FPGA Optimization Guide for Intel® oneAPI Toolkits

User Guide

Rev. 12

April 2022

Order No.: fpgaoptguide-12

Contents

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Product and Performance Information
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex. Notice revision #20201201

Unless stated otherwise, the code examples in this document are provided to you under an MIT license, the terms of which are as follows:

1.0 Introduction To FPGA Design Concepts

The FPGA Optimization Guide for Intel® oneAPI Toolkits provides guidance on leveraging the functionalities of SYCL* to optimize your design.

This document assumes that you are familiar with SYCL* concepts and application programming interfaces (APIs), as described in the SYCL* Specification version 1.2.1 by the Khronos* Group. It also assumes that you have experience in creating SYCL* applications.

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

For more information, refer to the Intel® oneAPI Programming Guide and SYCL* Specification version 1.2.1 (https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf).

1.1 FPGA Architecture Overview

A field-programmable gate array (FPGA) is a reconfigurable semiconductor integrated circuit (IC).

FPGAs occupy a unique computational niche relative to other computing devices, such as central and graphics processing units (CPUs and GPUs), and custom accelerators, such as application-specific integrated circuits (ASICs). CPUs and GPUs have a fixed hardware structure to which a program maps. Conversely, ASICs and FPGAs can build custom hardware to implement a program.

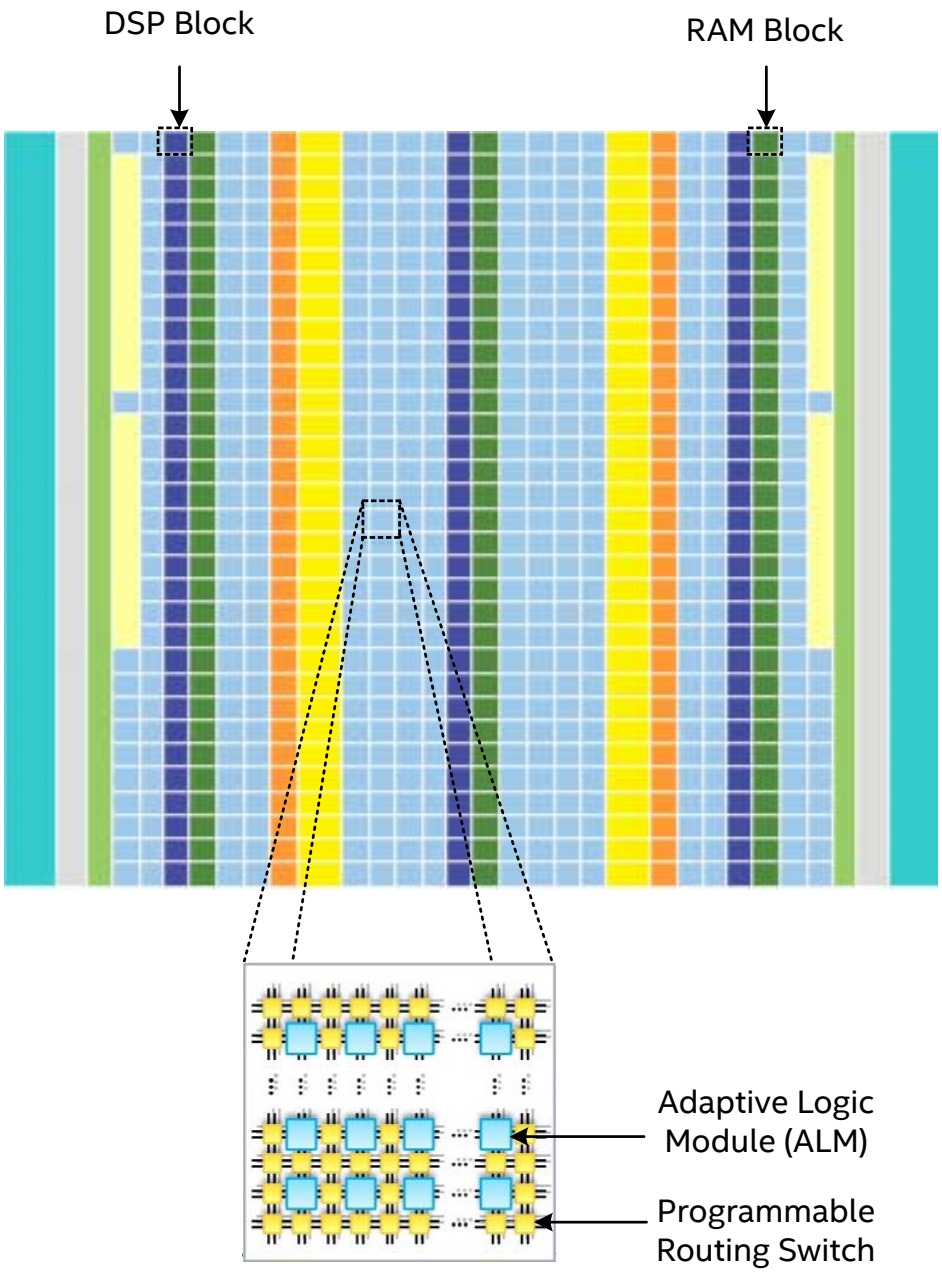
While a custom ASIC generally outperforms an FPGA on a specific task, they take significant time and money to develop. However, FPGAs are a cheaper off-the-shelf alternative that you can reprogram for each new application.

An FPGA is made up of a grid of configurable logic, known as adaptive logic modules (ALMs), and specialized blocks, such as digital signal processing (DSP) blocks and random-access memory (RAM) blocks. These programmable blocks are combined via configurable routing interconnects to implement complete digital circuits.

The total number of ALMs, DSP blocks, and RAM blocks used by a design is often referred to as the *FPGA area* or *area* that the design uses.

The following image illustrates a high-level architectural view of an FPGA:

Figure 1. FPGA Architecture



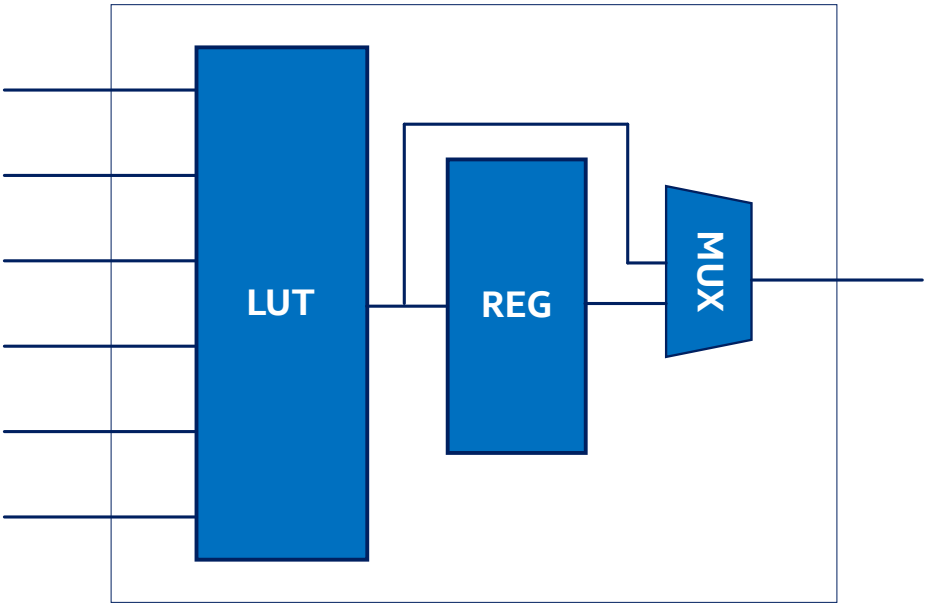
1.1.1 Adaptive Logic Module (ALM)

The basic building block in an FPGA is an adaptive logic module (ALM).

A simplified ALM consists of a lookup table (LUT) and an output register from which the compiler can build any arbitrary Boolean logic circuit.

The following figure illustrates a simplified ALM:

Figure 2. Adaptive Logic Module



1.1.2 Lookup Table (LUT)

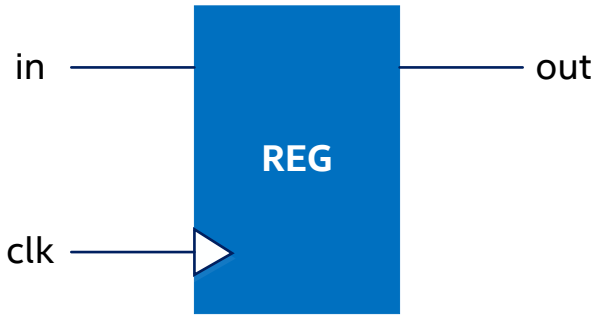
A *lookup table (LUT)* that implements an arbitrary Boolean function of N inputs is often referred to as an *N-LUT*.

1.1.3 Register

A register is the most basic storage element in an FPGA. It has an input (`in`), an output (`out`), and a clock signal (`clk`). It is synchronous, that is, it synchronizes output changes to a clock. In an ALM, a register may store the output of the LUT.

The following figure illustrates a register:

Figure 3. Register

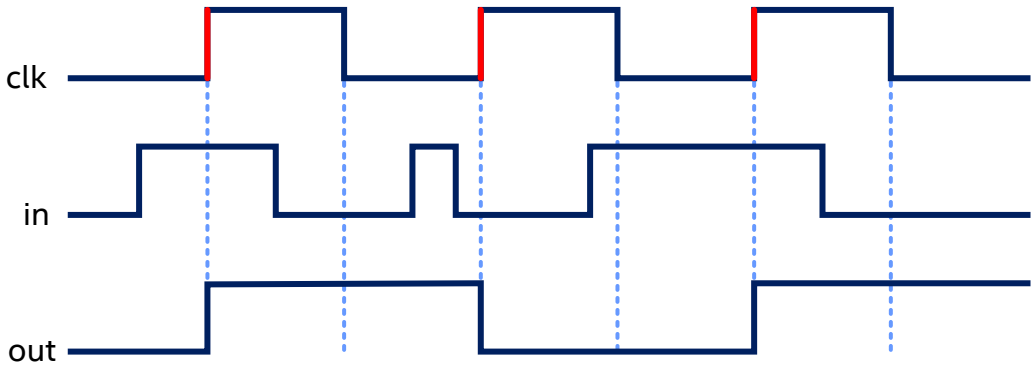


NOTE

The clock signal is implied and not shown in some figures.

The following figure illustrates the waveform of register signals:

Figure 4. Waveform of Register Signals



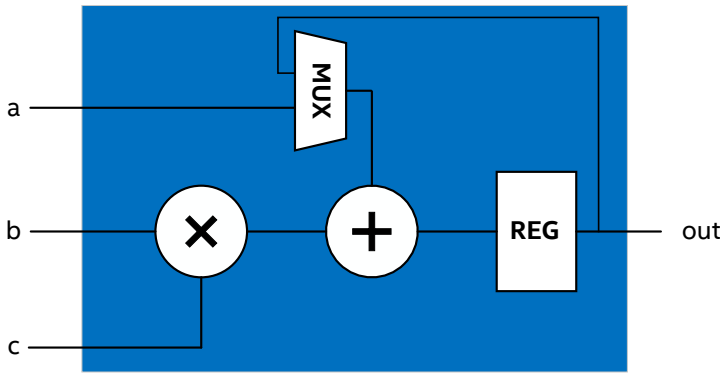
The input data propagates to the output on every clock cycle. The output remains unchanged between clock cycles.

1.1.4 Digital Signal Processing (DSP) Block

A digital signal processing (DSP) block implements specific support for common fixed-point and floating-point arithmetic, which reduces the need to build equivalent logic from general-purpose ALMs.

The following figure illustrates a simplified three-input DSP block consisting of a multiplier (×) and an adder (+):

Figure 5. DSP Block



1.1.5 Random Access Memory (RAM) Blocks

A random access memory (RAM) block implements memory by using a high density of memory cells.

For more information, refer to Memory Types.

1.2 Concepts of FPGA Hardware Design

This section describes the following FPGA hardware design concepts:

- Maximum Frequency (f_{MAX}) on page 11
- Latency on page 12
- Pipelining on page 12
- Throughput on page 13
- Datapath on page 13
- Control Path on page 13
- Occupancy on page 14

1.2.1 Maximum Frequency (f_{MAX})

The maximum clock frequency at which a digital circuit can operate is called its f_{MAX}. This is the maximum rate at which the outputs of registers are updated.

The physical propagation delay of the signal across combinational logic between two consecutive register stages limits the clock speed. This propagation delay is a function of the complexity of the combinational logic in the path. The path with the most combinational logic elements (and the highest delay) limits the speed of the entire circuit. This speed-limiting path is often referred to as the *critical path*.

The f_{MAX} is calculated as the inverse of the critical path delay. You may want to have high f_{MAX} since it results in high performance in the absence of other bottlenecks.

1.2.2 Latency

Latency is the measure of how long it takes to complete one or more operations in a digital circuit. You can measure latency at different granularities. For example, you can measure the latency of a single operation or the latency of the entire circuit.

You can measure latency in time (for example, microseconds) or clock cycles. Typically, clock cycles are the preferred way to express latency because measuring latency in clock cycles disconnects latency from your circuit clock frequency. By expressing latency independent of circuit clock frequency, it is easier to discern the true impact of circuit changes to the circuit's performance.

For more information and an example, refer to Pipelining on page 12.

1.2.3 Pipelining

Pipelining is a design technique used in synchronous digital circuits to increase f_{MAX}. Pipelining involves adding registers to the critical path, which decreases the amount of logic between each register. Less logic takes less time to execute, which enables an increase in f_{MAX}.

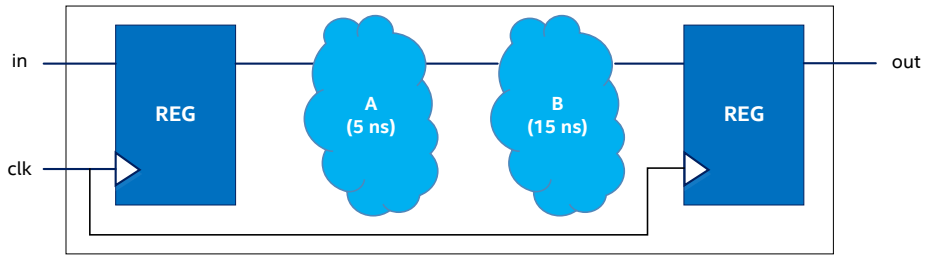
The critical path in a circuit is the path between any two consecutive registers with the highest latency. That is, the path between two consecutive registers where the operations take the longest to complete.

Pipelining is especially useful when processing a stream of data. A pipelined circuit can have different stages of the pipeline operating on different input stream data in the same clock cycle, which leads to better data processing throughput.

Example

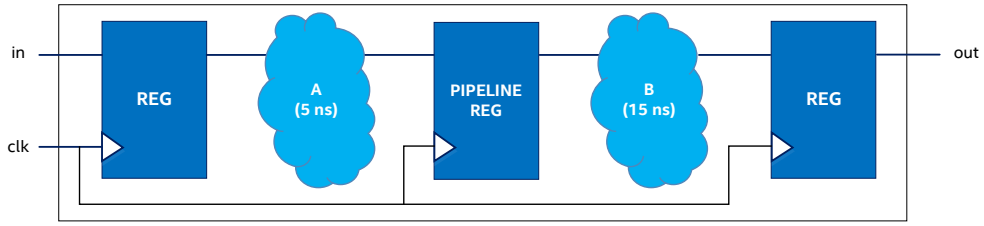
Consider a simple circuit with operations A and B on the critical path. If operation A takes 5 ns to complete and operation B takes 15ns to complete, then the time delay on the critical path is 20 ns. This results in an f_{MAX} of 50 MHz (1/`max_delay`).

Figure 6. Unpipelined Logic Block with the f_{MAX} of 50 MHz and Latency of Two Clock Cycles



If a pipeline register is added between A and B, the critical path changes. The delay on the critical path is now 15ns. Pipelining this block results in an f_{MAX} of 66.67 MHz, and the maximum delay between two consecutive registers is 15 ns.

Figure 7. Pipelined Logic Block with an f_{MAX} of 66.67 MHz and Latency of Three clock cycles



While pipelining generally results in a higher f_{MAX}, it increases latency. In the previous example, the latency of the block containing A and B increases from two to three clock cycles after pipelining.

1.2.4 Throughput

Throughput of a digital circuit is the rate at which data is processed. In the absence of other bottlenecks, higher f_{MAX} results in higher throughput (for example, samples/second).

Throughput is a good measure of the performance of a circuit, and throughput and performance are often used interchangeably when discussing a circuit.

1.2.5 Datapath

A datapath is a chain of registers and combinational logic in a digital circuit that performs computations.

For example, the datapath in Pipelined Logic Block with an f_{MAX} of 100 MHz and Latency of Three clock cycles consists of all elements shown, from the input register to the last output register.

In contrast, memory blocks are outside the datapath and reads and writes to memory are also considered to be outside of the datapath.

1.2.6 Control Path

While the datapath is the path on which computations occur, the control path is the path of signals that control the datapath circuitry.

The control path is the logic added by the compiler to manage the flow of data through your design. Control paths include controls such as the following:

Control	Description
Handshaking flow control	Handshaking ensures that one part of your design is ready and able to accept data from another part of your design.
Loop control	Loop controls control the data flow through the hardware generated for loops in your code, including any loop carried dependencies.
Branch control	Branch controls implement conditional statements in your code. Branch control can include parallelizing parts of conditional statements to improve performance.

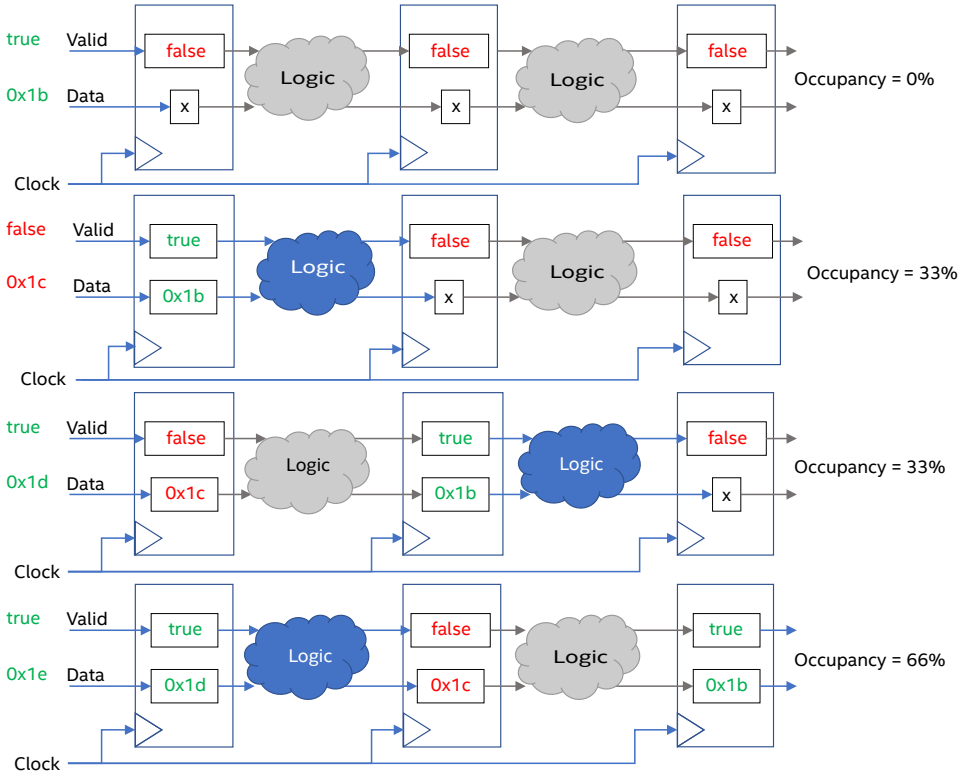
1.2.7 Occupancy

The occupancy of a datapath at a point in time refers to the proportion of the datapath that contains valid data. The occupancy of a circuit over the execution of a program is the average occupancy over time from the moment the program starts to run until it has been completed.

Unoccupied portions of the datapath are often referred to as *bubbles*. Bubbles are analogous to no-operation (no-ops) instructions for a CPU that have no effect on the final output.

Decreasing bubbles increase occupancy. In the absence of other bottlenecks, maximizing occupancy of the datapath results in higher throughput.

Figure 8. A Datapath Through Four Iterations Showing A Bubble Traveling Through



1.3 Methods of Hardware Design

Traditionally, you program an FPGA using a hardware description language (HDL) such as Verilog or VHDL. However, a recent trend is to use higher-level languages. Higher levels of abstraction can reduce the design time and increase the portability of the resulting design.

The rest of this chapter discusses how the compiler maps high-level languages to a hardware datapath.

1.4 How Source Code Becomes a Custom Hardware Datapath

Based on your source code and the following principles, the Intel® oneAPI DPC++/C++ Compiler builds a custom hardware datapath in the FPGA's logic:

- The datapath is functionally equivalent to the C++ program described by the source. Once the datapath is constructed, the compiler orchestrates work items and loop iterations such that the hardware is effectively occupied.
- The compiler builds the custom hardware datapath while minimizing the area of the FPGA resources (ALMs, DSPs, and so on) used by the design.

1.4.1 Mapping Source Code Instructions to Hardware

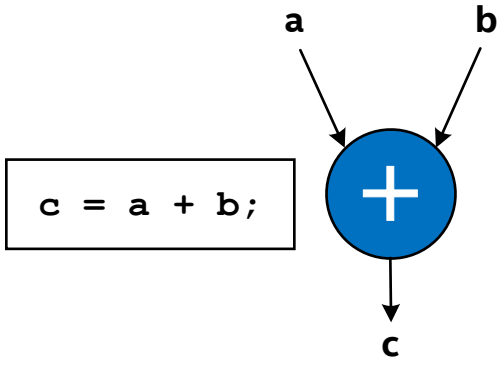
For fixed architectures such as CPUs and GPUs, the source code is compiled by the compiler into a set of instructions that run on functional units with a fixed functionality. For these fixed architectures to be useful in a broad range of applications, some of their available functional units are not useful to every program. Unused functional units mean that your program does not fully occupy the fixed architecture hardware.

FPGAs are not subject to these restrictions of fixed functional units. On an FPGA, you can synthesize a specialized hardware datapath that can be fully occupied for an arbitrary set of instructions, which means you can be more efficient with the chip's silicon area.

By implementing your algorithm in hardware, you can fill your chip with custom hardware that is always (or almost always) working on your problem instead of having idle functional units.

The Intel® oneAPI DPC++/C++ Compiler maps statements from the source code to individual specialized hardware operations, as shown in the example in the following image:

Figure 9. Mapping Source Code Instructions to Hardware



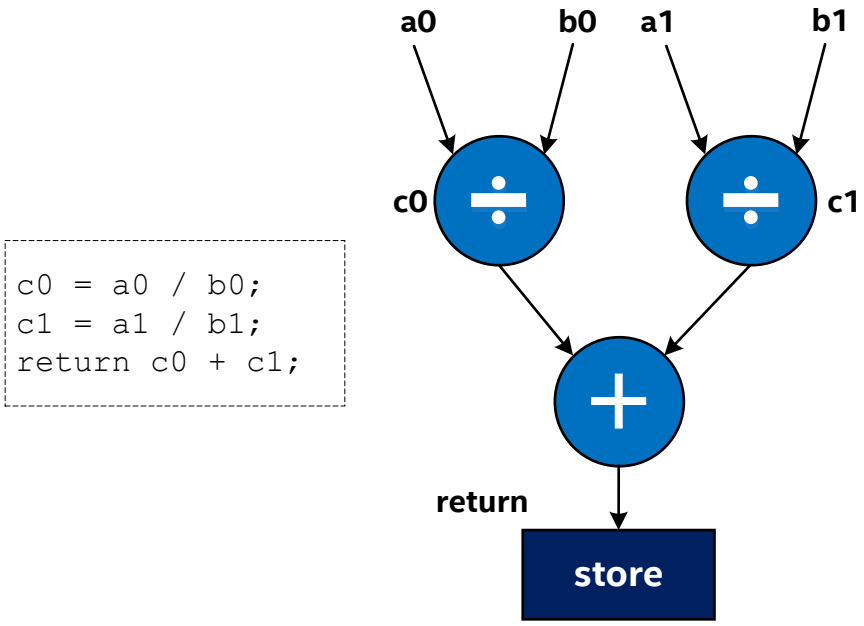
In general, each instruction maps to its own unique instance of a hardware operation. However, a single statement may map to more than one hardware operation, or multiple statements may combine into a single hardware operation when the compiler finds that it can generate more efficient hardware.

The latency of hardware operations is dependent on the complexity of the operation and the target f_{MAX}.

The compiler then takes these hardware operations and connects them into a graph based on their dependencies. When operations are independent, the compiler automatically infers parallelism by executing those operations simultaneously in time.

The following figure illustrates a dependency graph created for the hardware datapath:

Figure 10. Dependency Graph



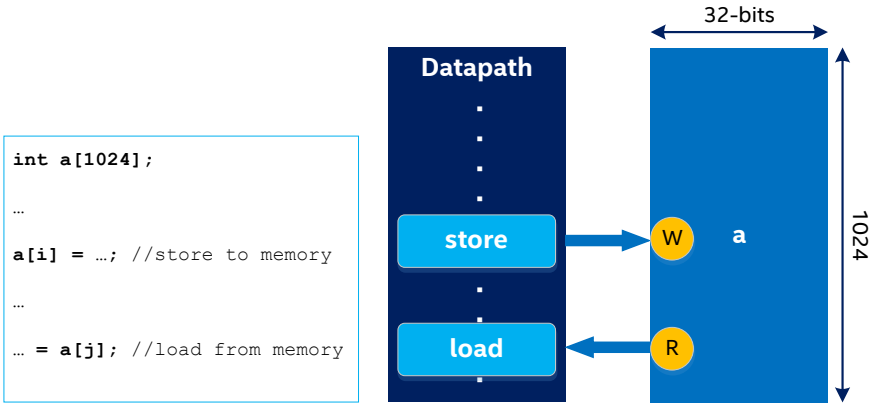
The dependency graph illustrates how the instruction is mapped to hardware operations and how the hardware operations are connected based on their dependencies. The loads in this example instruction are independent of each other and can therefore run simultaneously.

1.4.2 Mapping Arrays and Their Accesses to Hardware

Similar to mapping statements to specialized hardware operations, the compiler maps arrays to hardware memories based on memory access patterns and variable sizes. The datapath interacts with this memory through load/store units (LSUs), which are inferred from array accesses in the source code.

The following figure illustrates a simple example of mapping arrays and their accesses to hardware:

Figure 11. Mapping Arrays and Their Accesses to Hardware



A RAM can have a limited number of read ports and write ports, but a datapath can have many LSUs. When the number of LSUs does not match the available number of read and write ports, the compiler uses techniques like replication, double-pumping, sharing, and arbitration. For more information, refer to Kernel Memory on page 36.

NOTE

FPGAs provide specialized hardware block RAMs that you can configure and combine to match the size of your arrays. Doing so can provide many terabytes per second of on-chip memory bandwidth because each of these memories can interact with the datapath simultaneously.

Arrays might also be implemented in your kernel datapath. In this case, the array contents are stored as registers in the datapath when your algorithm is pipelined (as discussed in Pipelining on page 12). Storing array contents as registers in the datapath can improve performance in some cases, but it is a design decision whether to implement an array as registers or as memories.

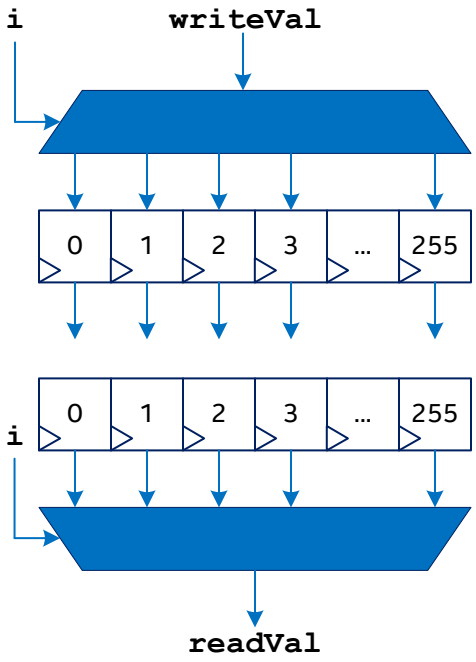
When you access an array that is implemented as registers, LSUs are not used. The compiler might choose to use a select or a barrel shifter instead.

Figure 12. Select or a Barrel Shifter

```
int a[256];

a[i] = writeVal; //store to array

readVal = a[j]; // load from array
```



1.5 Scheduling

Scheduling refers to the process of determining clock cycles at which each operation in the datapath executes. Pipelining is the outcome of scheduling.

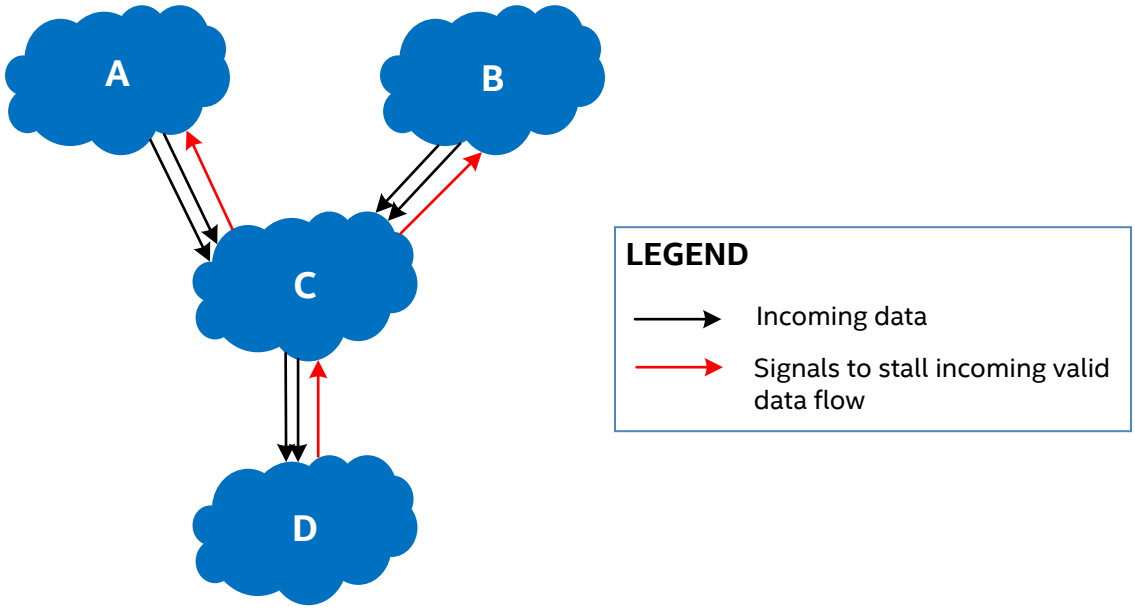
1.5.1 Dynamic Scheduling

The Intel® oneAPI DPC++/C++ Compiler generates pipelined datapath that is dynamically scheduled. A dynamically scheduled portion of the datapath does not pass data to its successor until its successor signals that it is ready to receive it. This signaling is accomplished using handshaking control logic. For example, a variable latency load from memory may refuse to accept its predecessors' data until the load is complete.

Handshaking helps in removing bubbles in the pipeline, thereby increasing occupancy. For more information about bubbles, refer to Occupancy on page 14.

The following figure illustrates four regions of dynamically scheduled logic:

Figure 13. Dynamically Scheduled Logic

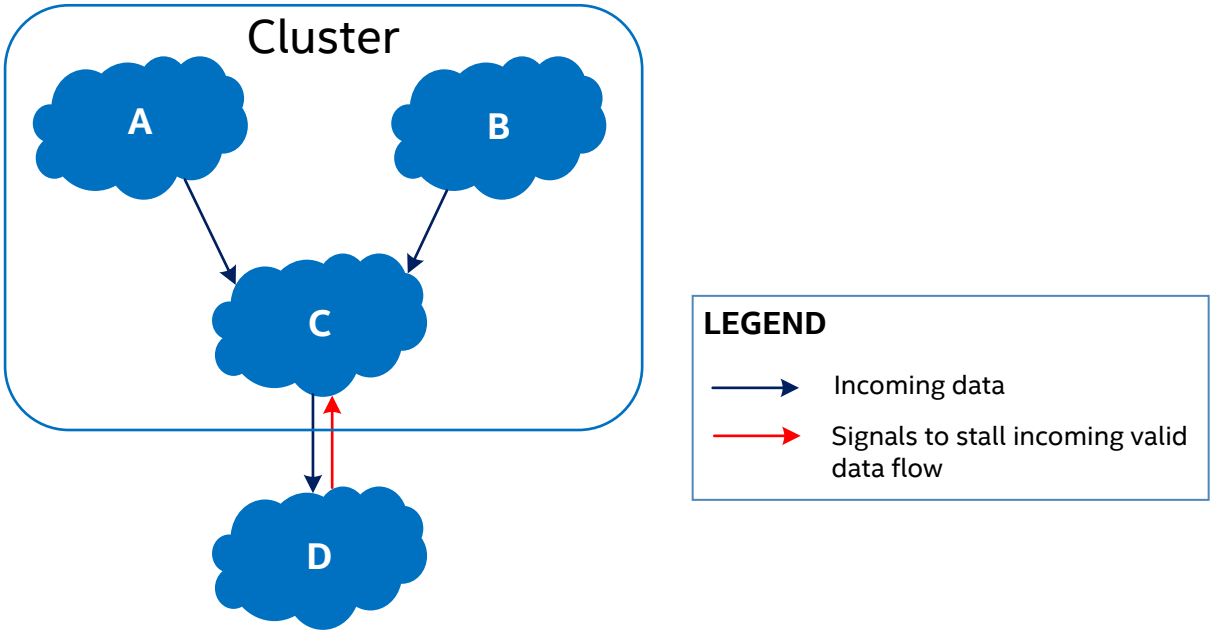


1.5.2 Clustering the Datapath

Dynamically scheduling all operations adds overhead in the form of additional FPGA areas required to implement the required handshaking control logic.

To reduce this overhead, the compiler groups fixed latency operations into clusters. A cluster of fixed latency operations, such as arithmetic operations, requires fewer handshaking interfaces, thereby reducing the area overhead.

Figure 14. Clustered Logic



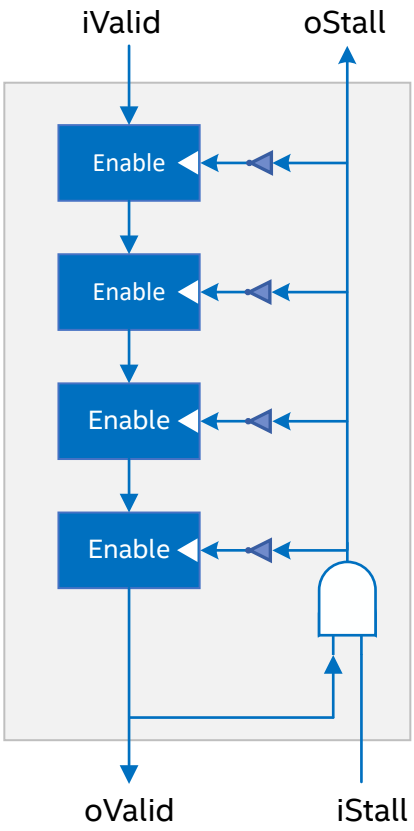
If A, B, and C from Figure 13 on page 19 do not contain variable latency operations, the compiler can cluster them together, as illustrated in Figure 14 on page 20. Clustering the logic reduces area by removing the need for signals to stall data flow in addition to other handshaking logic within the cluster.

Cluster Types

The Intel® oneAPI DPC++/C++ Compiler can create the following types of clusters:

- **Stall-Enable Cluster (SEC)**: This cluster type passes the handshaking logic to every pipeline stage in the cluster in parallel. This means that if the cluster is stalled by logic from further down in the datapath, all logic in the SEC stalls simultaneously.

