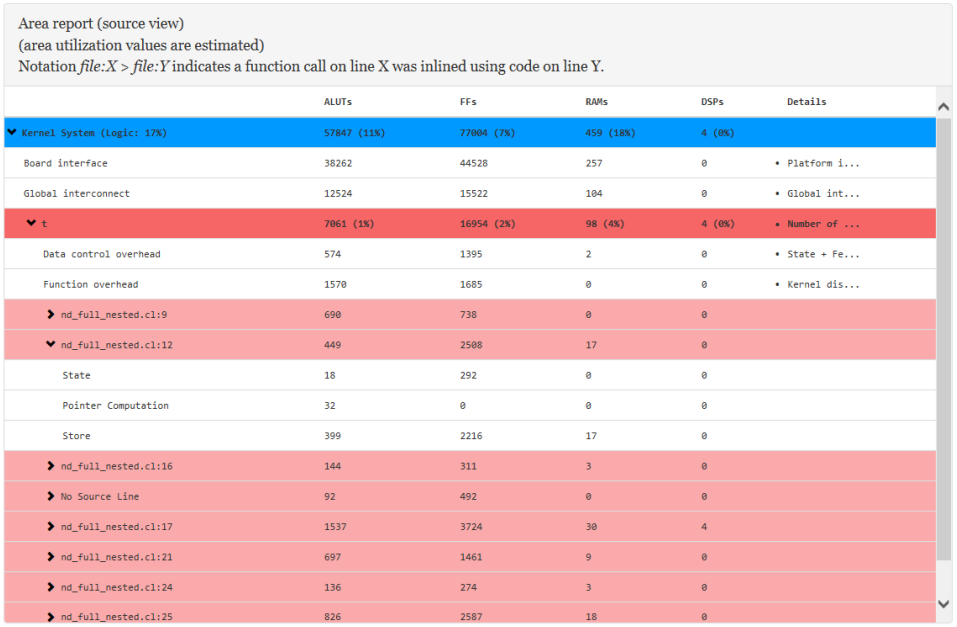
The area analysis by source report shows an approximation how how each line of the source code affects area usage. In this view, the area information is displayed hierarchically.

OpenCL kernel example that includes four loops:

```
 1  // ND-Range kernel with unrolled loops
 2  __attribute ((reqd_work_group_size(1024,1,1)))
 3  kernel void t (global int * out, int N) {
 4    int i = get_global_id(0);
 5    int j = 1;
 6    for (int k = 0; k < 4; k++) {
 7      #pragma unroll
 8      for (int n = 0; n < 4; n++) {
 9        j += out[k+n];
10      }
11    }
12    out[i] = j;
13
14    int m = 0;
15    #pragma unroll 1
16    for (int k = 0; k < N; k++) {
17      m += out[k/3];
18    }
19    #pragma unroll
20    for (int k = 0; k < 6; k++) {
21      m += out[k];
22    }
23    #pragma unroll 2
24    for (int k = 0; k < 6; k++) {
25      m += out[k];
26    }
27    out[2] = m;
28  }
```

The area report below lists the area usage for the kernel system, board interface, and global interconnects. These elements are system-level IP and are dependent on the Custom or Reference Platform that your design targets. The kernel `t` is within the hierarchy of the kernel system and is where the source code begins. The report specifies all the variables declared within the source code under kernel `t` and sorts the remaining area information by line number.

Figure 24. Source View of an Example Area Report



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

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 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 ...
Data control overhead	574	1395	2	0	• State + Fe...
Function overhead	1570	1685	0	0	• Kernel dis...
nd_full_nested.cl:9	690	738	0	0	
nd_full_nested.cl:12	449	2508	17	0	
State	18	292	0	0	
Pointer Computation	32	0	0	0	
Store	399	2216	17	0	
nd_full_nested.cl:16	144	311	3	0	
No Source Line	92	492	0	0	
nd_full_nested.cl:17	1537	3724	30	4	
nd_full_nested.cl:21	697	1461	9	0	
nd_full_nested.cl:24	136	274	3	0	
nd_full_nested.cl:25	826	2587	18	0	

In this example, for the code line for the code line `j += out[k+n]` (line 9), the Intel FPGA SDK for OpenCL Offline Compiler calculates the estimated area usage based on the area required to perform the addition and to load data from global memory. For the code line `out[i] = j` (line 12), the offline compiler calculates the estimated area usage based on the area required to compute the pointer value and then store it back to global memory.

2.4.2. Area Analysis of System

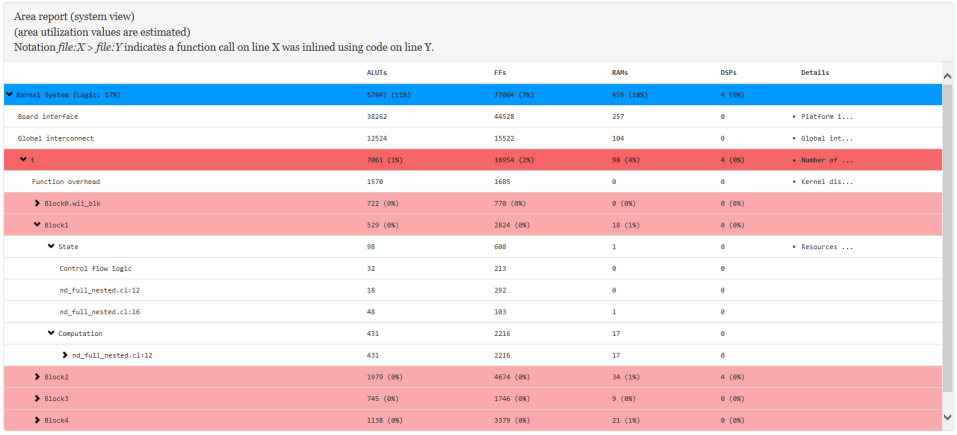
The area analysis of system report shows an area breakdown of your OpenCL system that is the closest approximation to the hardware that is implemented in the FPGA.

OpenCL kernel example that includes four loops:

```
 1  // ND-Range kernel with unrolled loops
 2  __attribute((reqd_work_group_size(1024,1,1)))
 3  kernel void t (global int * out, int N) {
 4    int i = get_global_id(0);
 5    int j = 1;
 6    for (int k = 0; k < 4; k++) {
 7      #pragma unroll
 8      for (int n = 0; n < 4; n++) {
 9        j += out[k+n];
10      }
11    }
12    out[i] = j;
13
```

```
14    int m = 0;
15    #pragma unroll 1
16    for (int k = 0; k < N; k++) {
17      m += out[k/3];
18    }
19    #pragma unroll
20    for (int k = 0; k < 6; k++) {
21      m += out[k];
22    }
23    #pragma unroll 2
24    for (int k = 0; k < 6; k++) {
25      m += out[k];
26    }
27    out[2] = m;
28  }
```

Figure 25. System View of an Example Area Report



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. In this example, area information for the code line `out[i] = j` (line 12) is available under Block1. The estimated area usage for line 12 in the system view is the same as the estimation in the source view.

2.5. Verifying Information on Memory Replication and Stalls

The *<your_kernel_filename>*`/reports/report.html` file provides a stall point graph that includes load and store information between kernels and different memories, channels connected between kernels, and loops.

The system viewer shows an abstracted netlist of your OpenCL system. Reviewing the graphical representation of your OpenCL design in the system viewer allows you to verify memory replication, and identify any load and store instructions that are stallable.

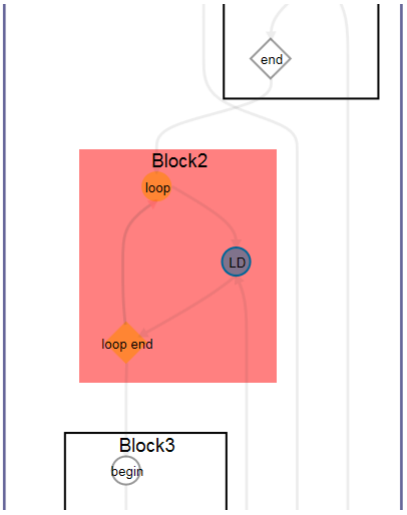
1. In the report menu's **View reports...** pull-down menu, select **System viewer**.

2.5.1. Features of the System Viewer

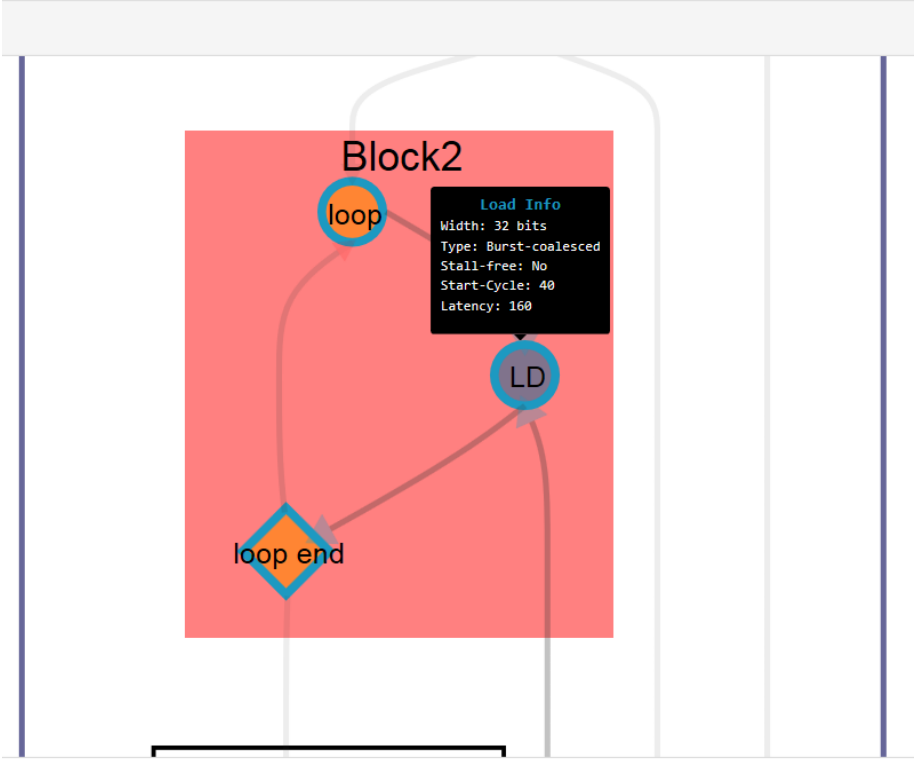
The system viewer is an interactive graphical report of your OpenCL system that allows you to review information such as the sizes and types of loads and stores, stalls, and latencies.

You may interact with the system viewer in the following ways:

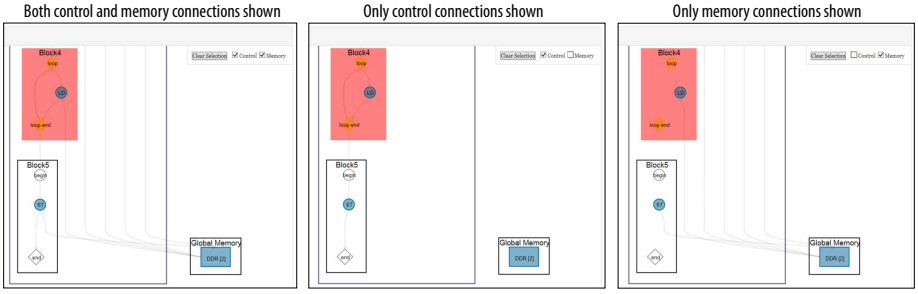
- Use the mouse wheel to zoom in and out within the system viewer.
- Review portions of your design that are associated with red logic blocks. For example, a logic block that has a pipelined loop with a high initiation interval (II) value might be highlighted in red because the high II value might affect design throughput.



- Hover over any node within a block to view information on that node in the tooltip and in the details pane.



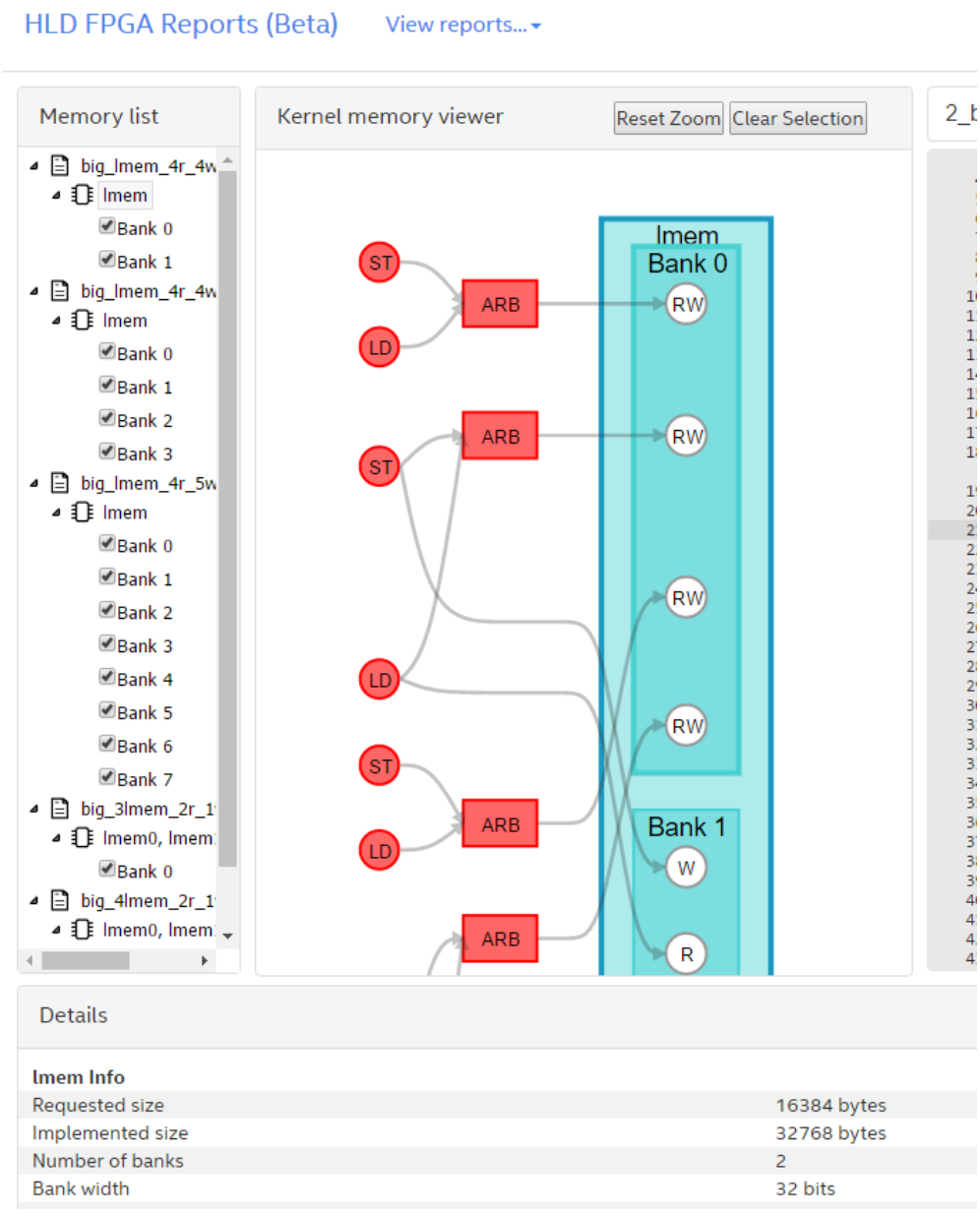
- Select the type of connections you wish to include in the system viewer by unchecking the type of connections you wish to hide. By default, both **Control** and **Memory** are checked in the system viewer. **Control** refers to connections between blocks and loops. **Memory** refers to connections to and from global or local memories. If your design includes connections to and from read or write channels, you also have a **Channels** option in the system viewer.



2.5.2. Features of the Kernel Memory Viewer

Data movement is often 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 across the memory system of your kernel. Use the Kernel Memory Viewer to help you identify data movement bottlenecks in your kernel design.

Also, 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 find where you might have unwanted arbitration in the LSUs.



The Kernel Memory Viewer has the following panes:

Memory List

The Memory List pane shows you a hierarchy of kernels, memories in that kernel, and the corresponding memory banks.

Clicking a memory name in the list displays a graphical representation of the memory in the Kernel memory viewer pane. Also, the line in your code where you declared the memory is highlighted in the Source Code pane.

Clearing a check box for a memory bank collapses that bank in the Kernel Memory Viewer pane, which can help you to focus on specific memory banks when you view a complex memory design. By default, all banks in kernel memory are selected and shown in the Kernel Memory Viewer pane.

Kernel Memory Viewer

The Kernel Memory Viewer pane shows you connections between loads and stores to specific logical ports on the banks in a memory system. The following types of nodes might be shown in the Kernel Memory Viewer pane, depending on the kernel memory system:

- Memory node: The kernel memory.
- Bank node: A bank in the memory. Only banks selected in the Memory List pane are shown. Select banks in the Memory List pane to help you focus on specific on memory banks when you view a complex memory design.
- Port node: 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.
- 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.

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 of the ports that the LSU connects to.

Hover over a port node to highlight the path from the port node to all of the LSUs that store to the port node.

Click on a node to select it and have the node attributes displayed in the Details pane.

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 shows information such as the width and depths of the memory banks, as well as any user-defined 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.

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

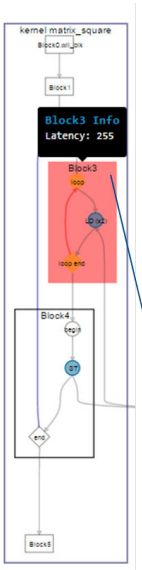
A guide on how to leverage the information in the HTML report to optimize an OpenCL kernel.

OpenCL design example that performs matrix square AxA:

```
// performs matrix square A*A
// A is a square len*len matrix

kernel void matrix_square (global float* restrict A,
                           unsigned len,
                           global float* restrict out) {
  for(unsigned oi = 0; oi < len*len; oi++) {
    float sum = 0;
    int row = oi % len;
    for (int col = 0; col < len; col++) {
      unsigned i = (row * len) + col;  // 0, 1, 2, 3, 4,...
      unsigned j = (col * len) + row;  // 0, 3, 6, 9, 1,...
      sum += A[i] * A[j];
    }
    out[oi] = sum;
  }
}
```

The system view of the area report for the kernel `matrix_square` indicates that the estimated usages of flipflops (FF) and RAMs for Block3 are high. Further examination of Block3 in the system viewer shows that Block3 also has a high latency value.