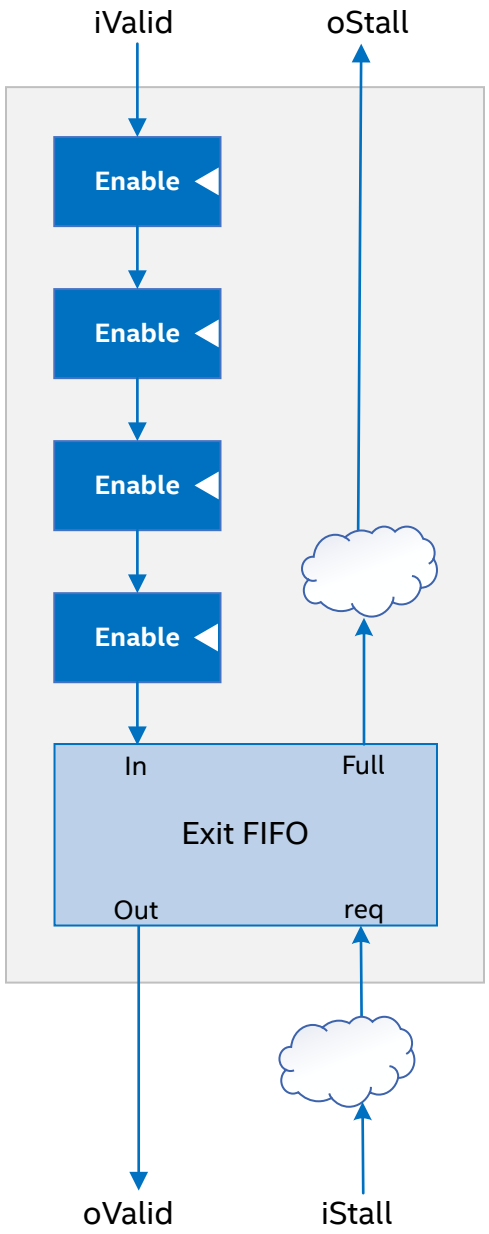
Figure 15. Stall-Enable Cluster



- **Stall-Free Cluster (SFC)**: This cluster type adds a first in, first out (FIFO) buffer to the end of the cluster that can accommodate the entire latency of the pipeline in the cluster. This FIFO is often called an *exit FIFO* because it is attached to the exit of the cluster datapath.

  Because of this FIFO, the pipeline stages in the cluster do not require any handshaking logic. The stages can run freely and drain into the capacity FIFO, even if the cluster is stalled from logic further down in the datapath.

Figure 16. Stall-Free Cluster



Cluster Characteristics

The exit FIFO of the stall-free cluster results in some of the following tradeoffs:

- **Area**: Because an SEC does not use an exit FIFO, it can save FPGA area compared to an SFC. If you have a design with many small, low-latency clusters, you can save a substantial amount of area by asking the compiler to use SECs instead of SFCs.

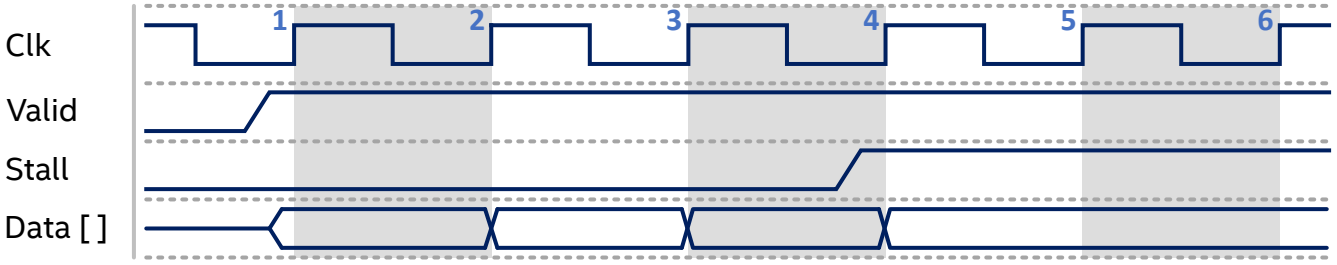
- **Latency**: Logic that uses SFCs might have a larger latency than logic that uses SECs because of the write-read latency of the exit FIFO. If you use a zero-latency FIFO for the exit FIFO, you can mitigate the latency, but f_{MAX} or FPGA area use might be negatively impacted. For additional information, refer to Global Control of Exit FIFO Latency of Stall-free Clusters (`-Xssfc-exit-fifo-type=<value>`) on page 172.
- **F_{MAX}**: In an SFC, the `oStall` signal has less fanout than in an SEC. For a cluster with many pipeline stages, you can improve your design f_{MAX} by using an SFC.
- **Handshaking**: The exit FIFO in SFCs allow them to take advantage of hyper-optimized handshaking between clusters. For more information, refer to Hyper Optimized Handshaking on page 23. SECs do not support this capability.
- **Bubble Handling**: SECs remove only leading bubbles in the pipeline under limited circumstances. A leading bubble is a bubble that arrives before the first piece of valid data arrives in the cluster. SECs do not remove any arriving afterward.

  SFCs can use the capacity FIFO to remove all bubbles from the pipeline if the SFC gets a downstream `stall` signal.
- **Stall Behavior**: When an SEC receives a downstream stall, it stalls any logic upstream of it within one clock cycle. When an SFC receives a downstream stall, the exit FIFO allows it to consume additional valid data depending on how deep the exit FIFO is and how many bubbles are in the cluster datapath.

1.5.3 Handshaking Between Clusters

By default, the handshaking protocol between clusters is a simple stall/valid protocol. Data from the upstream cluster is consumed when the `stall` signal is low, and the `valid` signal is high.

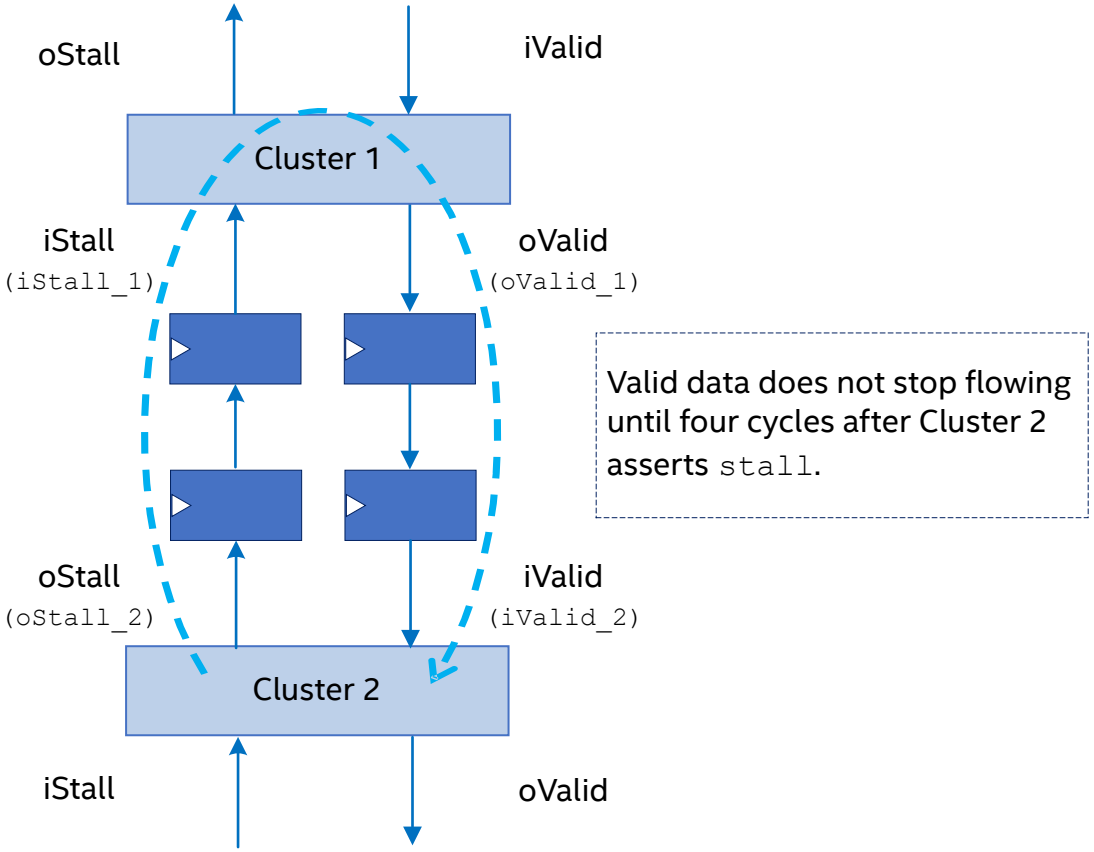
Figure 17. Handshaking Between Clusters



Hyper Optimized Handshaking

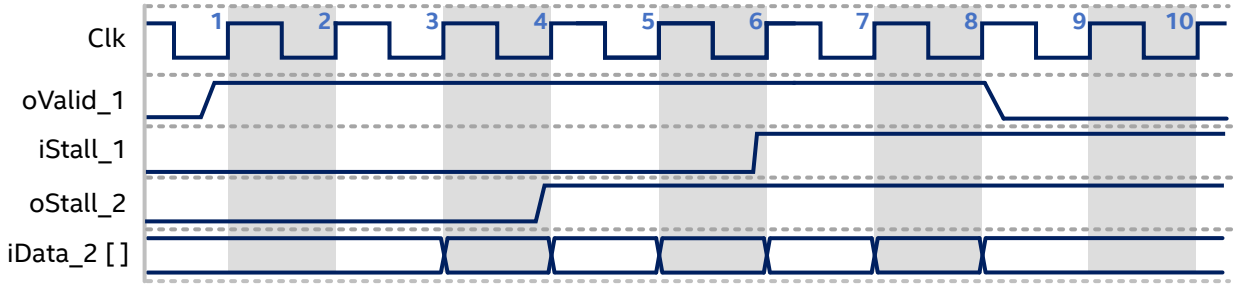
If the distance across the FPGA between these two clusters is large, handshaking may become the critical path that affects peak f_{MAX} in the design. To improve these cases, the Intel® oneAPI DPC++/C++ Compiler can add pipelining registers to the stall/valid protocol to ease the critical path and improve f_{MAX}. This enhanced handshaking protocol is called *hyper-optimized handshaking*.

Figure 18. Hyper-Optimized Handshaking Data Flow



The following timing diagram illustrates an example of upstream cluster 1 and downstream cluster 2 with two pipelining registers inserted in-between:

Figure 19. Hyper-Optimized Handshaking



RESTRICTION

Hyper-optimized handshaking is currently available only for the Intel® Agilex™ and Intel® Stratix® 10 device families.

1.6 Mapping Parallelism Models to FPGA Hardware

This section describes various mechanisms for mapping parallelism models to FPGA hardware:

- Data Parallelism on page 25
- Task Parallelism on page 35

1.6.1 Data Parallelism

Traditional instruction set architecture (ISA)-based accelerators, such as GPUs, derive data parallelism from vectorized instructions and execute the same operation on multiple processing units. In comparison, FPGAs derive their performance by taking advantage of their spatial architecture. FPGA compilers do not require you to vectorize your code. The compiler vectorizes your code automatically whenever it can.

The generated hardware implements data parallelism in the following ways:

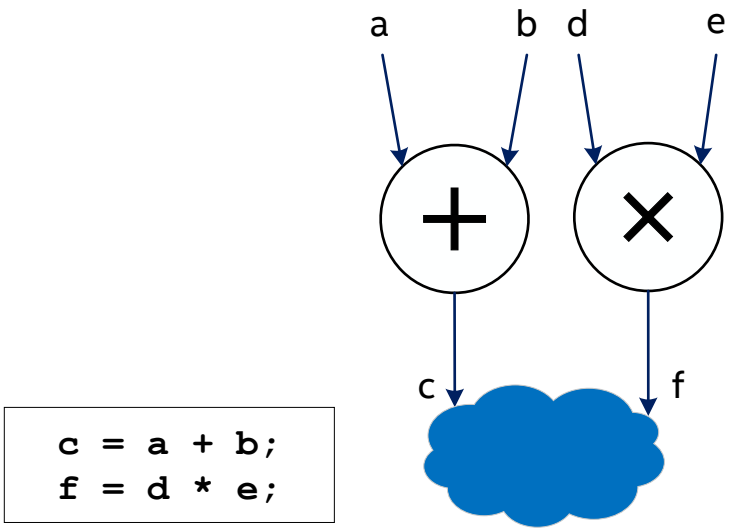
- Executing Independent Operations Simultaneously on page 25
- Pipelining on page 27

1.6.1.1 Executing Independent Operations Simultaneously

As described in Mapping Source Code Instructions to Hardware, the compiler can automatically identify independent operations and execute them simultaneously in hardware. This, when combined with pipelining (explained below), is how performance through data parallelism is achieved on the FPGA.

The following image illustrates an example of an adder and a multiplier, which are scheduled to execute simultaneously while operating on separate inputs:

Figure 20. Automatic Vectorization in the Generated Hardware Datapath



This automatic vectorization is analogous to how a superscalar processor takes advantage of instruction-level parallelism, but this vectorization happens statically at compile time instead of dynamically, at runtime. This means that there is no hardware or runtime cost of dependency checking for the generated hardware datapath. Additionally, the flexible logic and routing of an FPGA mean that only the available resources (ALMs, DSPs, and so on) of the FPGA bound the number of independent operations operating simultaneously.

Unrolling Loops

You can unroll loops in the design by using loop attributes. Loop unrolling decreases the number of iterations executed at the expense of increasing hardware resource consumption corresponding to executing multiple iterations of the loop simultaneously. Once unrolled, the hardware resources are scheduled as described in Scheduling on page 18.

NOTE

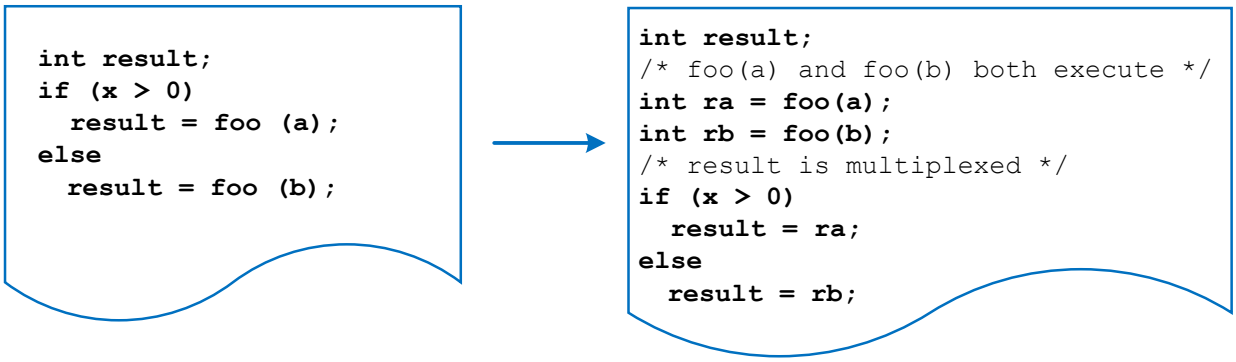
The Intel® oneAPI DPC++/C++ Compiler never attempts to unroll any loops in your source code automatically. You must always control loop unrolling by using the corresponding pragma. For more information, refer to Unroll Loops on page 89, Loop Directives on page 200, and Unrolling Loops tutorial.

Conditional Statements

The Intel® oneAPI DPC++/C++ Compiler attempts to eliminate conditional or branch statements as much as possible. Conditionally executed code becomes predicated in the hardware. Predication increases the possibilities for executing operations simultaneously and achieving better performance. Additionally, removing branches allows the compiler to apply other optimizations to the design.

In the following example, the function `foo` can be run unconditionally. The code that cannot be run unconditionally, like the memory assignments, retains a condition.

Figure 21. Conditional Statements



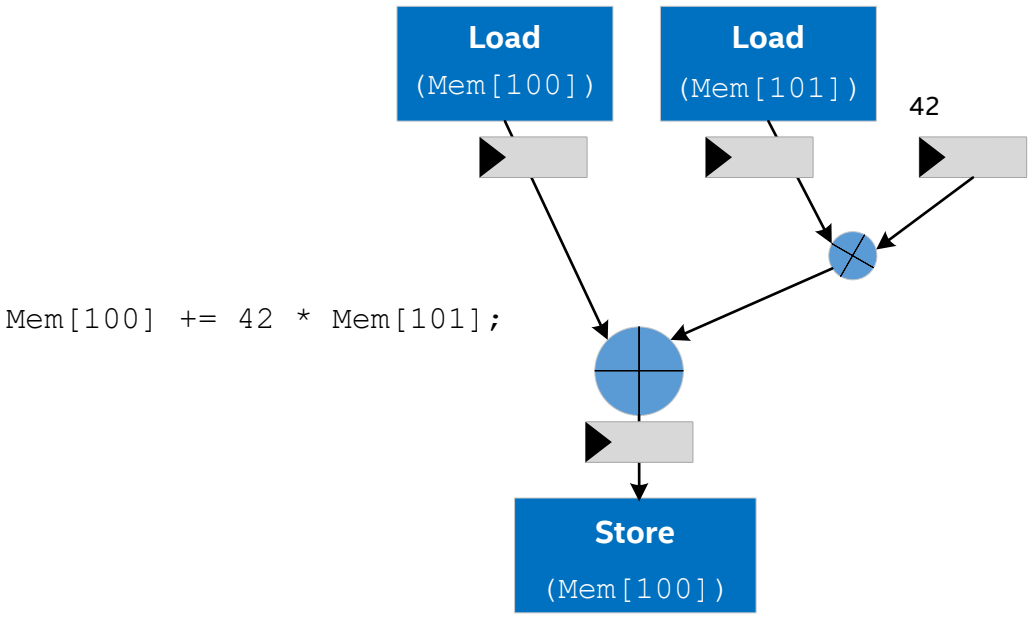
1.6.1.2 Pipelining

Similar to implementing a CPU with multiple pipeline stages, the compiler generates a deeply pipelined hardware datapath. For more information, refer to Concepts of FPGA Hardware Design on page 11 and How Source Code Becomes a Custom Hardware Datapath on page 15. Pipelining allows for many data items to be processed concurrently (in the same clock cycle) while efficiently using the hardware in the datapath by keeping it occupied.

Example 1. Example of Vectorization of the Datapath vs. Pipelining the Datapath

Consider the following example of code mapping to hardware:

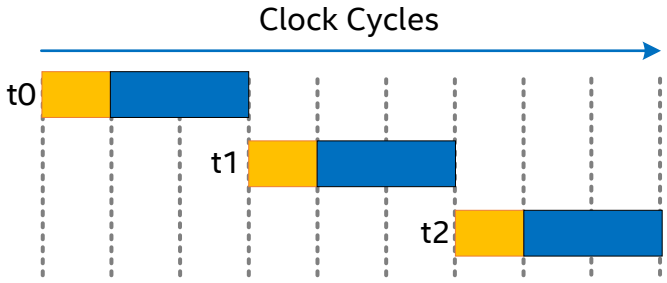
Figure 22. Example Code Mapping to Hardware



Multiple invocations of this code when running on a CPU would not be pipelined. The output of an invocation is completed before inputs are passed to the next invocation of the code. On an FPGA device, this kind of unpipelined invocation results in poor throughput and low occupancy of the datapath because many of the operations are sitting idle while other parts of the datapath are operating on the data.

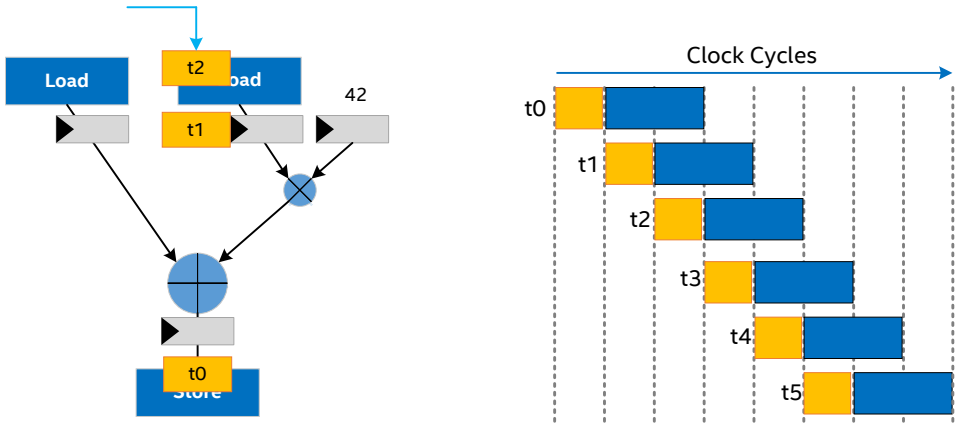
The following figure shows what throughput and occupancy of invocations look like in this scenario:

Figure 23. Unpipelined Execution Resulting in Low Throughput and Low Occupancy



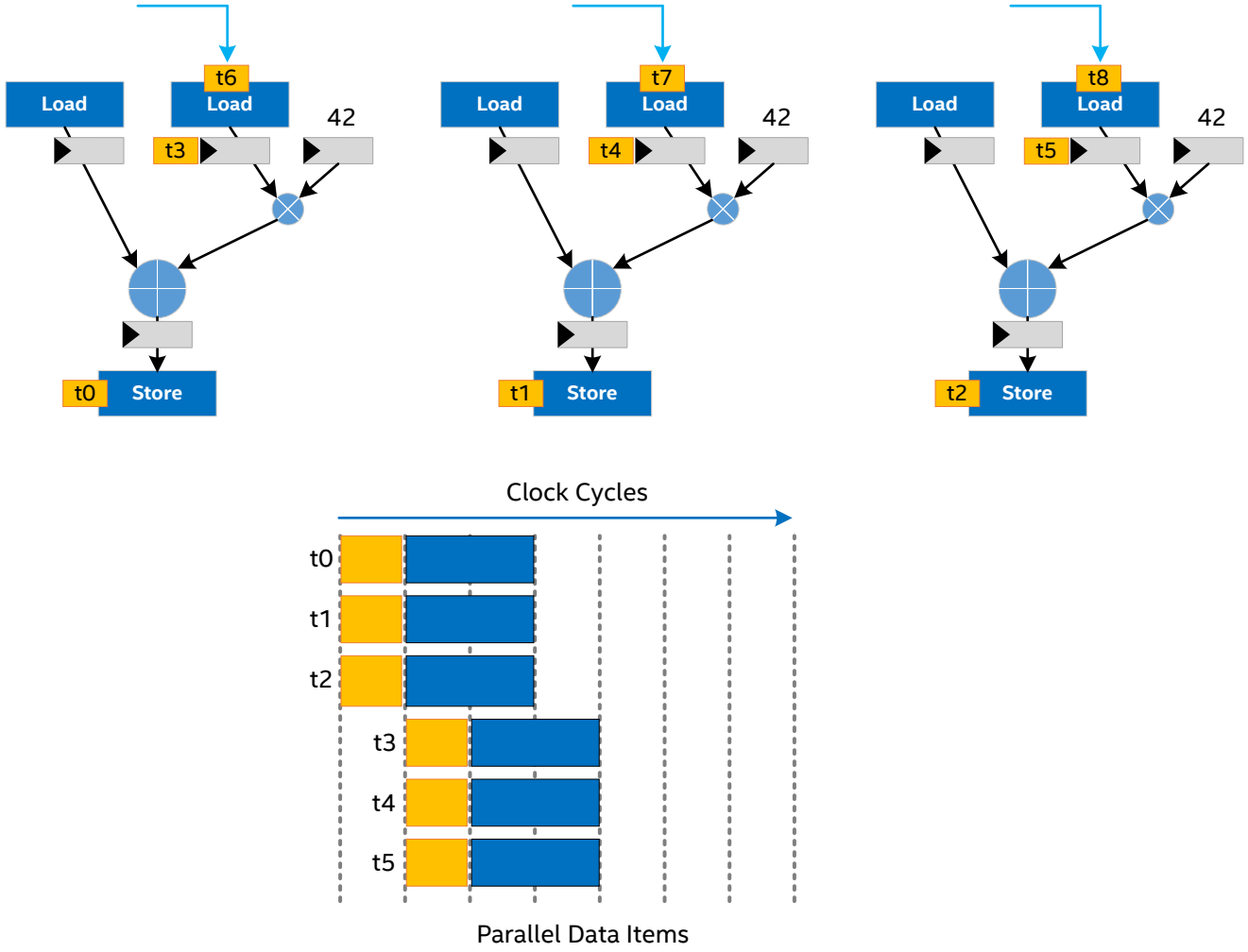
The Intel® oneAPI DPC++/C++ Compiler pipelines your design as much as possible. New inputs can be sent into the datapath each cycle, giving you a fully occupied datapath for higher throughput, as shown in the following figure:

Figure 24. Pipelining the Datapath Results in High Throughput and High Occupancy



You can gain even further throughput by vectorizing the pipelined hardware. Vectorizing the hardware improves throughput but requires more FPGA area for the additional copies of the pipelined hardware:

Figure 25. Vectorizing the Datapath Resulting in High Throughput and High Occupancy



Understanding where the data you need to pipeline is coming from is key to achieving high-performance designs on the FPGA. You can use the following sources of data to take advantage of pipelining:

- Work items
- Loop iterations

Pipelining Loops Within a Single Work Item

Within a single work item kernel, loops are the primary source of pipeline parallelism. When the Intel® oneAPI DPC++/C++ Compiler pipelines a loop, it attempts to schedule the loop's execution such that the next iteration of the loop enters the pipeline before the previous iteration has completed. This pipelining of loop iterations can lead to higher performance.

The number of clock cycles between iterations of the loop is called the *Initiation Interval* (II). For the highest performance, a new iteration of the loop would start every clock cycle corresponding to an II of 1. Data dependencies that are carried from

one iteration of the loop to another can affect the ability to achieve an II of 1. These dependencies are called *loop carried dependencies*. The II of the loop must be high enough to accommodate all loop carried dependencies.

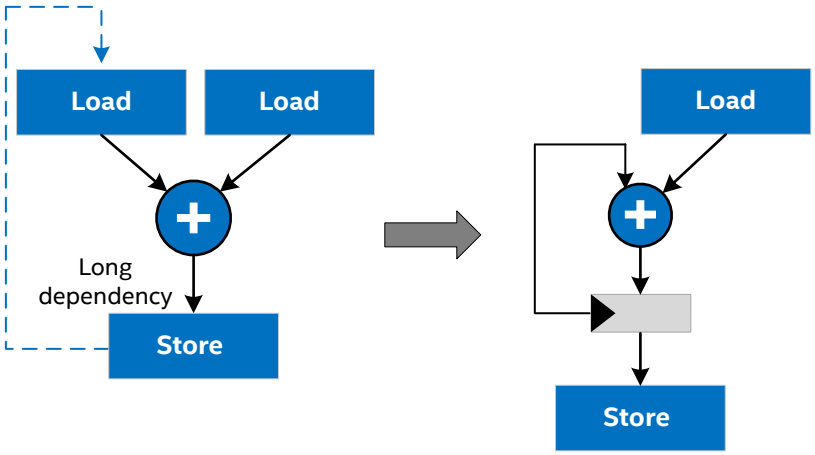
NOTE

The II required to satisfy this constraint is a function of the f_{MAX} of the design. If the f_{MAX} is lower, the II might also be lower. Conversely, if the f_{MAX} is higher, a higher II might be required. The Intel® oneAPI DPC++/C++ Compiler automatically identifies these dependencies and builds hardware to resolve them while minimizing the II, subject to the target f_{MAX}.

Naively generating hardware for the code in Figure 26 on page 31 results in two loads: one from memory b and one from memory c. Because the compiler knows that the access to `c[i-1]` was written to in the previous iteration, the load from `c[i-1]` can be optimized away.

Figure 26. Pipelining a Datapath with Loop Iteration

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



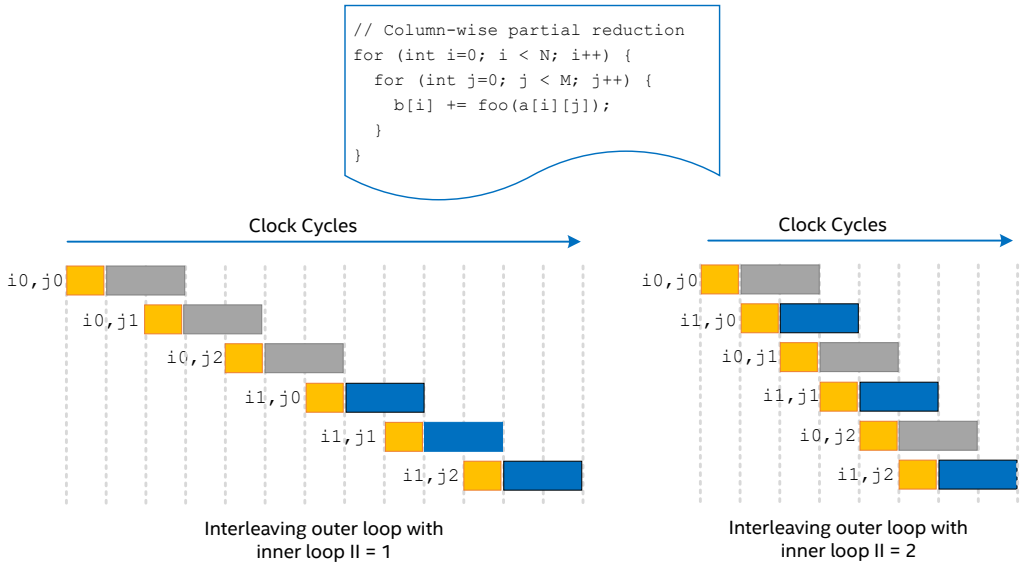
NOTE

The dependency on the value stored to `c` in the previous iteration is resolved in a single clock cycle, so an II of 1 is achieved for the loop even though the iterations are not independent.

In cases where the Intel® oneAPI DPC++/C++ Compiler cannot initially achieve II of 1, you have several optimization strategies to choose from:

- **Interleaving**: When a loop nest with an inner loop II is greater than 1, the Intel® oneAPI DPC++/C++ Compiler can attempt to interleave iterations of the outer loop into iterations of the inner loop to utilize the hardware resources better and achieve higher throughput.

Figure 27. Interleaving



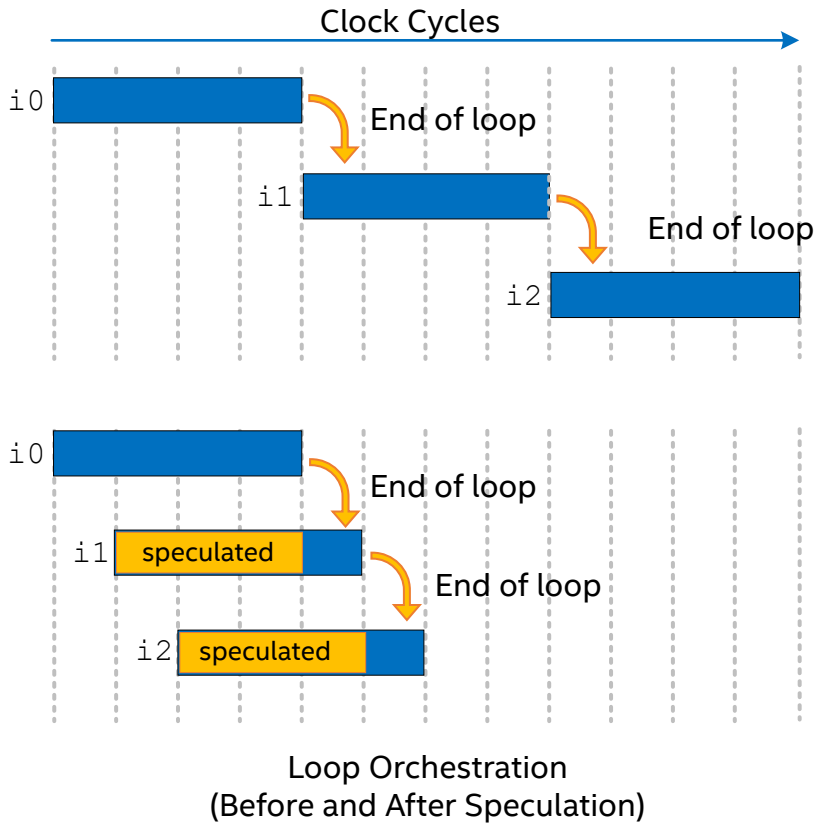
For additional information about controlling interleaving in your kernel, refer to `max_interleaving` Attribute on page 207.

- **Speculative Execution**: When the critical path affecting II is the computation of the loop's exit condition and not a loop carried dependency, the Intel® oneAPI DPC++/C++ Compiler can attempt to relax this scheduling constraint by speculatively continuing to execute iterations of the loop while the exit condition is being computed. If it is determined that the exit condition is satisfied, the effects of these extra iterations are suppressed. This speculative iteration can achieve lower II and higher throughput, but it can incur additional overhead between the loop invocations (equivalent to the number of speculated iterations). A larger loop trip count helps to minimize this overhead.

NOTE

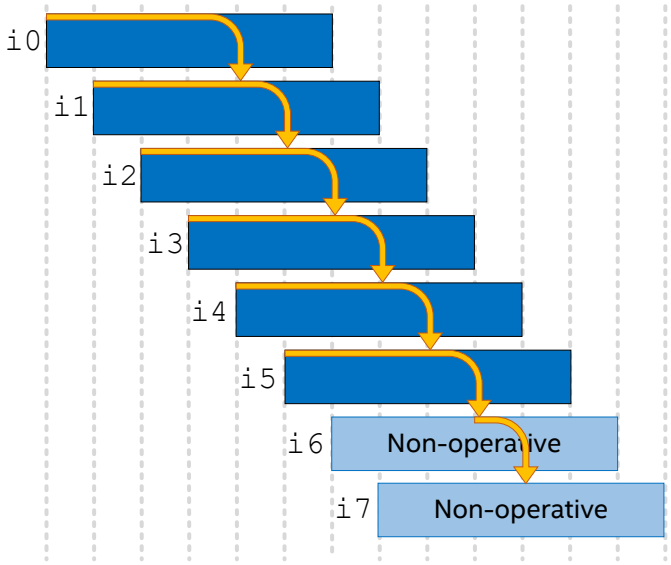
A loop invocation is what starts a series of loop iterations. One loop iteration is one execution of the body of a loop.

Figure 28. Loop Orchestration Without Speculative Execution



Loop Orchestration
(Before and After Speculation)

Figure 29. Loop Orchestration With Speculative Execution



These optimizations are applied automatically by the Intel® oneAPI DPC++/C++ Compiler, and they can also be controlled through pragma statements and loop attributes in the design. For additional information, refer to `speculated_iterations` Attribute on page 209

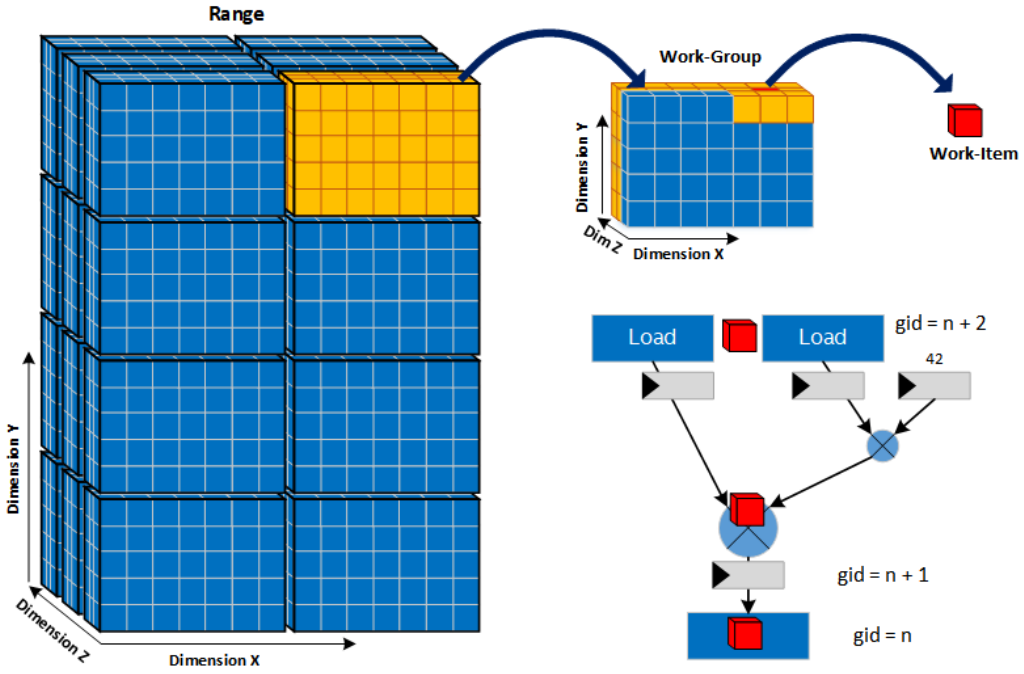
Pipelining Across Multiple Work Items

A range of work items represents a large set of independent data that can be processed in parallel. Since there are no implicit dependencies between work items, at every clock cycle, the next work item can always enter kernel's datapath before previous work items have completed, unless there is a dynamic stall in the datapath. The following figure illustrates the pipeline in Figure 22 on page 28 filled with work items:

NOTE

Loops are not pipelined for kernels that use more than one work item in the current version of the compiler. This will be relaxed in a future release.