Figure 26. Area Report and System Viewer Results for the Unoptimized Kernel matrix_square



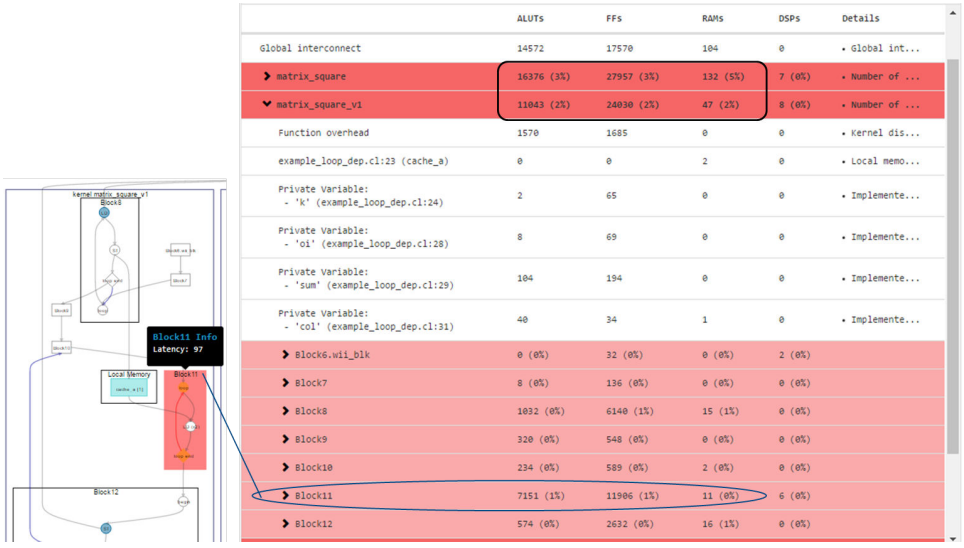
Area report (system view)
(area utilization values are estimated)
Notation *file:X > file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
Kernel System (Logic: 32%)	96988 (18%)	159749 (15%)	674 (26%)	31 (2%)	
Board interface	38262	44528	257	0	• Platform i...
Global interconnect	14572	17570	104	0	• Global int...
matrix_square	16376 (3%)	27957 (3%)	132 (5%)	7 (0%)	• Number of ...
Function overhead	1570	1685	0	0	• Kernel dis...
Private Variable: - 'oi' (example_loop_dep.cl:6)	8	69	0	0	• Implemente...
Private Variable: - 'sum' (example_loop_dep.cl:7)	104	194	0	0	• Implemente...
Private Variable: - 'col' (example_loop_dep.cl:9)	32	27	1	0	• Implemente...
Block0.wii_blk	0 (0%)	32 (0%)	0 (0%)	2 (0%)	
Block1	0 (0%)	102 (0%)	0 (0%)	0 (0%)	
Block2	234 (0%)	589 (0%)	2 (0%)	0 (0%)	
Block3	13662 (3%)	22339 (2%)	113 (4%)	5 (0%)	
Block4	766 (0%)	2920 (0%)	16 (1%)	0 (0%)	

The cause for these performance bottlenecks is because 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 modified code below.

```
kernel void matrix_square_v1 (global float* restrict A,
                              unsigned len,
                              global float* restrict out)
{
  // 1. preload the data into local memory
  //    - suppose we know the max size is 4X4

  local int cache_a[16];
  for(unsigned k = 0; k < len*len; k++) {
    cache_a[k] = A[k];
  }

  for(unsigned oi = 0; oi < len*len; oi++) {
    float sum = 0;
    int row = oi % len;
    for(int col = 0; col < len; col++) {
      unsigned i = (row * len) + col;  // 0, 1, 2, 3, 4,...
      unsigned j = (col * len) + row;  // 0, 3, 6, 9, 1,...
      sum += cache_a[i] * cache_a[j];
    }
    out[oi] = sum;
  }
}
```

Figure 27. Area Report and System Viewer Results for the Modified Kernel matrix_square_v1



	ALUTs	FFs	RAMs	DSPs	Details
Global interconnect	14572	17570	104	0	• Global int...
matrix_square	16376 (3%)	27957 (3%)	132 (5%)	7 (0%)	• Number of ...
matrix_square_v1	11043 (2%)	24030 (2%)	47 (2%)	8 (0%)	• Number of ...
Function overhead	1570	1685	0	0	• Kernel dis...
example_loop_dep.cl:23 (cache_a)	0	0	2	0	• Local memo...
Private Variable: - 'k' (example_loop_dep.cl:24)	2	65	0	0	• Implemente...
Private Variable: - 'oi' (example_loop_dep.cl:28)	8	69	0	0	• Implemente...
Private Variable: - 'sum' (example_loop_dep.cl:29)	104	194	0	0	• Implemente...
Private Variable: - 'col' (example_loop_dep.cl:31)	40	34	1	0	• Implemente...
Block6.wii_blk	0 (0%)	32 (0%)	0 (0%)	2 (0%)	
Block7	8 (0%)	136 (0%)	0 (0%)	0 (0%)	
Block8	1032 (0%)	6140 (1%)	15 (1%)	0 (0%)	
Block9	320 (0%)	548 (0%)	0 (0%)	0 (0%)	
Block10	234 (0%)	589 (0%)	2 (0%)	0 (0%)	
Block11	7151 (1%)	11906 (1%)	11 (0%)	6 (0%)	
Block12	574 (0%)	2632 (0%)	16 (1%)	0 (0%)	

As shown in the area report and the system viewer results, preloading the data into local memory reduces the RAM usage by one-third and lowers the latency value from 255 to 97.

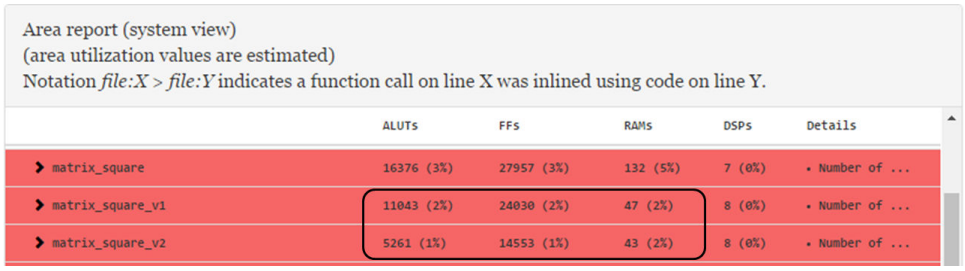
Further examination of the area report of `matrix_square_v1` shows that the code line `int row = oi % len`, which is line 30 in the area report below, uses an unusually large amount of area because of the modulus computation.

Figure 28. Source View of the Area Report for 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 usages by 50%.

```
kernel void matrix_square_v2 (global float* restrict A,
                              unsigned len,
                              global float* restrict out)
{
  // 1. preload the data into local memory
  //    - suppose we know the max size is 4X4
  // 2. remove the modulus computation

  local int cache_a[16];
  for (unsigned k = 0; k < len*len; k++) {
    cache_a[k] = A[k];
  }

  unsigned row = 0;
  unsigned ci = 0;
  for (unsigned oi = 0; oi < len*len; oi++) {
    float sum = 0;

    // keep a column counter to know when to increment row
    if (ci == len) {
      ci = 0;
      row += 1;
    }
    ci += 1;

    for (int col = 0; col < len; col++) {
      unsigned i = (row * len) + col;   // 0, 1, 2, 3, 4,...
      unsigned j = (col * len) + row;   // 0, 3, 6, 9, 1,...
      sum += cache_a[i] * cache_a[j];
    }
    out[oi] = sum;
  }
}
```

Figure 29. Comparison of Area Usages between Modified Kernels matrix_square_v1 and 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) + col` and `unsigned j = (col * len) + row`, respectively) have very different ALUT and FF usage estimations. In addition, the area report also shows that these two computations uses digital signal processing (DSP) blocks.

Figure 30. Area Report of Modified Kernel matrix_square_v2 Showing Different Area Usages for Different Index Computations

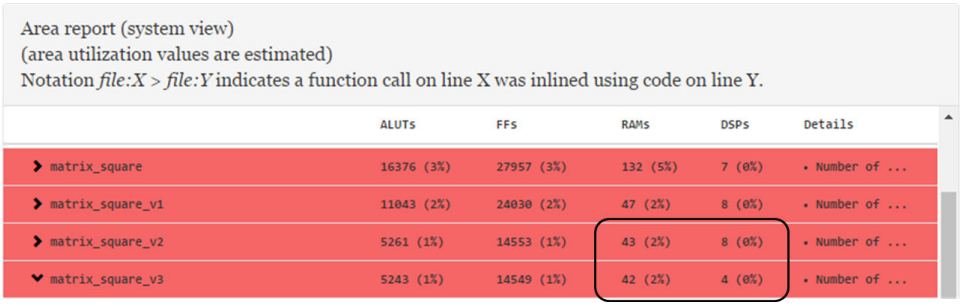
A way to optimize DSP and RAM block usages for index calculation is to remove the multiplication computation and simply keep track of the addition, as shown in the modified kernel `matrix_square_v3` below.

```
kernel void matrix_square_v3 (global float* restrict A,
                              unsigned len,
                              global float* restrict out) {
  // 1. preload the data into local memory
  //    - suppose we know the max size is 4X4
  // 2. remove the modulus computation
  // 3. remove DSP and RAM blocks for index calculation helps reduce the
latency

  local int cache_a[16];
  for (unsigned k = 0; k < len*len; k++) {
    cache_a[k] = A[k];
  }
```

```
  unsigned row_i = 0;
  unsigned row_j = 0;
  unsigned ci = 0;
  for (unsigned oi = 0; oi < len*len; oi++) {
    float sum = 0;
    unsigned i, j;

    // keep a column counter to know when to increment row
    if (ci == len) {
      ci = 0;
      row_i += len;
      row_j += 1;
    }
    ci += 1;

    i = row_i;  // initialize i and j
    j = row_j;
    for (int col = 0; col < len; col++) {
      i += 1;       // 0, 1, 2, 3, 0,...
      j += len;     // 0, 3, 6, 9, 1,...
      sum += cache_a[i] * cache_a[j];
    }
    out[oi] = sum;
  }
}
```

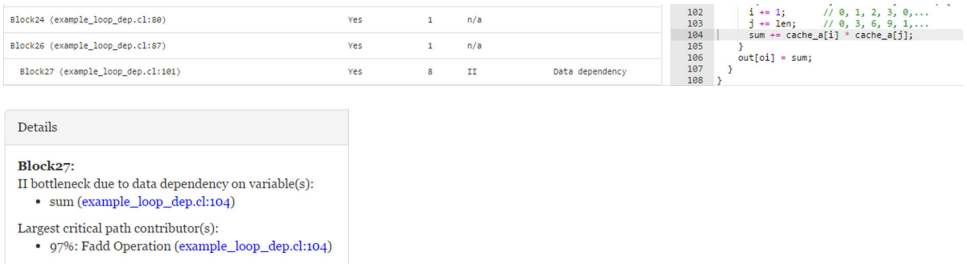
By removing the multiplication step, you can reduce DSP usage by 50%, as shown in the area report below. In addition, the modification helps reduce latency.

Figure 31. Comparison of Area Usages between Modified Kernels matrix_square_v2 and matrix_square_v3



To further reduce latency, you can review the loop analysis report of the modified kernel `matrix_square_v3`. As shown below, the analysis pane and the details pane of the report indicate that the line `sum += cache_a[i] * cache_a[j]` has a loop-carried dependency, causing Block27 to have an II bottleneck.

Figure 32. Loop Analysis of the Modified Kernel matrix_square_v3

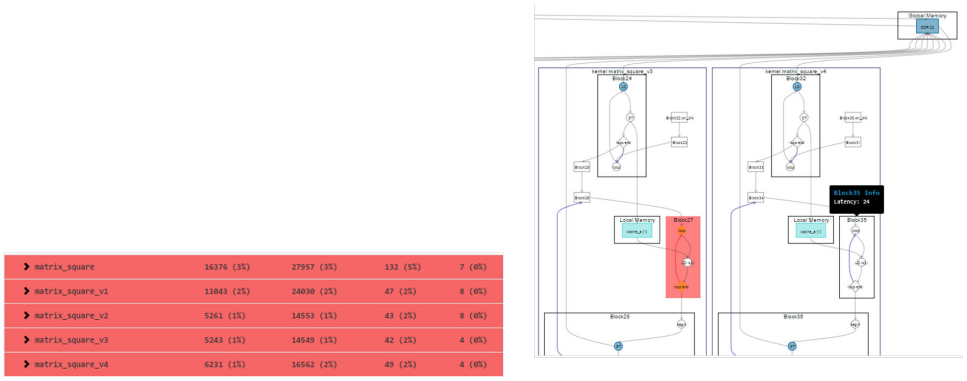


To resolve the loop-carried dependency, you can separate the multiplication and the addition portions of the computation, as shown in the highlighted code below in the modified kernel `matrix_square_v4`.

```
kernel void matrix_square_v4 (global float* restrict A,
                              unsigned len,
                              global float* restrict out)
{
  // 1. preload the data into local memory
  //    - suppose we know the max size is 4X4
  // 2. remove the modulus computation
  // 3. remove DSP and RAM blocks for index calculation helps reduce the
latency
  // 4. remove loop-carried dependency 'sum' to improve throughput by trading
off area

  local int cache_a[16];
  for (unsigned k = 0; k < len*len; k++) {
    cache_a[k] = A[k];
  }

  unsigned row_i = 0;
  unsigned row_j = 0;
  unsigned ci = 0;
  for (unsigned oi = 0; oi < len*len; oi++) {
    float sum = 0;
    unsigned i, j;

    float prod[4];   // make register
    #pragma unroll
    for (unsigned k = 0; k < 4; k++) {
      prod[k] = 0;
    }

    // keep a column counter to know when to increment row
    if (ci == len) {
      ci = 0;
      row_i += len;
      row_j += 1;
    }
    ci += 1;

    i = row_i;  // initialize i and j
    j = row_j;
    for (int col = 0; col < len; col++) {
      i += 1;       // 0, 1, 2, 3, 0,...
      j += len;     // 0, 3, 6, 9, 1,...
      prod[col] = cache_a[i] * cache_a[j];
    }

    sum = prod[0];
    #pragma unroll
    for (unsigned k = 1; k < 4; k++) {
      sum += prod[k];
    }

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

As shown in the area report and system viewer 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 II value of the loops to 1, and reduces the latency from 30 to 24.

Figure 33. Area Report and System Viewer Results of the Modified Kernel matrix_square_v4



2.7. HTML Report: Area Report Messages

After you compile your OpenCL application, review the area report that the Intel FPGA SDK for OpenCL Offline Compiler generates. In addition to summarizing the applications resource usage, the area report offers suggestions on how to modify your design to improve efficiency.

Area Report Message for Board Interface on page 42
Area Report Message for Function Overhead on page 42
Area Report Message for State on page 43
Area Report Message for Feedback on page 43
Area Report Message for Constant Memory on page 43
Area Report Messages for Private Variable Storage on page 43

2.7.1. 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 1. Additional Information on Area Report Message

Message	Notes
Platform interface logic.	—

2.7.2. 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 2. 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.

2.7.3. 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

2.7.4. 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.

2.7.5. 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 3. 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.	—

2.7.6. 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 4. 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*> and depth <*Z*> [(depth was increased by a factor of <*N*> due to a loop initiation interval of <*M*>.)] - ...	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*> register(s) of width <*Y*> and depth <*Z*> - ...	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*> and depth <*Z*> [(depth was increased by a factor of <*N*> due to a loop initiation interval of <*M*>.)] - ...	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 usage of the private variable. The report shows additional area usage information on the lines where the variable is accessed.

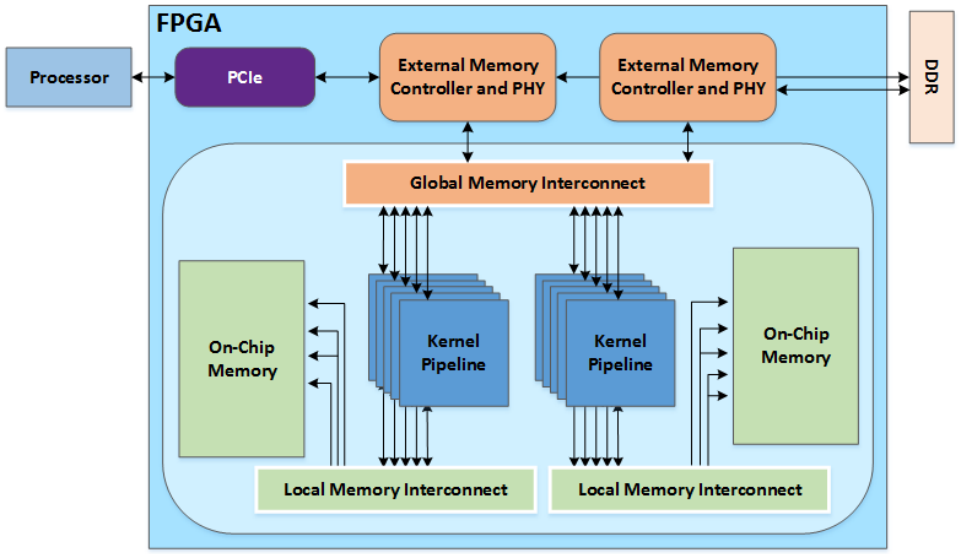
Note:

- 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.

2.8. HTML Report: 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.

Figure 34. OpenCL Design Components



Kernels on page 45
Global Memory Interconnect on page 46
Local Memory on page 47
Nested Loops on page 54
Loops in a Single Work-Item Kernel on page 61
Channels on page 68
Load-Store Units on page 68

2.8.1. Kernels

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 on 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 usually needs to be a large number of threads.

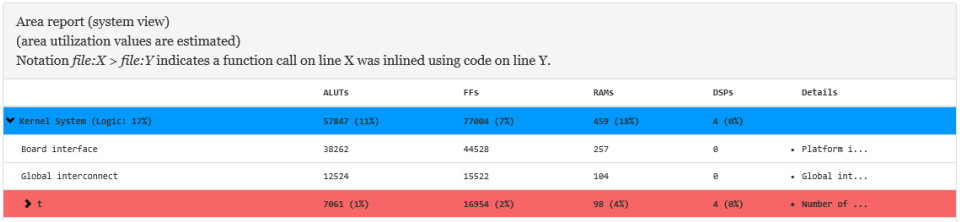
2.8.2. Global Memory Interconnect

The ability to maximize memory bandwidth for read and write accesses is crucial for high performance computing. There are various types of global memory interconnect that can exist in an OpenCL system. A memory interconnect is sometimes referred to as a *load-store unit* (*LSU*).

Unlike a GPU, an FPGA can build any custom LSU that is most optimal 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.

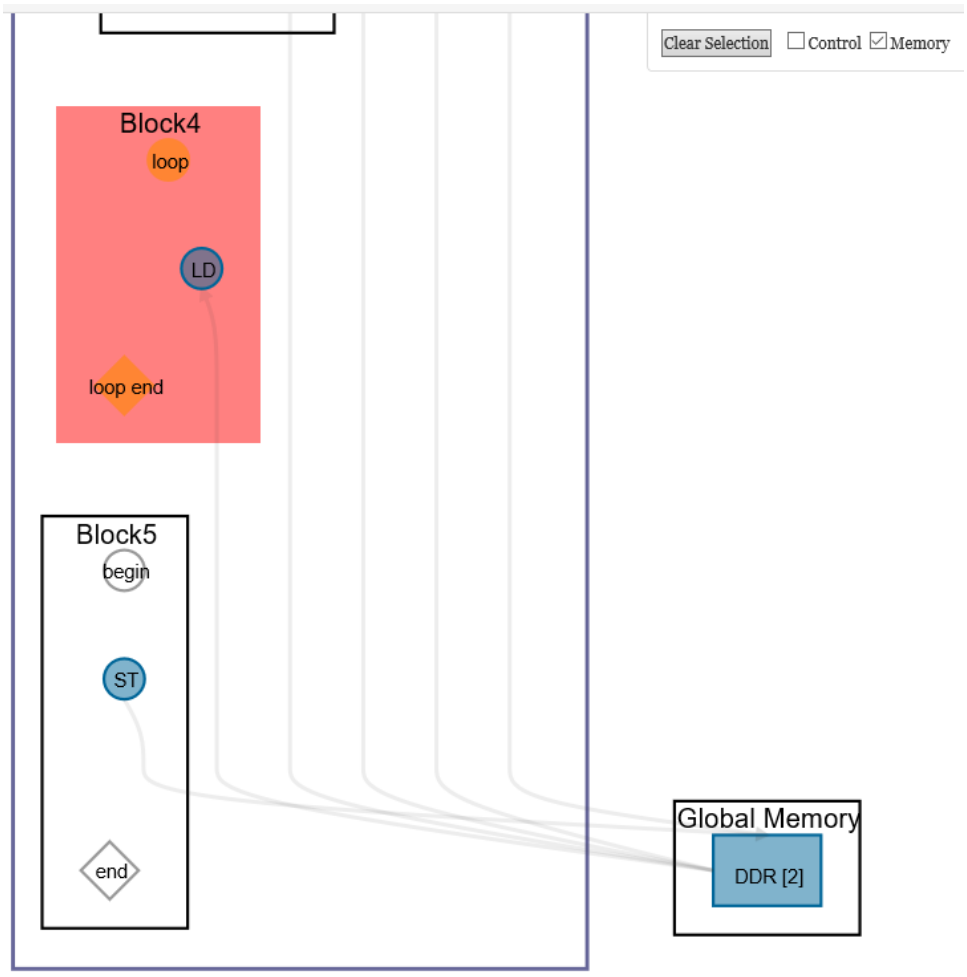
Figure 35. HTML Area Report Showing the Size of the Global Memory Interconnect in an OpenCL Design


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 memory system viewer depicts global memory interconnects as loads (LD), stores (ST), and connections (gray lines).

Figure 36. System Viewer 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 Accesses on page 133 and Optimize Global Memory Accesses on page 132 illustrate the difference in access patterns between contiguous and burst-interleaved memory accesses, respectively.

2.8.3. Local Memory

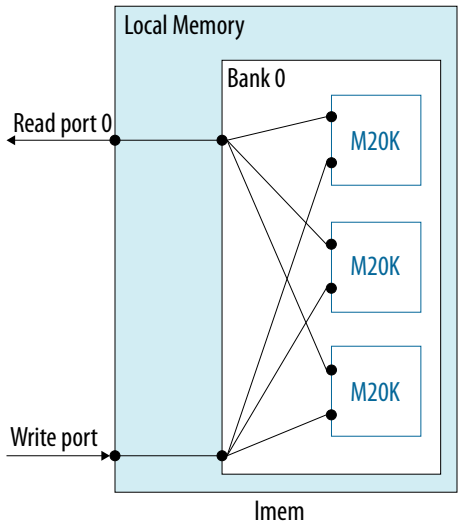
Local memory is a complex system. Unlike typical GPU architecture where there are different levels of caches, FPGA implements local memory in dedicated memory blocks inside the FPGA.

Local Memory Characteristics

- Ports—Each bank of local memory has one write port and one read port that your design can access simultaneously.
- Double pumping—The double-pumping feature allows each local memory bank to support up to three read ports. Refer to the *Double Pumping* section for more information.

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

Figure 37. Implementation of Local Memory in One or Multiple M20K Blocks



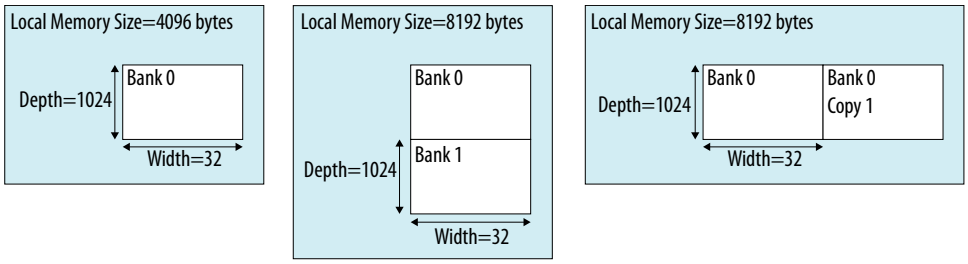
In your kernel code, declare local memory as a variable with type `local`:

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

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

The diagrams below illustrate these basic local memory properties: size, width, depth, banks, and replication.

Figure 38. Local Memory Examples with No Replication and Two Banks that Are Replicated Two Times

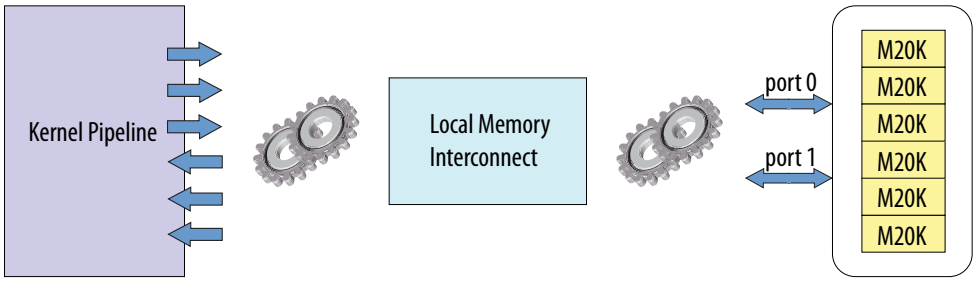


In the HTML report, the overall state of the local memory is reported as optimal, good but replicated, and potentially inefficient.

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

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 a local memory load-store unit (LSU) to arbitrate the memory access. However, inferring an LSU might cause inefficiencies. Refer to *Local Memory LSU* for more information.

Figure 39. Complex Local Memory Systems



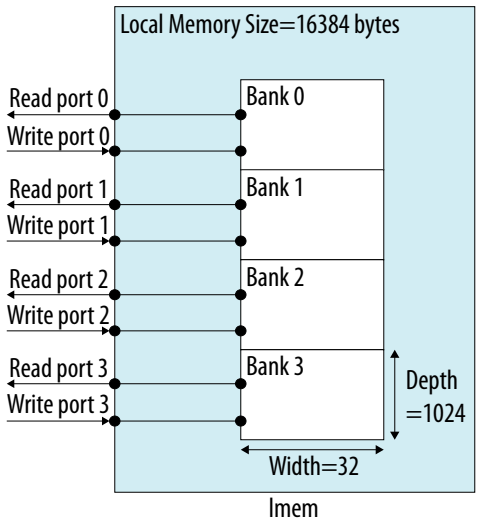
The offline compiler does not always implement local memory with the exact size that you specified. Since FPGA RAM blocks have specific dimensions, the offline compiler implements a local memory size that rounds up to the next supported RAM block dimension. Refer to device-specific information for more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

Local Memory Banks

Local memory banking only works on the lowest dimension by default. Having multiple banks allows simultaneous writes to take place. The figure below illustrates the implementation of the following local variable declaration:

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

Figure 40. Implementation of lmem[1024][4]



Each local memory access in a loop has a separate address. In the following code example, the offline compiler can infer `lmem` to create four separate banks. The loop allows four simultaneous accesses to `lmem[][]`, which achieves the optimal configuration.