Figure 30. Pipelining a Datapath with SYCL* Work Items

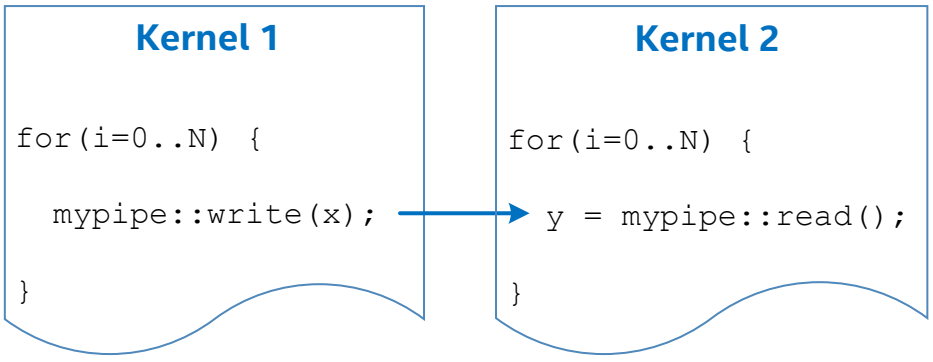


1.6.2 Task Parallelism

While the compiler achieves concurrency by scheduling independent individual operations to execute simultaneously, it does not achieve concurrency at coarser granularities (for example, across loops).

For larger code structures to execute in parallel, you must write them as separate kernels that launch simultaneously. These kernels then run asynchronously with each other, and you can achieve synchronization and communication using pipes, as illustrated in the following figure:

Figure 31. Multiple Kernels Running Asynchronously



This is similar to how a CPU program can leverage threads running on separate cores to achieve simultaneous asynchronous behavior.

1.7 Memory Types

The compiler maps the user-defined arrays in the source code to hardware memories. You can classify these hardware memories into Kernel Memory and Global Memory.

1.7.1 Kernel Memory

If you declare a private array, a group local memory, or a local accessor, then the Intel® oneAPI DPC++/C++ Compiler creates a kernel memory in hardware. Kernel memory is sometimes referred to as *on-chip memory* because it is created from memory sources (such as RAM blocks) available on the FPGA. The following source code snippet illustrates both a kernel and a global memory and their accesses:

```
constexpr int N = 32;
Q.submit([&](handler &cgh) {
  // Create an accessor for device global memory from buffer buff
  accessor acc(buff, cgh, write_only);
  cgh.single_task<class Test>([=]() {
    // Declare a private array
    int T[N];

    // Write to private memory
    for (int i = 0; i < N; i++)
      T[i] = i;

    // Read from private memory and write to global memory through the accessor
    for (int i = 0; i < N; i+=2)
      acc[i] = T[i] + T[i+1];
  });
});
```

To allocate local memory that is accessible to and shared by all work items of a workgroup, define a group-local variable at the function scope of a workgroup using the `group_local_memory_for_overwrite` function, as shown in the following example:

```
Q.submit([&](handler &cgh) {
  cgh.parallel_for<class Test>(
    nd_range<1>(range<1>(128), range<1>(32)), [=](nd_item<1> item) {
      auto ptr = group_local_memory_for_overwrite<int[64]>(item.get_group());
      auto& ref = *ptr;
      ref[2 * item.get_local_linear_id()] = 42;
    });
});
```

The example above creates a kernel with four workgroups, each containing 32 work items. It defines an `int[64]` object as a group-local variable, and each work-item in the workgroup obtains a `multi_ptr` to the same group-local variable.

The compiler performs the following to build a memory system:

- Maps each array access to a load-store unit (LSU) in the datapath that transacts with the kernel memory through its ports.
- Builds the kernel memory and LSUs and retains complete control over their structure.

- Automatically optimizes the kernel memory geometry to maximize the bandwidth available to loads and stores in the datapath.
- Attempts to guarantee that kernel memory accesses never stall.

These are discussed in detail in later sections of this guide.

Stallable and Stall-Free Memory Systems

Accesses to a memory (read or write) can be stall-free or stallable:

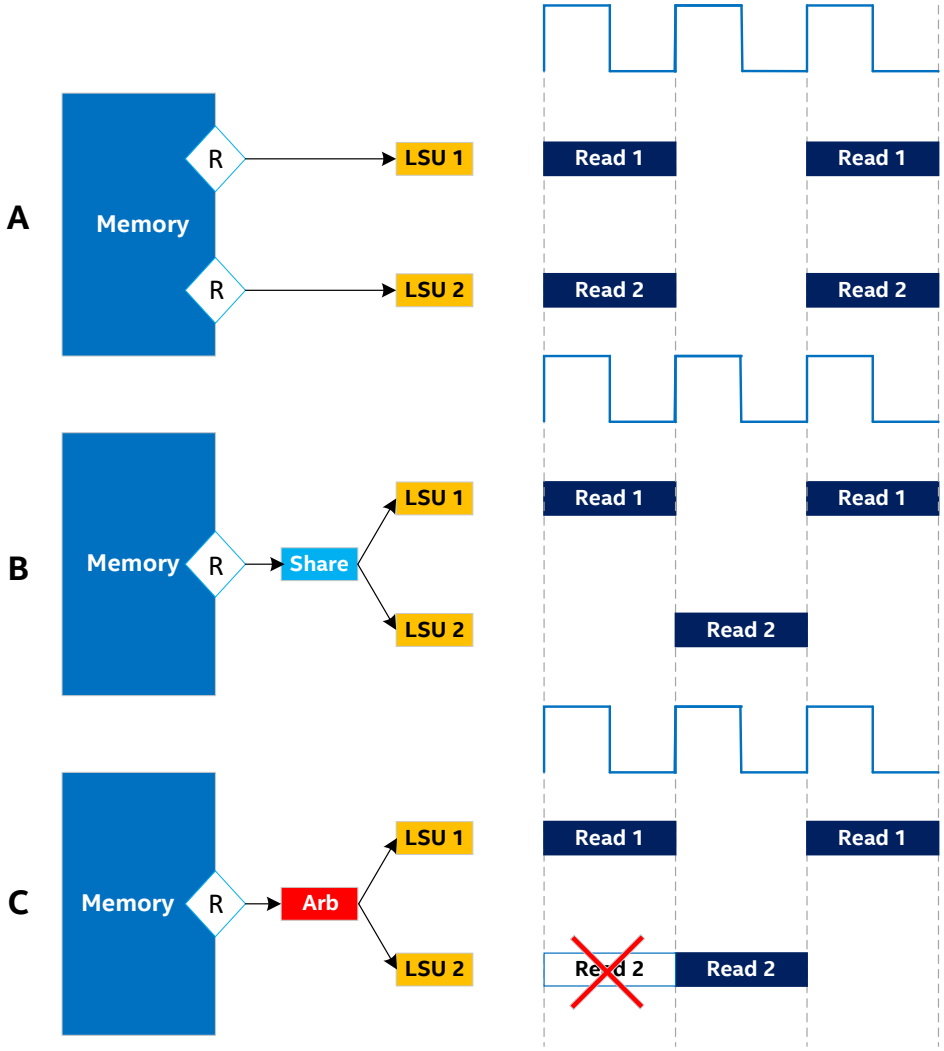
Table 1. Memory Systems

Memory Access	Description
Stall-free	A memory access is stall-free if it has contention-free access to a memory port. This is illustrated in Figure 32 on page 38. A memory system is stall-free if each of its memory operations has contention-free access to a memory port.
Stallable	A memory access is stallable if it does not have contention-free access to a memory port. When two datapath LSUs attempt to transact with a memory port in the same clock cycle, one of those memory accesses is delayed (or stalled) until the memory port in contention becomes available.

As much as possible, the Intel® oneAPI DPC++/C++ Compiler attempts to create stall-free memory systems for your kernel.

A read or write is stall-free if it has contention-free access to a memory port, as shown in the following figure:

Figure 32. Examples of Stall-free and Stallable Memory Systems



The Figure 32 on page 38 shows the following example memory systems:

- **A**: A stall-free memory system

  This memory system is stall-free because, even though the reads are scheduled in the same cycle, they are mapped to different ports. There is no contention for accessing the memory systems.

- **B**: A stall-free memory system

  This memory system is stall-free because the two reads are statically scheduled to occur in different clock cycles. The two reads can share a memory port without any contention for the read access.

- **C**: A stallable memory system

This memory system is stallable because two reads are mapped to the same port in the same cycle. The two reads happen at the same time. These reads require collision arbitration to manage their port access requests, and arbitration can affect throughput.

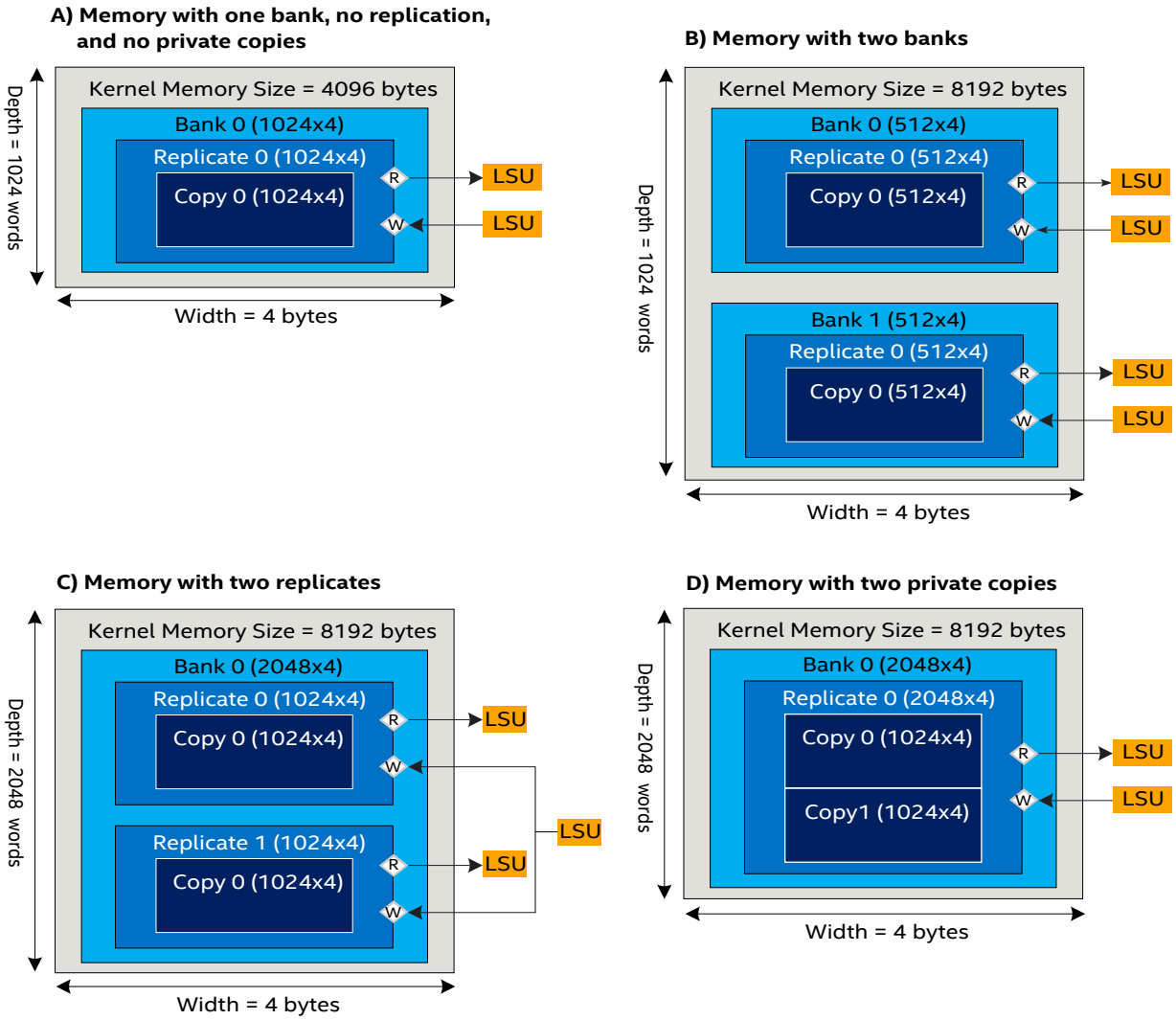
A kernel memory system consists of the following parts:

Table 2. Parts of a Kernel Memory System

Part	Description
Port	A memory *port* is a physical access point into a memory. A port is connected to one or more load-store units (LSUs) in the datapath. An LSU can connect to one or more ports. A port can have one or more LSUs connected.
Bank	A memory *bank* is a division of the kernel memory system that contains a subset of the data stored. That is, all of the data stored for a kernel is split across banks, with each bank containing a unique piece of the stored data. A memory system always has at least one bank.
Replicate	A memory bank *replicate* is a copy of the data in the memory bank with its own ports. All replicates in a bank contain the same data. Each replicate can be accessed independent of the others A memory bank always has at least one replicate.
Private Copy	A *private copy* is a copy of the data in a replicate that is created for nested loops to enable concurrent iterations of the outer loop. A *replicate* can comprise multiple private copies, with each iteration of an outer loop having its own private copy. Because each outer loop iteration has its own private copy, private copies are not expected to contain the same data.

The following figure illustrates the relationship between banks, replicates, ports, and private copies:

Figure 33. Schematic Representation of Kernel Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies



Strategies That Enable Concurrent Stall-Free Memory Accesses

The compiler uses a variety of strategies to ensure that concurrent accesses are stall-free including:

- Adjusting the number of ports the memory system has. This can be done either by replicating the memory to enable more read ports or by using double pumping to enable four ports instead of two per replicate. All of the replicate's physical access ports can be accessed concurrently.

- Partitioning memory content into one or more banks, such that each bank contains a subset of the data contained in the original memory (corresponds to the top-right box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies). The banks of a kernel memory can be accessed concurrently by the datapath.
- Replicating a bank to create multiple coherent replicates (corresponds to the bottom-left box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies). Each replicate in a bank contains identical data. The replicates are loaded concurrently.
- Creating private copies of an array declared inside a loop nest (corresponds to the bottom-right box of Schematic Representation of Local Memories Showing the Relationship between Banks, Replicates, Ports, and Private Copies). This enables loop pipelining, as each pipeline-parallel loop iteration accesses its own private copy of the array declared within the loop body. It is not expected for the private copies to contain the same data.

Despite the compiler's best efforts, the kernel memory system can still be stallable. This might happen due to resource constraints or memory attributes defined in your source code. In that case, the compiler tries to minimize the hardware resources consumed by the arbitrated memory system.

1.7.2 Global Memory

If the kernel code accesses a host-allocated buffer, the compiler creates a hardware interface through which the datapath accesses the buffer in global memory. A host-allocated buffer resides in device global memory off-chip. The code snippet in the Kernel Memory section shows a device global memory and its accesses within the kernel.

Unlike kernel memory, the compiler does not define the structure of a buffer in global memory. The compiler instantiates a specialized LSU for each access site based on the memory access pattern to maximize the efficiency of data accesses.

All accesses to global memory must go through the hardware interface. The compiler connects every LSU to an existing hardware interface through which it transacts with device global memory. Since the compiler cannot alter that interface or create more such interfaces, it must share the interface between multiple datapath reads or writes, which can limit the throughput of the design. The strategies used by the compiler to maximize efficient use of available memory interface bandwidth include (but are not limited to) the following:

- Eliminating unnecessary accesses.
- Statically coalescing contiguous accesses.
- Generating specialized LSUs that can perform the following:
  - Dynamically coalesced accesses that fall within the same memory word (as defined by the interface).
  - Prefetch and cache memory contents.

2.0 Analyze Your Design

When compiling an FPGA hardware image, the Intel® oneAPI DPC++/C++ Compiler provides object files, FPGA early image object, FPGA image object, and executables as checkpoints to allow you to inspect errors and modify your source code without performing a full compilation on each iteration.

At the FPGA early image object checkpoint, you can view the optimization report that the compiler generates. At the FPGA image object checkpoint, the compiler generates a complete FPGA image.

For detailed information about the FPGA compilation, refer to the FPGA Flow in the *Intel® oneAPI Programming Guide*.

2.1 Analyze the FPGA Early Image

The FPGA early image provides early access to compiler feedback about your design, including performance and area estimates.

2.1.1 Review the `report.html` File

Review the `<project_dir>/reports/report.html` file of your application to determine whether the estimated kernel performance data is acceptable. The HTML report also provides suggestions on how you can modify kernels to increase performance. For detailed information about generating the report, refer to the FPGA Flow section of the *Intel® oneAPI Programming Guide*.

2.1.1.1 Loop Analysis

The `report.html` file contains information about all loops in your design and their unroll statuses. The Loop Analysis report helps you examine whether the Intel® oneAPI DPC++/C++ Compiler can maximize your kernel's throughput.

To view the Loop Analysis report, click **Throughput Analysis ➤ Loop Analysis**. The purpose of this view is to show estimates of performance indicators (such as II) and potential performance bottlenecks. For each loop, you can identify the following using the report:

- Whether the loop is pipelined
- Whether the loop uses a hyper-optimized loop structure
- Any pragma or attribute applied to the loop
- II of the loop

NOTES

- Loop Analysis report does not report anything about loops in NDRange kernels.
- The F_{MAX} II report is now deprecated and merged with the Loop Analysis report.

The left-hand **Loops List** pane of the Loop Analysis report displays the following types of loops:

- Fused loops (see Fuse Loops to Reduce Overhead and Improve Performance on page 91)
- Fused subloops
- Coalesced loops
- Fully unrolled loops
- Partial unrolled loops
- Regular loops

Loop Pragma and Attributes

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

- `disable_loop_pipelining` Attribute on page 200
- `initiation_interval` Attribute on page 201
- `ivdep` Attribute on page 202
- `loop_coalesce` Attribute on page 204
- `max_concurrency` Attribute on page 205
- `max_interleaving` Attribute on page 207
- `speculated_iterations` Attribute on page 209
- `unroll` Pragma on page 210

Key Performance Metrics

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

- **Source Location**: Indicates the loop location in the source code.
- **Pipelined**: Indicates whether a loop is pipelined. Pipelining allows for many data items to be processed concurrently (in the same clock cycle) while efficiently using of the hardware in the datapath by keeping it occupied.
- **II**: Shows the sustainable initiation interval (II) of the loop. Processing data in loops is an additional source of pipeline parallelism. When you pipeline a loop, the next iteration of the loop begins before previous iterations complete. You can determine the number of clock cycles between iterations by the number of clock cycles you require to resolve any dependencies between iterations. You can refer to this number as the initiation interval (II) of the loop. The Intel® oneAPI DPC++/C++ Compiler automatically identifies these dependencies and builds hardware to resolve these dependencies while minimizing the II.
- **Scheduled f_{MAX}**: Shows the scheduled maximum clock frequency at which the loop operates. The f_{MAX} is the maximum rate at which the outputs of registers are updated.

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

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

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

NOTE

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

- **Latency**: Shows the number of clock cycles a loop takes to complete one or more instructions. Typically, you want to have low latency. However, lowering latency often results in decreased f_{MAX}.
- **Speculated Iterations**: Shows the loop speculation. Loop speculation is an optimization technique that enables more efficient loop pipelining by allowing future iterations to be initiated before determining whether the loop was exited already.
- **Max Interleaving Iterations**: Indicates the number of interleaved invocations of an inner loop that can be executed simultaneously. For more information, refer to `max_interleaving` Attribute on page 207.

Example 2. Example

The following is a SYCL* kernel example that includes three loops:

```
 1  cgh.single_task<class example>([=]() {
 2    #pragma unroll
 3    for (int i = 0; i < 10; i++) {
 4      acc_data[i] += i;
 5    }
 6    #pragma unroll 1
 7    for (int k = 0; k < N; k++) {
 8      #pragma unroll 5
 9      for (int j = 0; j < N; j++) {
10        acc_data[j] = j + k;
11      }
12    }
13  });
```

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

The Intel® oneAPI DPC++/C++ Compiler implements the following loop unrolling strategies based on the source code:

- Fully unrolls the first inner loop (line 3) because of the `#pragma unroll` specification.
- Does not unroll the second loop (line 7), which is an outer loop because of the `#pragma unroll 1` specification.
- Unrolls the third loop (line 9, an inner loop of the second loop) five times because of the `#pragma unroll 5` specification.

For more examples, refer to Loops on page 83 section.

2.1.1.2 Bottlenecks Viewer

The Bottlenecks viewer, when used with the Loop Analysis and Schedule Viewer, provides information about the throughput bottleneck(s) in your design. This viewer lists all loops that result in a bottleneck for the currently selected system, kernel, or task. You can select these loops to view more details about the bottleneck in the Details pane. For more information about the concept of bottlenecks, refer to Remove Loop Bottlenecks on page 93.

The Bottlenecks viewer identifies the following categories of bottlenecks:

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

For example, consider a simple loop with memory dependency from the Triangular Loops example:

```
for (int y = x + 1; y < n; y++) {
  local_buf[y] = local_buf[y] + SomethingComplicated(local_buf[x]);
}
```

In the Loop Analysis viewer, this loop is identified as having an II of 30. When you select this loop in the Loop Analysis report, you can find the following message in the Details pane:

```
Compiler failed to schedule this loop with smaller II due to memory dependency:
     From: Load Operation (triangular_loop.cpp: 78)
     To: Store Operation (triangular_loop.cpp: 78)
Compiler failed to schedule this loop with smaller II due to memory dependency:
     From: Load Operation (triangular_loop.cpp: 78)
```

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

2.1.1.3 Area Analysis of System

The `<project_dir>/reports/report.html` file contains information about area use of your DPC++ system.

The report provides the following information:

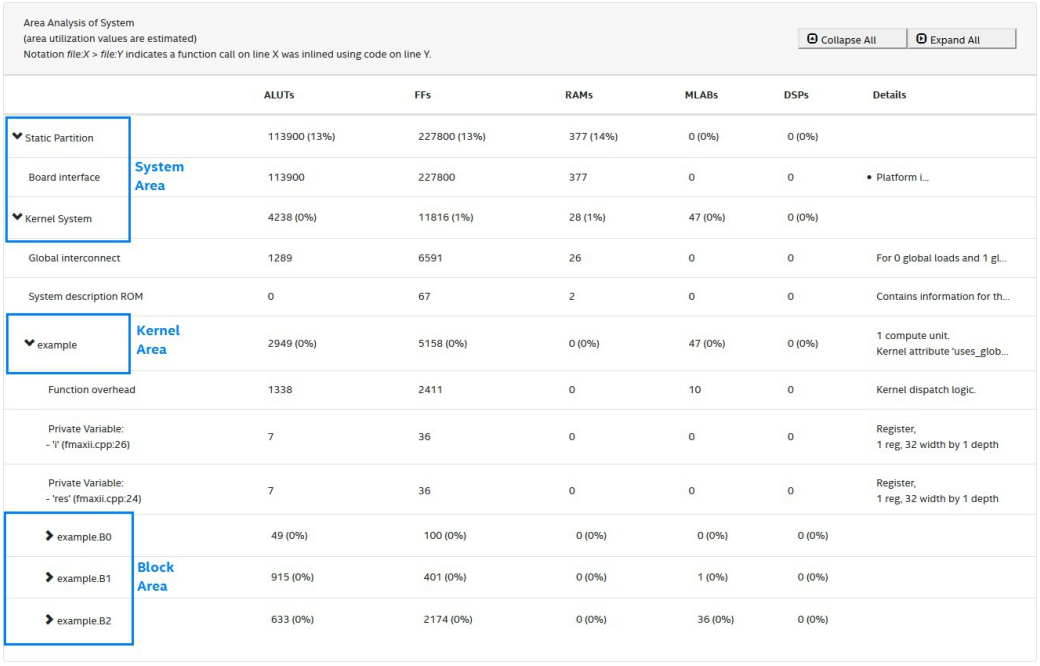
- Detailed area breakdown of the whole DPC++ system, mapped to your source code where possible.
- Architectural details to give insight into the generated hardware and offers actionable suggestions to resolve potential inefficiencies.

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

As you can observe in the following figure, the report is divided into three levels of hierarchy:

- **System area**: Used by all kernels, pipes, interconnects, and board logic.
- **Kernel area**: Used by a specific kernel, including overheads, for example, dispatch logic.
- **Block area**: Used by a specific block within a kernel. A block represents a branch-free section of your source code (for example, a loop body). To view the area, use information from the source code lines associated with a block and expand the report entry for that block.

Figure 34. Area Analysis of System Report Hierarchy



NOTE

The area use data are estimates that the Intel® oneAPI DPC++/C++ Compiler generates. These estimates might differ from the final area utilization results.

Messages in the Area Analysis of System Report

After you compile your DPC++ application, review the Area Analysis of System report that the Intel® oneAPI DPC++/C++ Compiler generates. In addition to summarizing the application's resource use, the Area Analysis of System report offers suggestions on modifying your design to improve efficiency. Refer to the following sections that describe various messages reported in the Area Analysis of System report.

Message for Board Interface

The Area Analysis of System report identifies the amount of logic that the Intel® oneAPI DPC++/C++ 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.

Message for Function Overhead

The Area Analysis of System report identifies the amount of logic that the Intel® oneAPI DPC++/C++ Compiler generates for dispatching kernels.

Message for State

The Area Analysis of System report identifies the number of resources 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.

Message for Feedback

The Area Analysis of System 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.

Messages for Private Variable Storage

The Area Analysis of System report provides information on the implementation of private memory based on your DPC++ design. For single work-item kernels, the Intel® oneAPI DPC++/C++ Compiler implements private memory differently, depending on the types of variable. The Intel® oneAPI DPC++/C++ Compiler implements scalars and small arrays in registers of various configurations (for example, plain registers, shift registers, and barrel shifter). The Intel® oneAPI DPC++/C++ Compiler implements larger arrays in block RAM.

The following table lists messages and notes of different private variable storage types:

Table 3. Additional Information About Area Analysis of System Report Message

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

NOTES

- The Area Analysis of System report annotates memory information on the line of code that declares or uses private memory, depending on its implementation.
- When the Intel® oneAPI DPC++/C++ Compiler implements private memory in on-chip block RAM, the Area Analysis of System report displays relevant local-memory-specific messages to private memory systems.

2.1.1.4 System Viewer

The `<project_dir>/reports/report.html` file provides a stall-point graph that includes sizes and types of loads and stores, stalls, latencies, load and store information between kernels and different memories, pipes connected between kernels, and loops.

The System Viewer (see Figure 35 on page 50) shows an abstracted netlist of your DPC++ system in a hierarchical graphical report consisting of system, global memory, block, and cluster views. It allows you to review information such as sizes, types, dependencies, and schedules of instructions, FIFO created by feedback nodes, FIFO created for capacity balancing in stallable regions, properties of interfaces such as pipe and memory interface, and view variables with loop-carried dependencies.

Access the System Viewer by selecting **Views ➢ System Viewer** in the `report.html`.

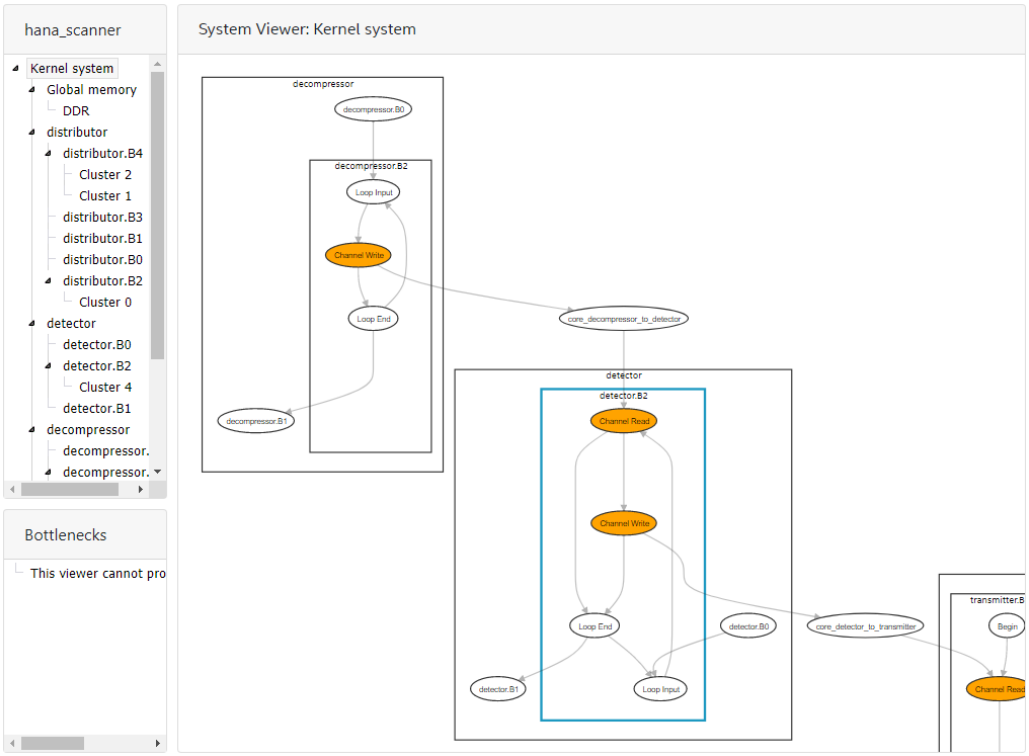
You can interact with the System Viewer in the following ways:

- Use the mouse wheel to zoom in and out within the System Viewer.
- Navigate through the following hierarchical views in the left-hand pane:
  - System
  - Global memory
  - Block
  - Cluster
- Click a node to display its location in the source code in the Code pane and node details in the Details pane.

System View

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

Figure 35. Kernel System View of the System Viewer Report



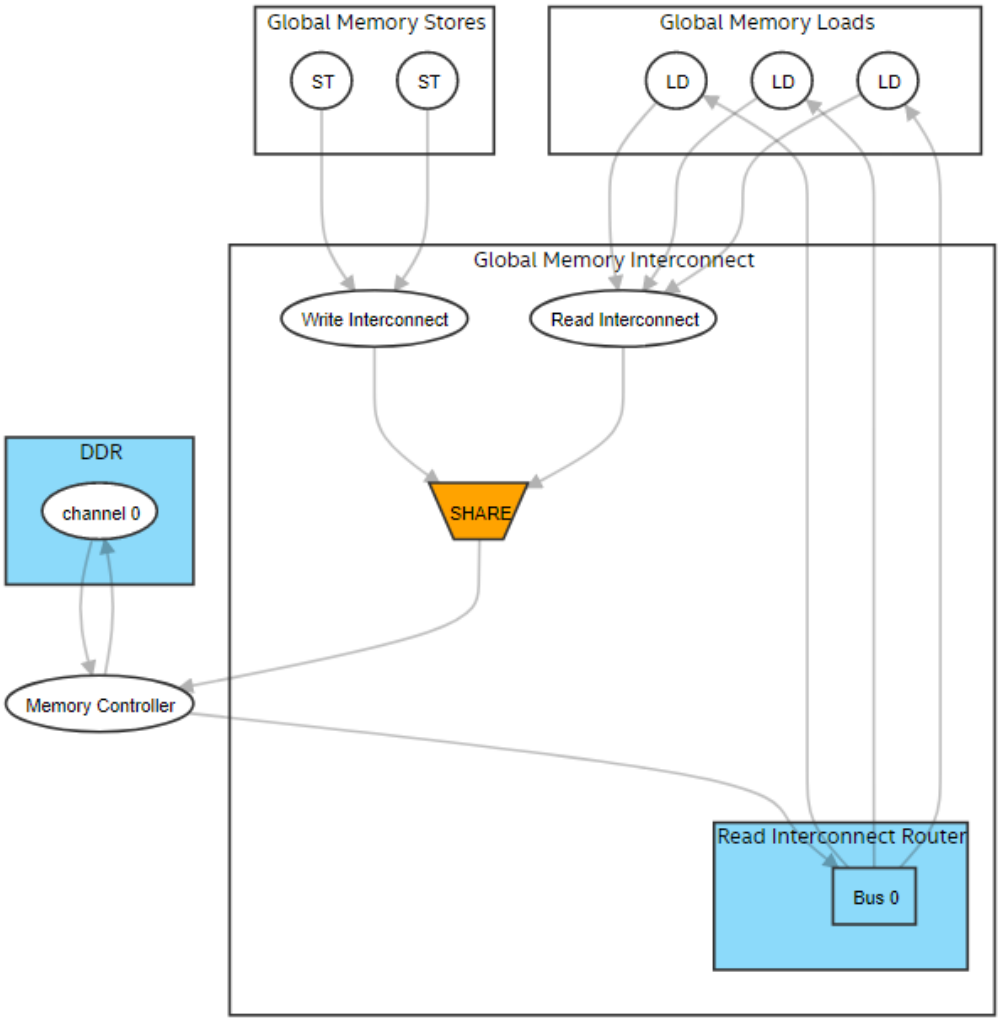
Global Memory View

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

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

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

Figure 36. Graphical Representation of the Global Memory in the System Viewer



In Figure 36 on page 51:

- When you select stores or loads, you can view respective lines in the source code and details about the LSU type and LSU-level bandwidth.
- For the write interconnect block, you can view the interconnect style, number of writes to the global memory, status of the `-Xsforce-single-store-ring` compiler flag, and number of store rings.
- For the read interconnect block, you can view the interconnect style and number of reads from the global memory.
- For the read interconnect router block, you can view the status of the `-Xsnum-reorder` flag, total number of buses, and all connections between buses and load LSUs. Buses in this block provide read data from the memory to load LSUs.
- For the global memory (DDR in Figure 36 on page 51), you can view the status of interleaving, interleaving size, number of channels, maximum bandwidth the BSP can deliver, and channel width.

- For the memory controller block, you can view the maximum bandwidth the BSP can deliver, sum of the load/store throughput, and read/write bandwidth. For additional information about how global memory bandwidth use is calculated, refer to Global Memory Bandwidth Use Calculation on page 110 in this guide. It describes the formulas used in calculating the bandwidth.
- LSUs using USM pointers show up twice in both host and device global memory views as they can access both memories.

Block View

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

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

NOTE

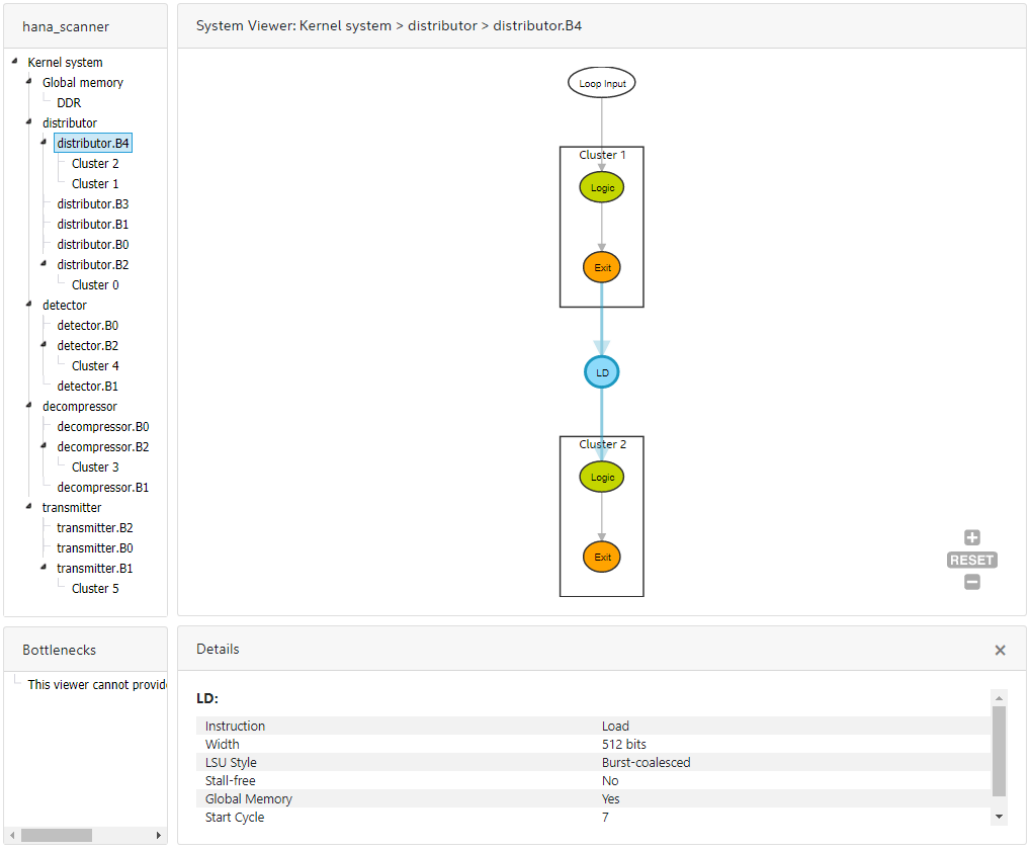
The schedule information is relative to the start of each block. Because the Intel® oneAPI DPC++/C++ Compiler cannot statically infer the trip counts of blocks, if your design consists of multiple blocks, the compiler cannot compute the absolute schedule information by considering the trip counts. Moreover, the schedule information provides estimated values from empirical measurements for the stallable instructions (such as pipe RD/WR or memory LD/ST). The real schedule is likely to be different, and you must verify it with a hardware, or a simulation run.