```
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 >> 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;
}
```

If you do not want to bank on the lowest dimension, specify 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. The specified `bank_bits` attribute also implies which memory bank contains which data element:

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

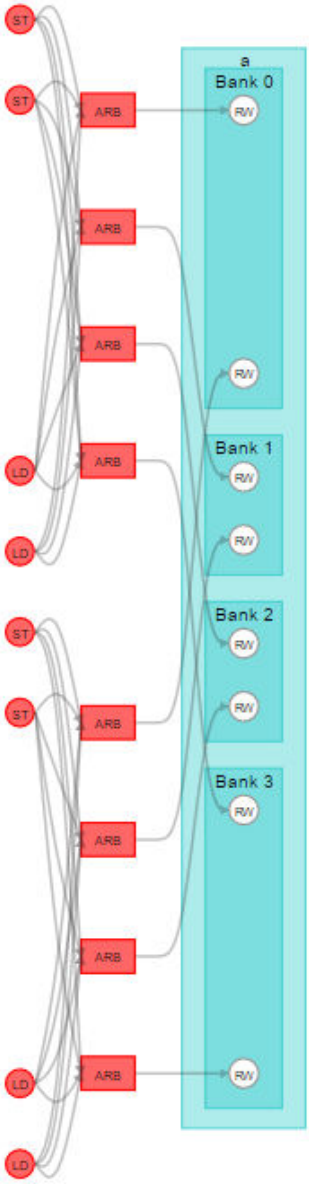
In the following example, the banking is done on the seventh and eighth bits instead of the lowest two dimensions:

```
#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 >> 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. However, if you specify the wrong bits to bank on, the memory arbitration logic changes.

The following view of the memory results from specifying the memory as follows:

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



If the compiler cannot infer the local memory accesses to separate addresses, it uses a local memory interconnect to arbitrate the accesses, which degrades performance.

Local Memory Replication

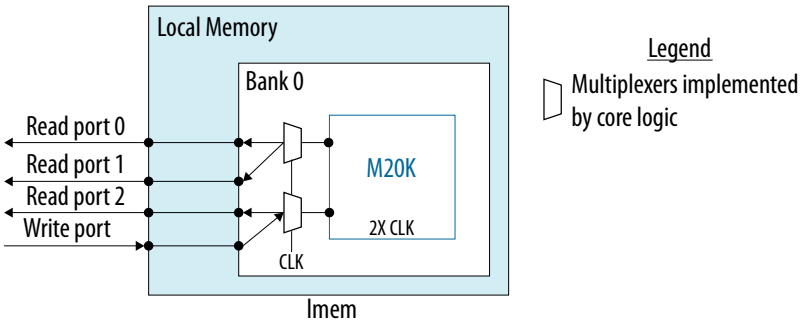
Local memory replication allows for simultaneous read operations to occur. The offline compiler optimizes your design for efficient local memory access in order to maximize overall performance. Although memory replication leads to inefficient hardware in some cases, memory replication does not always increase RAM usage.

When the offline compiler recognizes that more than two work groups are reading from local memory simultaneously, it replicates the local memory. If local memory replication increases your design area significantly, consider reducing the number of barriers in the kernel or increasing the `max_work_group_size` value to help reduce the replication factor.

Double Pumping

By default, each local memory bank has one read port and one write port. The double pumping feature allows each local memory bank to support up to three read ports.

Figure 41. Hardware Architecture of Double Pumping in Local Memory



The underlying mechanism that enables double pumping is in the M20K hardware. During the first clock cycle, the M20K block is double clocked. Then, during the second clock cycle, the ports are multiplexed to create two more read ports.

By enabling the double pumping feature, the offline compiler trades off area versus maximum frequency. The offline compiler uses heuristic algorithms to determine the optimal memory configurations.

Advantages of double pumping:

- Increases from one read port to three read ports
- Saves RAM usage

Disadvantages of double pumping:

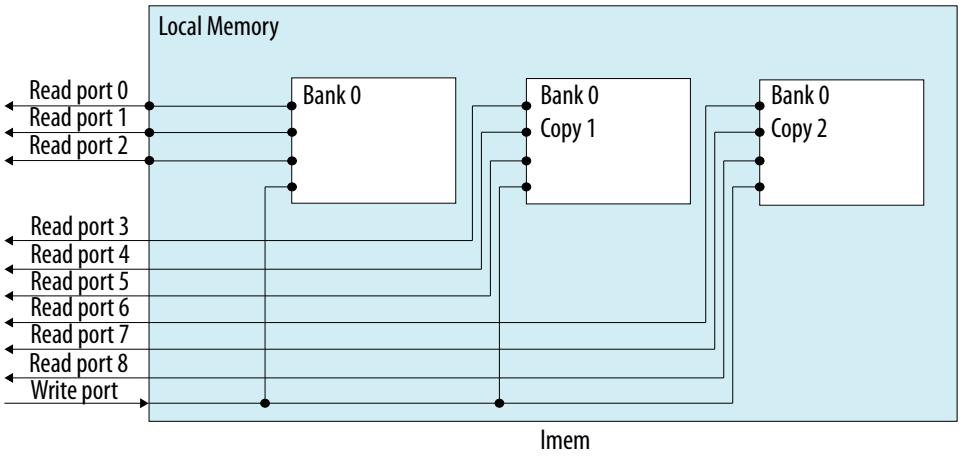
- Implements additional logic
- Might reduce maximum frequency

The following code example illustrates the implementation of local memory with eight read ports and one write port. The offline compiler enables double pumping and replicates the local memory three times to implement a memory configuration that can support up to nine read ports.

```
#define NUM_WRITES   1
#define NUM_READS    8
#define NUM_BARRIERS 1

  local int lmem[1024];
  int li = get_local_id(0);

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

```
}

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

res = 0;
#pragma unroll
for (int i = 0; i < NUM_READS; i++) {
  res ^= lmem[li - i];
}
```

Figure 42. Intel FPGA SDK for OpenCL Offline Compiler's Implementation of lmem[] with Eight Read Ports and One Write Port



2.8.4. 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.

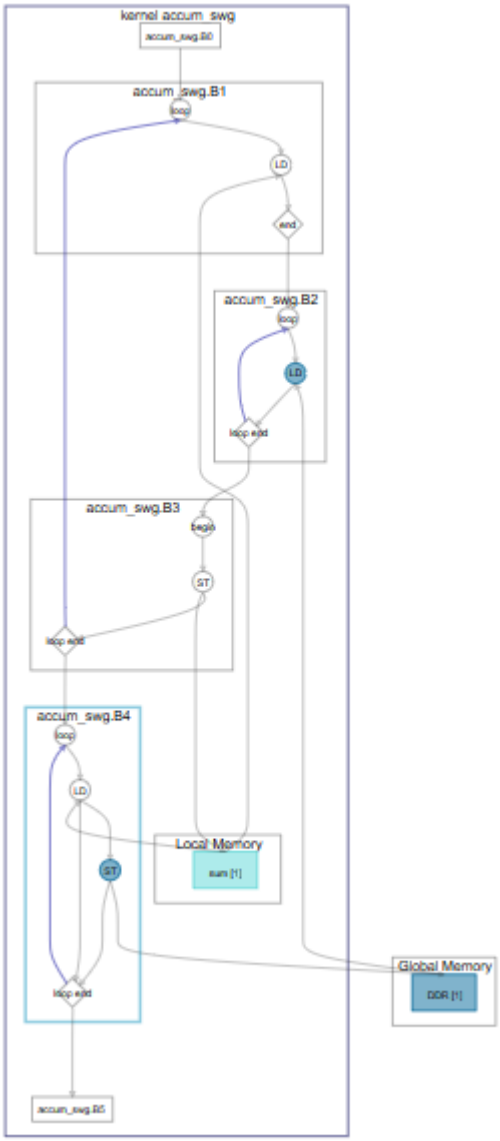
Note: The launch frequency of a new loop iteration is called the initiation interval (II). II refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. An optimally unrolled loop has an II value of 1 because one loop iteration is processed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

Figure 43. Loop Analysis Report

Loops analysis — ☑ Show fully unrolled loops

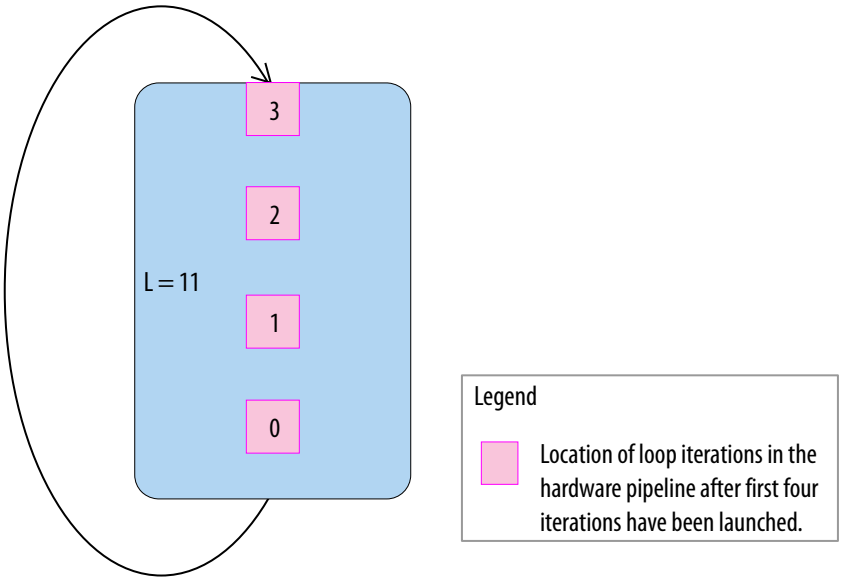
	Pipelined	II	Bottleneck	Details
Kernel: accum_swg (accum_swg.cl:3)				Single work-item execution
accum_swg.B1 (accum_swg.cl:5)	Yes	>=1	n/a	
accum_swg.B2 (accum_swg.cl:6)	Yes	~1	n/a	II is an approximation.
accum_swg.B4 (accum_swg.cl:12)	Yes	~1	n/a	II is an approximation.

Figure 44. System View of Single-Work Item Kernel



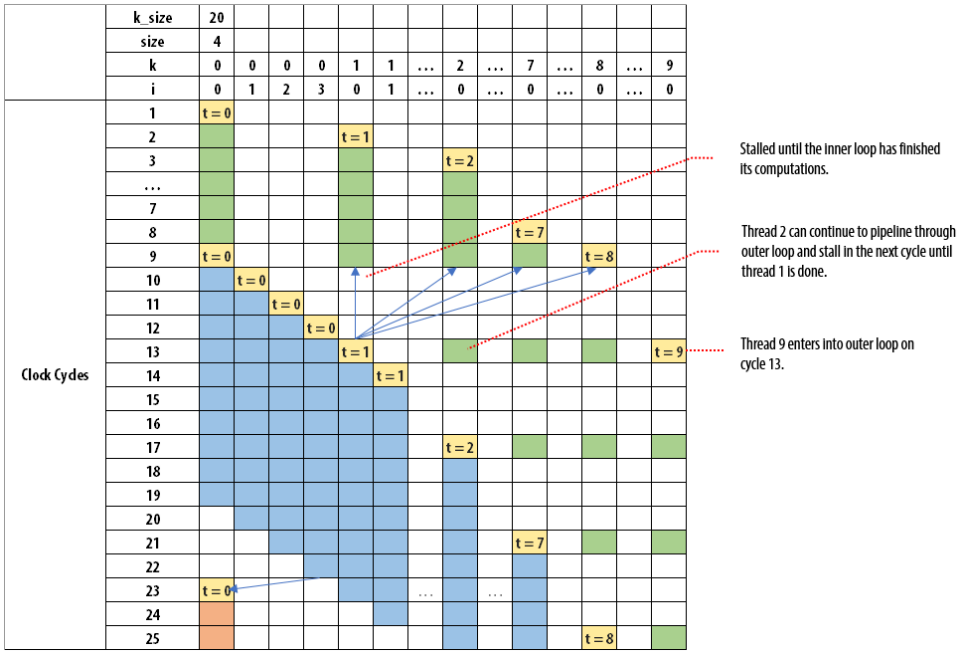
The following figure illustrates how each iteration of i enters into the block:

Figure 45. Inner Loop `accum_swg.B2` Execution



When we 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 will take 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 will enter into the loop at every four clock cycles, which is the value of `size`. Through this example, we 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 46. Single Work-Item Execution



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;
      }
    }
  }
}
```

Out-of-Order Loop Iterations

The number of iterations of an inner loop can differ for each iteration of the outer loop. Consider the following code example:

```
__kernel void order( __global unsigned* restrict input,
                     __global unsigned* restrict output
                     int N ) {
  unsigned sum = 0;
  for (unsigned i = 0; i < N; i++) {
    for (unsigned j = 0; j < i; j++) {
      sum += input[i+j];
```

```
    }
  }
  output[0] = sum;
}
```

This example shows that for `i = 0`, inner loop `j` iterates zero times. For `i = 1`, `j` iterates once, and so on. Because the number of iterations changes for the inner loop, the offline compiler cannot infer pipelining.