If your design has loops, the Intel® oneAPI DPC++/C++ Compiler encapsulates the loop control logic into loop orchestration nodes and the initial condition of the loops into loop input nodes and their connection to the datapath.

Inside a block, there are often pipe RD/WR or memory LD/ST nodes connecting to computation nodes or clusters. You can click on the computation nodes and view the Details pane (or hover over the nodes) to see specific instructions and the bit width. You can click on the RD/WR or LD/ST nodes to see information such as instruction type, width, depth, LSU type, stall-free global memory, scheduled start cycle, estimated latency, and schedule of a pipe or an LSU from the Details pane. For stallable nodes, the latency value provided is an estimate. Perform a simulation or hardware run for more accurate latency values.

If your design has clusters, a cluster has a FIFO in its exit node to store any pipelined data in-flight. You can click on the cluster exit node to find the exit FIFO width and depth attribute. The cluster exit FIFO size is also available in the cluster view of the System Viewer.

Figure 37. Block View of the System Viewer Report



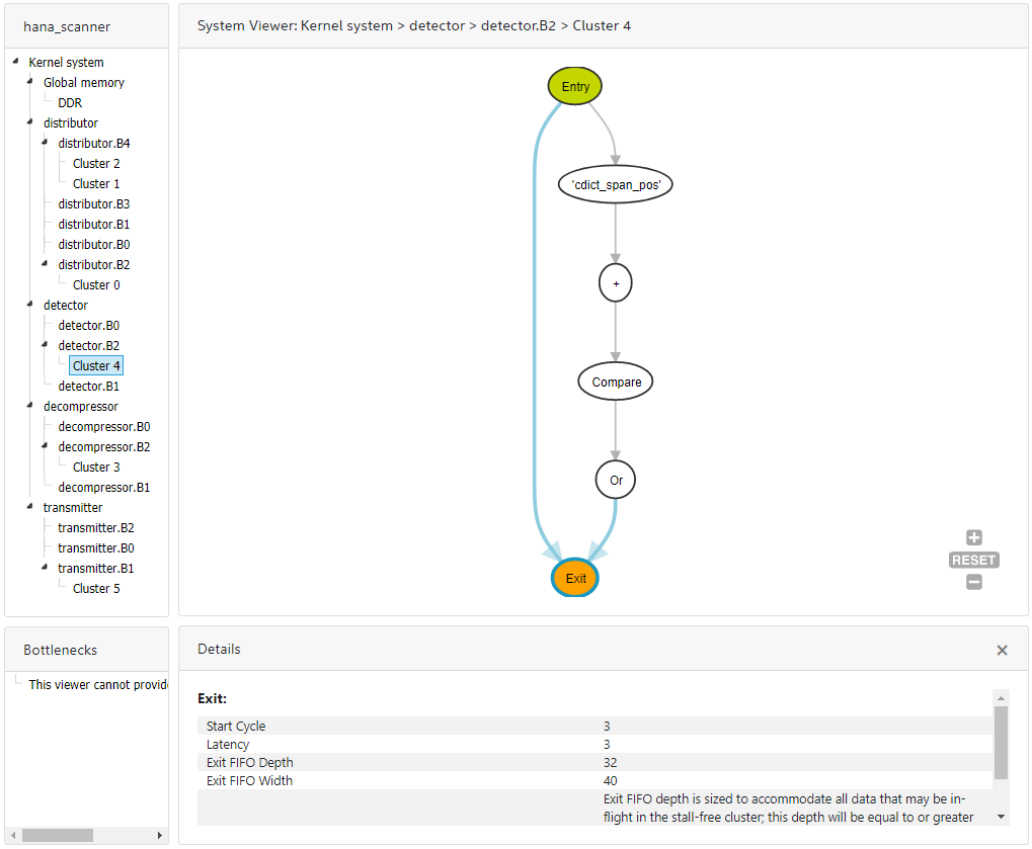
Cluster View

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

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

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

Figure 38. Cluster View of the System Viewer



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

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

NOTE

The dependency distance is the number of iterations between successive load/store nodes that depend on each other.

2.1.1.5 Kernel Memory Viewer

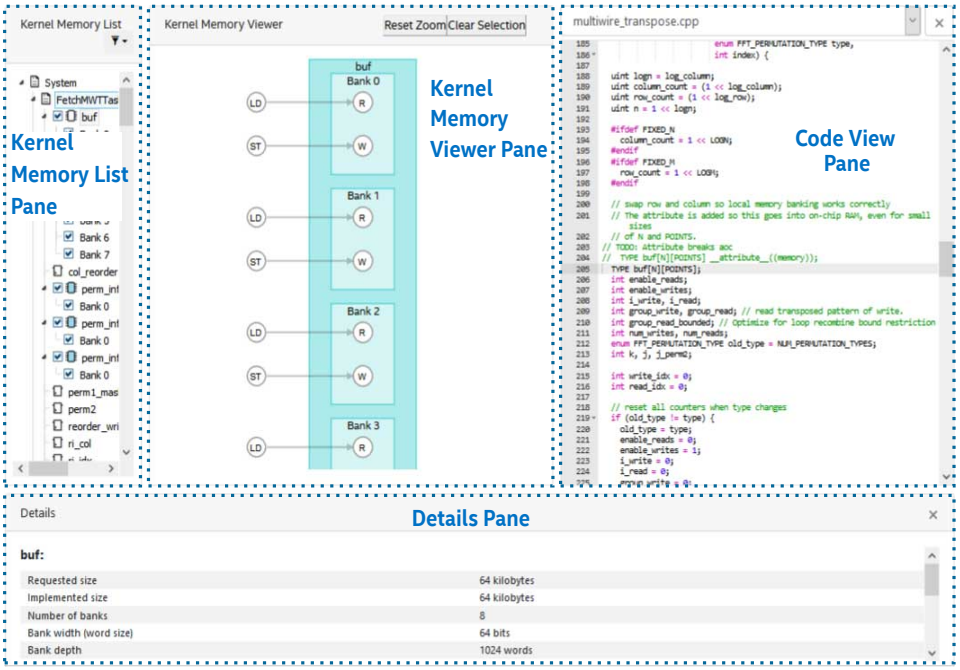
Data movement is a bottleneck in many algorithms. The Kernel Memory Viewer shows you how the Intel® oneAPI DPC++/C++ Compiler interprets the data connections and synthesizes memory for your kernel. Use the Kernel Memory Viewer to help identify data movement bottlenecks in your kernel design.

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

From the **Reports** menu's **System Viewers** drop-down menu, select **Kernel Memory Viewer** to analyze your SYCL* system.

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

Figure 39. Kernel Memory Viewer Layout



The Kernel Memory Viewer has the following panes:

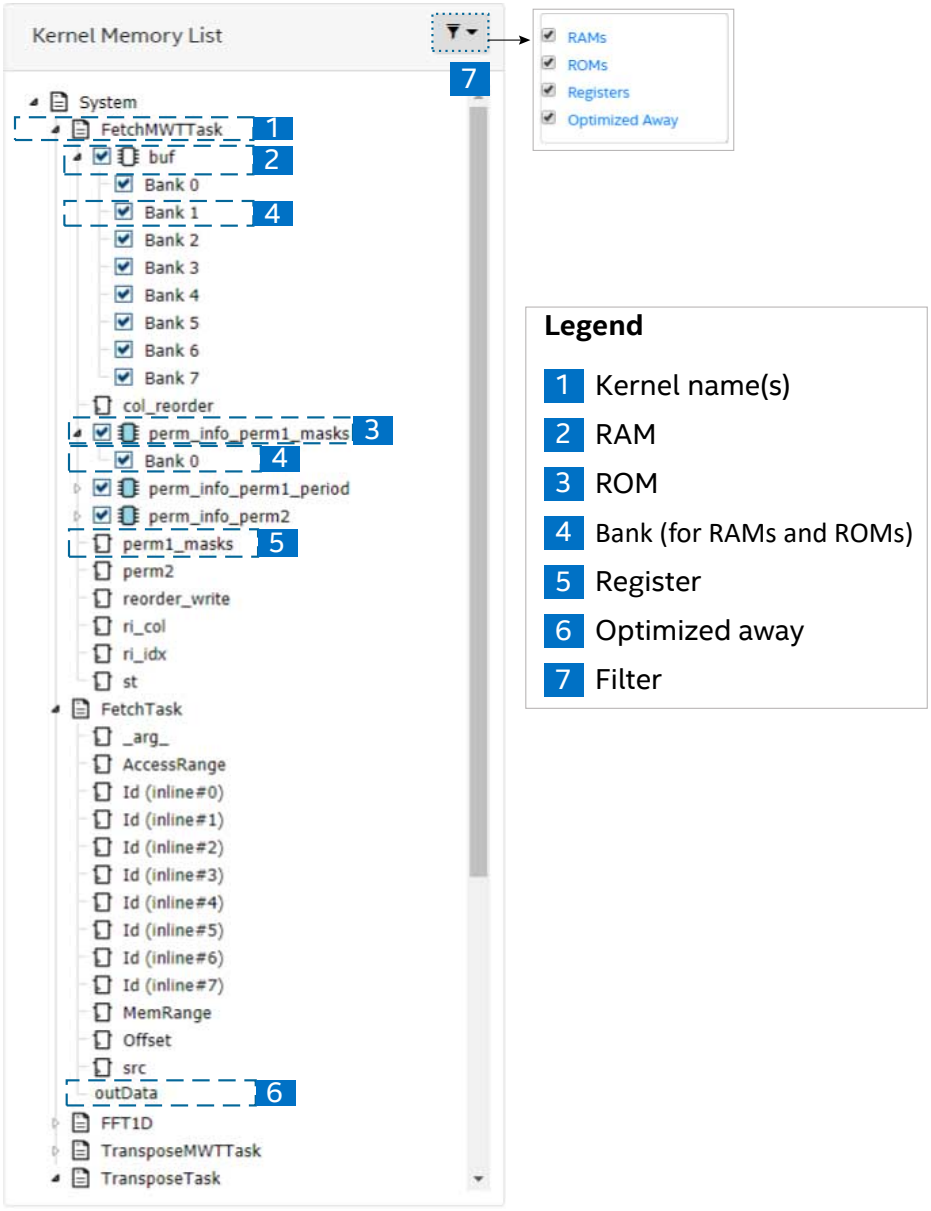
Table 4. Kernel Memory Viewer Panes

Pane	Description
Kernel Memory List	Lists all memories present in your design. When you select a memory name, you can view its graphical representation in the Kernel Memory Viewer pane.
Kernel Memory Viewer	Shows a graphical representation of the memory system or memory bank selected in the Kernel Memory List pane.
Code View	Shows the source code file for which the reports are generated.
Details	Shows the details of the memory system or memory bank selected in the Kernel Memory List pane.

Kernel Memory List

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

Figure 40. Features and Details of the Kernel Memory List Pane



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

Table 5. Kernel Memory List Pane Icons and Labels

No.	Icon or Label	Name	Description
1		Kernel name	You can expand or collapse the list of memories in your kernel. Memories that do not belong to any kernel are displayed under **(Other)**.
2		RAM	A RAM is a memory that has at least one write to it. The name of the RAM memory is the same as its name in your design. When you select a memory name, you can view a logical representation of the RAM in the Kernel Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select banks that you want the Kernel Memory Viewer pane to display: • Expand the memory name. • Clear the memory name check box to collapse all memory banks in the view. • Select the memory name check box to show all memory banks in the view.
3		ROM	A ROM is a read-only memory. The name of the ROM memory is same as its name in your design. When you select a memory name, you can view a logical representation of the ROM in the Kernel Memory Viewer pane. By default, only the first bank of the memory system is displayed. To select banks that you want the Kernel Memory Viewer pane to display: • Expand the memory name. • Clear the memory name check box to collapse all memory banks in the view. • Select the memory name check box to show all memory banks in the view.
4	**Bank #num**	Bank	A memory bank is always associated with a RAM or a ROM. Each bank is named **Bank #num**, where #num is the memory bank's ID starting from 0. • Click on the bank name to display the bank view in the Kernel Memory Viewer pane, which displays a graphical representation of the bank with all its replicates and private copies. This view can help you focus on specific memory banks of a complex memory design. • Clear the memory bank name check-box to collapse the bank in the logical representation of the memory. • Select the memory bank name check-box to display the bank in the logical representation of the memory.
5		Register	A register is a kernel variable carried through the pipeline in registers (rather than being stored in a RAM or ROM). The name of the register is the same as its name in your design. A register variable is implemented either exclusively in FFs or in a combination of FFs and RAM-based FIFOs.
6	**Text label**	Optimized Away	A kernel variable can be optimized away because it is unused in your design, or compiler optimizations have transformed all uses of the variable such that it is unnecessary. The name of the optimized away variable is the same as its name in your design.
7		Filter	Use the Kernel Memory List filter to selectively view the list of RAMs, ROMs, registers, and optimized away variables in your design.

continued...

No.	Icon or Label	Name	Description
			When you clear the checkbox associated with an item in the filter, you hide all occurrences of that kind of item in the Kernel Memory List. Filter your Memory List to help you focus on a specific type of memory in your design.

Kernel Memory Viewer

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

Table 6. Node Types Observed in the Kernel Memory Viewer Pane

Node Type	Description
Memory node	The memory system for a given variable in your design.
Bank node	A bank in the memory system. A memory system contains at least one bank. A memory bank can connect to one or more port nodes.
Replication node	A replication node shows memory bank replicates created to support multiple accesses to a local memory efficiently. A bank contains at least one replicate. You can view replicate nodes when you view a memory bank by clicking its name in the Kernel Memory List pane.
Private-copy node	A private-copy node shows private copies within a replicate created to allow simultaneous execution of multiple loop iterations. A replicate contains at least one private copy. You can view private-copy nodes when you view a memory bank by clicking its name in the Kernel Memory List pane.
Port node	Each read or write access to local memory is mapped to a port. The logical port for a bank. There are three types of ports: • **R**: A read-only port • **W**: A write-only port • **RW**: A read and write port
LSU node	A store (ST) or load (LD) node connected to the memory through port nodes.
Arbitration node	An arbitration (ARB) node shows that LSUs compete for access to a shared port node, which can lead to stalls.
Port-sharing node	A port-sharing node (SHARE) shows that LSUs have mutually exclusive access to a shared port node, so the load-store units are free from stalls.

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

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

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

Figure 41. Logical Representation of a Memory in the Kernel Memory Viewer Pane

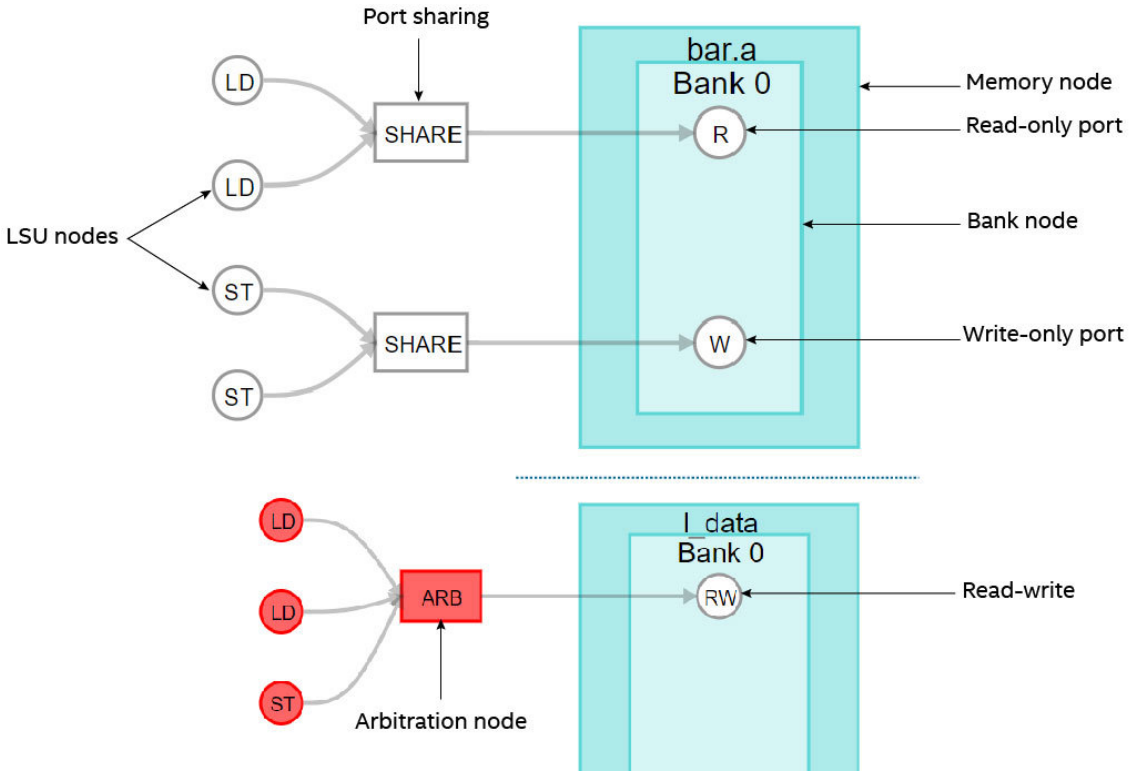
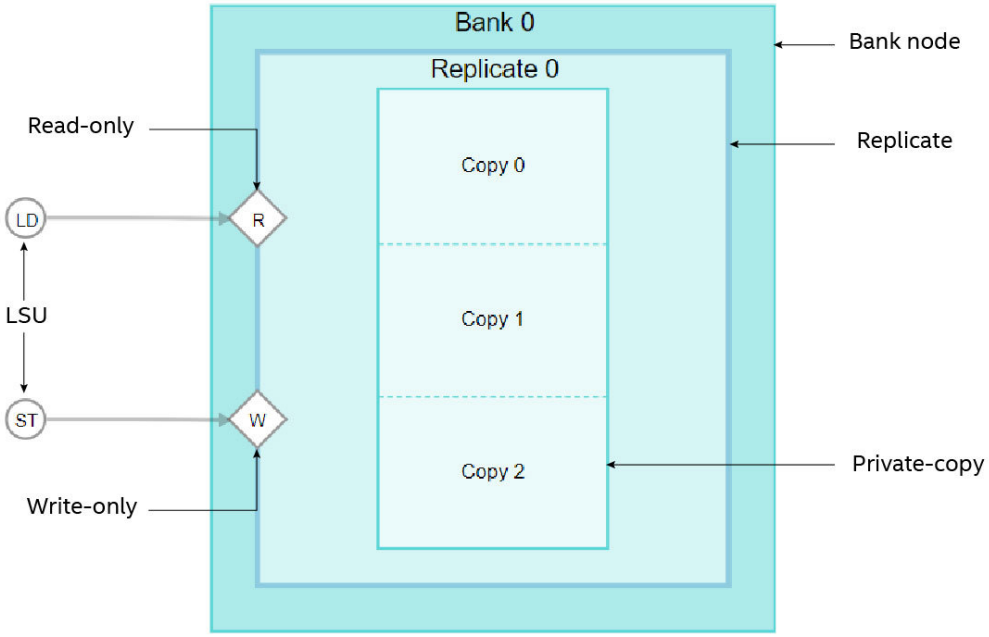


Figure 42. Bank View of a Memory Bank in the Kernel Memory Viewer Pane



Code View

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

Details

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

- Width and depths of memory banks
- Memory layout
- Address-bit mapping
- Memory attributes that you specified in your source code

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

2.1.1.6 Schedule Viewer

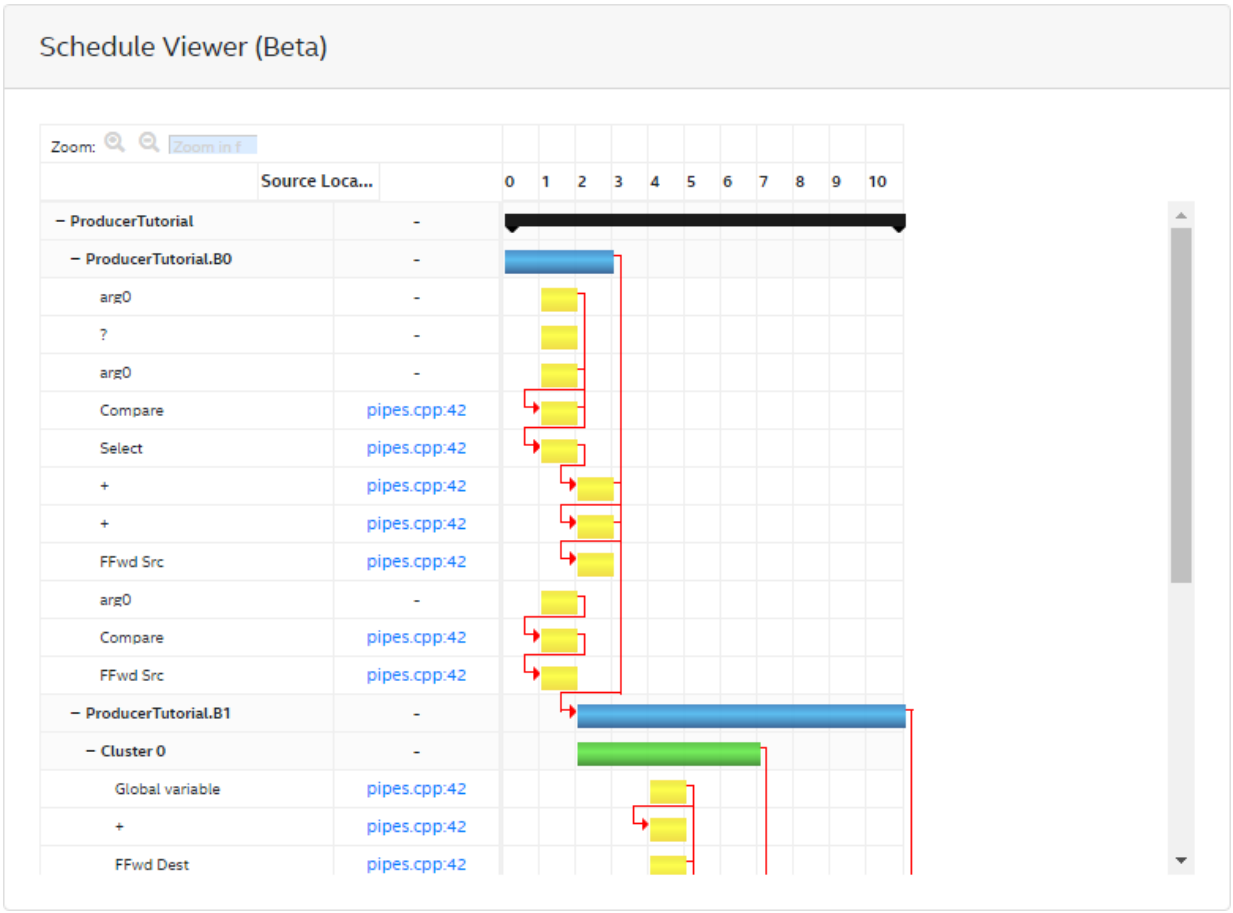
The Schedule Viewer displays a static view of the scheduled cycle and latency of a clustered group of instructions in your design. Use this report to view loop bottlenecks such as f_{MAX}/II bottlenecks, memory dependency, and occupancy limiter.

Access the Schedule Viewer by selecting **Views ➢ Schedule Viewer** in the `report.html` file.

In the Schedule Viewer:

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

Figure 43. Schedule Viewer



2.1.2 Access HLD FPGA Reports in JSON Format

In addition to the `report.html` file, SYCL* also provides the HLD FPGA report data in JSON files.

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

Table 7. JSON Files in the `<project_dir>/reports/lib/json` Directory

File	Description
`area.json`	Area Analysis of System
`block.json`	Block view of the System Viewer
`bottleneck.json`	Bottleneck view of the Loop Analysis report
`gmv.json`	Global memory view of the System Viewer
`info.json`	Summary of project name, compilation command, versions, and timestamps

continued...

File	Description
`loops.json`	Navigation tree of the Loop Analysis report
`loops_attr.json`	Loop Analysis
`mav.json`	System view of the System Viewer
`new_lmv.json`	Kernel Memory Viewer
`pipeline.json`	Cluster view of the System Viewer
`quartus.json`	Intel® Quartus® Prime compilation summary
`schedule.json`	Schedule Viewer
`summary.json`	Kernel compilation name mapping
`tree.json`	Navigation tree of the System Viewer
`warnings.json`	Compilation warning messages

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

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

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

```
"name":"<block name|Kernel: kernel name>  # Find the loops which do not begin
with "Kernel:"
"data":[<Yes|No>, <#|n/a>, <II|n/a>]      # The data field corresponds to
"Pipelined", "II", "Bottleneck"
```

2.2 Analyze the FPGA Image

An FPGA image provides the actual FPGA resource utilization of your design (ALMs, DSPs, and so on) and can be run on FPGA hardware to collect performance measurements for further analysis.

2.2.1 Quartus (Static) Summary

After you generate the FPGA image, sections that summarize the compilation results are populated in the `report.html`. The following sections appear on the **Summary** page:

- **Quartus Fit Clock Summary**: Shows the maximum clock frequencies that you can achieve for the design.
- **Quartus Fit Resource Utilization Summary**: Shows the total area utilization both for the entire design and for each kernel individually. There is no breakdown of area information by source line.

Compile the entire design in the same folder. The **Summary** includes everything, as shown in the following report:

Clock Frequency Summary

	Quartus Fitter: Clock Frequency (MHz)	Compile Target Frequency (MHz)	Compile Estimated Frequency (MHz)
Kernel clock	283.00	240.00	240.00
Clock 2x	566.00		

System Resource Utilization Summary

Quartus Fitter Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
k0_ZTS20ProducerBeforeKernel		1446.1		2635	2	13	0
k1_ZTS20ConsumerBeforeKernel		3908.3		7575	59	0	3
k2_ZTS19ProducerAfterKernel		2200.8		3854	13	13	2
k3_ZTS19ConsumerAfterKernel		2992.6		5608	43	0	1
Quartus Fitter: Kernel System		16591.7		36565	121	118	6
Device		427200		1708800		2713	1518

Compile Estimated Kernel Resource Utilization Summary

Name	Source Location	ALM	ALUT	REG	MLAB	RAM	DSP
ConsumerAfterKernel			2309	5790	53	0	1
ProducerAfterKernel			2515	4062	25	13	2
ConsumerBeforeKernel			3158	7252	69	0	3
ProducerBeforeKernel			1793	2972	14	13	0
Global Interconnect			2295	2568	0	61	0
System description ROM			0	67	0	2	0
Pipe resources			22	204	4	0	0
Compile Estimated: Kernel System			12092	22915	165	89	6

Warnings Summary: No Warnings

2.2.1.1 Timing Failures

If your FPGA compile fails to meet timing requirements, the Intel® oneAPI DPC++/C++ Compiler prints an error message and returns an error code. This means that the generated FPGA image does not meet all timing constraints. The best solution is usually to recompile with a different seed (see `-Xsseed=<value>` in the *Intel® oneAPI Programming Guide*). However, some rare designs where the FPGA is

extremely full might require sweeping several seeds to find one that passes the timing checks. If your design has chronic timing failures and you cannot resolve with seed sweeps, consult your BSP vendor.

When a timing failure happens, the compiler generates a `*.failing_clocks.rpt` file. The path to this file and file name are dependent on your BSP. This `.rpt` file lists which clocks in your design had failing paths and the magnitude of failures.

CAUTION

If the magnitude of the failures is very small (a few ps up to a few tens of ps), your image might be safe to use for limited testing. However, remember that **ANY** timing failure means the resultant FPGA image is not guaranteed to work and could result in unpredictable failures. You must use an image with timing failures only for limited internal testing under lab conditions and never deployed to the field.

2.2.2 Intel® FPGA Dynamic Profiler for DPC++

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

2.2.2.1 Measure Kernel Performance

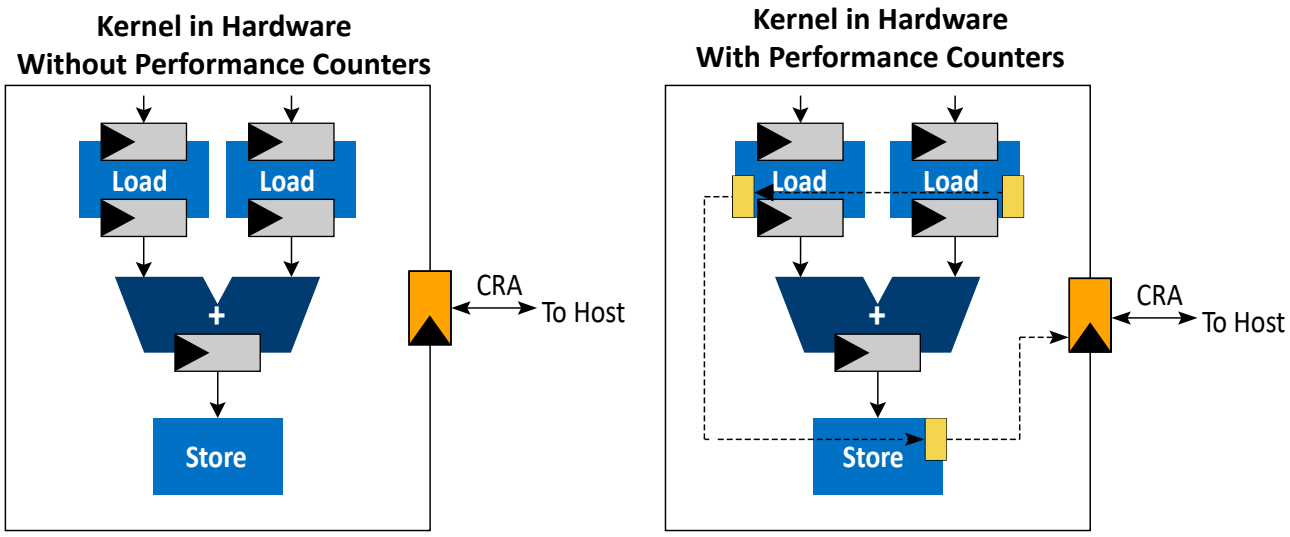
The Profiler instruments and connects performance counters in a daisy chain throughout the pipeline generated for the kernel program. The host then reads data collected by these counters. For example, in PCI Express® (PCIe®)-based systems, the host reads the Profiler data over the PCIe interface.

Consider the following SYCL example code:

```
// Vector Add Kernel
h.single_task<VectorAdd>([=]() {
  for (int i = 0; i < kSize; ++i) {
    r[i] = a[i] + b[i];
  }
});
```

The profiler instruments the pipeline created from this design as shown in Figure 44 on page 66. Performance counters are added to each load and store instruction, which are hooked together in a daisy chain that connects to the CRA interface.

Figure 44. Intel® FPGA Dynamic Profiler for DPC++: Performance Counters Instrumentation



Applications that use many pipes or memory accesses might stall frequently to enable the completion of memory transfers. The dynamic profiler collects various performance metrics such as stall, occupancy, idle, and bandwidth data at these points in the pipeline to help identify memory or pipe operations that create stalls.

2.2.2.2 Instrument the Kernel Pipeline with Performance Counters (`-Xsprofile`)

Enable profiling during design compilation to add profiling counters to the SYCL kernel pipeline. To instrument the SYCL code with performance counters, add the `-Xsprofile` flag to your compiler command.

NOTES

- Instrumenting the design with performance counters increases hardware resource utilization (that is, increases FPGA area use) and typically decreases performance.
- Ensure that all kernel names are unique so that the dynamic profiler interprets the results correctly.

CAUTION

In large designs, the overhead from profiling can cause f_{MAX} degradation. It may also prevent the design from fitting on the chip due to the area overhead of the profiling counters.

2.2.2.3 Obtain Profiling Data During Runtime

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

- Use the Profiler Runtime Wrapper from the command line to obtain the data. This data can later be imported into the Intel® VTune™ Profiler as described in Import FPGA Data collected with Profiler Runtime Wrapper. For more information about the Profiler Runtime Wrapper, refer to Invoke the Profiler Runtime Wrapper to Obtain Profiling Data on page 67.
- Run your host application in Intel® VTune™ Profiler using the **CPU/FPGA Interaction** view. For more information about how to configure and run your host application, refer to CPU/FPGA Interaction Analysis and Use Intel® VTune™ Profiler on page 68.

2.2.2.3.1 Invoke the Profiler Runtime Wrapper to Obtain Profiling Data

After compiling your SYCL* program using the Intel® oneAPI DPC++/C++ Compiler, you can profile your FPGA design using the Profiler Runtime Wrapper. The Profiler Runtime Wrapper calls your executable and collects profile information at a given sample rate. The performance counter data is saved in a `profile.mon` monitor description file that the Profiler Runtime Wrapper post-processes and outputs into a readable `profile.json` file. You are encouraged to use the `profile.json` for further data processing instead of the `profile.mon` file. However, both are available for use after host execution completes.

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

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

where:

- `[options]` are any additional flags you want to pass to the wrapper. Refer to `aocl profile –help` for a list of options and their uses.
- `/path/to/executable` is the path to the executable generated by the compiler.
- `[executable options]` are any options or arguments that need to be passed along to the executable.

CAUTION

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

Split the Execution and Data Post-Processing

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

- The `--no-json` flag only runs your executable and produces a `profile.mon` monitor file without post-processing it.

- The `--no-run <path to profile.mon file>` flag does not invoke your executable and instead just calls the post-processing step on the supplied `profile.mon` file.

Temporal Performance Collection

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

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

  CAUTION

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

- To turn off temporal profiling and instead collect performance data only once a kernel has finished executing, you can set the Profiler Runtime Wrapper's `-no-temporal` flag.

  NOTE

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

2.2.2.3.2 Use Intel® VTune™ Profiler

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

You can use the **CPU/FPGA Interaction View** in the Intel® VTune™ Profiler to determine performance information about your design in various graphical representations. You can view the following:

- Summarized or average data about your SYCL* kernels.
- A graphical representation of the overall kernel program execution process, including both host and device side events.
- Detailed statistics about memory and pipe accesses in both a source view format and timeline format.

The following tables describe types of performance data and information available in the **CPU/FPGA Interaction View** in the Intel® VTune™ Profiler:

Table 8. Types of Performance Data

Column	Description	Access Type
Attributes	Memory or pipe 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 pipe accesses
Stall%	Percentage of time the memory or pipe access is causing pipeline stalls. It is a measure of the ability of the memory or pipe access to fulfill an access request.	All memory and pipe accesses
Occupancy%	Percentage of the overall profiled period when a valid work-item executes the memory or pipe instruction.	All memory and pipe 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

Related Links

Analyzing CPU and FPGA (Intel® Arria® 10 GX) Interaction

Profiling an FPGA-driven SYCL* Application

Interpret Performance Counter Data

Profiling information helps in identifying poor memory or pipe behaviors that lead to unsatisfactory kernel performance.

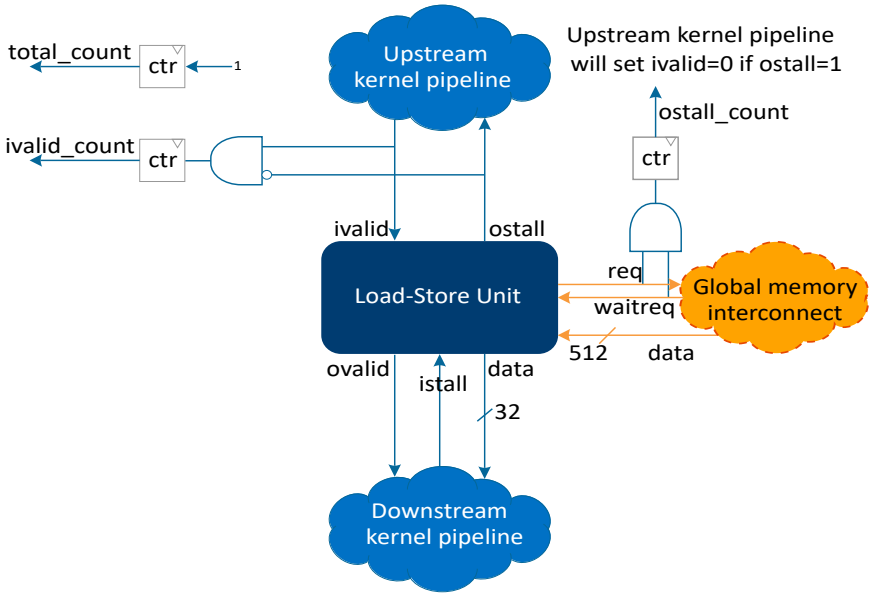
The following sections explain the Profiler metrics that the Profiler reports record.