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 47. System View of the Kernel

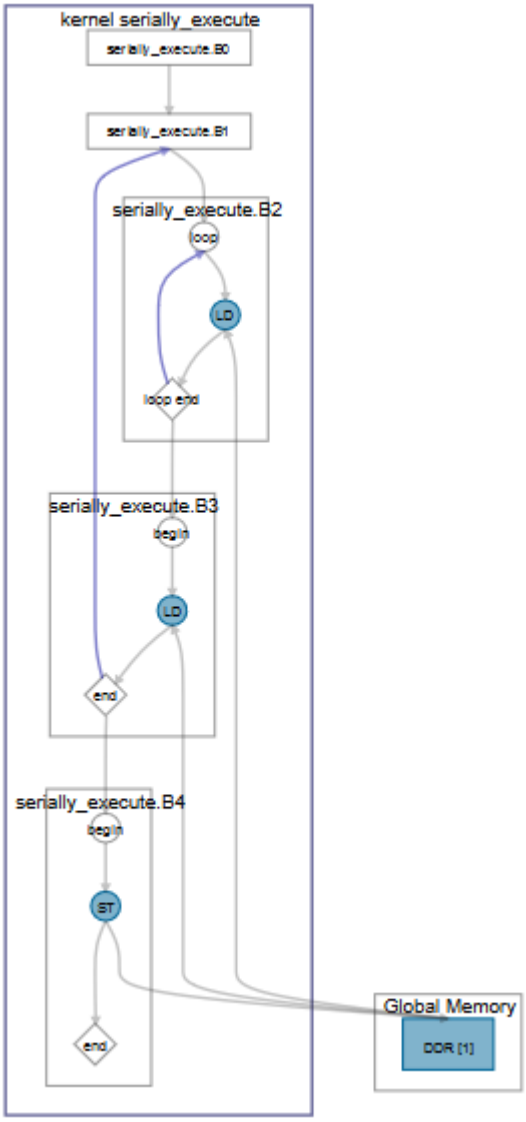
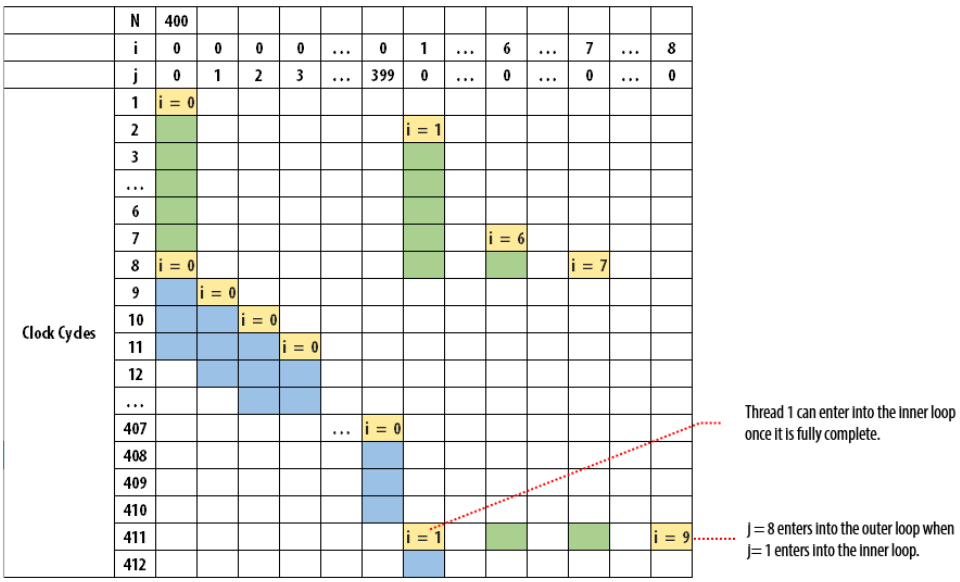


Figure 48. Serial Execution

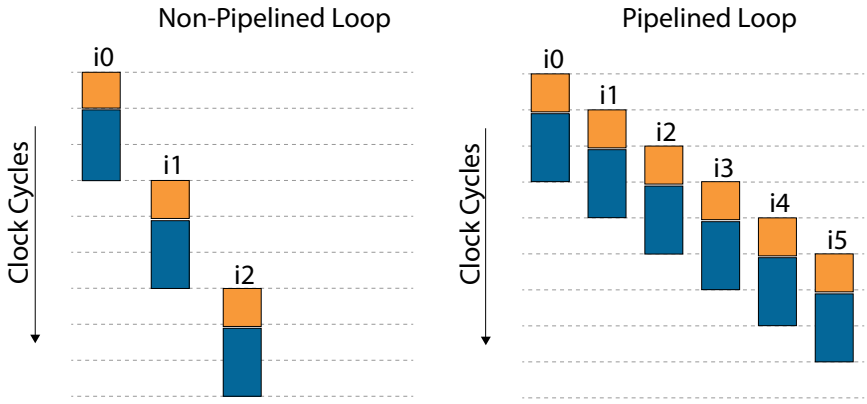


2.8.5. Loops in a Single Work-Item Kernel

The Intel FPGA SDK for OpenCL Offline Compiler implements an algorithm that optimizes kernels to maximize performance for data processing.

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 in flight. An optimally unrolled loop is a loop iteration that 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 49. Comparison of the Launch Frequency of a Loop Iteration Between a Non-Pipelined Loop and a Pipelined Loop



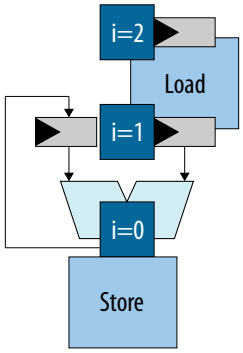
The launch frequency of a new loop iteration is called the initiation interval (II). II refers to the number of hardware clock cycles for which the pipeline must wait before it can process the next loop iteration. An optimally unrolled loop has an II value of 1 because one loop iteration is processed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

In the HTML report, the loop analysis of an optimally unrolled loop will state that the offline compiler has pipelined the loop successfully.

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 50. Hardware Representation of the Kernel `simple_loop`



The figure depicts how the offline compiler leverages parallel execution and loop pipelining to execute `simple_loop` efficiently. The loop analysis report of this `simple_loop` kernel will show that for loop "for.body", the **Pipelined** column will indicate Yes, and the **II** column will indicate 1.

Trade-Off Between Critical Path and Maximum Frequency

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

Consider the following example:

```
kernel void nd (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 logical diagram is a simplified representation of the actual, more complex hardware implementation of kernel nd.

Figure 51. Logical Diagram for the Loop in Kernel nd

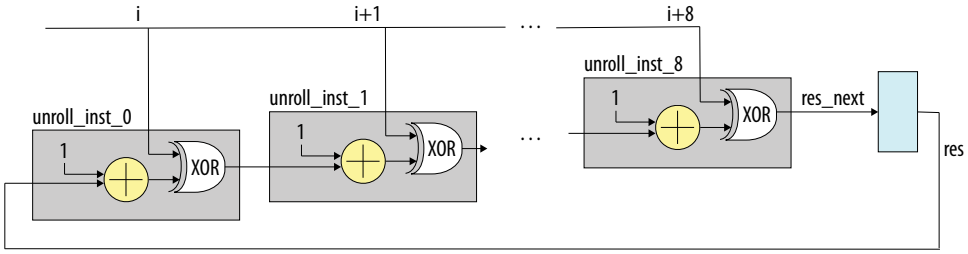
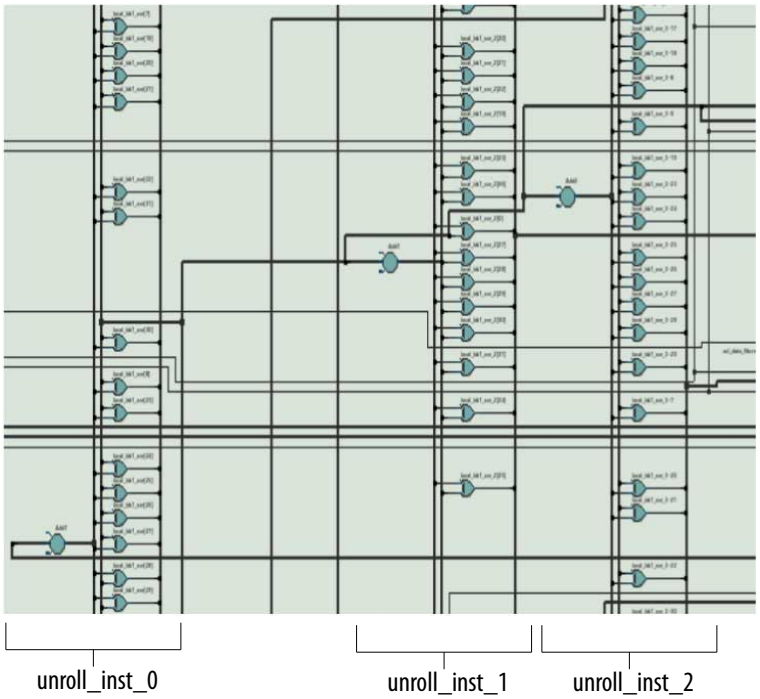


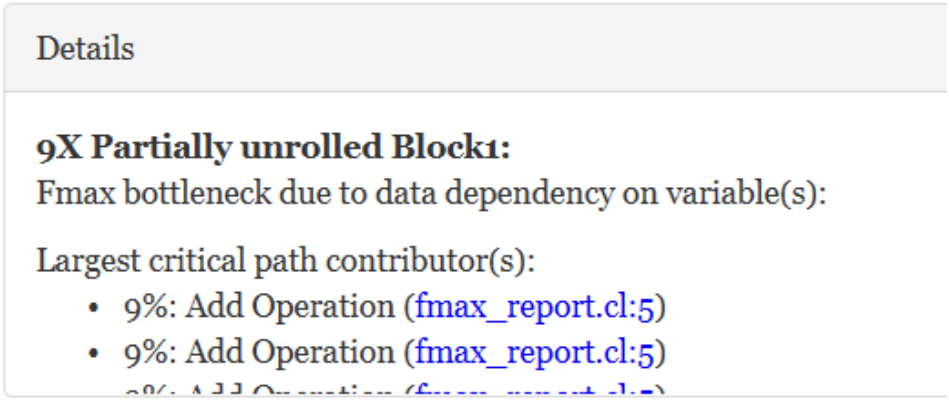
Figure 52. Actual Hardware Implementation of Kernel nd



The feedback with the addition operation and the XOR gate is the critical path that limits the offline compiler's ability to achieve the target frequency. The resulting HTML report describes a breakdown, in percentages, of the contributors that make up the critical path.

Figure 53. Details Pane of the Loop Analysis Report for Kernel nd

The line "9%: Add Operation (fmax_report.cl:5)" repeats nine times, one for each feedback.



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

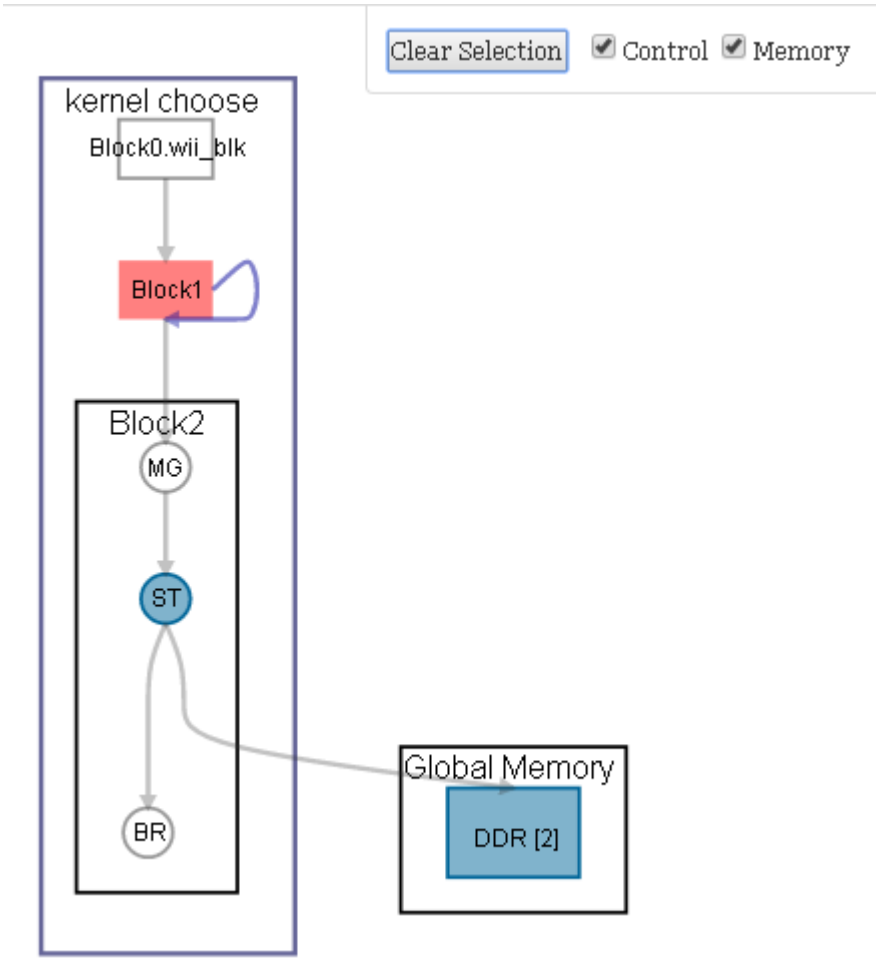
There are cases where a loop is pipelined but it does not achieve an II value of 1. These cases are usually caused by data dependencies or memory dependencies within a loop.

Data 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 will be greater than 1. Consider the following example:

```
1  // An example that shows data dependency
2  // choose(n, k) = n! / (k! * (n-k)!)
3
4  kernel void choose( unsigned n, unsigned k,
5                      global unsigned* restrict result )
6  {
7      unsigned product = 1;
8      unsigned j = 1;
9
10     for( unsigned i = k; i <= n; i++ ) {
11         product *= i;
12         if( j <= n-k ) {
13             product /= j;
14         }
15         j++;
16     }
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 figure below shows the logical view of the kernel `choose`, as it appears in the system viewer.

Figure 54. Logical View of the Kernel choose



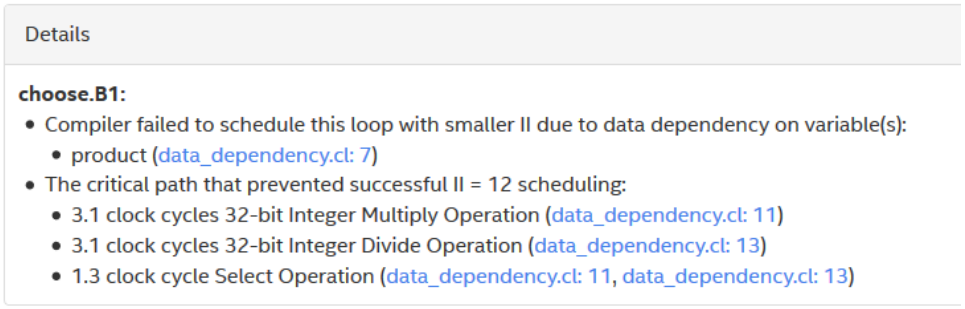
The loop analysis report for the kernel choose indicates that Block1 has an II value of 13. In addition, the details pane reports that the high II value is caused by a data dependency on product, and the largest contributor to the critical path is the integer division operation on line 13.

Figure 55. Loop Analysis Report of the Kernel choose

Loops analysis — ☑ Show fully unrolled loops

	Pipelined	II	Bottleneck	Details
Kernel: choose (data_dependency.cl:6)				Single work-item execution
choose.B1 (data_dependency.cl:10)	Yes	13	II	Data dependency

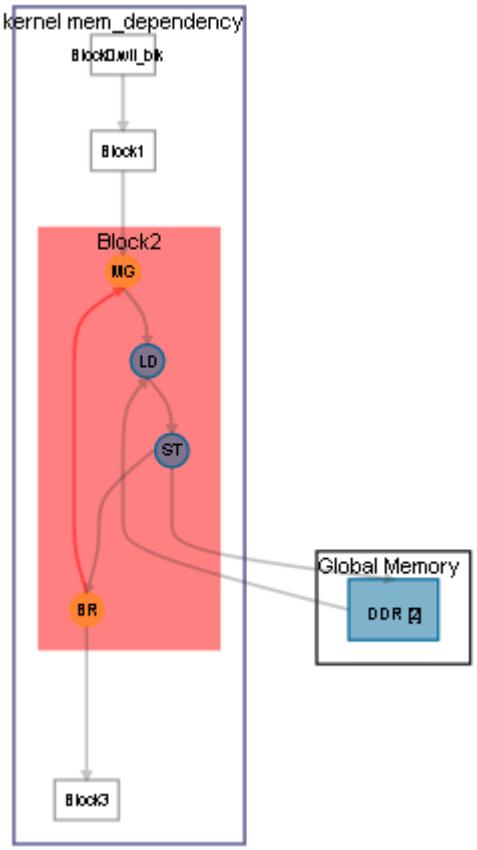
Figure 56. Information in the Details Pane of the Loop Analysis Report for the Kernel choose



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:

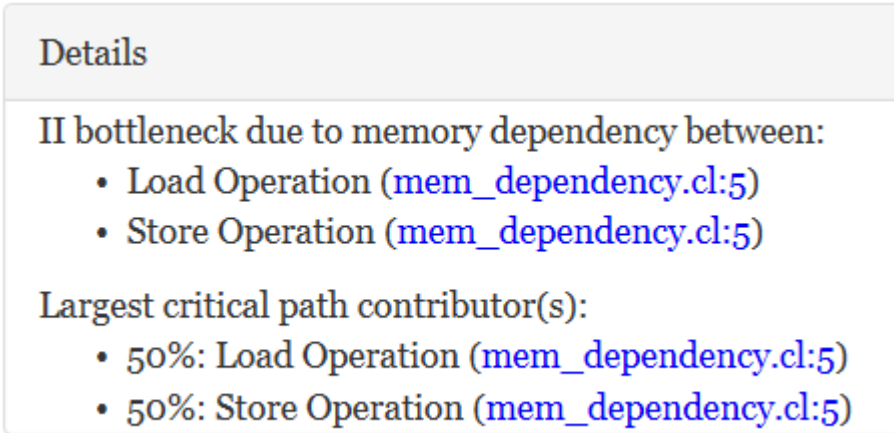
```
1  kernel void mirror_content( unsigned max_i,
2                              global int* restrict out)
3  {
4    for (int i = 1; i < max_i; i++) {
5      out[max_i*2-i] = out[i];
6    }
7  }
```

Figure 57. Logical View of the Kernel mirror_content



In the loop analysis for the kernel `mirror_content`, the details pane identifies the source of the memory dependency and the breakdown, in percentages, of the contributors to the critical path for Block2.