Stall, Occupancy, Bandwidth

The **CPU/FPGA Interaction View** shows stall percentage, occupancy percentage, and average memory bandwidth for specific lines of kernel code. For definitions of stall, occupancy, data transfer size, and bandwidth, refer to Table 8 on page 69.

SYCL* generates a pipeline architecture where work-items traverse through the pipeline stages sequentially. For more information, refer to Pipelining on page 27.

Figure 45. Simplified Representation of a Kernel Pipeline Instrumented with Performance Counters



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

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

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

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

NOTE

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

Ideal kernel pipeline conditions:

- Stall percentage equals 0%
- Occupancy percentage equals 100%

- Bandwidth equals the board's bandwidth

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

The Profiler reports a high occupancy percentage if the compiler generates a highly efficient pipeline from your kernel, where work-items or iterations are moving through the pipeline stages without stalling. The following notes may be helpful when interpreting profiling data:

- If all LSUs are accessed the same number of times, they have the same occupancy value.
- If work-items cannot enter the pipeline consecutively, bubbles are inserted into the pipeline.
- In loop pipelining, loop-carried dependencies also form bubbles in the pipeline because of bubbles that exist between iterations. For more information about bubbles, refer to Occupancy.
- If an LSU is accessed less frequently than other LSUs, such as when an LSU is outside a loop that contains other LSUs, this LSU has a lower occupancy value than the other LSUs. This rule also applies to pipes.

Stalling Pipes

Pipes provide a point-to-point communication link between two kernels.

Stalls occur if there is an imbalance between the read and write sides of the pipe 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 `ProducerToConsumerPipe::write` call in the write kernel.

2.2.2.4 Reduce Area Resource Use While Profiling

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

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

NOTE

You must invoke the Profiler Runtime Wrapper only once.

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

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

   If you do not include the `-shared-counters` flag, your design is run only once. So, you lack data for everything after the first shared signal.

CAUTION

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

2.2.2.5 Profiler Analyses of Example SYCL* Design Scenarios

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

High Stall Percentage

A high stall percentage implies that the memory or pipe instruction cannot fulfill the access request because of contention for memory bandwidth or pipe 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 access for possible improvements.

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

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

Low Occupancy Percentage

A low occupancy percentage implies that a work-item is accessing the load and store operations or the pipe infrequently. This behavior is expected for load and store operations or pipes that are in non-critical loops. However, if the memory or pipe 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:

```
queue_event = deviceQueue.submit([&](handler &cgh) {
   cgh.single_task<class LowOccupancyPercentageEx>([=]() {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < 1000; j++){
```

```
            Pipe1::write(data1);
         }
         for (int k =0; k < 3; k++) {
            Pipe2::write(data2);
         }
      }
   });
});
```

Assuming all loops are pipelined, the first inner loop with a trip count of 1000 is the critical loop. The second inner loop with a trip count of three is executed infrequently. As a result, you can expect that the occupancy and activity percentages for `Pipe1` are high and for `Pipe2` are low.

In addition, 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 generally empty for the duration of kernel execution, which leads to poor performance.

High Stall and High Occupancy Percentages

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

NOTE

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

Usually, the sum of stall and occupancy percentages approximately equals 100%. If a load and store operation or pipe has a high stall percentage, it means that the load and store operation or pipe 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 `-Xsno-interleaving=default` 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 with more bandwidth (for example, a board with three DDRs instead of two DDRs).

Solution for stalling local load and store operations:

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

Solutions for stalling pipes:

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

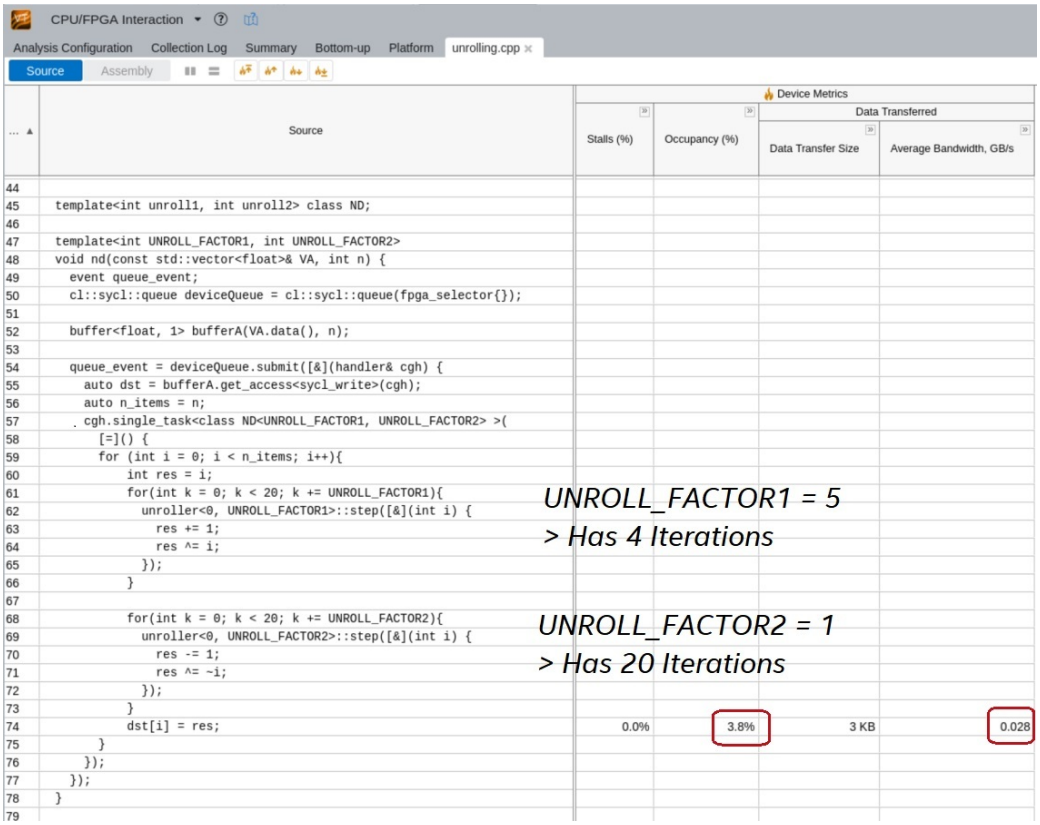
No Stalls, Low Occupancy Percentage, and Low Bandwidth

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

NOTE

An ideal kernel pipeline has a bandwidth that equals the board's available bandwidth.

Figure 46. Example SYCL 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.

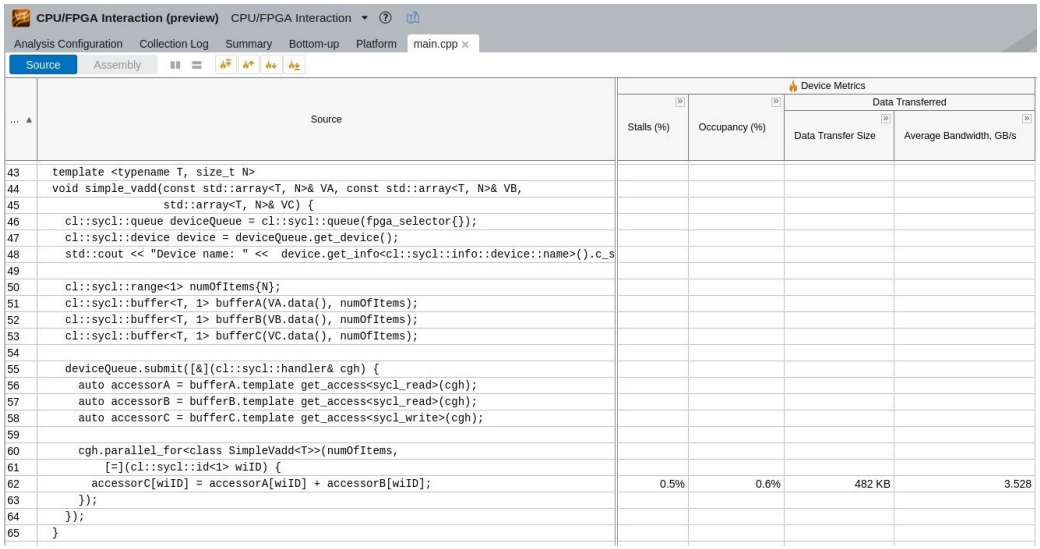
Solutions for resolving loop bottlenecks:

- Unroll the `FACTOR1` and `FACTOR2` loops evenly. Simply unrolling `FACTOR1` loop further does not resolve the bottleneck.
- Vectorize your kernel to allow multiple work-items to execute during each loop iteration.

No Stalls, High Occupancy Percentage, and Low Bandwidth

The structure of a kernel design might prevent it from leveraging all the available bandwidth that the accelerator board can offer.

Figure 47. Example SYCL 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 GB/s, which is 14% of the available bandwidth. To increase the bandwidth, increase the number of tasks performed in each clock cycle.

Solutions for low bandwidth:

- Automatically or manually, vectorize the kernel to make wider requests.
- Unroll the innermost loop to make more requests per clock cycle.
- Delegate some of the tasks to another kernel.

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 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 likely not a cause for concern.

2.2.2.6 Limitations

The Intel® FPGA Dynamic Profiler for DPC++ has some limitations:

- Profile data is not persistent across SYCL* programs or multiple devices.
- The Profiler is unique to each SYCL program and device, meaning, each program on each device has its own profiler information. If your host swaps a new kernel program in and out of the FPGA, the Profiler does not save any data.
- Profile data is not saved on a device for later profiling.

- All profiling data is read to the host during execution and is only stored on the device long enough to be read on the next readback. Any reprogramming of new designs or restarting the same design results in new profiling data, erasing any previous data that may have existed.
- Instrumenting the design with performance counters increases hardware resource utilization (that is, FPGA area use) and typically decreases performance.

2.2.3 System-level Profiling Using the Intercept Layer for OpenCL* Applications

The Intercept Layer for OpenCL* Applications is an open-source tool that you can use to profile oneAPI designs at a system-level. Although it is not part of the Intel® oneAPI Base Toolkit installation, it is freely available on GitHub*.

This tool serves the following purpose:

- Intercept host calls before they reach the device to gather performance data and log host calls.
- Provide data to visualize the calls through time and separate them into queued, submitted, and execution sections to better understand the execution.
- Identify gaps (using visualization) in the runtime that may be leading to inefficient execution and throughput drops.

NOTE

The Intercept Layer for OpenCL* Applications tool has a different purpose than the Intel® FPGA Dynamic Profiler for DPC++, which provides information about the kernels themselves and helps optimize the hardware. Together, you can use these tools to optimize both host and device-side execution.

The Intercept Layer has different options for capturing different aspects of the host run, and these options are described in its documentation. Call-logging and device timeline features are used to print information about the calls made by the host during execution.

You can view visualizations of this data in the following methods:

- Use JSON files generated by the Intercept Layer for OpenCL Applications that contain device timeline information. You can open these JSON files in the Google* Chrome trace event profiling tool, which provides a visualization of the data.
- Use the Intercept Layer for OpenCL Applications' python script that parses the timeline information into a Microsoft* Excel file, where it is presented both in a table format and in a bar graph.

Use the visualized data to identify gaps in the runtime where events are waiting for something else to finish executing. While it is not possible to eliminate all the gaps, you might be able to eliminate gaps caused by dependencies that can be avoided.

For example, double-buffering allows host-data processing and host transfers to the device-side buffer to occur in parallel with the kernel execution on the FPGA device. This parallelization is useful when the host performs any combination of the following actions between consecutive kernel runs:

- Preprocessing

- Postprocessing
- Writes to the device buffer

By running host and device actions in parallel, execution gaps between kernels are removed as they no longer have to wait for the host to finish its operation. You can clearly see the benefits of double-buffering with the visualizations provided by the Intercept Layer output.

NOTE

For additional information, refer to the FPGA tutorial sample "Using the OpenCL Intercept Layer to Profile" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

2.2.3.1 Set Up the Intercept Layer for OpenCL* Applications

The Intercept Layer for OpenCL* Applications is available on GitHub* at https://github.com/intel/opencl-intercept-layer

To set up the Intercept Layer for OpenCL Applications, perform the following steps:

1. Download Intercept Layer for OpenCL Applications version 2.2.1 or later from GitHub* at the following URL:

   https://github.com/intel/opencl-intercept-layer
2. Build the Intercept Layer according to the instructions provided in How to Build the Intercept Layer for OpenCL* Applications.
3. Ensure that you have set `ENABLE_CLILOADER=1` when running `cmake` command. For example, run `cmake -DENABLE_CLILOADER=1 ...`
4. Run the `make` command in the build directory. This step builds the `cliloader` loader utility.

   The `cliloader` executable should now exist in the `<path to opencl-intercept-layer-master download>/<build dir>/cliloader/` directory.
5. Add the directory to your `PATH` environment variable if you want to run multiple designs using `cliloader`.

   You can now pass your executables to `cliloader` to run them with the intercept layer. For details about the `cliloader` loader utility, see cliloader: A Intercept Layer for OpenCL* Applications Loader.
6. Set `cliloader` and other Intercept Layer options.

   If you run multiple designs with the same options, set up a `clintercept.conf` file in your home directory. You can also set the options as environment variables by prefixing the option name with `CLI_`. For example, the `DllName` option can be set through the `CLI_DllName` environment variable. For a list of options, see *Controls* in How to Use the Intercept Layer for OpenCL Applications.

Option/Variable	Description
`DllName=$CMPLR_ROOT/linux/lib/libOpenCL.so`	The intercept layer must know where `libOpenCL.so` file from the original oneAPI build is.
`DevicePerformanceTiming=1` and `DevicePerformanceTimelineLogging=1`	These options print out runtime timeline information in the output of the executable run.
`ChromePerformanceTiming=1`, `ChromeCallLogging=1`, `ChromePerformanceTimingInStages=1`	These variables set up the chrome tracer output and ensure the output has Queued, Submitted, and Execution stages.

These instructions set up the `cliloader` executable, which provides some flexibility by allowing for more control over when the layer is used or not used. If you prefer a local installation (for a single design) or a global installation (always ON for all designs), follow the instructions at How to Install the Intercept Layer for OpenCL Applications.

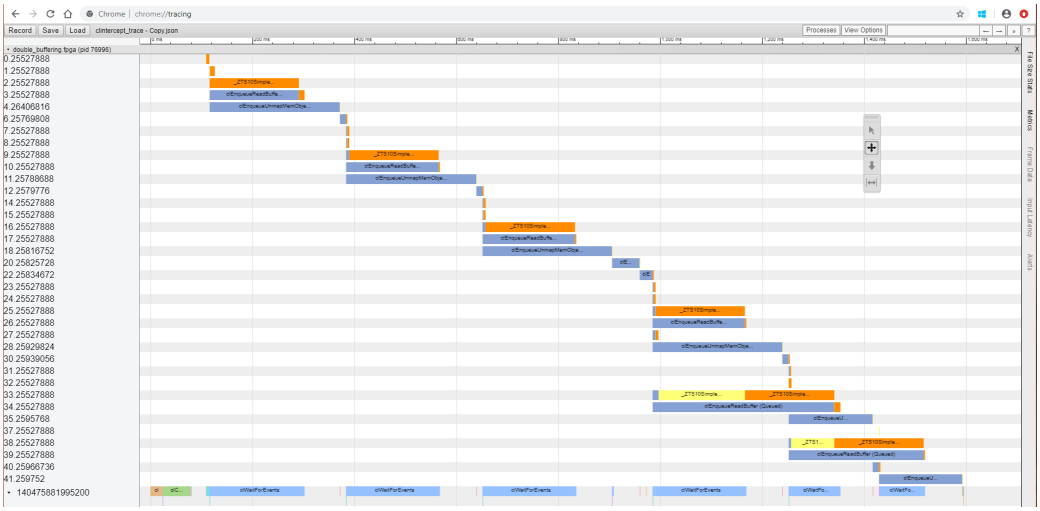
When you run the host executable with `cliloader <executable> [executable args]` command, the `stderr` output contains lines as shown in the following example:

```
Device Timeline for clEnqueueWriteBuffer (enqueue 1) = 63267241140401 ns
(queued),
63267241149579 ns (submit), 63267241194205 ns (start), 63267242905519 ns (end)
```

These lines give the timeline information about a variety of oneAPI runtime calls. After the host executable finishes running, there is also a summary of the performance information for the run. After the executable runs, the data collected is placed in the `CLIntercept_Dump` directory, which is in the home directory by default. Its location can be adjusted using the `DumpDir=<directory where you want the output files> cliloader` option. The `CLIntercept_Dump` directory contains a file called `clintercept_trace.json`. You can load this JSON file in the Google* Chrome trace event profiling tool (`chrome://tracing/`) to visualize the timeline data collected by the run.

The following is a sample visualization of timeline data:

Figure 48. OpenCL Intercept Layer Full Example Trace



This visualization shows different calls executed through time. The X-axis is time, with the scale shown near the top of the page. The Y-axis shows different calls that are split up in several ways.

The left side (Y-axis) has two different types of numbers:

- Numbers that contain a decimal point.
  - The part of the number before the decimal point orders the calls approximately by start time.
  - The part of the number after the decimal point represents the queue number the call was made in.
- Numbers that do not contain a decimal point. These numbers represent the thread ID of the thread being run on in the operating system.

The colors in the trace represent different stages of execution:

- Blue during the queued stage.
- Yellow during the submitted stage.
- Orange for the execution stage.

Identify gaps between consecutive execution stages and kernel runs to identify possible areas for optimization.

For an example use of Intercept Layer for OpenCL Applications, see Applying Double-Buffering Using the Intercept Layer for OpenCL* Applications on page 133.

3.0 Optimize Your Design

This chapter describes features and provides guidance on leveraging the functionalities of SYCL to optimize your designs.

In general, the methods you use to improve the performance of your 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 kernels can perform in hardware.

3.1 Throughput

For most designs, you can achieve high performance by optimizing the throughput of the hardware for a given algorithm. This section describes techniques to improve the design's throughput.

3.1.1 Single Work-item Kernels

When a kernel describes a single work item, the SYCL* host can execute the kernel as a single task. The SYCL* specification version 1.2.1 describes this mode of operation as task-parallel programming. A task is equivalent to a kernel executed with one workgroup that contains one work item. For more information, refer to Concepts of Hardware Design and Mapping Parallelism Models to Hardware.

3.1.1.1 Single Work-item Kernel Design Guidelines

If your kernels contain loop structures, follow the Intel®-recommended guidelines to construct the kernels in a way that allows the Intel® oneAPI DPC++/C++ Compiler to analyze them effectively. Well-structured loops are particularly important to aid the compiler in generating a pipeline parallel datapath for loops.

Avoid Pointer Aliasing

If your kernels have pointer arguments, you can improve the throughput of the design if the Intel® oneAPI DPC++/C++ Compiler can prove those arguments never point to the same memory location. It is possible to provide the compiler with information about pointer arguments in kernels. For more information, refer to Ignoring Dependencies Between Accessor Arguments on page 114.

Construct "Well-Formed" Loops

A *well-formed* loop has an exit condition that compares against an integer bound and has a simple induction increment. Including *well-formed* loops in your kernel improves performance because the Intel® oneAPI DPC++/C++ Compiler can analyze these loops efficiently.

The following example is a *well-formed* loop:

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

NOTE

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 following loop structure creates a loop-carried dependence because each loop iteration reads data written by the previous iteration:

```
for (int i = 0; i < N; i++) {
  A[i] = A[i - 1] + i;
}
```

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 Intel® oneAPI DPC++/C++ Compiler can achieve, which reduces kernel performance.

The Intel® oneAPI DPC++/C++ Compiler performs a static memory dependence analysis on loops to determine the extent of parallelism that it can achieve. In some cases, the Intel® oneAPI DPC++/C++ Compiler might assume loop-carried dependence:

- Between two array accesses and as a result, extract less pipeline parallelism.
- If it cannot resolve the dependencies at compilation time because of unknown variables or complex indexing expressions.

To minimize loop-carried dependencies, follow these guidelines 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, affine array indexes.** Avoid the following types of complex array indexes because the Intel® oneAPI DPC++/C++ Compiler cannot analyze them effectively, which might lead to suboptimal compiler output:

— Non-constants 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 Intel® oneAPI DPC++/C++ Compiler can analyze the array index `A[i][j]` effectively because the index variables are in different subscripts.

Avoid Complex Loop Exit Conditions

The Intel® oneAPI DPC++/C++ Compiler evaluates exit conditions to determine if subsequent loop iterations can enter the loop pipeline. Occasionally, the Intel® oneAPI DPC++/C++ Compiler requires memory accesses or complex operations to evaluate the exit condition. In these cases, subsequent iterations cannot launch until the evaluation completes, decreasing the overall loop performance.

Convert Nested Loops into a Single Loop

To maximize performance, combine nested loops into a single form whenever possible. Restructuring nested loops into a single loop reduces hardware footprint and computational overhead between loop iterations.

The following code examples illustrate the conversion of a nested loop into a single loop:

Table 10. 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`
`  }`
`  //statements`
`}`
	`for (i = 0; i < N*M; i++) {`
	`  //statements`
	`}`

Avoid Conditional Loops

To maximize performance, avoid declaring conditional loops. Conditional loops are tuples of loops that are declared within conditional statements such that one and only one of the loops is expected to be reached. These loops cannot be efficiently parallelized and result in a serialized implementation.

The following code examples illustrate the conversion of conditional loops to a more optimal implementation:

Table 11. Conversion of a Conditional Loop to an Optimized Loop

Conditional Loops	Converted Loop

`if (condition) {`
`  for (int i = 0; i < m; i++) {`
`  // statements`
`  }`
`}`
	`for (int i = 0; i < m; i++) {`
	`  if (condition) {`
	`  // statements`
	`  }`
	`  else {`

Conditional Loops	Converted Loop

`else {`
`  for (int i = 0; i < m; i++) {`
`  // statements`
`  }`
`}`
	`  // statements`
	`  }`
	`}`

Declare Variables in the Deepest Scope Possible

To reduce 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 use because the Intel® oneAPI DPC++/C++ Compiler does not need to preserve the variable data across loops that do not use 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 use, 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 resources necessary to present the variable.

3.1.1.2 Loops

The Intel® oneAPI DPC++/C++ Compiler attempts to maximize the occupancy of the datapath of a loop within a task kernel by executing iterations in a pipeline parallel method. The following sections provide guidelines and describe techniques for writing loops in task kernels such that the Intel® oneAPI DPC++/C++ Compiler can best extract pipeline parallelism from these loops.

3.1.1.2.1 Refactor the Loop-Carried Data Dependency

Based on the feedback from the optimization report, you can restructure your program to reduce the critical path of a pipelined loop by reducing the distance between the first time you use a loop-updated variable in the loop body and the last time you define it within a single iteration.

Consider the following code:

```
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  accessor A(A_buf, cgh, read_only);
  accessor B(B_buf, cgh, read_only);
  accessor Result(Result_buf, cgh, write_only);
  cgh.single_task<class unoptimized>([=]() {
```

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

- The report indicates that the Intel® oneAPI DPC++/C++ Compiler successfully infers pipelined execution for the outer loop, and a new loop iteration launches every other cycle.
- The only a single loop iteration will execute message in the first row of the Details pane indicates that the loop executes a single iteration at a time across the subloop because of the 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. Moreover, the serialization is enforced by a critical path spanning from the first use of sum in the body of the loop i to the last definition of sum at the end of the body of loop j.
- The second row of the report notifies you that the inner loop executes in a pipelined manner with no performance-limiting loop-carried dependencies.

NOTE

For recommendations on how to structure your single work-item kernel, refer to Single Work-item Kernel Design Guidelines on page 80.

To optimize performance of this kernel, reduce the length of the critical path induced by 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.

Following code illustrates the restructured `optimized` kernel snippet:

```
constexpr int N = 128;
 queue.submit([&](handler &cgh) {
   accessor A(A_buf, cgh, read_only);
   accessor B(B_buf, cgh, read_only);
   accessor Result(Result_buf, cgh, write_only);
   cgh.single_task<class optimized>([=]() {
     int sum = 0;
     for (int i = 0; i < N; i++) {
       // Step 1: Definition
       int sum2 = 0;
       // Step 2: Accumulation of array A values for one outer
       // loop iteration
       for (int j = 0; j < N; j++) {
          sum2 += A[i * N + j];
       }
```

```
      // Step 3: Addition of array B value for an outer loop iteration
      sum += sum2;
      sum += B[i];
    }
    Result[0] = sum;
  });
});
```

3.1.1.2.2 Relax Loop-Carried Dependency

Example 1: Multiply Chain

Based on the feedback from the optimization report, you can relax a loop-carried dependency by allowing the compiler to infer a shift register and increase the dependence distance. Increase the dependence distance by increasing the number of loop iterations that occur between the generation of a loop-carried value and its use.

Consider the following code example:

```
constexpr int N = 128;
 queue.submit([&](handler &cgh) {
   accessor A(A_buf, cgh, read_only);
   accessor result(result_buf, cgh, write_only);
   cgh.single_task<class unoptimized>([=]() {
     float mul = 1.0f;
     for (int i = 0; i < N; i++)
       mul *= A[i];

     result[0] = mul;
   });
 });
```

The optimization report shows that the Intel® oneAPI DPC++/C++ 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 8 (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 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 copies to `mul` and write the final value to the result.

The following code illustrates the restructured kernel:

```
constexpr int N = 128;
 constexpr int M = 8;
 queue.submit([&](handler &cgh) {
   accessor A(A_buf, cgh, read_only);
   accessor result(result_buf, cgh, write_only);
   cgh.single_task<class optimized>([=]() {
```

```
    float mul = 1.0f;

    // Step 1: Declare multiple copies of variable mul
    float mul_copies[M];

    // Step 2: Initialize all copies
    for (int i = 0; i < M; i++)
      mul_copies[i] = 1.0f;

    for (int i = 0; i < N; i++) {
      // Step 3: Perform multiplication on the last copy
      float cur = mul_copies[M-1] * A[i];

      // Step 4a: Shift copies
      #pragma unroll
      for (int j = M-1; j > 0; j--)
        mul_copies[j] = mul_copies[j-1];

      // Step 4b: Insert updated copy at the beginning
      mul_copies[0] = cur;
    }

    // Step 5: Perform reduction on copies
    #pragma unroll
    for (int i = 0; i < M; i++)
      mul *= mul_copies[i];

    result[0] = mul;
  });
});
```

Example 2: Accumulator

Similar to Example 1, this example also applies the same technique and demonstrates how to write single work-item kernels that carry out double precision floating-point operations efficiently by inferring a shift register.

Consider the following example:

```
queue.submit([&](handler &cgh) {
  accessor arr(arr_buf, cgh, read_only);
  accessor result(result_buf, cgh, write_only);
  accessor N(N_buf, cgh, read_only);
  cgh.single_task<class unoptimized>([=]() {
    double temp_sum = 0;
    for (int i = 0; i < *N; ++i)
      temp_sum += arr[i];
      result[0] = temp_sum;
  });
});
```

The kernel `unoptimized` is an accumulator that sums the elements of a double precision floating-point array `arr[i]`. For each loop iteration, the Intel® oneAPI DPC++/C++ Compiler takes 10 cycles to compute the result of the addition and then stores it in the variable `temp_sum`. Each loop iteration requires the value of `temp_sum` from the previous loop iteration, which creates a data dependency on `temp_sum`.

To relax the data dependency, infer the array `arr[i]` as a shift register.

The following is the restructured kernel `optimized`:

```
//Shift register size must be statically determinable
constexpr int II_CYCLES = 12;
queue.submit([&](handler &cgh) {
  accessor arr(arr_buf, cgh, read_only);
  accessor result(result_buf, cgh, write_only);
  accessor N(N_buf, cgh, read_only);
  cgh.single_task<class optimized>([=]() {
    //Create shift register with II_CYCLE+1 elements
    double shift_reg[II_CYCLES+1];

    //Initialize all elements of the register to 0
    for (int i = 0; i < II_CYCLES + 1; i++) {
      shift_reg[i] = 0;
    }

    //Iterate through every element of input array
    for(int i = 0; i < N; ++i){
      //Load ith element into end of shift register
      //if N > II_CYCLE, add to shift_reg[0] to preserve values
      shift_reg[II_CYCLES] = shift_reg[0] + arr[i];

      #pragma unroll
      //Shift every element of shift register
      for(int j = 0; j < II_CYCLES; ++j)
      {
        shift_reg[j] = shift_reg[j + 1];
      }
    }

    //Sum every element of shift register
    double temp_sum = 0;

    #pragma unroll
    for(int i = 0; i < II_CYCLES; ++i)
    {
      temp_sum += shift_reg[i];
    }

    result[0] = temp_sum;
  });
});
```

3.1.1.2.3 Transfer Loop-Carried Dependency to Local Memory

Loop-carried dependencies can adversely affect the loop initiation interval or II (refer to Pipelining Across Multiple Work Items on page 34). 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 example:

```
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  accessor A(A_buf, cgh, read_write);
  cgh.single_task<class unoptimized>([=]() {
    for (int i = 0; i < N; i++)
      A[N-i] = A[i];
    }
  });
});
```

Global memory accesses have long latencies. In this example, the loop-carried dependency on the array `A[i]` causes long latency. The optimization report reflects this latency with an II of 185. 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.

Following is the restructured kernel `optimized`:

```
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  accessor A(A_buf, cgh, read_write);
  cgh.single_task<class optimized>([=]() {
    int B[N];
    for (int i = 0; i < N; i++)
      B[i] = A[i];

    for (int i = 0; i < N; i++)
      B[N-i] = B[i];

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

3.1.1.2.4 Minimize the Memory Dependencies for Loop Pipelining

Intel® oneAPI DPC++/C++ 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 workgroup.

Loop dependencies might introduce bottlenecks for single work-item kernels due to latency associated with the memory accesses. The Intel® oneAPI DPC++/C++ Compiler defers a memory operation until a dependent memory operation completes. This could affect the loop initiation interval (II). The Intel® oneAPI DPC++/C++ Compiler indicates the memory dependencies in the optimization report.

To minimize the impact of memory dependencies for loop pipelining:

- Ensure that the Intel® oneAPI DPC++/C++ Compiler does not assume false dependencies.
- When the static memory dependence analysis fails to prove that dependency does not exist, the Intel® oneAPI DPC++/C++ Compiler assumes that a dependency exists and modifies the kernel execution to enforce the dependency. The 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.
  - The Intel® oneAPI DPC++/C++ Compiler can fully resolve the read-after-write (control dependency) operation.

- Override the static memory dependence analysis by adding the line `[[intel::ivdep]]` before the loop in your kernel code if you are sure that it carries no dependencies. For more information, refer to `ivdep` Attribute on page 202

3.1.1.2.5 Unroll Loops

You can control the way the Intel® oneAPI DPC++/C++ Compiler translates SYCL kernel descriptions to hardware resources. If your SYCL kernel contains loop iterations, increase performance by unrolling the loop. Loop unrolling decreases the number of iterations that the Intel® oneAPI DPC++/C++ Compiler executes at the expense of increased hardware resource consumption.

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

```
queue.submit([&](handler &cgh) {
  accessor x(x_buf, cgh, read_only);
  accessor sum(sum_buf, cgh, write_only);
  cgh.single_task<class unoptimzed>([=]() {
    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;
  });
});
```

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

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

The Intel® oneAPI DPC++/C++ Compiler arranges these operations in a pipeline according to the data flow semantics of the SYCL* kernel code. For example, the Intel® oneAPI DPC++/C++ 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 SYCL 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 following code example:

NOTE

Loop unrolling significantly changes the structure of the compute unit that the Intel® oneAPI DPC++/C++ Compiler creates.

```
queue.submit([&](handler &cgh) {
  accessor x(x_buf, cgh, read_only);
  accessor sum(sum_buf, cgh, write_only);
  cgh.single_task<class unoptimzed>([=]() {
    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 Intel® oneAPI DPC++/C++ Compiler to unroll four iterations of the loop completely. To accomplish the unrolling, the Intel® oneAPI DPC++/C++ 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 in every clock cycle after the completion of the initial load operations and additions. The Intel® oneAPI DPC++/C++ Compiler further optimizes this kernel by coalescing the four load operations so that the compute unit can load all 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® oneAPI DPC++/C++ Compiler might fail to meet scheduling because it cannot unroll this nested loop structure easily, resulting in a high II. In this case, the Intel® oneAPI DPC++/C++ Compiler issues the following error message along with the line number of the outermost loop:

```
Kernel <function> exceeded the Max II. The Kernel's resource usage is estimated
to be much larger than FPGA capacity. It 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 load operations from global memory allow the hardware implementation of the kernel to perform more operations per clock cycle.

The Intel® oneAPI DPC++/C++ 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 Intel® oneAPI DPC++/C++ 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 Intel® oneAPI DPC++/C++ Compiler unrolls. Refer to Single Work-item Kernel Design Guidelines on page 80 for tips on constructing well-structured loops.

3.1.1.2.6 Fuse Loops to Reduce Overhead and Improve Performance

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

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

Unfused Loops

```
for (int i = 0 ; i < 10;  i++ ) {
  for(int j = 0 ; j < 300 ; j++){
     a[j] = localMem[j] + 3;
  }
  for(int k = 0 ; k < 300 ; k++){
     b[k] = localMem[k] + 4;
  }
}
```

Fused Loops

```
for (int i = 0; i < 10; i++ ) {
  for(int j = 0 ; j < 300 ; j++){
    int localMemVal = localMem[j];
    a[j] = localMemVal + 3;
    b[k] = localMemVal + 4;
  }
}
```

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

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

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

Fusion Criteria

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

- Loops must be adjacent. That is, you cannot have a statement S_i with side-effects such that S_i executes after L_j and before L_k.
- Each loop must have a single-entry point and a single exit point. For example, loops that contain break statements are not considered for fusion.
- Loops must have no negative-distance dependencies. That is, for loops L_j and L_k where L_j is defined before L_k, iteration m of loop L_k does not depend on values calculated in iteration $m+n$ (where $n>0$) of loop L_j.

Automatic Loop Fusion

The Intel® oneAPI DPC++/C++ Compiler fuses loops with the same trip counts automatically if the compiler analysis of your kernel determines that fusing the loops is profitable.

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

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

The Loop Analysis report indicates when loops were fused.

3.1.1.2.7 Optimize Loops With Loop Speculation

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

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

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

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

The `speculated_iterations` attribute allows you to directly control the number of speculated iterations for a loop. If the exit condition calculation is the bottleneck to lowering II (as shown in the Loop Analysis report), increasing the number of speculated iterations may improve the II (this is not guaranteed as other bottlenecks may be uncovered). For more information, refer to `speculated_iterations` Attribute on page 209.

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

the `speculated_iterations` attribute can reduce this overhead, but be aware that choosing an attribute value that is too low may increase the II (due to not having enough time to evaluate the exit condition).

Consider the following example:

```
while (m*m*m < N) {
  m+=1;
}
dst[0] = m;

[[intel::speculated_iterations(7)]]
while (m*m*m < N) {
  m+=1;
}
dst[0] = m;

[[intel::speculated_iterations(0)]]
while (m*m*m < N) {
  m+=1;
}
dst[0] = m;
```

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

NOTE

For additional information, refer to the FPGA tutorial sample `Speculated Iterations` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

3.1.1.2.8 Remove Loop Bottlenecks

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

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

NOTE

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

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

Simple Loop Example

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

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

Consider a simple loop with memory dependency from the Triangular Loops example:

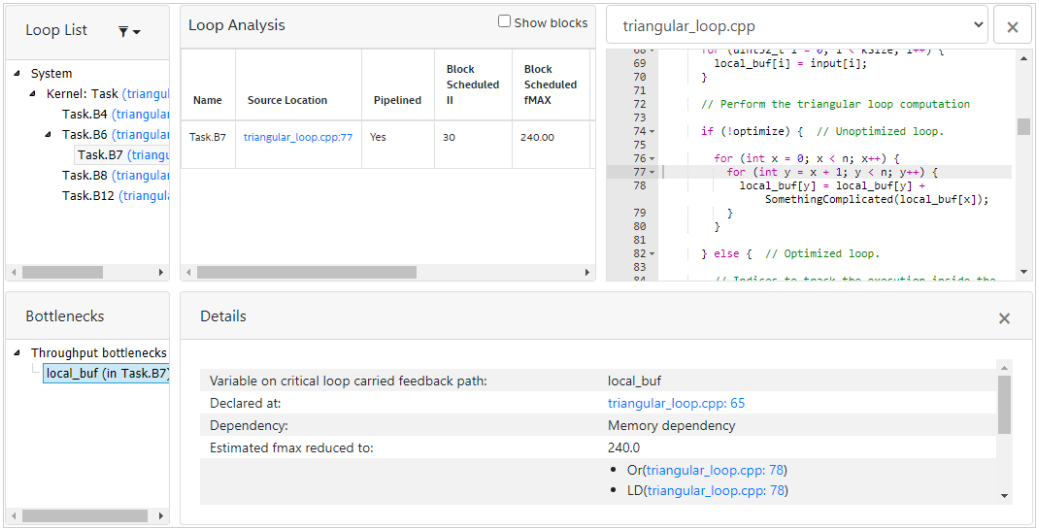
```
for (int y = x + 1; y < n; y++) {
  local_buf[y] = local_buf[y] + SomethingComplicated(local_buf[x]);
}
```

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

Loop Analysis — Show blocks

Name	Source Location	Pipelined	Block Scheduled II	Block Scheduled fMAX	Latency	Speculated Iterations	Max Interleaving Iterations	Brief Info
Task.B7	triangular_loop.cpp:77	Yes	30	240.00	36.000000	1	1	Memory dependency

The `for` loop on line 77 has a latency of 36, maximum interleaving iterations of 1, and initiation interval of 30. So, the maximum concurrency is ~1 ((Latency of 36 x Maximum interleaving iteration of 1) / II of 30). The bottleneck causing this low maximum concurrency results from a loop carried dependency caused by a memory dependency. This is reported in the Bottlenecks Viewer on page 45 as shown in the following image:



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

```
int res = N;
for (int i = 0; i < N; i++) {
  res += 1;
  res ^= i;
}
```

Nested Loop Example

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

```
01 void TriangularLoop(std::unique_ptr<queue>& q, buffer<uint32_t>& input_buf,
02                     buffer<uint32_t>& output_buf, uint32_t n, event& e,
03                     bool optimize) {
04
05   e = q->submit([&](handler& h) {
06
07     accessor input(input_buf, h, read_only);
08     accessor output(output_buf, h, write_only, no_init);
09
10     h.single_task<Task>([=]() [[intel::kernel_args_restrict]] {
11
12       const int real_iterations = (n * (n + 1) / 2 - 1);
13       const int extra_dummy_iterations = (kM - 2) * (kM - 1) / 2;
14       const int loop_bound = real_iterations + extra_dummy_iterations;
15
16       uint32_t local_buf[kSize];
17
18       for (uint32_t i = 0; i < kSize; i++) {
19         local_buf[i] = input[i];
20       }
21
22       if (!optimize) {  // Unoptimized loop.
23
24         for (int x = 0; x < n; x++) {
25           for (int y = x + 1; y < n; y++) {
26             local_buf[y] = local_buf[y] + SomethingComplicated(local_buf[x]);
27           }
28         }
```

In this example, the bottleneck is resulted from a loop carried dependency caused by a memory dependency on the variable `local_buf`. The `local_buf` variable must finish loading in the loop on line 27 before the next outer loop (line 24) iteration is launched. Therefore, the maximum concurrency of the outer loop is 1. This information is reported in the details sections of the Loop Analysis and Schedule Viewer reports.

Addressing Bottlenecks

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

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

- `[[intel::initiation_interval(n)]]` (See `initiation_interval` Attribute on page 201)
- `[[intel::ivdep(safelen)]]` (See `ivdep` Attribute on page 202)

- `[[intel::max_concurrency(n)]]` (See `max_concurrency` Attribute on page 205)
- `[[intel::private_copies(N)]]` (See Memory Attributes on page 197)

Consider the previous Simple Loop Example where the concurrency is 1 and the initiation interval is 30. Applying the `[[intel::ivdep(M)]]` attribute, as shown in the following code snippet, comes at an expense of lowered predicted f_{MAX} from 240 MHz to 194.4 MHz. The original nested loop is manually coalesced or merged into a single loop. Since the loop is restructured to ensure that a minimum of `M` iterations is executed, the `[[intel::ivdep(M)]]` is used to inform the compiler that at least `M` iterations always separate any pair of dependent iterations. This new modified loop now has a maximum concurrency of 35 ((Latency of 35 x Maximum interleaving iteration of 1) / II of 1) and an initiation interval of 1.

```
// Indices to track the execution in the merged loop
int x = 0, y = 1;

// Total iterations of the merged loop
const int loop_bound = TotalIterations(M, n);

[[intel::ivdep(M)]]
for (int i = 0; i < loop_bound; i++) {

  // Determine if this is a real or dummy iteration
  bool compute = y > x;
  if (compute) {
    local_buf[y] = local_buf[y] + SomethingComplicated(local_buf[x]);
  }

  y++;
  if (y == n) {
    x++;
    y = Min(n - M, x + 1);
  }
}
```

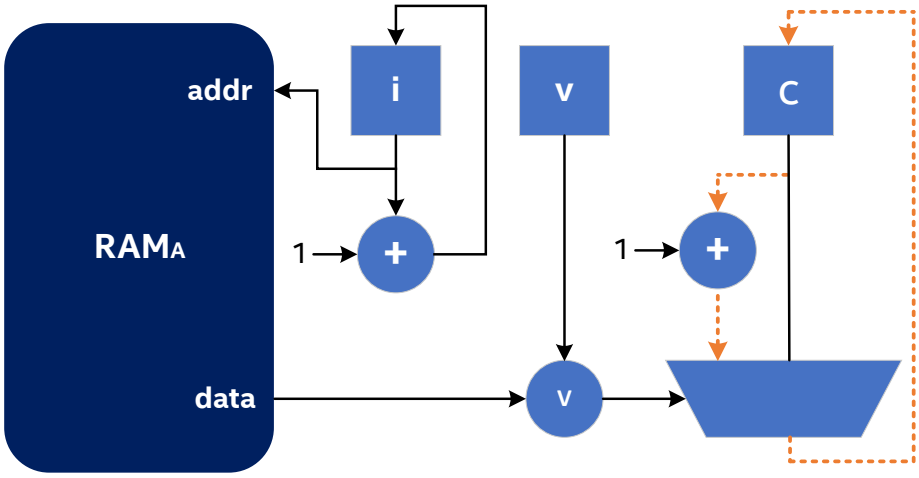
3.1.1.2.9 Shannonization to Improve F_{MAX}/II

Shannonization (named after Claude Shannon) is a loop optimization technique that improves the f_{MAX}/II of a design by precomputing operations in a loop and removing the operations from the critical path. To demonstrate shannonization, consider the following example, which counts the number of elements in an array `A` that are less than some runtime value `v`:

```
int A[SIZE] = {/*...*/};
int v = /*some dynamic value*/
int c = 0;
for (int i = 0; i < SIZE; i++) {
  if (A[i] < v) {
    c++;
  }
}
```

A possible circuit diagram for this algorithm is shown in the following image, where the orange dotted line represents a possible critical path in the circuit:

Figure 50. Circuit Diagram for the Example Shannonization Algorithm

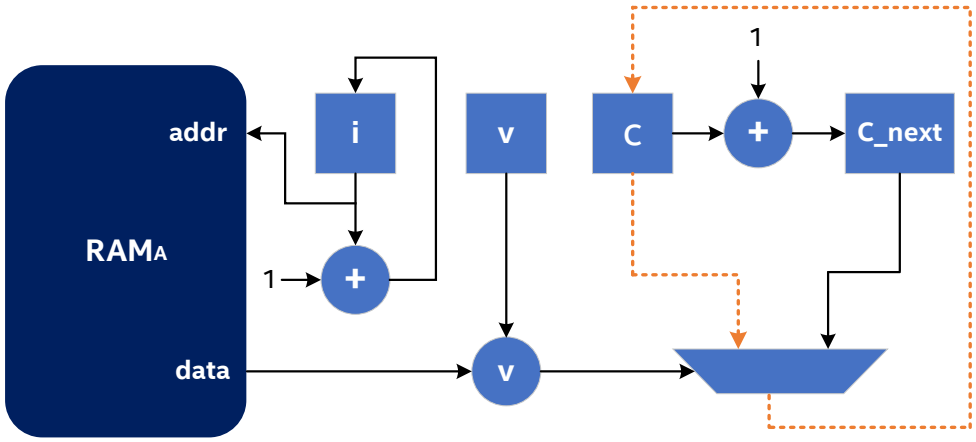


The goal of the shannonization optimization is to remove operations from the critical path. In this case, precompute the next value of `c` (fittingly named `c_next`) for a later iteration of the loop to use when required (that is, the next time `A[i] < v`). This optimization is shown in the following code sample:

```
int A[SIZE] = {/*...*/};
int v = /*some dynamic value*/
int c = 0;
int c_next = 1;
for (int i = 0; i < SIZE; i++) {
  if (A[i] < v) {
    // these operations can happen in parallel!
    c = c_next;
    c_next++;
  }
}
```

A possible circuit diagram for this optimized algorithm is shown in the following image, where the orange dotted line represents a possible critical path in the circuit:

Figure 51. Circuit Diagram for the Optimized Shannonization Algorithm



In the Figure 51 on page 98, the + operation from the critical path is removed. This diagram assumes that the critical path delay through the multiplexer is higher than through the adder. This may not be the case, and the critical path could be from the `c` register to the `c_next` register through the adder, in which case, the multiplexer from the critical path is removed. Regardless of which operation has the longer critical path delay (the adder or the multiplexer), an operation from the critical path is removed by precomputing and storing the next value of `c`. This allows in reducing the critical path delay at the expense of area (in this case, a single 32-bit register).

NOTE

For additional information, refer to the FPGA tutorial sample `Shannonization` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

3.1.1.2.10 Optimize Inner Loop Throughput

In this topic, you learn how to optimize throughput of an inner loop with a low trip count. A *low* trip count is relative. For understanding the concept, consider *low* to be on the order of 100 or fewer iterations.

NOTE

Prior to reading the rest of this topic, you must review Maximum Frequency (f_{MAX}) on page 11 and Optimize Loops With Loop Speculation on page 92.

Consider the following example code snippet:

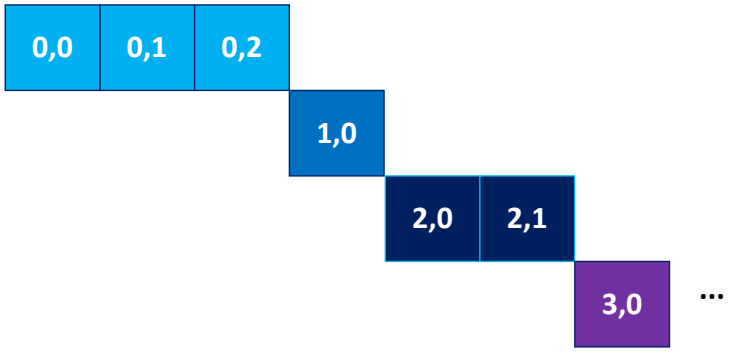
```
for (int i; i < kOuterLoopBound; i++) {
  int inner_loop_iterations = rand() % kInnerLoopBound;
  for (int j; j < inner_loop_iterations; j++) {
    /* ... */
  }
}
```

Before you optimize the inner loops with low trip counts, assume the following:

- `kOuterLoopBound` is a number greater than one million.
- `kInnerLoopBound` is 3. This means that the value of `inner_loop_iterations` is dynamic, but you know that it is in the range `(0,3)`.
- II of the inner loop is 1, which means that a new inner loop iteration can start every cycle. This means that the outer loop II is dynamic and depends on how many inner loop iterations the previous outer loop iteration must start.

A possible timing diagram for this loop structure is shown in the following figure, where the numbers in the squares are the values of `i` and `j`, respectively:

Figure 52. Timing Diagram for the Loop



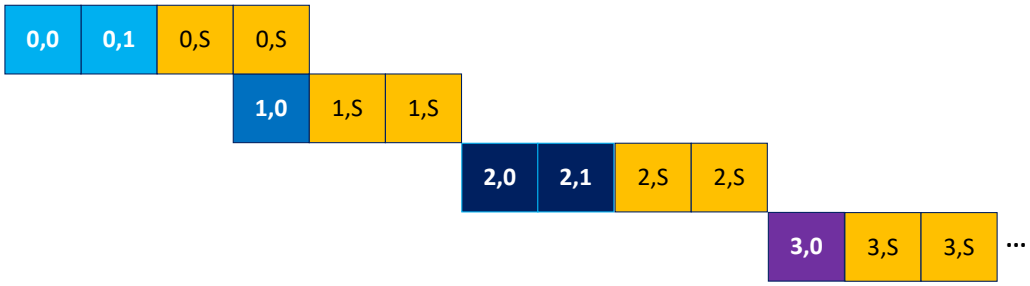
In general, the Intel® oneAPI DPC++/C++ Compiler optimizes loops for throughput with the assumption that the loop has a high trip count. These optimizations include (but are not limited to) speculating iterations and inserting pipeline registers in the circuit that starts loops. The next two sections describe how these optimizations can substantially decrease throughput and how you can disable them to improve your design when applied to inner loops with low trip counts.

Speculated Iterations

Loop Speculation enables loop iterations to be initiated before determining whether they should have been initiated. Speculated iterations are the iterations of a loop that launch before the exit condition computation has been completed. This is beneficial when the computation of the exit condition is preventing effective loop pipelining. However, when an inner loop has a low trip count, speculating iterations result in a relatively high proportion of invalid loop iterations.

For example, consider that the compiler speculated two inner loop iterations for the code above. In that case, the timing diagram appears as shown in the following diagram, where the orange blocks with the S denote an invalid speculated iteration:

Figure 53. Timing Diagram with Invalid Speculated Iteration



In this case where the inner loop iteration count is in the range `(0,3)`, speculating two iterations can cause up to a 3x reduction in the design's throughput. This happens when each outer loop iteration launches one inner loop iteration (`inner_loop_iterations` is always 1), but two iterations are speculated. For this reason, Intel® advises to force the compiler to not speculate iterations for inner loops with known small trip counts using the `[[intelfpga::speculated_iterations(0)]]` attribute.

Dynamic Trip Counts

As mentioned earlier, the compiler's default behavior is to optimize loops for throughput. However, as you saw in the previous section, loops with low trip counts have unique throughput characteristics that lead to the compiler making different optimizations. The compiler attempts its best to determine if a loop has a high or low trip count and optimizes accordingly. However, in some circumstances, you may need to provide it with more information to make a better decision.

In the previous section, this additional information was the `speculated_iterations` attribute. However, it is not just speculated iterations that cause delays in the launching of inner loops. The compiler uses other heuristics. For example, the compiler may attempt to improve the f_{MAX} of a loop circuit by adding a pipeline register on the circuit path that starts a loop, which results in a one-cycle delay in starting the loop. For outer loops with large trip counts, this one cycle delay is negligible. However, for inner loops with small trip counts, this one cycle delay can cause throughput degradation. Like the speculated iteration case discussed in the previous section, this one cycle delay can result in up to a 2x reduction in the design's throughput.

If the inner loop bounds are known to the compiler, it decides whether to turn on/off this delay register depending on the (known) trip count. However, in the code snippet above, the inner loop's trip count is not a constant (`inner_loop_iterations` is a random number at runtime). In cases like this, Intel recommends explicitly bounding the trip count of the inner loop. This is illustrated in the example code snippet in the following, where `j < kInnerLoopBound` exit condition is added to the inner loop. This gives the compiler more explicit information about the loop's trip count and allows it to optimize accordingly.

```
for (int i; i < kOuterLoopBound; i++) {
  int inner_loop_iterations = rand() % kInnerLoopBound;
  for (int j; j < inner_loop_iterations && j < kInnerLoopBound; j++) {
    /* ... */
  }
}
```

NOTE

For additional information, refer to the FPGA tutorial sample `Optimize Inner Loop` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

3.1.1.2.11 Improve Loop Performance by Caching On-Chip Memory

In SYCL* task kernels for FPGA, the main objective is to achieve an initiation interval (II) of 1 on performance-critical loops. This means that a new loop iteration is launched on every clock cycle, thereby maximizing the loop's throughput. When the loop contains a loop-carried variable implemented in on-chip memory, the Intel® oneAPI DPC++/C++ Compiler often cannot achieve II=1 because the memory access takes more than one clock cycle. If the updated memory location is necessary on the next loop iteration, the next iteration must be delayed to allow time for the update, hence II > 1.

The on-chip memory cache technique breaks this dependency by storing recently-accessed values in a cache capable of a one-cycle read-modify-write operation. The cache is implemented in FPGA registers rather than on-chip memory. By pulling memory accesses preferentially from the register cache, the loop-carried dependency is broken.

When is the On-chip Memory Cache Technique Applicable?

You can apply the on-chip memory cache technique in the following situations:

- **Failure to achieve II=1 because of a loop-carried memory dependency in on-chip memory**

  The on-chip memory cache technique is applicable if the compiler could not pipeline a loop with II=1 because of an on-chip memory dependency. If the compiler could not achieve II=1 because of a global memory dependency, this technique does not apply as the access latencies are too great.

  To check this for a given design, view the Loop Analysis report in the design's optimization report. The Loop Analysis report lists the II of all loops and explains why a lower II is not achievable. Check whether the reason given resembles `the compiler failed to schedule this loop with smaller II due to memory dependency`. The report describes the `most critical loop feedback path during scheduling`. Check whether this includes on-chip memory load/store operations on the critical path.

- **An II=1 loop with a load operation of latency 1**

  The compiler is capable of reducing the latency of on-chip memory accesses to achieve II=1. In doing so, the compiler makes a trade-off by sacrificing f_{MAX} to improve the II.

  In a design with II=1 critical loops but lower than the desired f_{MAX}, the on-chip memory cache technique might still be applicable. It can help recover f_{MAX} by enabling the compiler to achieve II=1 with a higher latency memory access. To check whether this is the case for a given design, view the Kernel Memory Viewer report in the design's optimization report. Select the desired on-chip memory from the Kernel Memory List, and mouse over the load operation `LD` to check its latency. If the latency of the load operation is 1, this is a clear sign that the compiler has attempted to sacrifice f_{MAX} to improve loop II.

Implement the On-chip Memory Cache Technique

Consider the FPGA design example in `onchip_memory_cache.cpp`, which demonstrates the technique using a program that computes a histogram. The histogram operation accepts an input vector of values, separates the values into buckets, and counts the number of values per bucket. For each input value, an output bucket location is determined, and the count for the bucket is incremented. This count is stored in the on-chip memory, and the increment operation requires reading from memory, performing the increment, and storing the result. This read-modify-write operation is the critical path that can result in II > 1.

To reduce II, the idea is to store recently-accessed values in an FPGA register-implemented cache that is capable of a one-cycle read-modify-write operation. If the memory location required on a given iteration exists in the cache, it is pulled from there. The updated count is written back to both the cache and the on-chip memory. The `ivdep` attribute is added to inform the compiler that if a loop-carried variable (namely, the variable storing the histogram output) is required within `CACHE_DEPTH` iterations, it is guaranteed to be available right away.

Select the Cache Depth

While any value of `CACHE_DEPTH` results in functional hardware, the ideal value of `CACHE_DEPTH` requires some experimentation. The depth of the cache must roughly cover the latency of the on-chip memory access. To determine the correct value, Intel® recommends starting with a value of 2 and then increase it until both II = 1 and load latency > 1. In the `onchip_memory_cache.cpp` example, a `CACHE_DEPTH` of 5 is necessary. It is important to find the minimal value of `CACHE_DEPTH` that results in a maximal performance increase. Unnecessarily large values of `CACHE_DEPTH` consume unnecessary FPGA resources and can reduce f_{MAX}. Therefore, at a `CACHE_DEPTH` that results in II=1 and load latency = 1, if further increases to `CACHE_DEPTH` show no improvement, do not increase `CACHE_DEPTH` any further.

NOTE

For additional information, refer to the FPGA tutorial sample `Onchip Memory Cache` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

3.1.1.3 Single-Cycle Floating-Point Accumulator for Single Work-Item Kernels

Single work-item kernels that perform accumulation in a loop can leverage the single-cycle floating-point accumulator feature of the Intel® oneAPI DPC++/C++ Compiler. The compiler searches for these kernel instances and attempts to map an accumulation that executes in a loop into the accumulator structure.

The compiler supports an accumulator that adds or subtracts a value. To leverage this feature, describe the accumulation in a way that allows the compiler to infer the accumulator, which must be part of a loop, must have an initial value of 0, and cannot be conditional.

NOTE

The accumulator is available only on Intel® Arria® 10 devices.

3.1.1.3.1 Strategies for Inferring the Accumulator

To leverage the single cycle floating-point accumulator feature, you can modify the accumulator description in your kernel code to improve efficiency or work around programming restrictions.

Describe an Accumulator Using Multiple Loops

Consider a case where you want to describe an accumulator using multiple loops, with some of the loops being unrolled:

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
  #pragma unroll
  for (j = 0; j < 16; j++)
    acc += (x[i+j]*y[i+j]);
}
```

With fast math enabled by default, the Intel® oneAPI DPC++/C++ Compiler automatically rearranges operations in a way that exposes the accumulation.

Modify a Multi-Loop Accumulator Description

In cases where you cannot compile an accumulator description using the `-Xsfp-relaxed` compiler command option, rewrite the code to expose the accumulation.

For the code example above, rewrite it in the following manner:

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
  float my_dot = 0.0f;
  #pragma unroll
  for (j = 0; j < 16; j++)
    my_dot += (x[i+j]*y[i+j]);
    acc += my_dot;
}
```

Modify an Accumulator Description Containing a Variable or Non-Zero Initial Value

Consider a situation where you might want to apply an offset to a description of an accumulator that begins with a non-zero value:

```
float acc = array[0];
for (i = 0; i < k; i++) {
  acc += x[i];
}
```

Because the accumulator hardware does not support variable or non-zero initial values in a description, you must rewrite the description.

```
float acc = 0.0f;
for (i = 0; i < k; i++) {
  acc += x[i];
}
acc += array[0];
```

Rewriting the description in the above manner enables the kernel to use an accumulator in a loop. The loop structure is then followed by an increment of `array[0]`.

3.1.2 NDRange Kernels

If your program naturally tends to describe multiple concurrent threads operating in a data-parallel manner, specify your kernel to operate in parallel instances over a work-item index-space (NDRange).

Avoid Work-Item ID-Dependent Backward Branching

The Intel® oneAPI DPC++/C++ Compiler collapses conditional statements into single bits that indicate when a particular functional unit becomes active. The Intel® oneAPI DPC++/C++ Compiler eliminates simple control flow paths that do not involve looping structures, resulting in a flat control structure and more efficient hardware use.

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
}
```

3.1.3 Memory Accesses

Memory access efficiency often dictates the overall performance of your SYCL* kernel. Refer to Memory Types for an introduction to memory accesses.

The pipeline parallel nature of SYCL execution on FPGA means that memory loads and stores in your SYCL code compete for access to memory resources (global, local, and private memories). If your SYCL kernel performs a large number of memory accesses, the compiler must generate arbitration logic to share the available memory bandwidth between memory access sites in your kernel's datapath. If the bandwidth demanded by the datapath exceeds what the memory and arbitration logic can provide, the datapath stalls. This degrades kernel’s throughput because the compute pipeline must wait for a memory access before resuming.

When optimizing your design, it is important to understand whether your kernel's throughput is limited by memory accesses (a memory-bound kernel) or by the structure of the kernel datapath (a compute-bound kernel). These situations require different optimization techniques. The following sections discuss memory access optimization in detail.

Consider the following when developing your SYCL code:

- The maximum computation bandwidth of an FPGA is much larger than the available global memory bandwidth.
- The available global memory bandwidth is much smaller than the local and private memory bandwidth.
- Minimize the number of global memory accesses.

3.1.3.1 Load-Store Units

The Intel® oneAPI DPC++/C++ Compiler generates several 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. Refer to the following topics for more information:

- Load-Store Unit Styles on page 105
- Load-Store Unit Modifiers on page 106
- Load-Store Unit Controls on page 107

3.1.3.1.1 Load-Store Unit Styles

The Intel® oneAPI DPC++/C++ Compiler generates different styles of load-store units (LSUs) based on:

- Inferred memory access pattern
- Types of memory available on the target platform
- Whether the memory accesses are to the local or global memory

The Intel® oneAPI DPC++/C++ Compiler can generate the following styles of LSUs:

- Burst-Coalesced Load-Store Units on page 105
- Prefetching Load-Store Units on page 106
- Pipelined Load-Store Units on page 106

Burst-Coalesced Load-Store Units

A burst-coalesced LSU is the default LSU style instantiated by the compiler for accessing global memory. It buffers contiguous memory requests until the largest possible burst can be made. For noncontiguous memory requests, a burst-coalesced LSU flushes the buffer between requests.

While a burst-coalesced LSU provides efficient, variable-latency access to global memory, a burst-coalesced LSU requires a considerable amount of FPGA resources.

The following example code results in the compiler instantiating burst-coalesced LSUs:

```
cgh.single_task<class Kernel>([=] {

  int x = input_accessor[RandomIndex]; //burst-coalesced
  output_accessor[0] = x;
});
```

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

- Cached on page 106
- Write-Acknowledge (write-ack) on page 107
- Nonaligned on page 107

Prefetching Load-Store Units

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

The following example code results in the compiler instantiating prefetching LSUs to access global memory:

```
cgh.single_task<class Kernel>([=] {
    int x = 1;
  for (int i = 0; i < VectorSize; i++) {
     x = x + input_accessor[i]; //prefetching
  }
  output_accessor[0] = x;
});
```

Pipelined Load-Store Units

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

```
cgh.single_task<class Kernel>([=] {

  const unsigned LMEM_SIZE = 128;
  int lmem[LMEM_SIZE];
  for (int i = 0; i < LMEM_SIZE; i++) {
    lmem[i] = i * 100; //pipelined
  }
  output_accessor[0] = lmem[input_accessor[0]];
});
```

The compiler might modify a local-pipelined LSU as a never-stall LSU. For more details, refer to Never-stall on page 107.

The Intel® oneAPI DPC++/C++ Compiler may also infer a pipelined LSU for global memory accesses that can be proven to be infrequent. The compiler uses a pipelined LSU for such accesses because a pipelined LSU is smaller than other LSU styles. While a pipelined LSU might have lower throughput, this throughput tradeoff is acceptable because memory accesses are infrequent.

3.1.3.1.2 Load-Store Unit Modifiers

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

Cached

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

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.

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.

Never-stall

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

3.1.3.1.3 Load-Store Unit Controls

The Intel® oneAPI DPC++/C++ Compiler allows you to control the LSU that is generated for global memory accesses via a set of templated options in the `ext::intel::lsu` class. The `ext::intel::lsu` class has two member functions, `load()` and `store()`, which allow loading from and storing to a global pointer, respectively.

NOTE

Not all LSU control options listed below work with both `load()` and `store()` members function. Not all styles and types are supported on every target device. Thus, the use of LSU control options results in a specific type of LSU.

Table 12. LSU Control Options Available in the `ext::intel::lsu` Class

Control	Syntax	Value	Default Value	Supported Functionality
Burst-Coalesce	`ext::intel::burst_coalesce<B>`	`B` is a boolean	False	Load and store
Static Coalescing	`ext::intel::statically_coalesce<B>`	`B` is a boolean	True	Load and store
Prefetch	`ext::intel::prefetch<B>`	`B` is a boolean	false	Only loads
Cache	`ext::intel::cache<N>`	`N` is an integer greater than or equal to 0.	0	Only loads
Pipeline	No option provided	-	Defaults above result in this.	Load and store

Currently, not every combination of LSU controls is supported by the compiler. The following rules apply:

- For store, the `ext::intel::cache` control must be 0 and the `ext::intel::prefetch` control must be false.
- For load, if the `ext::intel::cache` control is set to a value greater than 0, then the `ext::intel::burst_coalesce` control must be set to true.
- For load, exactly one of `ext::intel::prefetch` and `ext::intel::burst_coalesce` control options are allowed to be true.
- For load, exactly one of `ext::intel::prefetch` and `ext::intel::cache` control options are allowed to be true.

Burst-Coalesce

The burst-coalesce control helps the Intel® oneAPI DPC++/C++ Compiler infer a burst-coalesced LSU. It can be combined with other attributes such as *cache*. For more details about this LSU type, refer to Burst-Coalesced Load-Store Units on page 105.

Consider the following example of burst-coalesce LSU control:

```
using BurstCoalescedLSU =
  ext::intel::lsu<ext::intel::burst_coalesce<true>,
                  ext::intel::statically_coalesce<false>>;
BurstCoalescedLSU::store(output_ptr, X);
```

Static Coalescing

The static-coalescing control helps the Intel® oneAPI DPC++/C++ Compiler turn on or off static coalescing for global memory accesses. Static coalescing of memory accesses is the default behavior of the compiler, so this feature can be used to turn off static coalescing. It can be combined with other attributes such as *burst_coalesce*. For more details about this memory optimization modifier, refer to Static Memory Coalescing on page 116.

Consider the following example using the control to turn off static-coalescing LSU control:

```
using NonStaticCoalescedLSU =
  ext::intel::lsu<ext::intel::burst_coalesce<true>,
                  ext::intel::statically_coalesce<false>>;

int X = NonStaticCoalescedLSU::load(input_ptr);
```

Prefetch

The prefetch control helps the Intel® oneAPI DPC++/C++ Compiler infer a prefetch style LSU. The compiler honors this control only if it is physically possible. If there is a risk that an LSU is not functionally correct, it is not inferred. This control also works only for load accesses. For more details about this LSU type, refer to Prefetching Load-Store Units on page 106.

Consider the following example of prefetch LSU control:

```
using PrefetchingLSU =
  ext::intel::lsu<ext::intel::prefetch<true>,
                  ext::intel::statically_coalesce<false>>;

int X = PrefetchingLSU::load(input_ptr);
```

Cache

The cache control helps the Intel® oneAPI DPC++/C++ Compiler turn on or off caching behavior and set the size of the cache. This LSU modifier can be useful for `parallel_for` kernels. The cache control can be combined with other attributes such as *burst_coalesce*. For more details about this LSU modifier, refer to Cached on page 106.

Consider the following example of cache LSU control:

```
using CachingLSU =
  ext::intel::lsu<ext::intel::burst_coalesce<true>,
                        ext::intel::cache<1024>,
                        ext::intel::statically_coalesce<false>>;

int X = CachingLSU::load(input_ptr);
```

Pipeline

The pipeline control helps the Intel® oneAPI DPC++/C++ Compiler infer the LSU style of pipeline. It can be combined with other attributes such as *static coalesce*. For more details about this LSU type, refer to Pipelined Load-Store Units on page 106.

For example:

```
using PipelinedLSU = ext::intel::lsu<>;

PipelinedLSU::store(output_ptr,X);
```

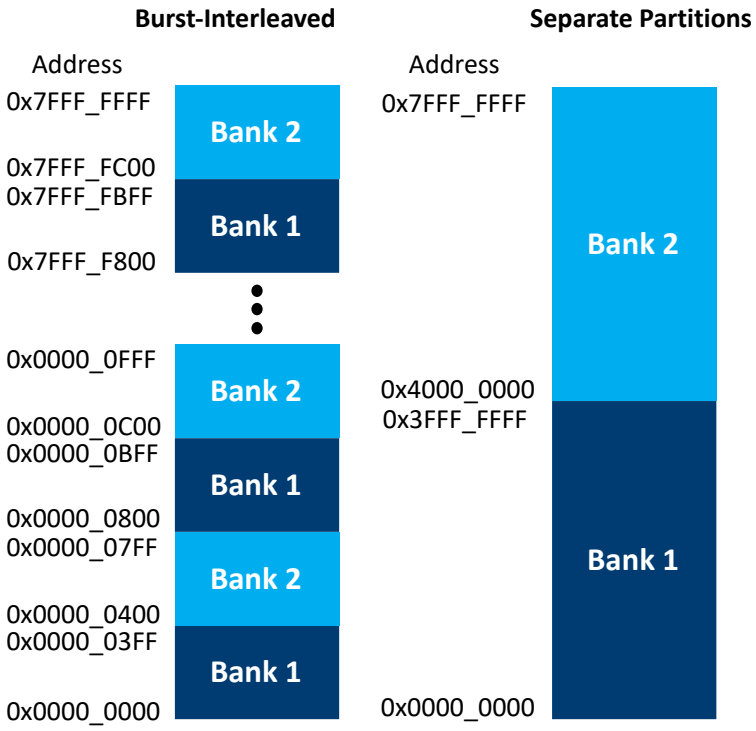
3.1.3.2 Global Memory Accesses Optimization

The Intel® oneAPI DPC++/C++ Compiler uses SDRAM as global memory. By default, the compiler configures global memory in a burst-interleaved configuration. The Intel® oneAPI DPC++/C++ 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 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 following figure illustrates the difference in memory mapping patterns between burst-interleaved and non-interleaved memory partitions:

Figure 54. Global Memory Partitions



Related Links

Global Memory Bandwidth Use Calculation on page 110

Manual Partition of Global Memory on page 112

Partitioning Buffers Across Different Memory Types (Heterogeneous Memory) on page 112

Partitioning Buffers Across Memory Channels of the Same Memory Type on page 113

Ignoring Dependencies Between Accessor Arguments on page 114

Contiguous Memory Accesses on page 115

Static Memory Coalescing on page 116

3.1.3.2.1 Global Memory Bandwidth Use Calculation

To ensure the global memory bandwidth listed in the board specification file is utilized completely, calculating the kernel bandwidth use is beneficial. The report.html file also displays the kernel bandwidth values in the global memory view of the System Viewer. The following formulas explain how you can calculate this value on a per-LSU basis:

Figure 55. Formulas for Calculating Kernel Bandwidth Use

$$\textbf{LSU bandwidth} = \min(BW_1, BW_2, BW_3) \text{ MB/s}$$

$$\textbf{BW}_1 = \text{KWIDTH} * \text{FMAX}$$

$$\textbf{BW}_2 = \text{MWIDTH} * \text{FMAX}$$

$$\textbf{BW}_3 = \frac{\text{MaxBandwidth} * \text{NUM\_INTERLEAVING\_CHANNELS}}{\text{NUM\_CHANNELS}}$$

The LSU bandwidth equation is the minimum of three bottlenecks you need to calculate the use of global memory bandwidth. The remaining equations represent three bottlenecks that can limit the LSU bandwidth. These formulas represent the theoretical maximum bandwidth an LSU may consume, ignoring all other LSUs. The actual bandwidth depends on the LSU's access pattern and the interconnect's arbitration between all LSUs. To get an estimate of the overall bandwidth, a sum of the LSU bandwidths is available in the controller of the global memory view of the System Viewer.

The following table describes the variables used in the above equations:

Table 13. Variables Used in Calculating Kernel Bandwidth

Variable	Description
KWIDTH	Byte-width of the LSU on the kernel. In the report.html file, it is referred to as WIDTH.
MWIDTH	Byte-width of the LSU facing the external memory. In the report.html file, it is referred to as the <Memory Name>_Width.
FMAX	Clock speed of the kernel in MHz. In the report.html file, you can identify this as the design's clock speed.
MaxBandwidth	Maximum bandwidth (measured in MB/s) the global memory can achieve. You can find this in the board_spec.xml file for the specific global memory.
NUM_CHANNELS	Number of interfaces an external memory has. You can find this by counting the number of interfaces listed in the board_spec.xml file under that memory.
NUM_INTERLEAVING_CHANNELS	When interleaving is enabled, this is the number of channels. Otherwise, this value is 1.
BW_1	Bottleneck at the kernel boundary. Therefore, BW_1 uses only kernel values, which means, values you can change by optimizing the design. If this is limiting the overall bandwidth use than it indicates, changing your design can improve the bottleneck at the kernel boundary.
BW_2	Bottleneck at the memory interface to the kernel. Therefore, BW_2 uses the size of the memory interface and the f_{MAX}, which means either improving f_{MAX} of your design or switching to a board with a wider memory interface can improve the bandwidth use.
BW_3	Bottleneck in the external memory. Therefore, BW_3 uses external memory properties exclusively, and if this is limiting your design, you have utilized the board bandwidth completely.

3.1.3.2.2 Manual Partition 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® oneAPI DPC++/C++ Compiler cannot burst-interleave across different memory types. To manually partition a specific type of global memory, compile your kernels with the `-Xsno-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 `-Xsno-interleaving=<global_memory_type>` option in the `dpcpp` command for each memory type that you want to partition manually.

For more information, refer to Disable Burst-Interleaving of Global Memory (-Xsno-interleaving=<global_memory_type>).

3.1.3.2.3 Partitioning Buffers Across Different Memory Types (Heterogeneous Memory)

The board support package for your FPGA board can assemble a global memory space consisting of different memory technologies (for example, DRAM or SRAM). The board support package designates one such memory (which might consist of multiple interfaces) as the default memory. All buffers reside in this heterogeneous memory.