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



Related Information

Single Work-Item Kernel versus NDRange Kernel on page 9

2.8.6. 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 that 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.

2.8.7. 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.

While you cannot explicitly choose the load-store unit type or modifier, you can affect the type of LSU the compiler instantiates by changing the memory access pattern in your code, the types of memory available, and whether the memory accesses are to local or global memory.

Load-Store Unit Types

The compiler can generate several different types of load-store unit (LSU) 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 69
- Prefetching Load-Store Units on page 69
- Streaming Load-Store Units on page 69
- Semi-Streaming Load-Store Units on page 70
- Local-Pipelined Load-Store Units on page 70
- Global Infrequent Load-Store Units on page 70
- Constant-Pipelined Load-Store Units on page 71
- Atomic-Pipelined Load-Store Units on page 71

Burst-Coalesced Load-Store Units

A burst-coalesced LSU is the default LSU type instantiated by the compiler. 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 on page 71
- Write-Acknowledge (write-ack) on page 72
- Nonaligned on page 72

Prefetching Load-Store Units

A prefetching LSU instantiates a FIFO (sometimes called a named pipe) 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.

```
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;
}
```

Streaming Load-Store Units

A streaming LSU instantiates a FIFO which burst reads large blocks from memory to keep the FIFO full of valid data. This block of data can be used only if memory accesses are in-order, and addresses can be calculated as a simple offset from the base address.

```
kernel void streaming (global int * restrict in,
                       global int * restrict out) {
  int i = get_global_id(0);
  int idx = out[i];           // Streaming LSU
  int cached_value = in[idx];
  out[i] = cached_value;      // Streaming LSU
}
```

Semi-Streaming Load-Store Units

A semi-streaming LSU instantiates a read-only cache. The cache will have an area overhead, but will provide improved performance in cases where you make repeated accesses to the same data location in the global memory. You must ensure that your data is not overwritten by a store within the kernel, as that would break the coherency of the cache. The LSU cache is flushed each time the associated kernels are started.

```
#define N 16
kernel void semi_streaming (global int * restrict in,
                            global int * restrict out) {
  #pragma unroll 1
  for (int i = 0; i < N; i++) {
    int value = in[i]; // Semi-streaming LSU
    out[i] = value;
  }
}
```

Local-Pipelined Load-Store Units

A local-pipelined LSU is a pipelined LSU that 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 local-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;                // Local-pipelined LSU
    res >>= 1;
  }

  barrier(CLK_GLOBAL_MEM_FENCE);

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

  out[gi] = res;
}
```

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

- Never-stall on page 72

Global Infrequent Load-Store Units

A global infrequent LSU is a pipelined LSU that is used for global memory accesses that can be proven to be infrequent. The global infrequent LSU is instantiated only for memory operations that are not contained in a loop, and are active only for a single thread in an NDRange kernel.

The compiler implements a global infrequent LSU as pipelined LSU because a pipelined LSU is smaller than other LSU types. While a pipelined LSU might have lower throughput, this throughput tradeoff is acceptable because the 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];                    // Global Infrequent 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 136.

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
}
```

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 can be disabled by simplifying the access pattern or marking the pointer as `volatile`.

```
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;
}
```

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
  out[3*i+0] = a3;
  out[3*i+1] = a2;
  out[3*i+2] = a1;
}
```

Never-stall

If a local-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.

In the following example, some of the 96-bit wide memory access span two memory words, which requires two full lines of data to be read from memory.

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

3. 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 the performance by scaling the hardware to fill the remainder of the FPGA. The hardware footprint of the kernel correlates with the time it takes for hardware compilation. Therefore, the more optimizations you can perform with a smaller footprint (that is, a single compute unit), the more hardware compilations you can perform in a given amount of time.

In addition to data processing and memory access optimizations, consider implementing the following design practices, if applicable, when you create your kernels.

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

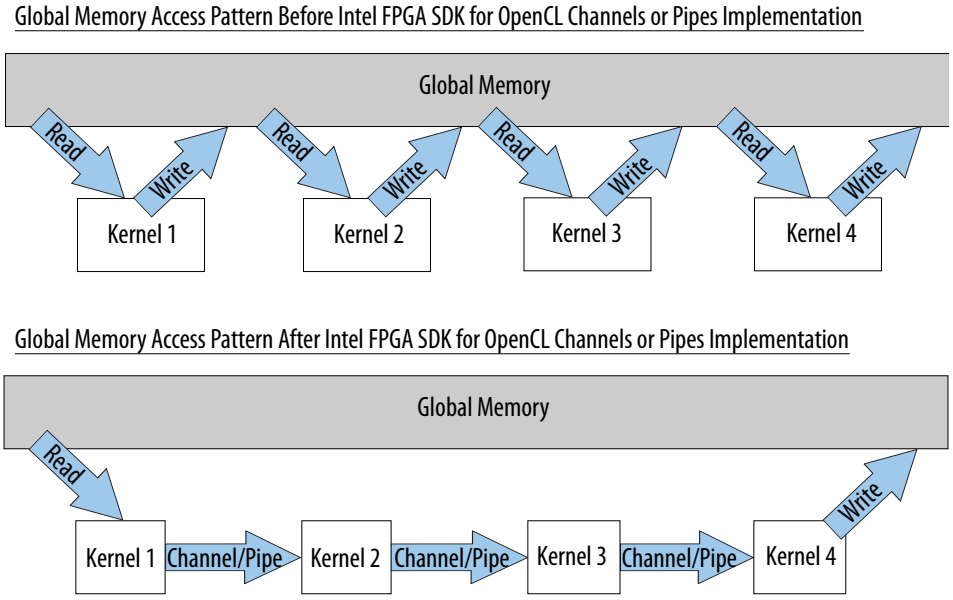
To increase data transfer efficiency between kernels, implement the Intel FPGA SDK for OpenCL channels extension in your kernel programs. If you want to leverage the capabilities of channels but have the ability to run your kernel program using other SDKs, implement OpenCL pipes.

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



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

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

Related Information

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

3.1.1. 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 Standard Edition 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 will only block when the pipe is empty, and the `write_pipe` calls will only 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 5. Blocking Channel and Pipe Calls for Kernel Synchronization

Kernel with Blocking Channel 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++)
  {
    ret_buf[i] = read_channel_intel(c0);
  }
}
```

Kernel with Blocking Pipe Call

```
__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++)
  {
    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

3.1.2. 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 6. 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++) {`
	`  read_pipe (c0, &dst[2*i]);`
	`  read_pipe (c1, &dst[2*i+2]);`
	`  }`
	`}`

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`.

3.1.3. 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 7. Buffer Inference Optimization for Channels and Pipes

Kernel with Channels

```
__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);
  }
}
```

Kernel with Pipes

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

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.

3.1.4. 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.

3.2. 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.

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 will issue 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 will perform 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 117

3.3. 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.

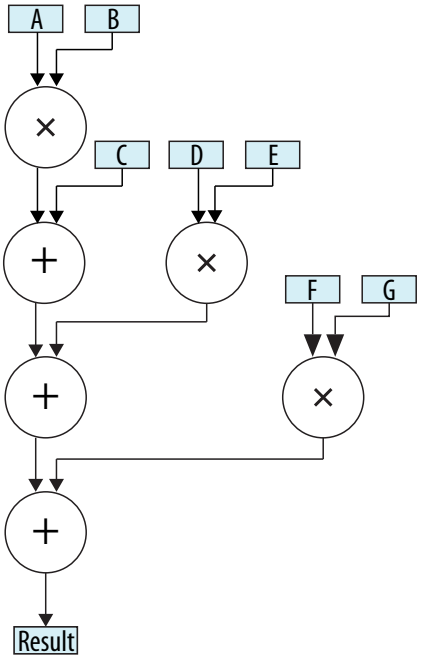
Tree Balancing

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);
```

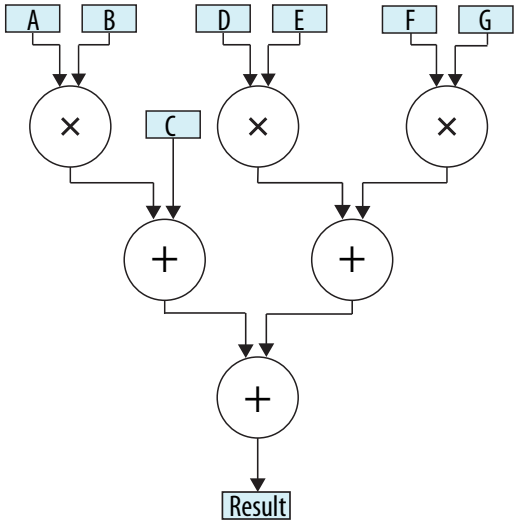
By default, the offline compiler creates an implementation that resembles a long vine for such computations:

Figure 60. 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 61. 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.

3.3.1. 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.

3.4. Allocating Aligned Memory

When allocating host-side memories that will be 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 kernel(s). For more information on 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

3.5. 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 will be 1-byte aligned. If the size of `mystruct` is 128 bytes, each load or store access will be 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 `mystruct4` is 16 bytes. Because `mystruct4` 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 will be efficient.

For more information on `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 Standard Edition Programming Guide*
- *Specifying the Alignment of a Struct* section of the *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*

Related Information

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

3.6. 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.
}
```

3.7. 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.

3.8. 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 on 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 80

3.9. 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
}
```

4. Profiling Your Kernel to Identify Performance Bottlenecks

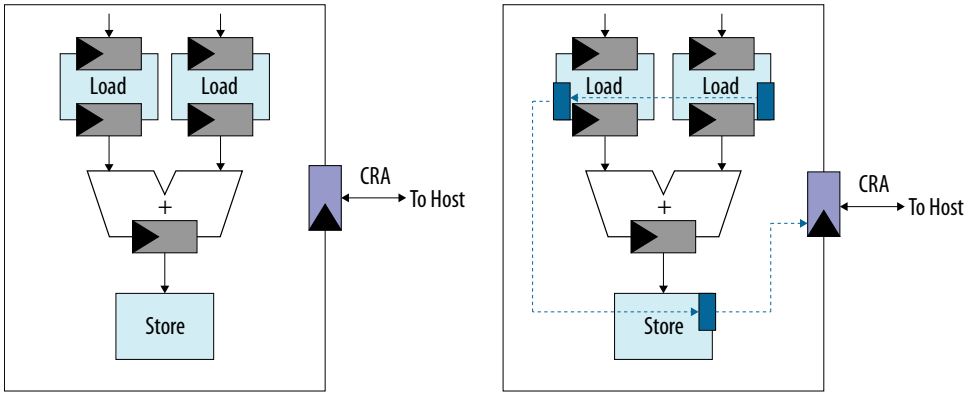
The Intel FPGA Dynamic Profiler for OpenCL generates data that helps you assess OpenCL kernel performance. The Intel FPGA Dynamic Profiler for OpenCL instruments the kernel pipeline with performance counters. These counters collect kernel performance data, which you can review via the profiler GUI.

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 62. 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.

For usage information on the Intel FPGA Dynamic Profiler for OpenCL, refer to the *Profiling Your OpenCL Kernel* section of the *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

Related Information

Profiling Your OpenCL Kernel

4.1. Intel FPGA Dynamic Profiler for OpenCL Best Practices

Intel recommends that you follow the Intel FPGA Dynamic Profiler for OpenCL 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.
- Periodically check kernel f_{max} and performance without using the Profiler.
- 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 Profiler's overhead takes over.
- Understand how all the load and store operations and channels are connected in the data flow.

4.2. Intel FPGA Dynamic Profiler for OpenCL GUI

The Intel FPGA Dynamic Profiler for OpenCL GUI displays statistical information collected from memory and channel or pipe accesses.

Table 8. Summary Heading in the Intel FPGA Dynamic Profiler for OpenCL GUI

Heading	Description
Board	Name of the accelerator board that the Intel FPGA SDK for OpenCL Offline Compiler uses during kernel emulation and execution.
Global Memory BW (DDR)	Maximum theoretical global memory bandwidth available for each memory type (for example, DDR).

Directly below the summary heading, you can view detailed profile information by clicking on the available tabs.

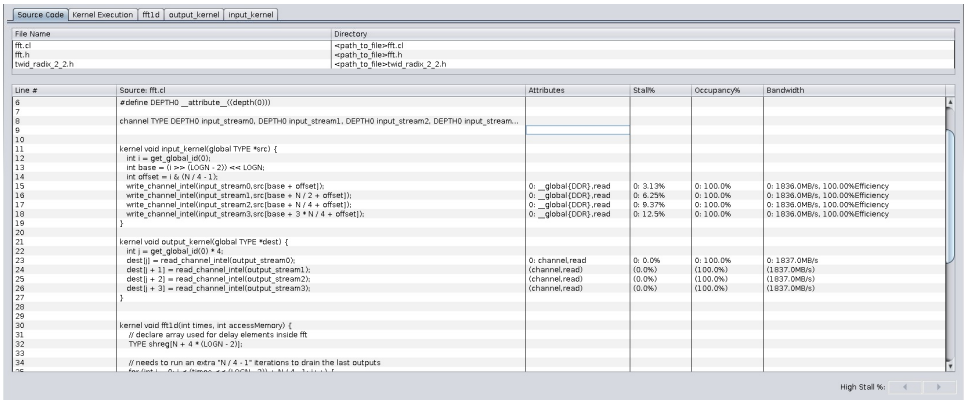
Important: In the following sections, information that relates to the SDK's channel extension also applies to OpenCL pipes.

Source Code Tab on page 90
Kernel Execution Tab on page 92
Autorun Captures Tab on page 94

4.2.1. Source Code Tab

The **Source Code** tab in the Intel FPGA Dynamic Profiler for OpenCL GUI contains source code information and detailed statistics about memory and channel accesses.

Figure 63. The Source Code tab in the Intel FPGA Dynamic Profiler for OpenCL GUI



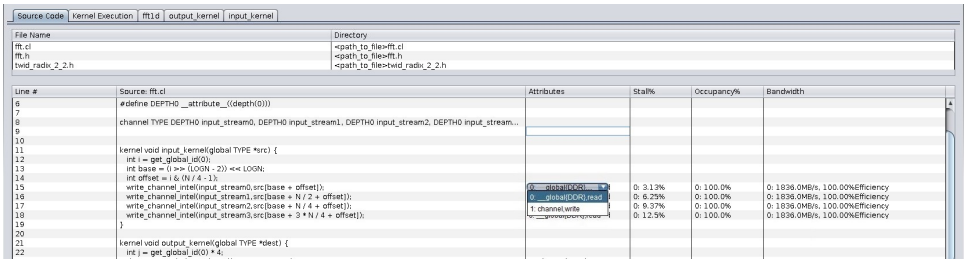
The **Source Code** tab provides detailed information on specific lines of kernel code.

Table 9. Types of Information Available in the Source Code Tab

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

If a line of source code instructs more than one memory or channel operations, the profile statistics appear in a drop-down list box and you may select to view the relevant information.

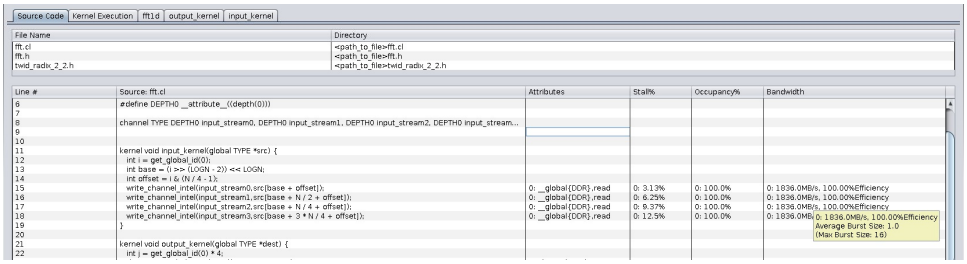
Figure 64. Source Code Tab: Drop-Down List for Multiple Memory or Channel Operations



4.2.1.1. Tool Tip Options

To obtain additional information about the kernel source code, hover your mouse over channel accesses in the code to activate the tool tip.

Figure 65. The Intel FPGA Dynamic Profiler for OpenCL GUI: Source Code Tab Tool Tip



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

Table 10. Types of Information that a Source Code Tab Tool Tip Might Provide

Column	Tool Tip	Description	Example Message	Access Type
Attributes	Cache Hits	The number of memory accesses using the cache. A high cache hit rate reduces memory bandwidth utilization.	Cache Hit%=30%	Global memory
	Unaligned Access	The percentage of unaligned memory accesses. A high unaligned access percentage signifies inefficient memory accesses. Consider modifying the access patterns in the kernel code to improve efficiency.	Unaligned Access %=20%	Global memory
	Statically Coalesced	Indication of whether the load or store memory operation is statically coalesced. Generally, static memory coalescing merges multiple memory accesses that access consecutive memory addresses into a single wide access.	Coalesced	Global or local memory
Occupancy%	Activity	The percentage of time a predicated channel or memory instruction is enabled (that is, when conditional execution is true). *Note:* The activity percentage might be less than the occupancy of the instruction.	Activity=20%	Global or local memory, and channels
Bandwidth	Burst Size	The average burst size of the memory operation. If the memory system does not support burst mode (for example, on-chip RAM), no burst information will be available.	Average Burst Size=7.6 (Max Burst=16)	Global memory

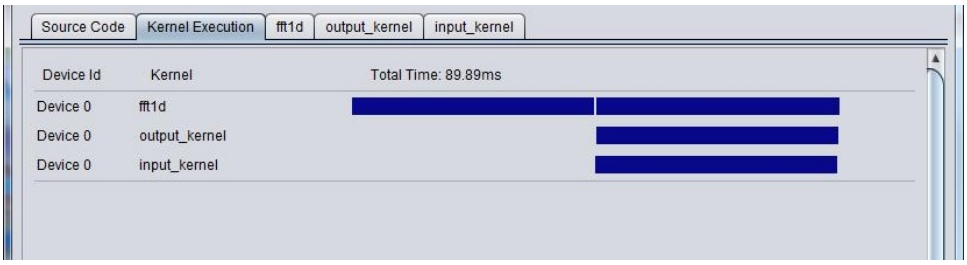
4.2.2. Kernel Execution Tab

The **Kernel Execution** tab in the Intel FPGA Dynamic Profiler for OpenCL GUI provides a graphical representation of the overall kernel program execution process. It illustrates the execution time of each kernel and provides insight into the interactions between different kernel executions.

For example, if you run the host application from a networked directory with slow network disk accesses, the GUI can display the resulting delays between kernel launches while the runtime stores profile output data to disk.

Attention: To avoid potential delays between kernel executions and increases in the overall execution time of the host application, run your host application from a local disk.

Figure 66. The Kernel Execution Tab in the Intel FPGA Dynamic Profiler for OpenCL GUI

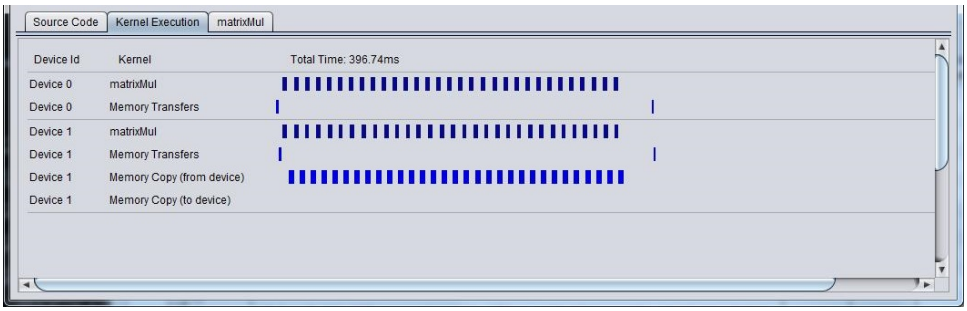


The horizontal bar graph represents kernel execution through time. The combination of the two bars shown in the first entry (fft1d) represents the total time. The second and last entries show kernel executions that occupy the time span. These bars represent the concurrent execution of `output_kernel` and `input_kernel`, and indicate that the kernels share common resources such as memory bandwidth.

Tip: You can examine profile data for specific execution times. In the example above, when you double-click the bar on the left for fft1d, another window opens to display profile data for that specific kernel execution event.

The **Kernel Execution** tab also displays information on memory transfers between the host and your devices, shown below:

Figure 67. Kernel Execution Tab: Host-Device Memory Transfer Information



Attention: Adjusting the magnification by zooming in or out might cause subtle changes to the granularity of the time scale.

To enable the display of memory transfer information, set the environment variable *ACL_PROFILE_TIMER* to a value of 1 and then run your host application. Setting the *ACL_PROFILE_TIMER* environment variable enables the recording of memory transfers. The information is stored in the `profile.mon` file and is then parsed by the Intel FPGA Dynamic Profiler for OpenCL GUI.

4.2.3. Autorun Captures Tab

To view the autorun statistical data, use the Intel FPGA Dynamic Profiler for OpenCL similar to how you view an enqueued kernel's data. Both autorun and enqueued kernel statistical data are stored in a single `profile.mon` file.

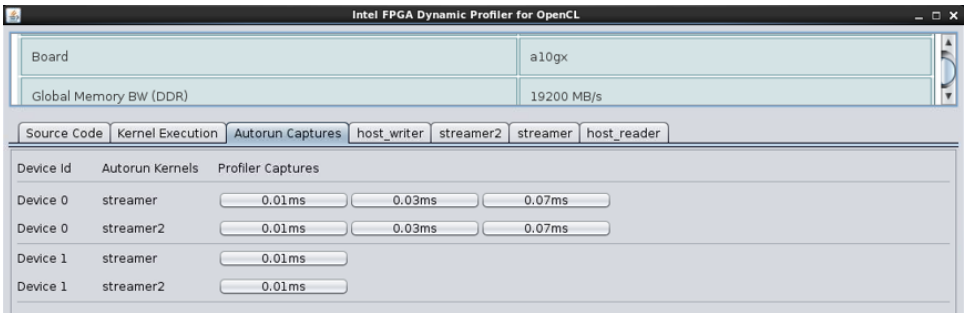
The autorun profile data is displayed similar to the enqueued profile data. However, autorun kernels does not have a runtime representation in the Execution Tab since autorun kernels run continuously.

Attention: If you profile autorun kernels multiple times, data displayed in the Intel FPGA Dynamic Profiler for OpenCL is an average total of all the profile instances.

If you profile autorun kernels at least once, the Autorun Captures tab appears in the Intel FPGA Dynamic Profiler for OpenCL GUI. This tab displays a table of all autorun profile captures organized by device and kernel. To view the profile data of an autorun kernel for a specific capture, select the associated button and a new profiler window opens to display data only from that autorun capture (instead of the overall average).

In the following figure, there are four autorun capture instances. If you want to view the autorun profile data from the capture done at 0.03ms for the `streamer` autorun kernel on device 0, then select the **0.03ms** button in the Device 0 streamer row.

Figure 68. Autorun Captures Tab



The Profiler Captures buttons are labelled with time during which the capture was started. This time is relative to the start of the host program.

Attention: Capture time does not correlate with the timeline found in the Kernel Execution tab, as this timeline is relative to the start of the first enqueued kernel and not the host program.

4.3. Interpreting the Profiling Information

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

Following are explanations on Intel FPGA Dynamic Profiler for OpenCL metrics that are recorded in the Profiler reports.

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

Stall, Occupancy, Bandwidth on page 95

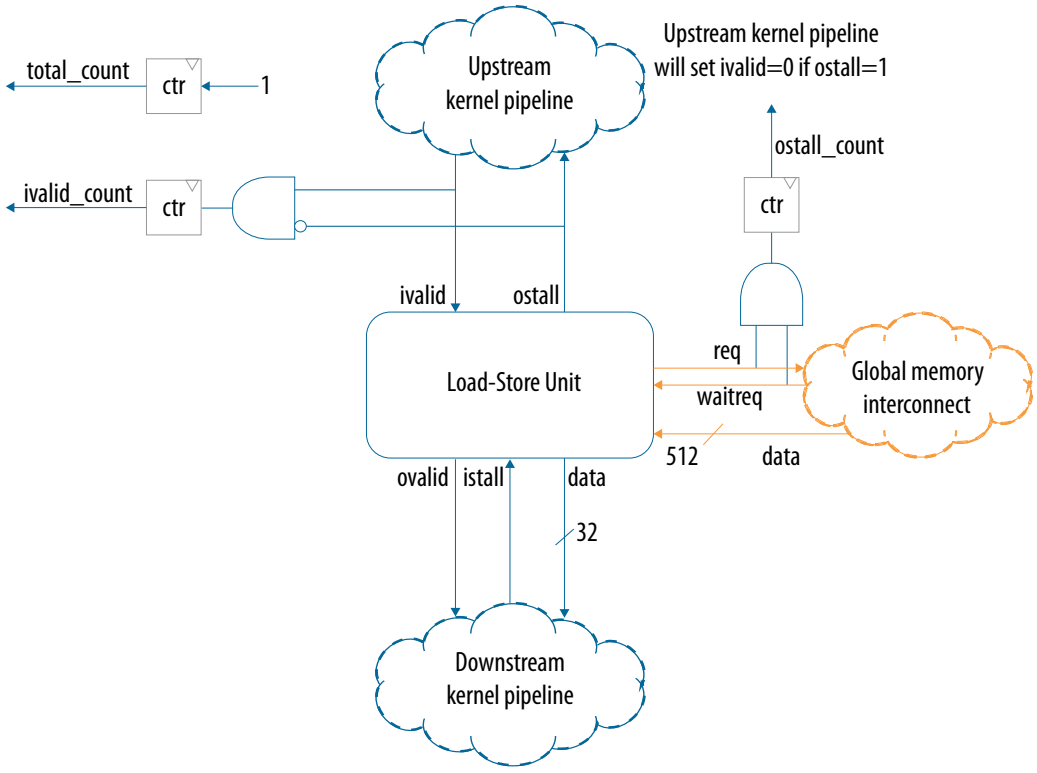
4.3.1. Stall, Occupancy, Bandwidth

For specific lines of kernel code, the **Source Code** tab in the Intel FPGA Dynamic Profiler for OpenCL GUI shows stall percentage, occupancy percentage, and average memory bandwidth.

For definitions of stall, occupancy, and bandwidth, refer to Table 9 on page 91.

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 69. Simplified Representation of a Kernel Pipeline Instrumented with Performance Counters



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

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

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

$$\text{Bandwidth} = \frac{data\_width \times ivalid\_count}{kernel\_time} \times 100\%$$

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 will 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 will have a lower occupancy value than the other LSUs.

The same rule regarding occupancy value applies to channels.

Related Information

Source Code Tab on page 90

4.3.1.1. 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 73

4.3.2. Activity

Activity measures the percentage of time that a predicated instruction is enabled, that is, the percentage of time an LSU receives data that it acts on.

In the **Source Code** tab of the Intel FPGA Dynamic Profiler for OpenCL GUI, the tool tip on the **Occupancy%** column might specify an Activity percentage. Activity differs from occupancy in that activity relates to predication, as explained below.

Each LSU has a predicate signal besides the ivalid signal. The ivalid signal indicates that upstream logic is providing valid data to the LSU. The predicate signal indicates that the LSU should act on the data that it receives. A work-item or loop iteration can occupy a memory instruction even if it is predicated. If the branch statements do not contain loops, the offline compiler converts the branches to minimize control flow, which leads to more efficient hardware. As part of the conversion, memory and channel instructions must be predicated and the output results much be selected through multiplexer logic.

Consider the following code example:

```
int addr = compute_address();
int x = 0;
if (some_rare_condition)
    x = src[addr];
```

The offline compiler will modify the code as follows:

```
int addr = compute_address();
int x = 0;
x = src[addr] if some_rare_condition;
```

In this case, `src[]` receives a valid address every clock cycle. Assuming `src[]` itself does not generate stalls into the pipeline, the `ivalid` signal for `src[]` will be high most of the time. In actuality, `src[]` only performs loading if the predicate signal `some_rare_condition` is true. Therefore, for this load operation, occupancy will be high but activity will be low.

Because activity percentages available in the tool tips do not account for predicated accesses, you can identify predicated instructions based on low activity percentages. Despite having low activity percentages, these instructions might have high occupancies.

Related Information

Tool Tip Options on page 92

4.3.3. Cache Hit

Cache hit rate measures the effectiveness of a private cache.

In the **Source Code** tab of the Intel FPGA Dynamic Profiler for OpenCL GUI, the tool tip on the **Attributes** column might specify a Cache Hit rate. For some global load units, the Intel FPGA SDK for OpenCL Offline Compiler might instantiate a private cache. In this case, the offline compiler creates an additional hardware counter to measure the effectiveness of this cache. Note that details of this private cache is available in the HTML area report.

4.3.4. Profiler Analyses of Example OpenCL Design Scenarios

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

4.3.4.1. 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.

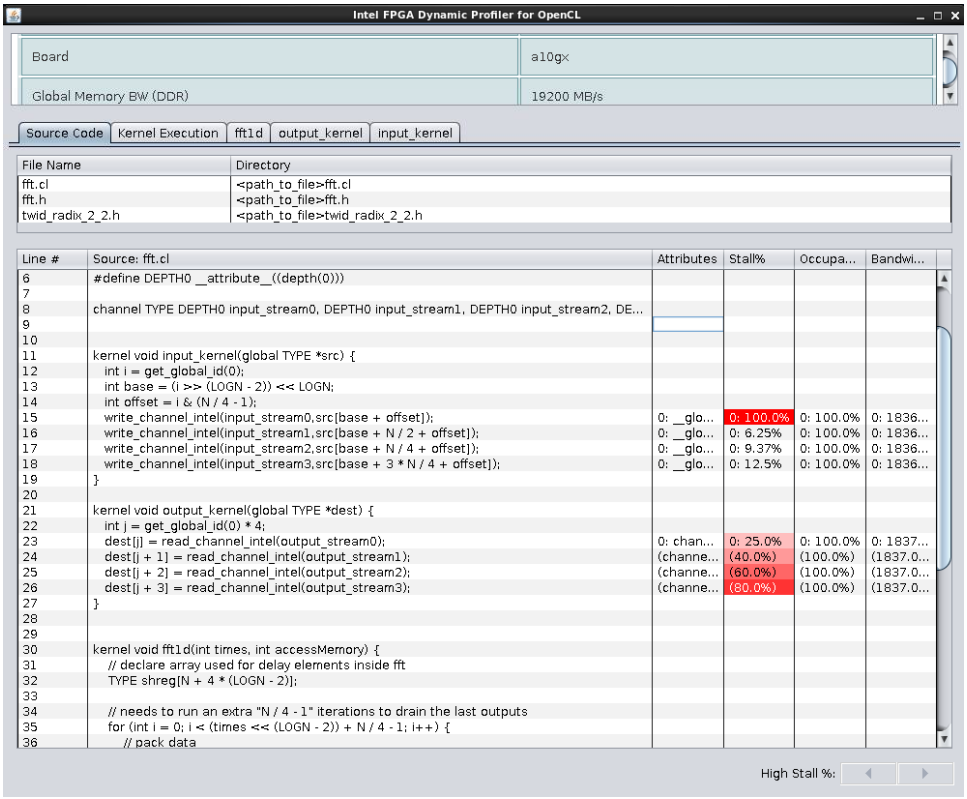
Memory instructions stall often whenever bandwidth usage is inefficient or if a large amount of data transfer is necessary during the execution of your application. Inefficient memory accesses lead to suboptimal bandwidth utilization. In such cases, analyze your kernel memory accesses for possible improvements.

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.

If a memory or channel access is causing high percentage pipeline stalls, the line in the source code that instructs the memory or channel is highlighted in red. A stall percentage of 20% or higher results in a high stall identification. The higher the stall percentage, the darker the red highlight will be. To easily traverse through high stall percentage values, right and left arrows can be found at the bottom right corner of the Source Code tab.

Figure 70. High Stall Percentage Identification



Related Information

- Transferring Data Via Intel FPGA SDK for OpenCL Channels or OpenCL Pipes on page 73
- Source Code Tab on page 90

4.3.4.2. 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 will be 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.

4.3.4.3. Low Bandwidth Efficiency

Low bandwidth efficiency occurs when excessive amount of bandwidth is necessary to obtain useful data. Excessive bandwidth usage generally occurs when memory accesses are poor (for example, random accesses), leading to unsatisfactory coalescing opportunities.

Review your memory accesses to see if you can rewrite them such that accesses to memory sites address consecutive memory regions.

Related Information

- Strategies for Improving Memory Access Efficiency on page 131
- Source Code Tab on page 90

4.3.4.4. 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:

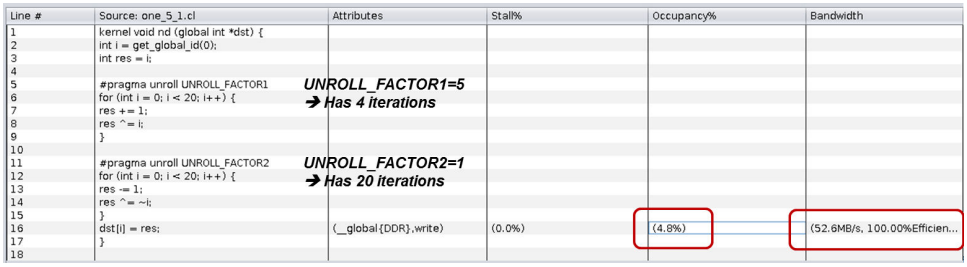
- Fix stalls on the other side of the channel. For example, if channel read stalls, it means that the writer to the channel is not writing data into the channel fast enough and needs to be adjusted.
- If there are channel loops in your design, specify the channel depth.

4.3.4.5. No Stalls, Low Occupancy Percentage, and Low Bandwidth Efficiency

Loop-carried dependencies might create a bottleneck in your design that causes an LSU or channel to have 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 71. Example OpenCL Kernel and Profiler Analysis



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 FACTOR1 loop further will not resolve the bottleneck
- Vectorize your kernel to allow multiple work-items to execute during each loop iteration

Related Information

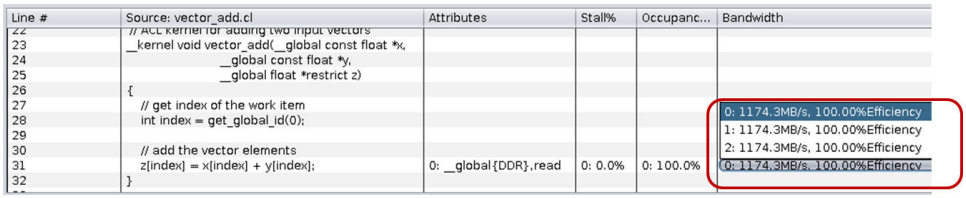
Kernel Vectorization on page 123

4.3.4.6. No Stalls, High Occupancy Percentage, and Low Bandwidth Efficiency

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 72. Example OpenCL Kernel and Profiler Analysis



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 MB/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

4.3.4.7. 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.

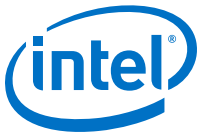
For example, if the kernel that reads is not launched concurrently with the kernel that writes, or if the read operations occur much slower than the write operations, the Profiler identifies a stall for the `write_channel_intel` call in the write kernel.

4.3.4.8. 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.

4.3.5. Autorun Profiler Data

Similar to enqueued kernels, you can view the autorun profiler statistical data by launching the Intel FPGA Dynamic Profiler for OpenCL GUI using the following `aocl` command:`aocl report <filename>.aocx profile.mon <filename>.source`



Related Information

Autorun Captures Tab on page 94

4.4. Intel FPGA Dynamic Profiler for OpenCL Limitations

The Intel FPGA Dynamic Profiler for OpenCL has some limitations.

- The Profiler can only extract one set of profile data from a kernel while it is running.

  If the Profiler collects the profile data after kernel execution completes, you can call the host API to generate the `profile.mon` file multiple times.

  For more information on how to collect profile data during kernel execution, refer to the *Collecting Profile Data During Kernel Execution* section of the *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

- 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 will not save the profile data.

- 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 Standard Edition Programming Guide*.

Related Information

- Collecting Profile Data During Kernel Execution
- Instrumenting the Kernel Pipeline with Performance Counters (-profile)

5. Strategies for Improving Single Work-Item Kernel Performance

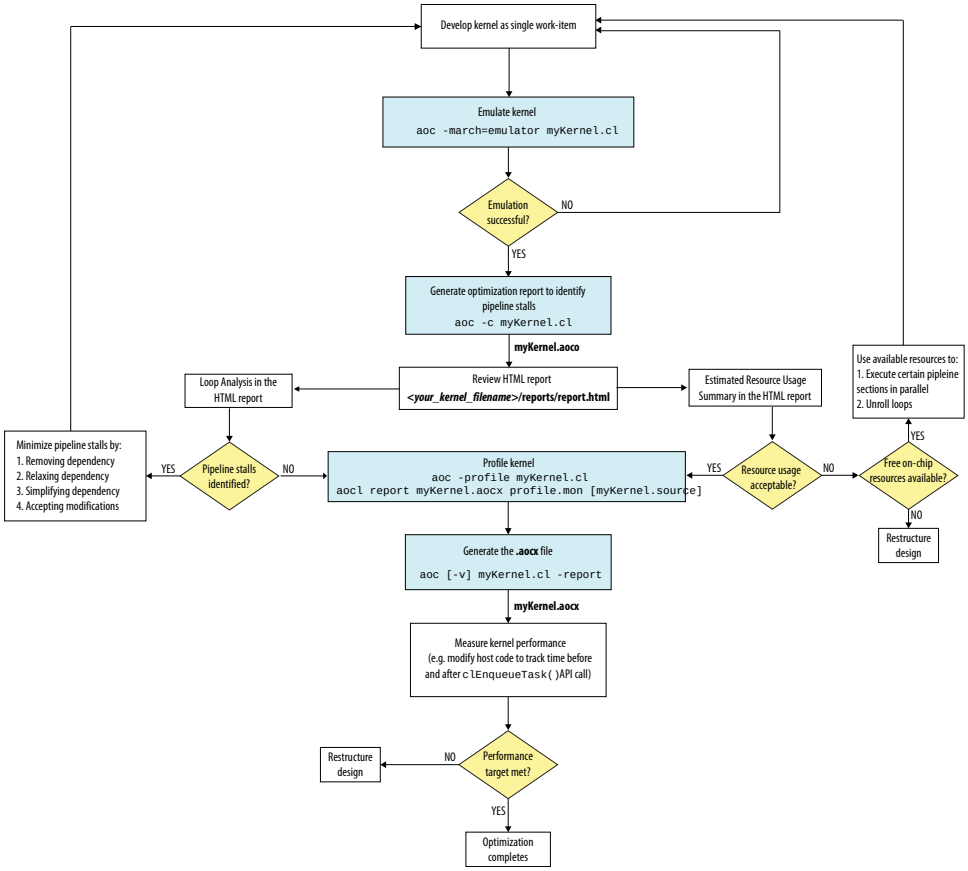
5.1. Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback

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.