To use the heterogeneous memory, perform the following steps to modify the code in your source file:

1. Determine the names of global memory types available on your FPGA board using one of the following methods:
   - Refer to the board vendor's documentation for information.
   - Identify the index of the global memory type in the `board_spec.xml` file of your Custom Platform. The index starts at 0 and follows the order in which the global memory appears in the `board_spec.xml`. For example, the first global memory type in the XML file has an index 0, the second has an index 1, and so on. For more information, refer to `global_mem` in the *Intel® FPGA SDK for OpenCL™ Pro Edition Custom Platform Toolkit User Guide*.
2. To instruct the host to allocate a buffer to a specific global memory type, insert the `buffer_location<index>` property in the accessor's property list.

   For example:

```
ext::oneapi::accessor_property_list PL{ext::intel::buffer_location<2>};
accessor acc(buffer, cgh, read_only, PL);
```

If you do not specify the `buffer_location` property, the host allocates the buffer to the default memory type automatically. To determine the default memory type, consult the documentation provided by your board vendor. Alternatively, in the `board_spec.xml` file of your Custom Platform, search for the memory type that is defined first or has the attribute `default=1` assigned to it. For more information, refer to *Intel® FPGA SDK for OpenCL™ Pro Edition Custom Platform Toolkit User Guide*.

NOTE

Streams support is limited if the target FPGA is used in a FPGA board with heterogeneous memory. With SYCL streams, all work items write to the same stream buffer in parallel, so atomics are leveraged to ensure race conditions. Atomics support for boards that have heterogeneous memory is limited. For more information about atomics and streams, refer to SYCL specifications available at https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf and https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

3.1.3.2.4 Partitioning Buffers Across Memory Channels of the Same Memory Type

By default, the Intel® oneAPI DPC++/C++ Compiler configures each global memory type in a burst-interleaved manner. Usually, the burst-interleaving configuration leads to the best load balancing between the memory channels. However, there might be situations where it is more efficient to partition the memory into non-interleaved regions. For additional information, refer to Global Memory Accesses Optimization on page 109.

The Figure 54 on page 110 illustrates the differences between burst-interleaved and non-interleaved memory partitions.

To manually partition some or all of the available global memory types, perform the following tasks:

1. Create a buffer with `property::buffer::mem_channel` specifying channel ID in its `property_list`.
   - Specify `property::buffer::mem_channel` with value 1 to allocate the buffer in the lowest available memory channel (default).
   - Specify `property::buffer::mem_channel` with value 2 or greater to allocate the buffer in the higher available memory channel.

   Here is an example buffer definition:

```
range<1> num_of_items{N};
buffer<T, 1> bufferA(VA.data(), num_of_items,
  {property::buffer::mem_channel{1}});
buffer<T, 1> bufferB(VB.data(), num_of_items,
  {property::buffer::mem_channel{2}});
buffer<T, 1> bufferC(VC.data(), num_of_items,
  {property::buffer::mem_channel{3}});
```

2. Compile your design kernel using the `-Xsno-interleaving=<global_memory_type>` flag to configure the memory channels of the specified memory type as separate addresses. For more information about the use of the `-Xsno-interleaving=<global_memory_type>` flag, refer to the Disable Burst-Interleaving of Global Memory (`-Xsno-interleaving=<global_memory_type>`) on page 166 section.

CAUTIONS

- Do not set more than one memory channel property on a buffer.
- If the memory channel specified is not available on the target board, the buffer is placed in the first memory channel.

3.1.3.2.5 Ignoring Dependencies Between Accessor Arguments

You can direct the Intel® oneAPI DPC++/C++ Compiler to ignore dependencies between accessor arguments in a SYCL* kernel in one of the following methods. Both methods allow the compiler to analyze dependencies between kernel memory operations more accurately, which can result in higher performance.

Method 1: Add the `[[intel::kernel_args_restrict]]` Attribute to Your Kernel

To direct the Intel® oneAPI DPC++/C++ Compiler to ignore dependencies between accessor arguments and USM pointers used in a kernel, add the `[[intel::kernel_args_restrict]]` attribute to your kernel. The use of the `[[intel::kernel_args_restrict]]` attribute is an assurance to the compiler that the accessor arguments and USM pointers used by the kernel do not alias with each other. This is an unchecked assertion by the programmer and results in an undefined behavior if violated.

Example 3. Example

```
device_queue.submit([&](handler& cgh) {
  // create accessors from global memory
  accessor in_accessor(in_buf, cgh, read_only);
  accessor out_accessor(out_buf, cgh, write_only);

  // run the task (note the use of the attribute here)
  cgh.single_task<KernelArgsRestrict>([=]() [[intel::kernel_args_restrict]] {
    for (int i = 0; i < N; i++) {
      out_accessor[i] = in_accessor[i];
    }
  });
});
```

TIP

For additional information, refer to the FPGA tutorial sample "Kernel Args Restrict" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

Method 2: Add the `no_alias` Property to an Accessor's Property List

The `no_alias` property notifies the Intel® oneAPI DPC++/C++ Compiler that all modifications to the memory locations accessed (directly or indirectly) by an accessor during kernel execution is done through the same accessor (directly or indirectly) and not by any other accessor or USM pointer in the kernel. This is an unchecked assertion by the programmer and results in an undefined behavior if it is violated. Effectively, applying `no_alias` to all accessors of a kernel is equivalent to applying the `[[intel::kernel_args_restrict]]` attribute to the kernel unless the kernel uses USM. You cannot apply the `no_alias` property on a USM pointer.

Example 4. Example

```
ext::oneapi::accessor_property_list PL{ext::oneapi::no_alias};
accessor acc(buffer, cgh, PL);
```

3.1.3.2.6 Contiguous Memory Accesses

The Intel® oneAPI DPC++/C++ Compiler attempts to dynamically coalesce accesses to adjacent memory locations to improve global memory efficiency. This is effective if consecutive work items access consecutive memory locations in a given load or store operation. The same is true in a `single_task` invocation if consecutive loop iterations access consecutive memory locations.

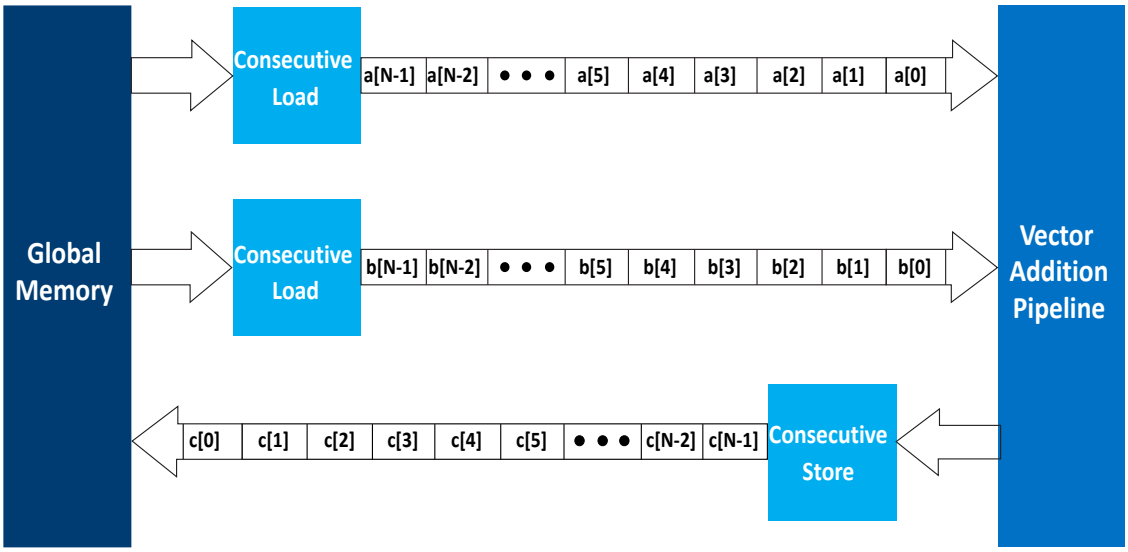
Consider the following code example:

```
q.submit([&](handler &cgh) {
  accessor a(a_buf, cgh, read_only);
  accessor b(b_buf, cgh, read_only);
  accessor c(c_buf, cgh, write_only, no_init);
  cgh.parallel_for<class SimpleVadd>(N, [=](id<1> ID) {
    c[ID] = a[ID] + b[ID];
  });
});
```

The load operation from the accessor `a` uses an index that is a direct function of the work-item global ID. By basing the accessor index on the work-item global ID, the Intel® oneAPI DPC++/C++ Compiler can ensure contiguous load operations. These load operations retrieve the data sequentially from the input array and send 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.

The following figure illustrates an example of the contiguous memory access optimization:

Figure 56. Contiguous Memory Access



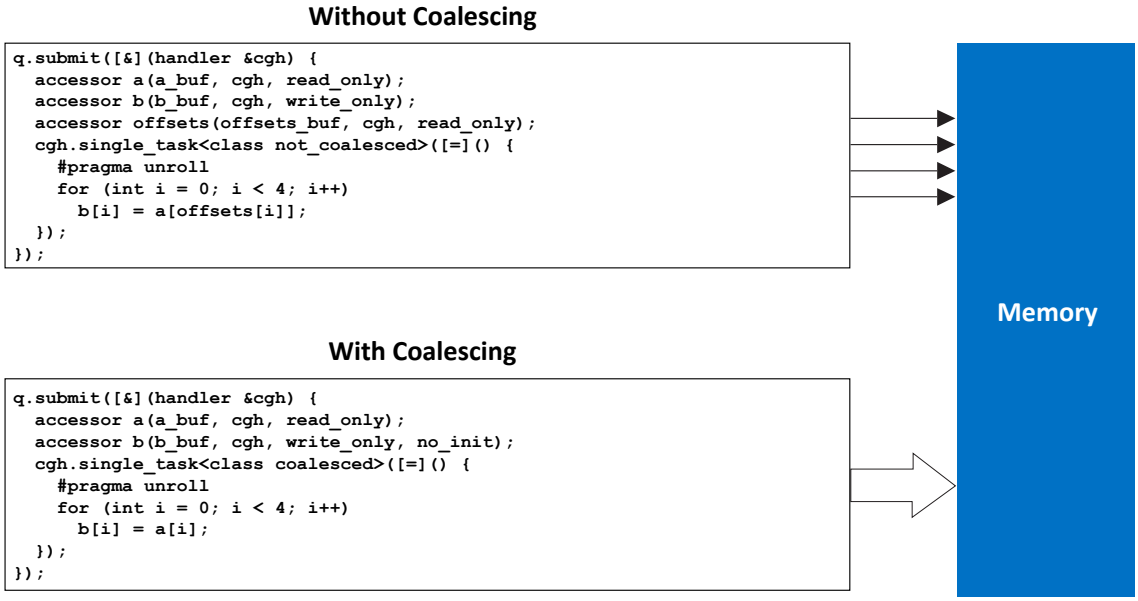
Contiguous load and store operations improve memory access efficiency because they lead to increased access speeds and reduced hardware resource needs. The data travels in and out of the computational portion of the pipeline concurrently, allowing overlaps between computation and memory accesses. Where possible, use work-item IDs that index accesses to arrays in global memory to maximize memory bandwidth efficiency.

3.1.3.2.7 Static Memory Coalescing

Static memory coalescing is an Intel® oneAPI DPC++/C++ Compiler optimization step that merges contiguous accesses to global memory into a single wide access. A similar optimization is applied to on-chip memory.

The figure below shows a common case where kernel performance might benefit from static memory coalescing:

Figure 57. Static Memory Coalescing



Consider the following vectorized kernel:

```
q.submit([&](handler &cgh) {
  accessor a(a_buf, cgh, read_only);
  accessor b(b_buf, cgh, write_only, no_init);
  cgh.single_task<class coalesced>([=]() {
    #pragma unroll
    for (int i = 0; i < 4; i++)
      b[i] = a[i];
  });
});
```

The kernel performs four load operations from buffer a that access consecutive locations in memory. Instead of performing four memory accesses to competing locations, the 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 compiler performs static memory coalescing automatically, you should use wide vector loads and stores in your SYCL* code whenever possible to ensure efficient memory accesses.

To allow static memory coalescing, you must write your code in such a way that the compiler can identify a sequential access pattern during compilation. The original kernel code shown in the figure above can benefit from static memory coalescing because all 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:

```
q.submit([&](handler &cgh) {
  accessor a(a_buf, cgh, read_only);
  accessor b(b_buf, cgh, write_only);
  accessor offsets(offsets_buf, cgh, read_only);
  cgh.single_task<class not_coalesced>([=]() {
```

```
    #pragma unroll
    for (int i = 0; i < 4; i++)
      b[i] = a[offsets[i]];
  });
});
```

The value `offsets[i]` is unknown at compilation time. As a result, the Intel® oneAPI DPC++/C++ Compiler cannot statically coalesce the read accesses to buffer `a`.

For more information, refer to Local and Private Memory Accesses Optimization on page 118.

3.1.3.3 Perform Kernel Computations Using Local or Private Memory

To optimize memory access efficiency, minimize the number of global memory accesses by performing kernel computations in local or private memory.

To minimize global memory accesses, it is often best to preload data from a group of computations from global memory to a local or private memory. Perform kernel computations on the preloaded data and write the results back to the global memory.

Preload Data into Local Memory or Private Memory

Local memory is considerably smaller than global memory, but it has significantly higher bandwidth 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 computation.

Store Variables and Arrays in Private Memory

The Intel® oneAPI DPC++/C++ Compiler implements private memory using FPGA registers in the kernel datapath, block RAMs, or MLABs. The Intel® oneAPI DPC++/C++ Compiler analyzes the private memory accesses and promotes them to register accesses. Scalar variables, for example `float`, `int` and `char`, are typically promoted. Aggregate data types are promoted if array-access indices are compile-time constants. Typically, private memory is useful for storing single variables or small arrays. Registers are plentiful hardware resources in FPGAs, and it is usually 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 (global and local).

For more information on the implementation of private memory using registers, refer to Inferring a Shift Register.

3.1.3.4 Local and Private Memory Accesses Optimization

To optimize local memory access efficiency, consider the following guidelines:

- Implementing certain optimization techniques, such as loop unrolling, might lead to more concurrent memory accesses.
  - Increasing the number of memory accesses can complicate memory systems and degrade performance.

- If you have function scope local data (defined in the body of a `parallel_for_work_group lambda` or by using `group_local_memory_for_overwrite`), the Intel® oneAPI DPC++/C++ Compiler statically sizes the local data that you define within a function body at compilation time.
- For accessors in the local space, the host assigns their memory sizes dynamically at runtime. However, the compiler must set these physical memory sizes at compilation time. By default, that size is 16 kB.
  - The host 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 Intel® oneAPI DPC++/C++ Compiler to make better inferences about access patterns through static code analysis. For information about the benefits of static coalescing, refer to Static Memory Coalescing on page 116. Complex address calculations and pointer math operations can prevent the Intel® oneAPI DPC++/C++ Compiler from identifying the memory system accessed by a given load or store, leading to increased area use 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 pointers are subsequently retrieved from memory. Storing pointers to memory usually leads to suboptimal area and performance results.
- Create local array elements that are a power of two bytes to allow the Intel® oneAPI DPC++/C++ Compiler to provide an efficient memory configuration.
  - Whenever possible, the Intel® oneAPI DPC++/C++ Compiler automatically pads the elements of the local memory to be a power of two to provide a more efficient memory configuration. For example, if you have a `struct` containing three chars, the Intel® oneAPI DPC++/C++ Compiler pads it to four bytes, instead of creating a narrower and deeper memory with multiple accesses (that is, a 1-byte wide memory configuration). However, there are cases where the Intel® oneAPI DPC++/C++ Compiler might not pad the memory, such as when the kernel accesses local memory indirectly through pointer arithmetic.
  - To determine if the Intel® oneAPI DPC++/C++ Compiler has padded the local memory, review the memory dimensions in the Memory Viewer. If the Intel® oneAPI DPC++/C++ Compiler fails to pad the local memory, it prints a warning message in the Area Analysis of System report.

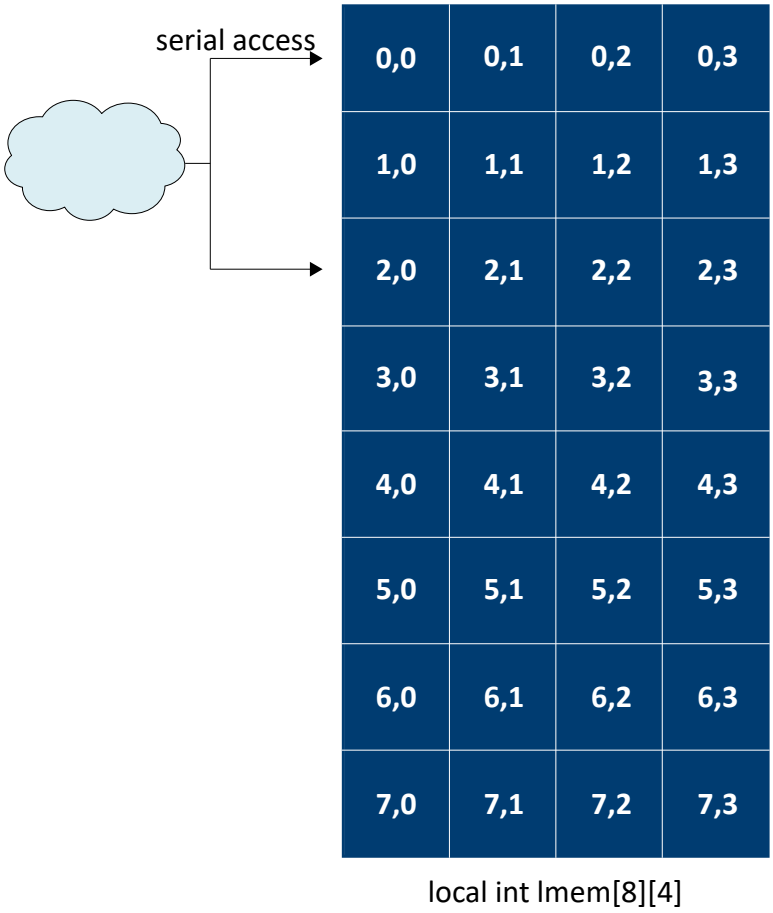
Local Memory Banks

Specifying the `numbanks(N)` and `bankwidth(M)` memory attributes allow you to configure the local memory banks for parallel memory accesses. The banking geometry described by these 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.

```
[[intel::fpga_memory]] int lmem[8][4];
#pragma unroll
for(int i = 0; i<4; i+=2) {
  lmem[i][x] = ...;
}
```

Figure 58. Serial Accesses to an 8 x 4 Local Memory System



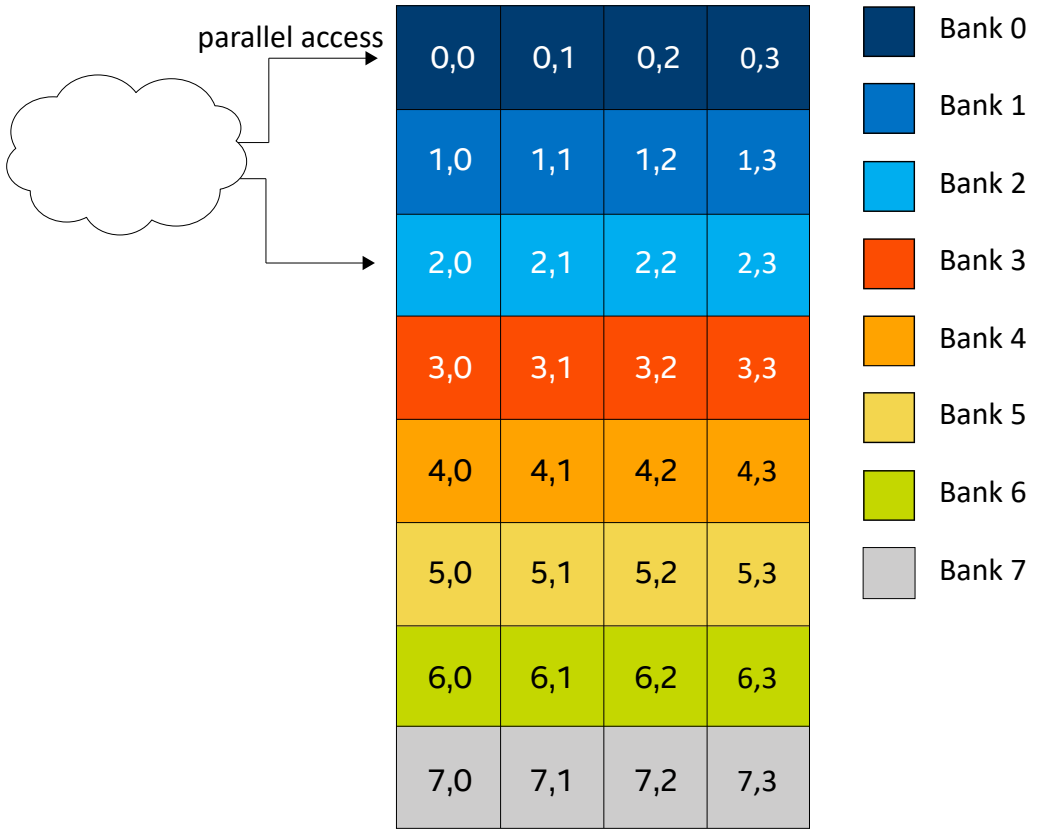
To improve performance, you can add `numbanks(N)` and `bankwidth(M)` in your code to define the number of memory banks and the bank widths in bytes. The following code implements eight memory banks, each 16-bytes wide. This memory bank configuration enables parallel memory accesses to the 8 x 4 array.

```
[[intel::numbanks(8), intel::bankwidth(16)]] int lmem[8][4];
#pragma unroll
for (int i = 0; i < 4; i+=2) {
  lmem[i][x & 0x3] = ...;
}
```

NOTE

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® oneAPI DPC++/C++ Compiler that `x` does not exceed the lower index bounds.

Figure 59. Parallel Access to an 8 x 4 Local Memory System with Eight 16-Byte-Wide Memory Banks



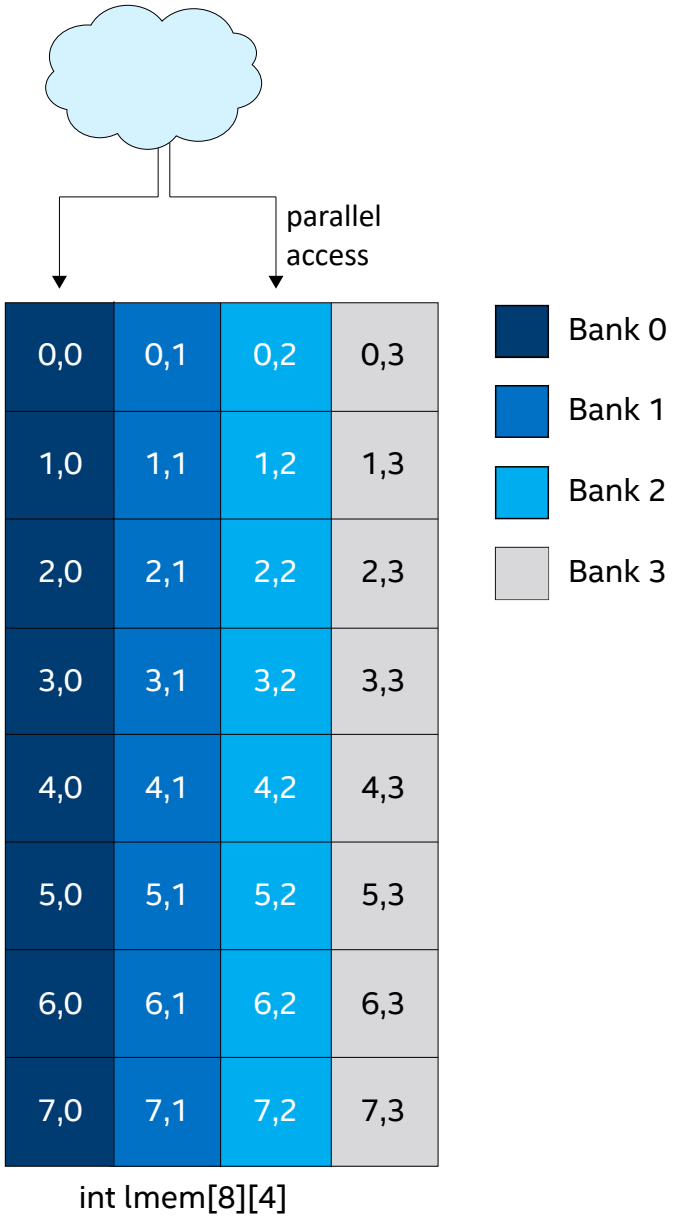
By specifying different values for the `numbanks(N)` and `bankwidth(M)` attributes, you can change the parallel access pattern. The following code implements four memory banks, each being four bytes wide:

NOTE

This memory bank configuration enables parallel memory accesses to the 8 x 4 array.

```
[[intel::numbanks(4), intel::bankwidth(4)]] int lmem[8][4];
#pragma unroll
for (int i = 0; i < 4; i+=2) {
  lmem[x][i] = ...;
}
```

Figure 60. Parallel Access to an 8 x 4 Local Memory System with Four 4-Byte-Wide Memory Banks

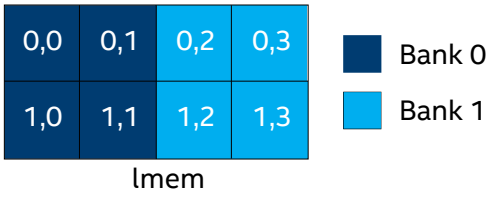


Optimize the Geometric Configuration of Local Memory Banks Based on Array Index

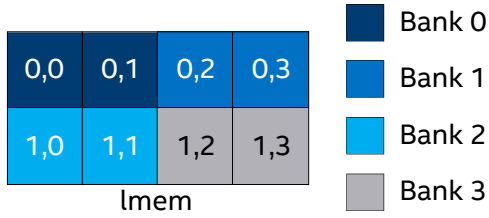
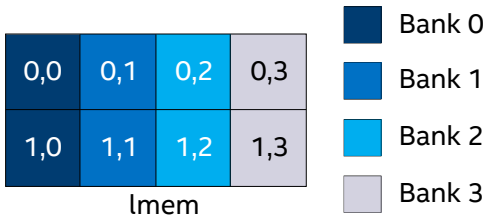
By default, the Intel® oneAPI DPC++/C++ Compiler attempts to improve performance by automatically banking a local memory system. The compiler includes advanced features that allow you to customize the banking geometry of your local memory system. To configure the geometry of local memory banks, include `numbanks(N)` and `bankwidth(M)` kernel attributes in your SYCL* kernel.

The table provides code examples illustrating how the bank geometry changes based on the values you assign to `numbanks` and `bankwidth`. The first and last rows of this table illustrate how to bank memory on the upper and lower indexes of a 2D array, respectively.

Table 14. Effects of `numbanks` and `bankwidth` on the Bank Geometry of 2 x 4 Local Memory System

Code Example	Bank Geometry
`[[intel::numbanks(2), intel::bankwidth(16)]] int lmem[2][4];`	
`[[intel::numbanks(2), intel::bankwidth(8)]] int lmem[2][4];`	

continued...

Code Example	Bank Geometry
`[[intel::numbanks(2), intel::bankwidth(4)]] int lmem[2][4];`	
`[[intel::numbanks(4), intel::bankwidth(8)]] int lmem[2][4];`	
`[[intel::numbanks(4), intel::bankwidth(4)]] int lmem[2][4];`	

3.1.3.5 Annotating Unified Shared Memory Pointers

When using Unified Shared Memory (USM) to perform a host allocation or a device allocation, Intel® recommends annotating raw USM pointers inside the kernel before accessing the pointers using the `host_ptr` or `device_ptr` object. The `host_ptr` and `device_ptr` objects are instances of the `multi_ptr` class in SYCL that provides constructors for address space qualified pointers.

Using `host_ptr` and `device_ptr` objects allow the compiler to perform better alias analysis, which typically leads to better throughput and smaller silicon area for your design. Also, host or device annotated pointers allow the compiler to infer simpler RTL, because Load-Store Units (LSUs) that want to access the USM pointers must be connected only to the host memory or only to the device memory, respectively. Without the annotations, the compiler is compelled to connect LSUs to both memories because the location of the pointer is unknown at compile time.

For example, when using the `malloc_device` function to define a pointer `Ptr`, construct a `device_ptr` object using the pointer `Ptr` inside the kernel, and access the `device_ptr` object directly instead of accessing the pointer `Ptr`:

```
T* ptr = malloc_device<T>(1024, Queue);
...
cgh.single_task<class DeviceAnnotation>([=]() {
  Ptr[0] = 42;  // load-store unit connected to both device and host memories
  device_ptr<T> DevicePtr(Ptr);
  DevicePtr[1] = 43;  // load-store unit connected only to the device memory
});
```

Similarly, when using the `malloc_host` function to define a pointer `Ptr`, construct a `host_ptr` object using the pointer `Ptr` inside the kernel and access the `host_ptr` object directly instead of accessing `Ptr`:

```
T* ptr = malloc_host<T>(1024, Queue);
...
cgh.single_task<class HostAnnotation>([=]() {
  Ptr[0] = 42;  // load-store unit connected to both device and host memories
  host_ptr<T> HostPtr(Ptr);
  HostPtr[1] = 43;  // load-store unit connected only to the host memory
});
```

CAUTION

Use annotations consistently and match them with the type of the runtime allocation used. Mismatches and inconsistencies when using the address space annotations are considered undefined behavior and may lead to incorrect results or hardware hangs.

Related Links

Unified Shared Memory (USM) interfaces

3.1.3.6 Zero-Copy Memory Access

Prior to the implementation of restricted USM, you had to access host's data from the device using one of the following methods:

- Through SYCL buffers
- By copying data between the host and the device using explicit USM

Both of these methods resulted in data transfers between the host and the device memory on discrete cards such as the Intel® FPGA Programmable Acceleration Card (PAC) D5005 (previously known as *Intel® FPGA Programmable Acceleration Card (PAC) with Intel® Stratix® 10 SX FPGA*).

With host allocations on devices supporting restricted USM, a kernel can directly access data over PCIe (no copying required). By using host allocations in designs that have infrequent random accesses to large pieces of data, you can improve throughput and latency of the design as a large piece of data no longer requires copying in full to the device. For detailed explanation of this concept, refer to the Zero-copy Data Transfer tutorial on GitHub.

- SYCL buffers created with host allocations set as `hostData` in the constructor still result in data copy from the host to the device memory. For more information, refer to Prepinning Memory on page 137.
- Shared allocation for the `pac_s10_usm` board that is in the `intel_s10sx_pac` BSP does not yield any change in the behavior or performance over host allocation. Both its host and shared allocations reside in the host.

NOTE

For additional information, refer to the FPGA tutorial sample `Zero-copy Data Transfer` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

Related Links

Unified Shared Memory (USM) Interfaces

3.1.3.7 Additional Recommendations

Optimizing memory accesses in your kernels can improve overall kernel performance. Consider implementing the following techniques for optimizing memory accesses:

- Avoid designing systems where one kernel writes an intermediate result to global memory and another kernel reads this data back from global memory. Instead, implement a SYCL* pipe (described in Pipes) between the producer and consumer kernels for direct data transfer. Alternatively, you can merge both kernels into a single larger kernel and use helper functions to logically separate the two original kernels.
- The Intel® oneAPI DPC++/C++ Compiler implements local memory in FPGAs differently than in GPUs. If your kernel contains code to avoid GPU-specific local memory bank conflicts, remove that code because the compiler generates hardware that avoids local memory bank conflicts automatically whenever possible.

3.1.4 Pipes

Pipes provide a mechanism for passing data between kernels and synchronizing kernels with high efficiency and low latency. Pipes allow kernels to communicate directly with each other using on-device FIFO buffers that are implemented using FPGA memory resources. The Intel® oneAPI DPC++/C++ Compiler supports concurrent kernel execution by launching each kernel in a separate command queue. Using pipes for data movement between concurrently executing kernels allows for data transfer without waiting for kernel completion, which can significantly increase the throughput of your design. Refer to Pipes Extension for more details about how to use pipes in your device code.

3.1.5 Host

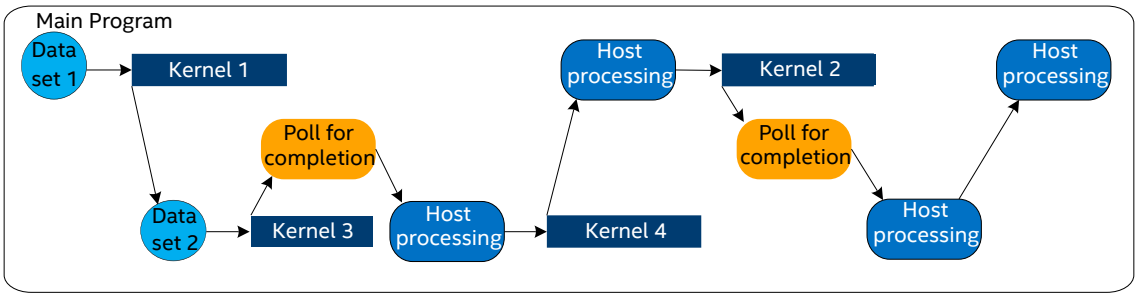
This section shows ways you can write your host to improve your application's throughput.

3.1.5.1 Multi-Threaded Host Application

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

The following figure illustrates how a single-threaded host application might process parallel, independent datapaths between kernel executions:

Figure 61. Kernel Execution by a Single-Threaded Host Application



The SYCL* runtime is thread safe and supports multithreaded applications. Thus, you can perform tasks on the host in parallel threads while still allowing those threads to access the SYCL APIs in a thread-safe way.

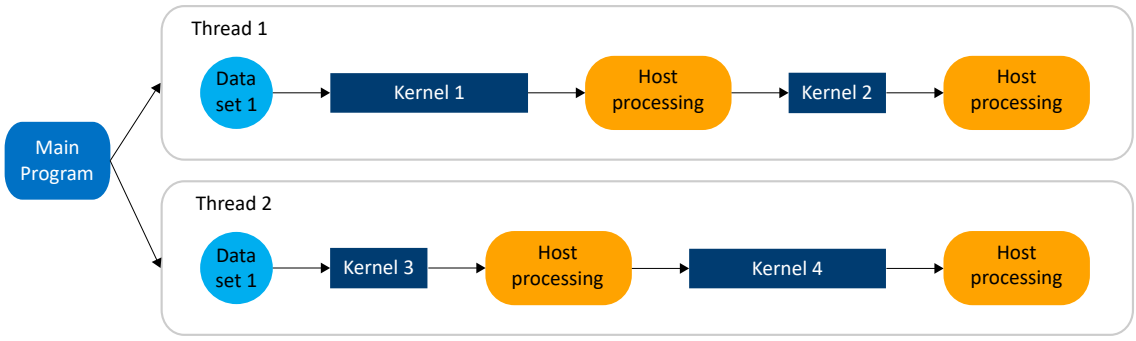
NOTE

A SYCL system that has FPGAs installed does not support multi-process execution. Creating a context opens the device associated with the context and locks it for that process. No other process may use that device. Some queries about the device through `device.get_info<>()` also opens up the device and locks it to that process since the runtime needs to query the actual device to obtain that information. The following are examples of queries that lock the device:

- `is_endian_little`
- `global_mem_size`
- `local_mem_size`
- `max_constant_buffer_size`
- `max_mem_alloc_size`
- `vendor`
- `name`
- `is_available`

The following figure illustrates how a multi-threaded host application processes parallel, independent datapaths between kernel executions:

Figure 62. Kernel Execution by a Multi-Threaded Host Application in a Thread-Safe Runtime Environment

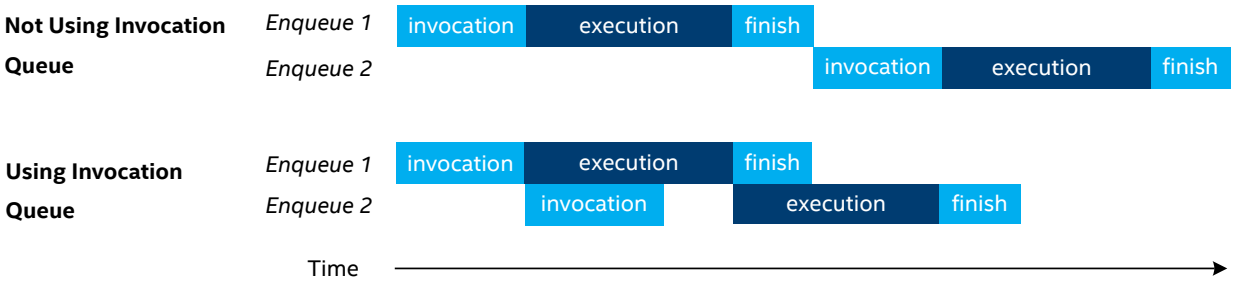


3.1.5.2 Utilizing Hardware Kernel Invocation Queue

Kernel invocation queue is a first-in first-out (FIFO) buffer used by the SYCL* runtime to store arguments for multiple kernel invocations on the device. Once the kernel finishes execution, the invocation queue allows the next invocation of the kernel to start immediately after. SYCL kernels are built with invocation queue to enable immediate launch of the next invocation.