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 Standard Edition 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 73. Optimization Work Flow of a Single Work-Item Kernel



1. Removing Loop-Carried Dependency on page 105
2. Relaxing Loop-Carried Dependency on page 108
3. Simplifying Loop-Carried Dependency on page 110
4. Transferring Loop-Carried Dependency to Local Memory on page 113
5. Removing Loop-Carried Dependency by Inferring Shift Registers on page 114

Related Information

- Profiling Your Kernel to Identify Performance Bottlenecks on page 89
- Emulating and Debugging Your OpenCL Kernel
- Profiling Your OpenCL Kernel

5.1.1. 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:

```
 1 #define N 128
 2
 3 __kernel void unoptimized (__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     for (unsigned j = 0; j < N; j++) {
11       sum += A[i*N+j];
12     }
13     sum += B[i];
14   }
15
16   * result = sum;
17 }
```

The optimization report for kernel `unoptimized` resembles the following:

```
================================================================================
Kernel: unoptimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file k.cl line 9)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has iterations
 |     launched at least 2 cycles apart.
 |     Having successive iterations launched every two cycles should still lead to
 |     good performance if the inner loop is pipelined well and has sufficiently
 |     high number of iterations.
 |
 | Iterations executed serially across the region listed below.
 | Only a single loop iteration will execute inside the listed region.
 | This will cause performance degradation unless the region is pipelined well
 | (can process an iteration every cycle).
 |
 |     Loop "Block2" (file k.cl line 10)
 |     due to:
 |     Data dependency on variable sum  (file k.cl line 7)
 |
 |
 |-+ Loop "Block2" (file k.cl line 10)
     Pipelined well. Successive iterations are launched every cycle.
```

- 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 will launch 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`:

```
 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
18     // Step 3: Addition of array B value for an outer loop iteration
19     sum += sum2;
20     sum += B[i];
21   }
22
23   * result = sum;
24 }
```

An optimization report similar to the one below indicates the successful removal of the loop-carried dependency on the variable `sum`:

```
================================================================================
Kernel: optimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file optimized.cl line 9)
 | Pipelined with successive iterations launched every 2 cycles due to:
 |
 |     Pipeline structure: every terminating loop with subloops has iterations
 |     launched at least 2 cycles apart.
 |     Having successive iterations launched every two cycles should still lead to
 |     good performance if the inner loop is pipelined well and has sufficiently
 |     high number of iterations.
 |
 |
 |-+ Loop "Block2" (file optimized.cl line 14)
     Pipelined well. Successive iterations are launched every cycle.


================================================================================
```

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 117

5.1.2. 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:

```
 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 }
```

```
================================================================================
Kernel: unoptimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file unoptimized.cl line 8)
   Pipelined with successive iterations launched every 6 cycles due to:

       Data dependency on variable mul  (file unoptimized.cl line 9)
       Largest Critical Path Contributor:
           100%: Fmul Operation  (file unoptimized.cl line 9)


================================================================================
=
```

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:

```
 1 #define N 128
 2 #define M 8
 3
 4 __kernel void optimized (__global float * restrict A,
 5                           __global float * restrict result)
 6 {
 7   float mul = 1.0f;
 8
 9   // Step 1: Declare multiple copies of variable mul
10   float mul_copies[M];
11
12   // Step 2: Initialize all copies
13   for (unsigned i = 0; i < M; i++)
14     mul_copies[i] = 1.0f;
15
16   for (unsigned i = 0; i < N; i++) {
17     // Step 3: Perform multiplication on the last copy
18     float cur = mul_copies[M-1] * A[i];
19
20     // Step 4a: Shift copies
21     #pragma unroll
22     for (unsigned j = M-1; j > 0; j--)
23       mul_copies[j] = mul_copies[j-1];
24
```

```
25     // Step 4b: Insert updated copy at the beginning
26     mul_copies[0] = cur;
27   }
28
29   // Step 5: Perform reduction on copies
30   #pragma unroll
31   for (unsigned i = 0; i < M; i++)
32     mul *= mul_copies[i];
33
34   * result = mul;
35 }
```

An optimization report similar to the one below indicates the successful relaxation of the loop-carried dependency on the variable *mul*:

```
================================================================================
Kernel: optimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Fully unrolled loop (file optimized2.cl line 13)
   Loop was automatically and fully unrolled.
   Add "#pragma unroll 1" to prevent automatic unrolling.


 + Loop "Block1" (file optimized2.cl line 16)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Fully unrolled loop (file optimized2.cl line 22)
     Loop was fully unrolled due to "#pragma unroll" annotation.


 + Fully unrolled loop (file optimized2.cl line 31)
   Loop was fully unrolled due to "#pragma unroll" annotation.
```

5.1.3. Simplifying Loop-Carried Dependency

In cases where you cannot remove or relax the loop-carried dependency in your kernel, you might be able to simplify the dependency to improve single work-item kernel performance.

Consider the following kernel example:

```
 1 #define N 128
 2 #define NUM_CH 3
 3
 4 channel uchar CH_DATA_IN[NUM_CH];
 5 channel uchar CH_DATA_OUT;
 6
 7 __kernel void unoptimized()
 8 {
 9   unsigned storage = 0;
10   unsigned num_bytes = 0;
11
12   for (unsigned i = 0; i < N; i++) {
13
14     #pragma unroll
15     for (unsigned j = 0; j < NUM_CH; j++) {
16       if (num_bytes < NUM_CH) {
17         bool valid = false;
18         uchar data_in = read_channel_nb_intel(CH_DATA_IN[j], &valid);
19         if (valid) {
```

```
20           storage <<= 8;
21           storage |= data_in;
22           num_bytes++;
23         }
24       }
25     }
26
27     if (num_bytes >= 1) {
28       num_bytes -= 1;
29       uchar data_out = storage >> (num_bytes*8);
30       write_channel_intel(CH_DATA_OUT, data_out);
31     }
32   }
33 }
```

This kernel reads one byte of data from three input channels in a nonblocking fashion. It then writes the data one byte at a time to an output channel. It uses the variable *storage* to store up to 4 bytes of data, and uses the variable *num_bytes* to keep track of how many bytes are stored in *storage*. If *storage* has space available, then the kernel reads a byte of data from one of the channels and stores it in the least significant byte of *storage*.

The optimization report below indicates that there is a loop-carried dependency on the variable *num_bytes*:

```
================================================================================
Kernel: unoptimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file unoptimized3.cl line 12)
 | Pipelined with successive iterations launched every 7 cycles due to:
 |
 |     Data dependency on variable num_bytes  (file unoptimized3.cl line 10)
 |     Largest Critical Path Contributors:
 |      16%: Integer Compare Operation  (file unoptimized3.cl line 16)
 |      16%: Integer Compare Operation  (file unoptimized3.cl line 16)
 |      16%: Integer Compare Operation  (file unoptimized3.cl line 16)
 |       7%: Integer Compare Operation  (file unoptimized3.cl line 27)
 |       6%: Add Operation  (file unoptimized3.cl line 10, line 22, line 28)
 |       6%: Add Operation  (file unoptimized3.cl line 10, line 22, line 28)
 |       6%: Add Operation  (file unoptimized3.cl line 10, line 22, line 28)
 |       3%: Non-Blocking Channel Read Operation  (file unoptimized3.cl line 18)
 |       3%: Non-Blocking Channel Read Operation  (file unoptimized3.cl line 18)
 |       3%: Non-Blocking Channel Read Operation  (file unoptimized3.cl line 18)
 |
 |
 |-+ Fully unrolled loop (file unoptimized3.cl line 15)
     Loop was fully unrolled due to "#pragma unroll" annotation.
```

The computation path of *num_bytes* is as follows:

1. Comparison on line 16 (`if (num_bytes < NUM_CH)`).
2. Computation of variable *valid* by the nonblocking channel read operation on line 18 (`uchar data_in = read_channel_nb_intel(CH_DATA_IN[j], &valid)`) for the comparison on line 19.
3. Addition on line 22 (`num_bytes++`).
4. Comparison on line 27 (`if (num_bytes >= 1)`).
5. Subtraction on line 28 (`num_bytes -= 1`).

Because of the `unroll` pragma on line 14, the Intel FPGA SDK for OpenCL Offline Compiler unrolls the loop, causing the comparisons and additions in the loop body to replicate three times. The optimization report shows that the comparisons are the most expensive operations on the computation path of *num_bytes*, followed by the additions on line 22.

To simplify the loop-carried dependency on *num_bytes*, consider restructuring the application to perform the following tasks:

1. Ensure that the kernel reads from the channels only if there is enough space available in *storage*, in the event that all channel read operations return data (that is, there is at least 3 bytes of empty space in *storage*).
   Setting this condition simplifies the computation path of the variable *num_bytes* by reducing the number of comparisons.
2. Increase the size of *storage* from 4 bytes to 8 bytes to satisfy the 3-byte space threshold more easily.

Below is the restructured kernel `optimized`:

```
 1 #define N 128
 2 #define NUM_CH 3
 3
 4 channel uchar CH_DATA_IN[NUM_CH];
 5 channel uchar CH_DATA_OUT;
 6
 7 __kernel void optimized()
 8 {
 9   // Change storage to 64 bits
10   ulong storage = 0;
11   unsigned num_bytes = 0;
12
13   for (unsigned i = 0; i < N; i++) {
14
15     // Ensure that we have enough space if we read from ALL channels
16     if (num_bytes <= (8-NUM_CH)) {
17       #pragma unroll
18       for (unsigned j = 0; j < NUM_CH; j++) {
19         bool valid = false;
20         uchar data_in = read_channel_nb_intel(CH_DATA_IN[j], &valid);
21         if (valid) {
22           storage <<= 8;
23           storage |= data_in;
24           num_bytes++;
25         }
26       }
27     }
28
29     if (num_bytes >= 1) {
30       num_bytes -= 1;
31       uchar data_out = storage >> (num_bytes*8);
32       write_channel_intel(CH_DATA_OUT, data_out);
33     }
34   }
35 }
```

An optimization report similar to the one below indicates the successful simplification of the loop-carried dependency on the variable *num_bytes*:

```
================================================================================
Kernel: optimized
================================================================================
The kernel is compiled for single work-item execution.
```

```
Loop Report:

 + Loop "Block1" (file optimized3.cl line 13)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Fully unrolled loop (file optimized3.cl line 18)
     Loop was fully unrolled due to "#pragma unroll" annotation.
```

5.1.4. 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:

```
1 #define N 128
2
3 __kernel void unoptimized( __global int* restrict A )
4 {
5     for (unsigned i = 0; i < N; i++)
6           A[N-i] = A[i];
7 }
```

```
================================================================================
Kernel: unoptimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file unoptimized4.cl line 5)
   Pipelined with successive iterations launched every 324 cycles due to:

       Memory dependency on Load Operation from: (file unoptimized4.cl line 6)
         Store Operation (file unoptimized4.cl line 6)
       Largest Critical Path Contributors:
           49%: Load Operation  (file unoptimized4.cl line 6)
           49%: Store Operation  (file unoptimized4.cl line 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 324 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 324 to 2:

```
================================================================================
Kernel: optimized
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file optimized4.cl line 7)
   Pipelined well. Successive iterations are launched every cycle.


 + Loop "Block2" (file optimized4.cl line 10)
   Pipelined with successive iterations launched every 2 cycles due to:

       Memory dependency on Load Operation from: (file optimized4.cl line 11)
         Store Operation (file optimized4.cl line 11)
       Largest Critical Path Contributors:
           65%: Load Operation  (file optimized4.cl line 11)
           34%: Store Operation  (file optimized4.cl line 11)


 + Loop "Block3" (file optimized4.cl line 13)
   Pipelined well. Successive iterations are launched every cycle.
```

5.1.5. Removing 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:

```
 1 __kernel void double_add_1 (__global double *arr,
 2                             int N,
 3                             __global double *result)
 4 {
 5   double temp_sum = 0;
 6
 7   for (int i = 0; i < N; ++i)
 8   {
 9       temp_sum += arr[i];
10   }
11
12   *result = temp_sum;
13 }
```

The optimization report for kernel `unoptimized` resembles the following:

```
================================================================================
Kernel: double_add_1
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Loop "Block1" (file unoptimized5.cl line 7)
   Pipelined with successive iterations launched every 11 cycles due to:

       Data dependency on variable temp_sum  (file unoptimized5.cl line 9)
       Largest Critical Path Contributor:
           97%: Fadd Operation  (file unoptimized5.cl line 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 remove the data dependency, infer the array `arr[i]` as a shift register.

Below is the restructured kernel `optimized`:

```
 1 //Shift register size must be statically determinable
 2 #define II_CYCLES 12
 3
 4 __kernel void double_add_2 (__global double *arr,
 5                             int N,
 6                             __global double *result)
 7 {
 8     //Create shift register with II_CYCLE+1 elements
 9     double shift_reg[II_CYCLES+1];
10
11     //Initialize all elements of the register to 0
12     for (int i = 0; i < II_CYCLES + 1; i++)
13     {
14         shift_reg[i] = 0;
15     }
16
17     //Iterate through every element of input array
18     for(int i = 0; i < N; ++i)
19     {
20         //Load ith element into end of shift register
21         //if N > II_CYCLE, add to shift_reg[0] to preserve values
22         shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
23
24         #pragma unroll
25         //Shift every element of shift register
26         for(int j = 0; j < II_CYCLES; ++j)
27         {
28             shift_reg[j] = shift_reg[j + 1];
29         }
30     }
31
32     //Sum every element of shift register
33     double temp_sum = 0;
34
35     #pragma unroll
36     for(int i = 0; i < II_CYCLES; ++i)
37     {
38         temp_sum += shift_reg[i];
39     }
```

```
40
41     *result = temp_sum;
42 }
```

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*:

```
================================================================================
Kernel: double_add_2
================================================================================
The kernel is compiled for single work-item execution.

Loop Report:

 + Fully unrolled loop (file optimized5.cl line 12)
   Loop was automatically and fully unrolled.
   Add "#pragma unroll 1" to prevent automatic unrolling.


 + Loop "Block1" (file optimized5.cl line 18)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Fully unrolled loop (file optimized5.cl line 26)
     Loop was fully unrolled due to "#pragma unroll" annotation.


 + Fully unrolled loop (file optimized5.cl line 36)
   Loop was fully unrolled due to "#pragma unroll" annotation.
```

5.2. 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 will 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.

- If all accesses to memory arrays that are inside a loop will 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 will 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]];
}
```

5.3. 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:

Nested Loop	Converted Single Loop

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

Nested Loop	Converted Single Loop

`    }`
`    //statements`
`}`	

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.

6. 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 121
2. Kernel Vectorization on page 123
3. Multiple Compute Units on page 126
4. Combination of Compute Unit Replication and Kernel SIMD Vectorization on page 129
5. Reviewing Kernel Properties and Loop Unroll Status in the HTML Report on page 130

6.1. 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.

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];
}
```

6.2. 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.

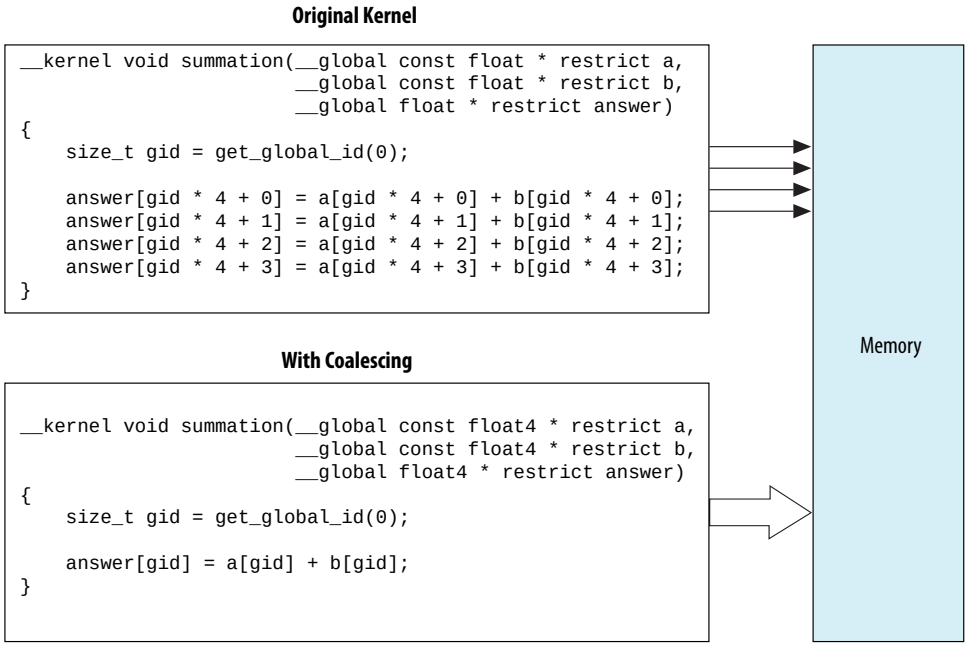
Attention: Each work-item handles four times as much work after you implement the manual optimizations. As a result, the host application must use an NDRange that is four times smaller than in the original example. On the contrary, you do not need to adjust the NDRange size when you exploit the automatic vectorization capabilities of the offline compiler. You can adjust the vector width with minimal code changes by using the `num_simd_work_items` attribute.

6.2.1. 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 74. Static Memory Coalescing



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`.

6.3. 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 as long as it has not reached its full capacity.

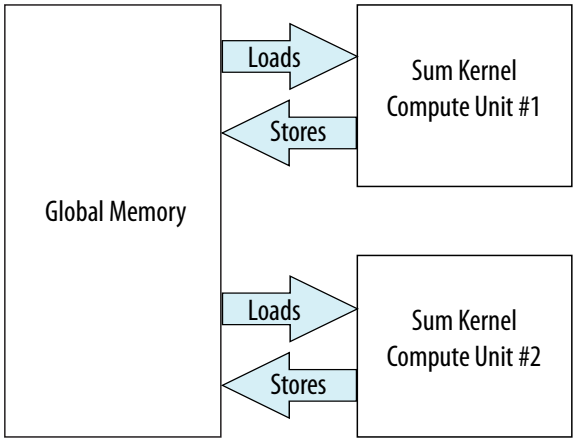
Assume each work-group takes the same amount of time to complete its execution. If the offline compiler implements two compute units, each compute unit executes half of the work-groups. Because the hardware scheduler dispatches the work-groups, you do not need to manage this process in your own code.

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 75. Data Flow with Multiple Compute Units



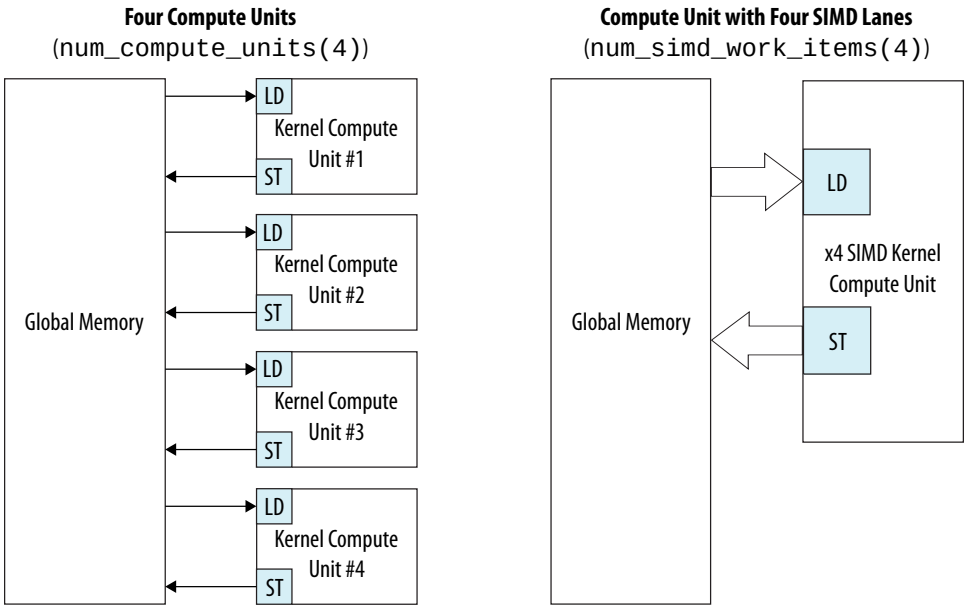
6.3.1. Compute Unit Replication versus Kernel SIMD Vectorization

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 76. 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.

6.4. 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.

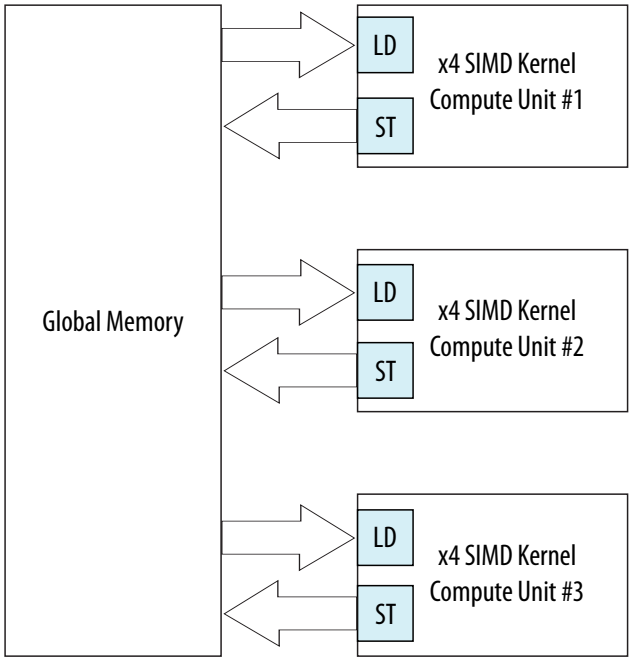
Consider a case where a kernel with a `num_simd_work_items` attribute set to 16 does not fit in the FPGA. The kernel might fit if you modify it by duplicating a narrower SIMD kernel compute unit. Determining the optimal balance between the number of compute units and the SIMD width might require some 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 77. Optimizing Throughput by Combining Compute Unit Replication and Kernel SIMD Vectorization



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.

6.5. 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 17

7. 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.

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 131
2. Optimize Global Memory Accesses on page 132
3. Performing Kernel Computations Using Constant, Local or Private Memory on page 135
4. Improving Kernel Performance by Banking the Local Memory on page 138
5. Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor on page 143
6. Minimizing the Memory Dependencies for Loop Pipelining on page 144

7.1. 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.

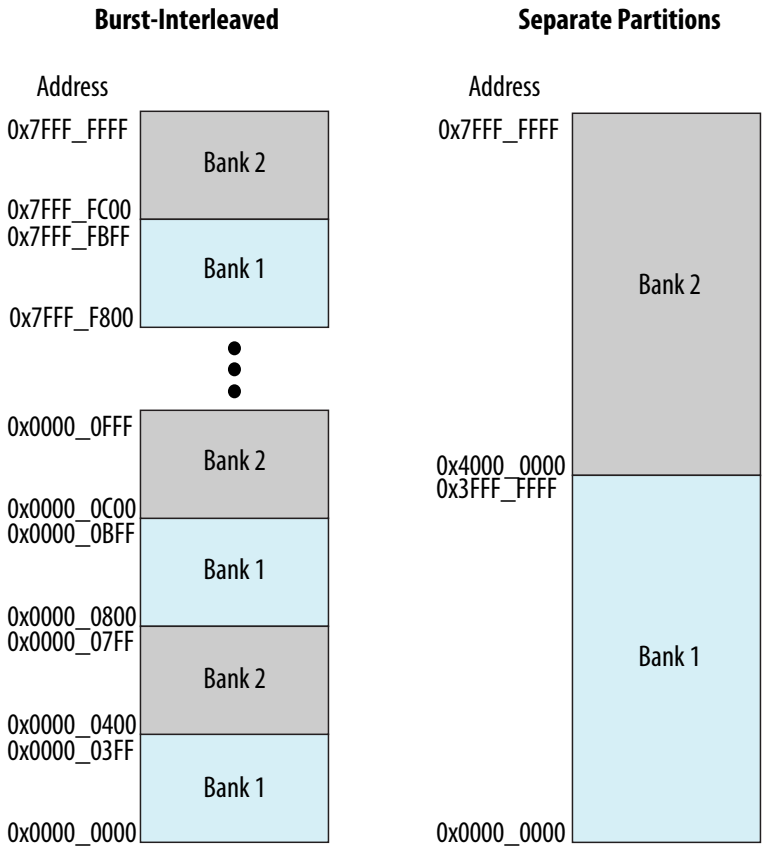
7.2. Optimize Global Memory Accesses

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 78. Global Memory Partitions



7.2.1. 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:

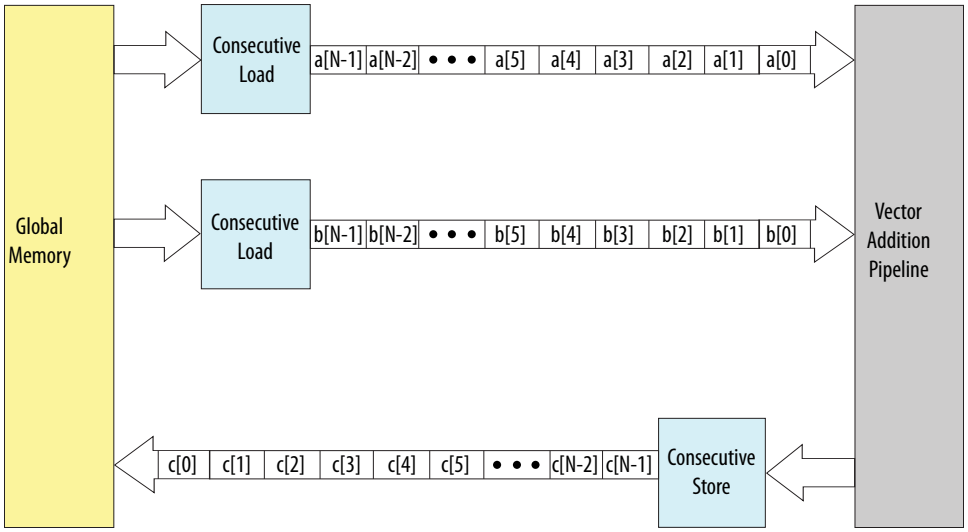
```
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict c )
{
    size_t gid = get_global_id(0);

    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 79. Contiguous Memory Access



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.

7.2.2. 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 on 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 Standard Edition Programming Guide*.

Related Information

Disabling Burst-Interleaving of Global Memory (-no-interleaving=<global_memory_type>)

7.2.2.1. 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 on how to allocate buffers in global memory and how to modify your host application to leverage 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 Standard Edition Programming Guide*.

Related Information

- Partitioning Buffers Across Different Memory Types (Heterogeneous Memory)
- Allocating OpenCL Buffer for Manual Partitioning of Global Memory

7.3. 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.

7.3.1. 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 on 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 Standard Edition Programming Guide*.

Related Information

Configuring Constant Memory Cache Size (-const-cache-bytes=<N>)

7.3.2. 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.

- For pointers to `__local` kernel arguments, the host assigns their memory sizes dynamically at runtime through `clSetKernelArg` calls. However, the offline compiler must set these physical memory sizes at compilation time.

  By default, pointers to `__local` kernel arguments are 16 kB in size. You can specify an allocation size by including the `local_mem_size` attribute in your pointer declaration.

  For usage information on the `local_mem_size` attribute, refer to the *Specifying Pointer Size in Local Memory* section of the *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

  Note: `clSetKernelArg` calls can request a smaller data size than has been physically allocated at compilation time, but never a larger size.

- 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.

Related Information

Programming Strategies for Optimizing Pointer-to-Local Memory Size

7.3.3. 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. The offline compiler promotes most scalar variablessuch as `float`, `int`, and `char`. It also promotes aggregate data types if accesses are constants at compilation time. 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 on the implementation of private memory using registers, refer to the *Inferring a Register* section of the *Intel FPGA SDK for OpenCL Standard Edition Programming Guide*.

Related Information

Inferring a Register

7.4. Improving Kernel Performance by Banking the Local Memory

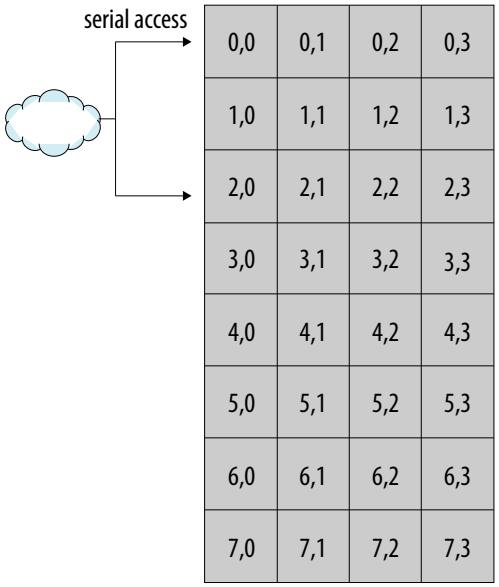
Specifying the `numbanks(N)` and `bankwidth(M)` advanced kernel attributes allows you to configure the local memory banks for parallel memory accesses.The banking geometry described by these advanced kernel attributes determines which elements of the local memory system your kernel can access in parallel.

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 80. Serial Accesses to an 8 x 4 Local Memory System

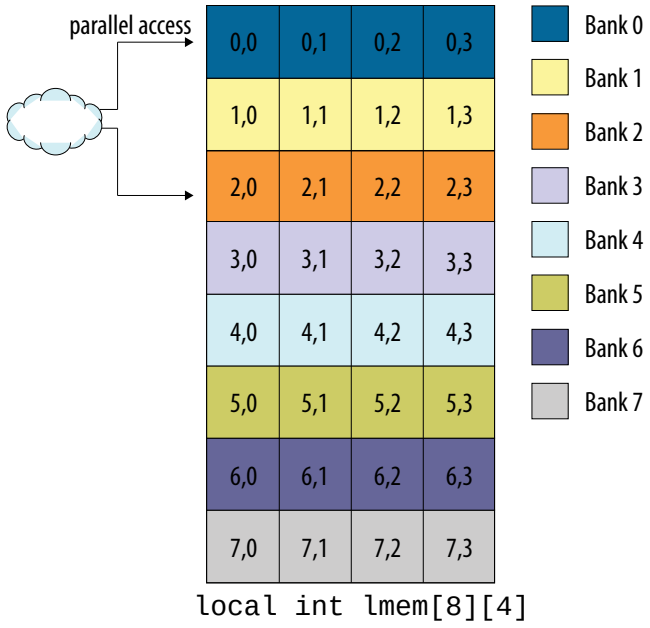


To improve performance, you can add `numbanks(`*N*`)` and `bankwidth(`*M*`)` in your code to define the number of memory banks and the bank widths in bytes. The following code implements eight memory banks, each 16-bytes wide. This memory bank configuration enables parallel memory accesses down the 8 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 will not exceed the lower index bounds.

Figure 81. 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 82. Parallel Access to an 8 x 4 Local Memory System with Four 4-Byte-Wide Memory Banks

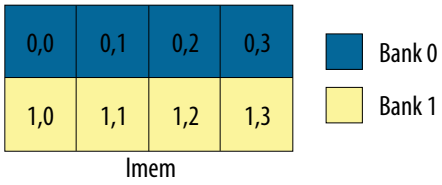
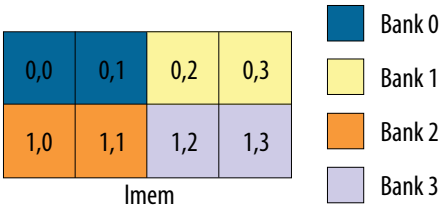
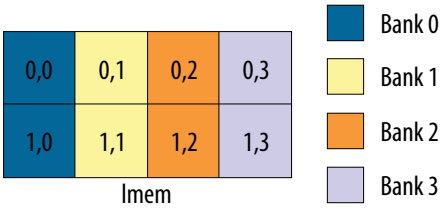
7.4.1. Optimizing the Geometric Configuration of Local Memory Banks Based on Array Index

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 11. 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];`	
`local int __attribute__((numbanks(2), bankwidth(8))) lmem[2][4];`	
`local int __attribute__((numbanks(2), bankwidth(4))) lmem[2][4];`	
`local int __attribute__((numbanks(4), bankwidth(8))) lmem[2][4];`	
`local int __attribute__((numbanks(4), bankwidth(4))) lmem[2][4];`	

Related Information

Kernel Attributes for Configuring Local and Private Memory Systems

7.5. Optimizing Accesses to Local Memory by Controlling the Memory Replication Factor

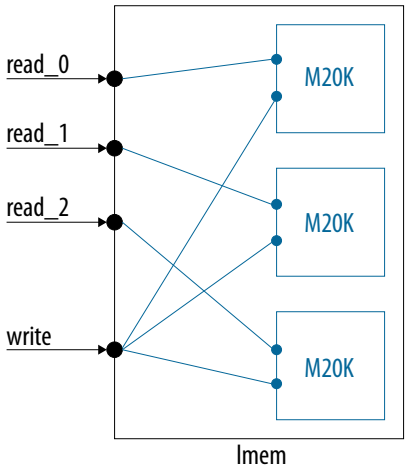
The memory replication factor is the number of M20K memory blocks that your design uses to implement the local memory system. To control the memory replication factor, include the `singlepump` or `doublepump` kernel attribute in your OpenCL kernel. The `singlepump` and `doublepump` kernel attributes are part Intel FPGA SDK for OpenCL's advanced features.

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 an example design where the kernel specifies three read ports and one write port for the local memory system, `lmem`. As shown in the code example below, including the `singlepump` kernel attribute in the local variable declaration indicates that the M20K blocks will run at the same frequency as the rest of the design.

```
int __attribute__((memory,
                   numbanks(1),
                   bankwidth(64),
                   singlepump,
                   numreadports(3),
                   numwriteports(1)))
                   lmem[16];
```

Figure 83. Accesses to Single-Pumped M20K Memory Blocks

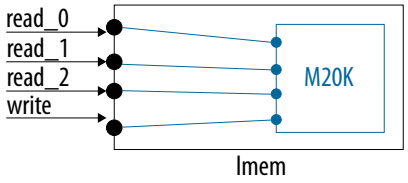


Each single-pumped M20K block will have two *logical* ports available. Each write port in the local memory system must be connected to all the M20K blocks that your design uses to implement the memory system. Each read port in the local memory system must be connected to one M20K block. Because of these connection constraints, there needs to be three M20K blocks to implement the specified number of ports in `lmem`.

If you include the `doublepump` kernel attribute in your local variable declaration, you specify that the M20K memory blocks will run at double the frequency as the rest of the design.

```
int __attribute__((memory,
                   numbanks(1),
                   bankwidth(64),
                   doublepump,
                   numreadports(3),
                   numwriteports(1)))
                   lmem[16];
```

Figure 84. Accesses to Double-Pumped M20K Memory Blocks



Each double-pumped M20K block will have four logical ports available. As such, there only needs to be one M20K block to implement all 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 or improves performance, or both.
- Stores must be connected to every replicate and must not suffer contention. 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

7.6. 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 could 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 dependences.

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 on 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 on floating-point operations, refer to *Optimize Floating-Point Operations*.

Related Information

- Optimizing Floating-Point Operations on page 80
- HTML Report: Area Report Messages on page 42
- Displaying the Estimated Resource Usage Summary On-Screen (-report)
- Relaxing the Order of Floating-Point Operations (-fp-relaxed)
- Reducing Floating-Point Rounding Operations (-fpc)

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 Standard 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. On the other hand, 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.

A. Additional Information

For additional information, demonstrations and training options, visit the Intel FPGA SDK for OpenCL product page.

Related Information

Intel FPGA SDK for OpenCL product page

A.1. Document Revision History for the Intel FPGA SDK for OpenCL Standard Edition Best Practices Guide

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1	• Maintenance release
2018.05.04	18.0	• Removed Intel FPGA SDK for OpenCL Pro Edition information. • In Preloading Data to Local Memory, added recommendation to create local array elements to be a power of 2 bytes. • Removed the topic *Resource-Driven Optimization* because it described an obsolete optimization behavior.

Table 12. 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 on page 94 — Autorun Profiler Data on page 102
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 95. • Added a new image to show comparison between parallel threads and loop pipelining, along with explanation to Multi-Threaded Host Application on page 15. • Added an FPGA architecture along with some explanation in FPGA Overview on page 5. • Added OpenCL Design Components image to HTML Report: Kernel Design Concepts on page 44. • Added an important note to Aligning a Struct with or without Padding on page 84 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 143. • Added Minimizing the Memory Dependencies for Loop Pipelining on page 144. • Added area report hierarchy details to Reviewing Area Information on page 28. • Added Best Practices for Channels and Pipes on page 78.

continued...

Date	Version	Changes
		• Updated Allocating Aligned Memory on page 83. • Added Reducing the Area Consumed by Nested Loops Using loop_coalesce on page 27. • Added Changing the Memory Access Pattern Example on page 23. • Updated the image Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 104. • In the following topics, implemented single dash and `-option=<value>` conventions for `aoc` command. — Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 104 — Optimizing Floating-Point Operations on page 80 — Manual Partitioning of Global Memory on page 134 — Constant Cache Memory on page 136 — Compilation Considerations on page 146 — High Stall and High Occupancy Percentages on page 100 • In Source Code Tab on page 90 and Tool Tip Options on page 92, updated the images to reflect Intel. • In High Stall Percentage on page 98, added a screenshot for high stall percentage identification along with relevant explanation. • In Local Memory on page 47, added a sentence about the overall state of the local memory as observed in the HTML report. • In Load-Store Units on page 68, 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 54. • Updated the code fragment in Pipelines on page 7 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.
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 68. • Added Reviewing the Report Summary on page 19. • Added Features of the Kernel Memory Viewer on page 33. • Revised the *Local Memory Banks* section of Local Memory on page 47 to include information about the `bank_bits` attribute. • Revised flowchart in Addressing Single Work-Item Kernel Dependencies Based on Optimization Report Feedback on page 104 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.

continued...

Date	Version	Changes
		• 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. • 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*.

continued...

Date	Version	Changes
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 `-03` 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.
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 compilerL 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*.

continued...

Date	Version	Changes
		• 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*. • 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*.

continued...

Date	Version	Changes
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.