As illustrated in the following figure, when the invocation queue is used, system and SYCL runtime environment overheads (from responding to the finish and sending in the next set of invocation arguments) are overlapped with the kernel executions. This allows kernels to execute continuously, maximizing the system level throughput.

Figure 63. Kernel Execution with and without Invocation Queue



SYCL kernel invocations are queued in hardware when the same SYCL kernel function is already running on the device, and the following are true:

- SYCL kernel was not compiled with hardware kernel invocation buffer disabled (`-Xsno-hardware-kernel-invocation-queue`). See Disable Hardware Kernel Invocation Queue (`-Xsno-hardware-kernel-invocation-queue`) on page 168

- SYCL kernel was not compiled with performance counters (`-Xsprofile`). See Instrument the Kernel Pipeline with Performance Counters (-Xsprofile) on page 66
- Any host to device synchronization operation (such as, host accessor, buffer destruction, and so on) is done between sequential kernel enqueues that requires first enqueue to finish.

Consider the following definitions of `simple_kernel()` and `check_output()` functions:

simple_kernel()

```
void simple_kernel(queue &deviceQueue,
                   buffer<cl_float, 1> &bufferA,
                   buffer<cl_float, 1> &bufferC)
{
  deviceQueue.submit([&](handler& cgh) {
    accessor accessorA(bufferA, cgh, read_only);
    accessor accessorC(bufferC, cgh, read_write, no_init);

    cgh.single_task<class SimpleAdd>([=]() {
      for (int i = 0; i < N; i++) {
        accessorC[i] = accessorA[i] + accessorA[i];
      }
    });
  });
}
```

check_output()

```
void check_output(buffer<cl_float, 1> &outBuffer) {
  accessor output_buf_acc(outBuffer, read_only);
  ...
  // Check output
  ...
}
```

Based on the function definitions of `simple_kernel()` on page 130 and `check_output()` on page 130, consider the following example code snippet where the kernel enqueue can be queued on the hardware kernel invocation queue:

```
// Example 1
main()
{
  ...
  simple_kernel(device_queue, bufferA, bufferC);
  simple_kernel(device_queue, bufferX, bufferZ);
  check_output(bufferC);
  check_output(bufferZ);
  ...
}
```

As soon as the first enqueue of `SimpleAdd` kernel is running, the second enqueue can be queued since they have no dependency.

Now, consider the following example code where kernel invocation cannot be queued on hardware:

```
// Example 2
main()
{
  ...
  simple_kernel(device_queue, bufferA, bufferC);
  check_output(bufferC);
  simple_kernel(device_queue, bufferX, bufferZ);
  check_output(bufferZ);
  ...
}
```

Creating the `output_buf_acc` accessor for the output buffer in check_output() function is a synchronization point that blocks the SYCL runtime until the first enqueue of `SimpleAdd` kernel is complete.

3.1.5.3 Double Buffering Host Utilizing Kernel Invocation Queue

Double buffering in SYCL* host application allows SYCL runtime environment to coalesce memory transfers and kernel execution.

In an application where the FPGA kernel is executed multiple times, the host must perform the following processing and buffer transfers before each kernel invocation.

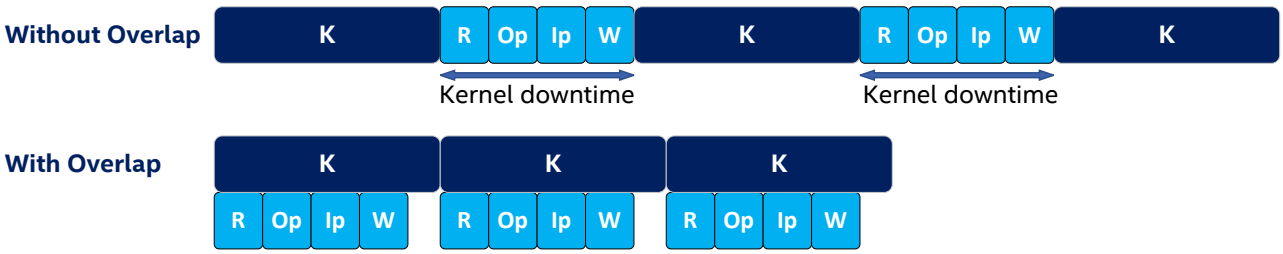
1. The output data from the *previous* invocation must be transferred from device to host and then processed by the host. Examples of this processing include:
   - Copying the data to another location
   - Rearranging the data
   - Verifying it in some way
2. The input data for the *next* invocation must be processed by the host and then transferred to the device. Examples of this processing include:
   - Copying the data from another location
   - Rearranging the data for kernel consumption
   - Generating the data in some way

Without double buffering, host processing and buffer transfers occur between kernel executions. Therefore, there is a gap in time between kernel executions, which you can refer as *kernel downtime* (See Figure 64 on page 132). If these operations overlap with kernel execution, the kernels can execute back-to-back with minimal downtime, thereby increasing overall application performance.

Determine When is Double Buffering Possible

Consider the following illustration:

Figure 64. Double Buffering Host



Following are the definitions of the required variables:

- ***R***: Time to transfer the kernel's output buffer from device to host.
- ***Op***: Host-side processing time of kernel output data (output processing)
- ***Ip***: Host-side processing time for kernel input data (input processing)
- ***W***: Time to transfer the kernel's input buffer from host to device.
- ***K***: Kernel execution time

In general, *R*, *Op*, *Ip*, and *W* operations must all complete before the next kernel is launched. To maximize performance, while one kernel is executing on the device, these operations should execute simultaneously on the host and operate on a second set of buffer locations. They should complete before the current kernel completes, thus allowing the next kernel to be launched immediately with no downtime. In general, to maximize performance, the host must launch a new kernel every *K*.

This leads to the following constraint:

```
R + Op + Ip + W <= K, to minimize kernel downtime
```

If the above constraint is not satisfied, a performance improvement may still be observed because of some overlap (perhaps not complete overlap) is still possible.

Measure the Impact of Double Buffering

You must get a sense of the kernel downtime to identify the degree to which this technique can help improve performance.

This can be done by querying the total kernel execution time from the runtime and comparing it to the overall application execution time. In an application where kernels execute with minimal downtime, these two numbers are close. However, if kernels have a lot of downtime, overall execution time notably exceeds kernel execution time.

Hardware Kernel Invocation Queue While Double Buffering Example

To utilize hardware kernel invocation queue while double buffering, write your host code as shown in the following code snippet:

```
main()
{ ...
  initialize_input(input_buf[0]);
  initialize_input(input_buf[1]);
```

```
  simple_kernel(device_queue, input_buf[0], output_buf[0]);
  for (int i = 1; i < TIMES; i++) {
    simple_kernel(device_queue,
                  input_buf[i%2],
                  output_buf[i%2]); // Launch the next kernel
    // Process output from previous kernel.
    // This will block on kernel completion.
    check_output(output_buf[(i-1)%2]);
    // Generate input for the next kernel.
    initialize_input(input_buf[(i-1)%2]);
  }
   ...
}
```

The following is the example function definition for `initialize_input()`:

```
void initialize_input (buffer<cl_float, 1> &inBuffer){
  accessor buf_acc(inBuffer, write_only, no_init);
  for (int i = 0; i < N; i++) {
    buf_acc[i] = rand();
  }
}
```

NOTE

For additional information, refer to the FPGA tutorial sample "Double Buffering" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

3.1.5.3.1 Applying Double-Buffering Using the Intercept Layer for OpenCL* Applications

The Intercept Layer for OpenCL* Applications is an open-source tool that you can use to profile oneAPI designs at a system-level. Although it is not part of the Intel® oneAPI Base Toolkit installation, it is freely available on GitHub*. For more information, refer to System-level Profiling Using the Intercept Layer for OpenCL* Applications on page 76.

The double-buffering optimization can help minimize or remove gaps between consecutive kernels as they wait for host processing to finish. These gaps are minimized or removed by having the host perform processing operations on a second set of buffers while the kernel executes. With this execution order, the host processing is done by the time the next kernel can run, so kernel execution is not held up waiting for the host. For additional information, refer to the FPGA tutorial sample "Double Buffering" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

In this example, the first three kernels are run without the double-buffer optimization, and the next three are run with it. The kernels were run on an Intel® Programmable Acceleration Card with Intel® Arria® 10 GX FPGA when the intercept layer data was collected. The change made by this optimization can be clearly seen in the Intercept Layer for OpenCL* Applications trace:

Figure 65. OpenCL Intercept Layer - With and Without Double Buffering

Here, the kernel runs named `_ZTS10SimpleVpow` can be recognized as the bars with the largest execution time (the large orange bars). Double buffering removes the gaps between the kernel executions that can be seen in the top trace image. This optimization improves the throughput of the design.

The Intercept Layer for OpenCL* Applications makes the need for and benefits of the optimization clear. Use the Intercept Layer tool on your designs to identify scenarios where you can apply double buffering (and other optimizations).

3.1.5.4 N-Way Buffering to Overlap Kernel Execution

N-way buffering is a generalization of the double buffering optimization technique. This system-level optimization enables kernel execution to occur in parallel with host-side processing and buffer transfers between host and device, improving application performance. N-way buffering can achieve this overlap even when the host-processing time exceeds kernel execution time.

In an application where the FPGA kernel is executed multiple times, the host must perform the following processing and buffer transfers before each kernel invocation.

1. The output data from the *previous* invocation must be transferred from the device to the host and then processed by the host. Examples of this processing include the following:
   - Copying the data to another location
   - Rearranging the data
   - Verifying it in some way
2. The input data for the *next* invocation must be processed by the host and then transferred to the device. Examples of this processing include:
   - Copying the data from another location
   - Rearranging the data for kernel consumption
   - Generating the data in some way

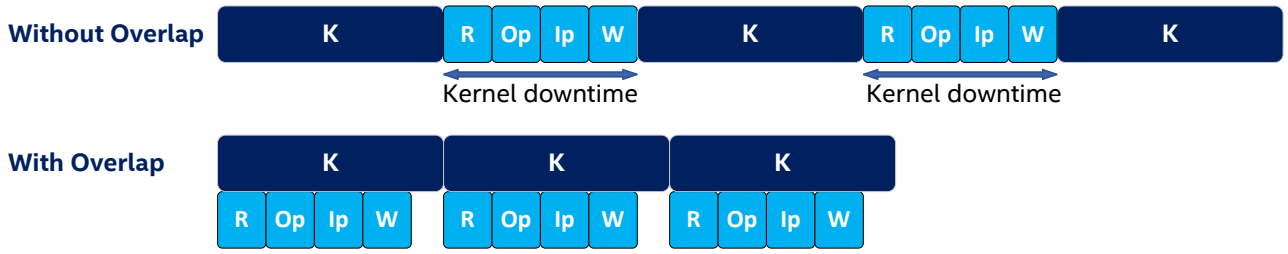
Without the N-way buffering, host processing and buffer transfers occur between kernel executions. Therefore, there is a gap in time between kernel executions, which you can refer as *kernel downtime* (See Figure 66 on page 135). If these operations overlap with kernel execution, the kernels can execute back-to-back with minimal downtime, thereby increasing the overall application performance.

Determine When is N-Way Buffering Possible

N-way buffering is frequently referred to as *double buffering* in the most common case where N=2.

Consider the following illustration:

Figure 66. N-Way Buffering Host



The following are the definitions of the required variables:

- ***R***: Time to transfer the kernel's output buffer from device to host.
- ***Op***: Host-side processing time of kernel output data (output processing).
- ***Ip***: Host-side processing time for kernel input data (input processing).
- ***W***: Time to transfer the kernel's input buffer from host to device.
- ***K***: Kernel execution time
- ***N***: The number of buffer sets used.
- ***C***: The number of host-side CPU cores.

In general, R, Op, Ip, and W operations must all complete before the next kernel is launched. To maximize performance, while one kernel is executing on the device, these operations must run in parallel and operate on a separate set of buffer locations. They must complete before the current kernel completes, thus allowing the next kernel to be launched immediately with no downtime. In general, to maximize performance, the host must launch a new kernel every K.

If these host-side operations are executed serially, this leads to the following constraint:

```
R + Op + Ip + W <= K, to minimize kernel downtime
```

In the above example, if the constraint is satisfied, the application requires two sets of buffers. In this case, N=2. However, the above constraint may not be satisfied in some applications if host-processing takes longer than the kernel execution time.

NOTE

A performance improvement may still be observed because kernel downtime may still be reduced (though perhaps not maximally reduced).

In this case, to further improve performance, reduce host-processing time through multithreading. Instead of executing the above operations serially, perform the input and output-processing operations in parallel using two threads, leading to the following constraint:

```
Max (R+Op, Ip+W) <= K

R + W <= K, to minimize kernel downtime
```

If the above constraint is still unsatisfied, the technique can be extended beyond two sets of buffers to N sets of buffers to help improve the degree of overlap. In this case, the constraint becomes:

```
Max (R + Op, Ip + W) <= (N-1)*K

R + W <= K, to minimize kernel downtime.
```

The idea of N-way buffering is to prepare N sets of kernel input buffers, launch N kernels, and when the first kernel completes, begin the subsequent host-side operations. These operations may take a long time (longer than K), but they do not cause kernel downtime because an additional N-1 kernels have already been queued and can launch immediately. By the time these first N kernels complete, the aforementioned host-side operations would have also completed, and the N+1 kernel can be launched with no downtime. As additional kernels complete, corresponding host-side operations are launched on the host, using multiple threads in a parallel fashion. Although the host operations take longer than K, if N is chosen correctly, they complete with a period of K, which is required to ensure you can launch a new kernel every K. To reiterate, this scheme requires multi-threaded host-operations because the host must perform processing for up to N kernels in parallel to keep up.

Use the above formula to calculate the N required to minimize downtime. However, there are some practical limits:

- N sets of buffers are required on both the host and device. Therefore, both must have the capacity for this many buffers.
- If the input and output processing operations are launched in separate threads, then (N-1)*2 cores are required so that C can become the limiting factor.

Measure the Impact of N-Way Buffering

You must get a sense of the kernel downtime to identify the degree to which this technique can help improve performance.

You can do this by querying the total kernel execution time from the runtime and comparing it to the overall application execution time. In an application where kernels execute with minimal downtime, these two numbers are close. However, if kernels have a lot of downtime, overall execution time notably exceeds the kernel execution time.

NOTE

For additional information, refer to the FPGA tutorial sample "N-Way Buffering" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

3.1.5.5 Prepinning Memory

You must consider how the transfer of data from the host to the device occurs when optimizing kernel memory accesses. For designs that have longer data transfer times than the compute time, the data transfer time may be a bottleneck. On devices supporting greater than a PCIe Gen3 x 8 transfer rates, prepinning the memory that is on the host prior to its transfer allows for it to transfer at a higher bandwidth. For example, on the Intel® FPGA Programmable Acceleration Card (PAC) D5005 (previously known as *Intel® FPGA Programmable Acceleration Card (PAC) with Intel® Stratix® 10 SX FPGA*) that has a PCIe Gen3 x16 transfer rate, memory transfer with prepinning achieves approximately 12 GB/s in half-duplex and 21 GB/s in full-duplex.

The following is an example of how to allocate buffers on prepinned memory:

```
intel::fpga_selector device_selector;
auto device_queue = queue(device_selector);
int* in = malloc_host<int>(1024, device_queue.get_context());
auto buf_pinned = buffer<int, 1>(in, 1024);
```

RESTRICTION

Prepinned memory is allocated through the unified shared memory (USM) call `malloc_host()`. Hence, a prepinned memory is available only on devices that support Restricted USM.

Pinned memory is a scarce resource on the system, so carefully consider which buffers you want to pin to avoid exceeding the system limit. In addition, pinning itself is an expensive operation, so for optimal performance, ensure that the creation of pinned buffers takes place outside the main compute loop.

3.1.5.6 Simple Host-Device Streaming

You can take advantage of SYCL Unified Shared Memory host allocation and zero-copy host memory to implement a streaming host-device design with low latency and high throughput.

TIP

Before you begin learning how to stream data, review Pipes on page 127 and Zero-Copy Memory Access on page 126 concepts.

NOTE

SYCL USM host allocations are only supported by some BSPs, such as the Intel® FPGA Programmable Acceleration Card (PAC) D5005 (previously known as *Intel® FPGA Programmable Acceleration Card (PAC) with Intel® Stratix® 10 SX FPGA*). Check with your BSP vendor to see if they support SYCL USM host allocations.

To understand the concept, consider the following design example:

- An offload design that maximizes throughput with no optimization for latency. See `DoWorkOffload` in `simple_host_streaming.cpp`.
- A single-kernel design that uses the methods described below to achieve a much lower latency while maintaining throughput. See `DoWorkSingleKernel` in `simple_host_streaming.cpp` and `single_kernel.hpp`.
- A multi-kernel design that uses the methods described below to achieve a much lower latency while maintaining throughput. See `DoWorkMultiKernel` in `simple_host_streaming.cpp` and `multi_kernel.hpp`.

Offload Processing

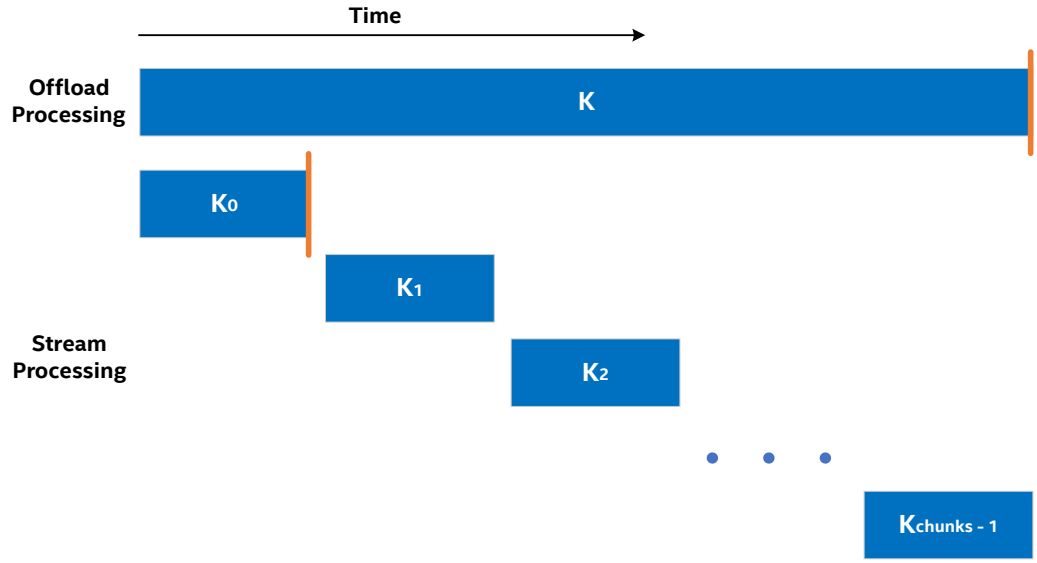
Typical SYCL designs perform offload processing. All of the input data is prepared by the CPU, and then transferred to an FPGA device. Kernels are started on the device to process the data. When the kernels finish, the CPU copies the output data from the FPGA back to its memory. Memory synchronization is achieved on the host after the device kernel signals its completion.

Offload processing achieves excellent throughput when the memory transfers and kernel computation are performed on large data sets, as the CPU's kernel management overhead is minimized. You can conceal data transfer overhead using double buffering or N-way buffering to maximize kernel throughput. However, a significant shortcoming of this design pattern is latency. The coarse-grain synchronization of waiting for the entire set of data to be processed results in a latency that is equal to the processing time of the entire data.

Host-device Streaming Processing

The method for achieving lower latency between the host and device is to break data set into smaller chunks and, instead of enqueueing a single long-running kernel, launch a set of shorter-running kernels. Together, these shorter-running kernels process the data in smaller batches. As memory synchronization occurs upon kernel completion, this strategy makes the output data available to the CPU in a more granular way. This is illustrated in the following figure, where the orange lines illustrate the time when the first set of data is available in the host:

Figure 67. Host-device Streaming Processing



In the streaming version, the first piece of data is available in the host earlier than in the offload version. To understand how much earlier, consider that you have `total_size` elements of data to process and you break the computation into `chunks` chunks of size `chunk_size=total_size/chunks` (as is the case in Figure 67 on page 139). Then, in an ideal situation, the streaming design achieves a latency that is `chunks` times better than the offload version.

Setting the `chunk_size`

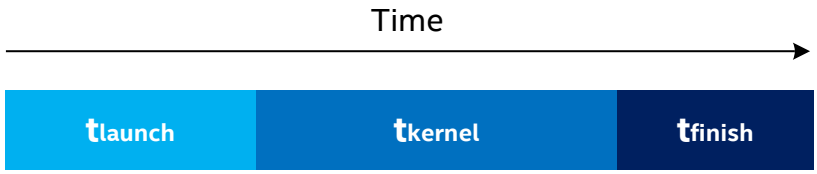
You might wonder if you can set the `chunk_size` to 1 (i.e., `chunks=total_size`) to minimize the latency. In the figure above, you may notice small gaps between the kernels in the streaming design (for example, between K_0 and K_1). This is caused by the overhead of launching kernels and detecting kernel completion on the host. These gaps increase the total processing time and therefore, decrease the throughput of the design (that is, when compared to the offload design, it takes more time to process the same amount of data). If these gaps are negligible, then the throughput is negligibly affected.

In the streaming design, the choice of the `chunk_size` is thus a tradeoff between latency (a smaller chunk size results in a smaller latency) and throughput (a smaller chunk size increases the relevance of the inter-kernel latency).

Lower Bounds on Latency

Lowering the `chunk_size` can reduce latency, sometimes at the expense of throughput. However, even if you are not concerned with throughput, there still exists a lower-bound on the latency of a kernel. Review the following figure:

Figure 68. Lower Bounds on Latency



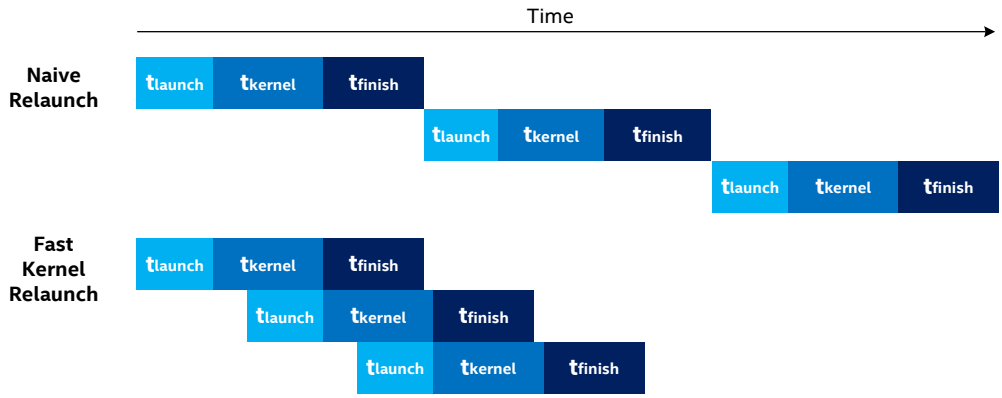
In the Figure 68:

- t_{launch} is the time for a kernel launch signal to go from the host to the device.
- t_{kernel} is the time for the kernel to execute on the device.
- t_{finish} is the time for the finished signal to go from the device to the host.

Even if you set `chunk_size` to 0 (i.e., launch a kernel that does nothing) and therefore t_{kernel} ~= 0, the latency is still t_{launch} + t_{finish}. In other words, the lower bound on kernel latency is the time needed for the *start* signal to go from the host to the device and for the *finished* signal to go from the device to the host.

In the *Setting the `chunk_size`* section above, you learned how gaps between kernel invocations can degrade throughput. In the Figure 68 on page 140, there appears to be a minimum t_{launch} + t_{finish} gap between kernel invocations. This is illustrated as the Naive Relaunch timeline in the following figure:

Figure 69. Naive Relaunch Timeline

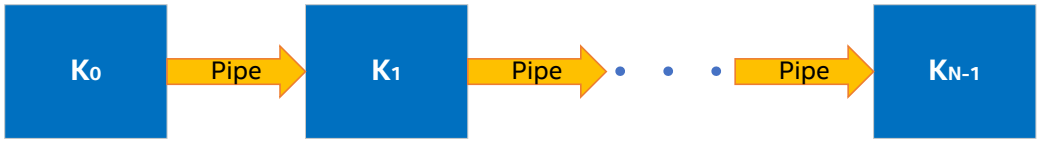


Fortunately, this overhead is circumvented by an automatic runtime kernel launch scheme that buffers kernel arguments on the device before the previous kernel finishes. This enables kernels to queue on the device and to begin execution without waiting for the previous kernel's *finished* to propagate back to the host. You can refer to this as *Fast Kernel Relaunch*, and it is also illustrated in the Figure 69. Fast kernel relaunch reduces the gap between kernel invocations and allows you to achieve lower latency while maintaining throughput.

Multiple Kernel Pipeline

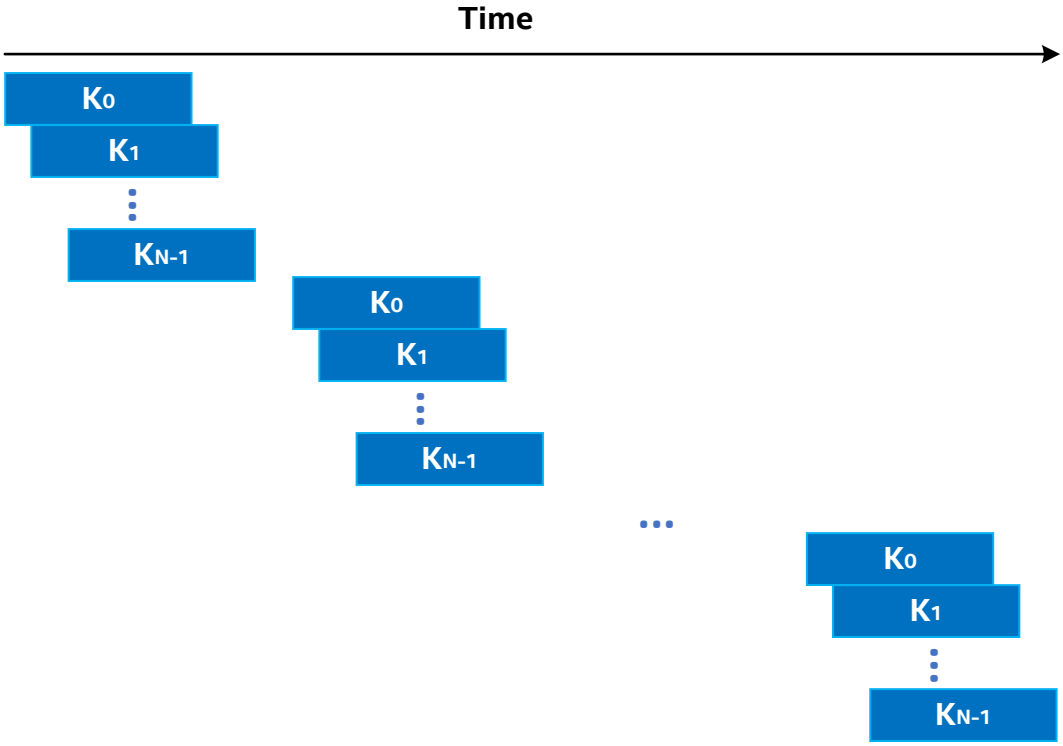
More complicated FPGA designs often instantiate multiple kernels connected by SYCL pipes (for examples, see FPGA Reference Designs). Suppose you have a kernel system of N kernels connected by pipes, as shown in the following figure:

Figure 70. Multiple Kernels Connected by Pipes



With the goal of achieving lower latency, you use the Host-device Streaming Processing technique to launch multiple invocations of your N kernels to process chunks of data. This gives you a timeline as shown in the following figure:

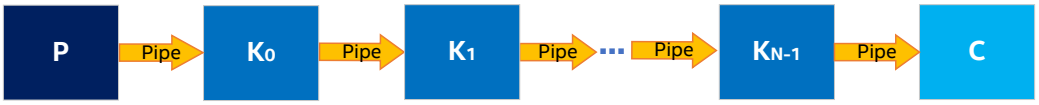
Figure 71. Multiple Invocation of Kernels to Process Chunks of Data



Notice the gaps between the start times of the N kernels for a single chunk. This is the t_{launch} time discussed in the Host-device Streaming Processing on page 138. However, the multi-kernel design introduces a potential new lower bound on the latency for a single chunk because processing a single chunk of data requires launching N kernels, which takes N x t_{launch}. If N (the number of kernels in your system) is sufficiently large, this limits your achievable latency.

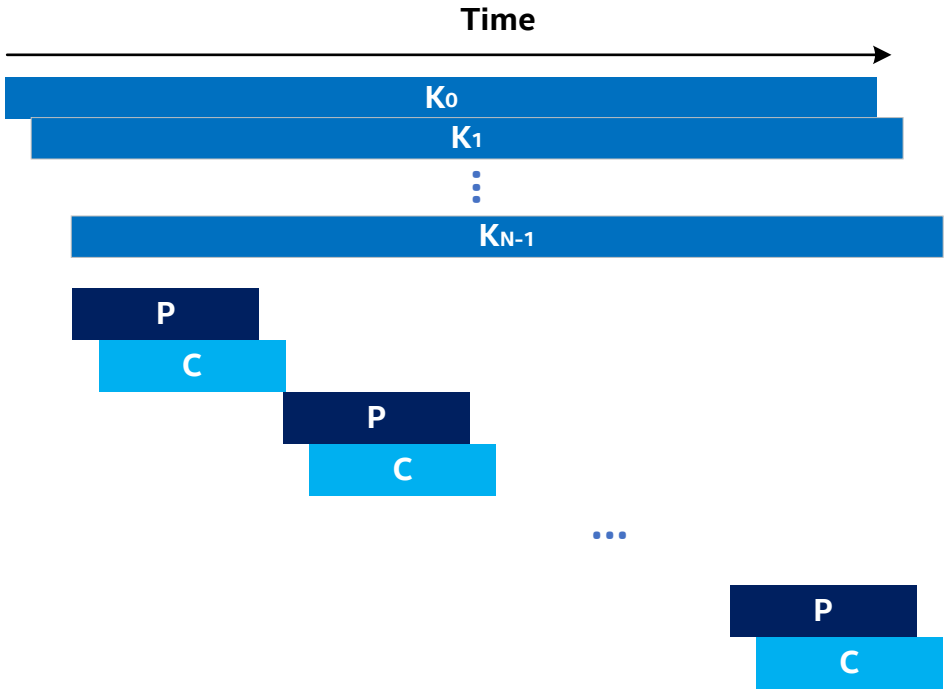
For designs with $N > 2$, Intel recommends a different approach. The idea is to enqueue your system of N kernels once, and to introduce Producer (P) and Consumer (C) kernels to handle the production and consumption of data from and to the host, respectively. This method is illustrated in the following figure:

Figure 72. Enqueuing a System of Kernels



The Producer streams data from the host and presents it to the kernel system through a SYCL pipe. The output of the kernel system is consumed by the Consumer and written back into host memory. To achieve low latency, you still process the data in chunks, but instead of having to enqueue N kernels for each chunk, you must only enqueue a single Producer and Consumer kernel per chunk. This enables you to reduce the lower bound on the latency to MAX(2 x t_{launch}, t_{launch} + t_{finish}). Observe that this lower bound does not depend on the number of kernels in your system (N).

Use this method only when N > 2. FPGA area is sacrificed to implement the Producer, Consumer, and their pipes to achieve lower overall processing latency.



Limitations

Fundamentally, the ability to stream data between the host and device is built around SYCL USM host allocations. The underlying problem is how to efficiently synchronize between the host and device to signal that some data is ready to be processed, or has been processed. In other words, how does the host signal to the device that some data is ready to be processed? Conversely, how does the device signal to the host that some data is done being processed?

One method to achieve this signaling is to use the start of a kernel to signal to the device that data is ready to be processed, and the end of a kernel to signal to the host that data has been processed. This is the approach taken in the examples discussed above. However, this method has two notable drawbacks. First, the latency to start and end kernels is high. To maintain high throughput, you must size the `chunk_size`

sufficiently large to hide the inter-kernel latency, resulting in a latency increase. Second, the programming model to achieve this performance is non-trivial as you must intelligently manage the SYCL device queue.

NOTE

For additional information, refer to the FPGA tutorial sample `Simple Host Streaming` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

3.1.5.7 Buffered Host-Device Streaming

In this topic, you learn how to optimize a full system design that streams data from the host to the device and back to the host and create heterogeneous designs that can achieve high throughput. The techniques described in this topic are not specific to a CPU-FPGA system, and you can apply them to GPUs, multi-core CPUs, and other processing units as well.

IMPORTANT

Prior to learning about buffered host-device streaming concept, you must review the following concepts:

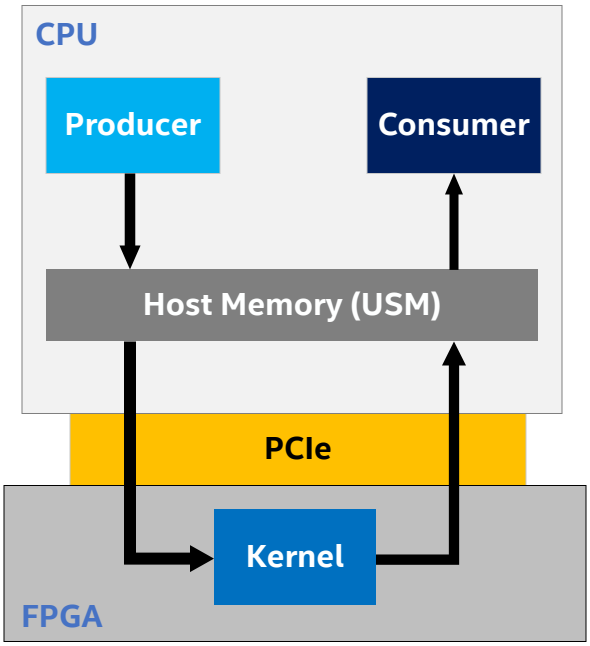
- Simple Host-Device Streaming on page 137
- Double Buffering Host Utilizing Kernel Invocation Queue on page 131
- N-Way Buffering to Overlap Kernel Execution on page 134
- Multithreaded C++ Programming

To understand the concept of buffered host-device streaming, consider an example design where a *Producer* (running on the CPU) produces data into USM host allocations, a *Kernel* (running on the FPGA) processes this data and produces output into host allocations, and a *Consumer* (running on the CPU) consumes the data. Data is shared between the host and FPGA device via host pointers (pointers to USM host allocations).

Figure 73. Example Design with a Producer, Kernel, and Consumer

In heterogeneous systems, it is important to understand how different compute architectures are connected (that is, how data is transferred from one to another) to better reason about and address performance bottlenecks in the system. The following figure is a slightly more detailed illustration of the processing pipeline in the example design, which shows the data flow between the Producer, Kernel, and Consumer. It illustrates that the Producer and Consumer operations both execute on the CPU and communicate with the FPGA kernel over a PCIe link (PCIe Gen3 x16).

Figure 74. Detailed Illustration of the Processing Pipeline



Roofline Analysis

When designing for throughput, you must first estimate the maximum throughput the system is capable of. This back-of-the-envelop calculation is called *roofline analysis*. To calculate the maximum achievable throughput of the Producer-Kernel-Consumer design shown in Figure 74, assume that the Producer and Consumer have infinite bandwidth (that is, they can produce or consume data at an infinite rate, respectively). Then, the bottleneck in the system is the FPGA kernel. Assume that the kernel has an f_{MAX} of 400MHz and processes 64 bytes per cycle. The following is kernel's maximum steady-state throughput:

400 MHz * 64 bytes/cycle = 25.6 GB/s ~= 26 GB/s.

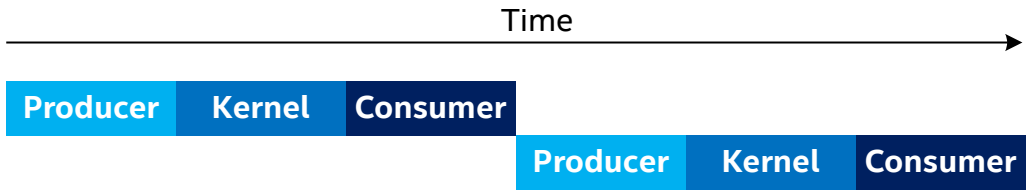
In reality, the rate of data transfer to and from the kernel depends on the bandwidth of the PCIe link. In Figure 74, you can observe that this depends on the bandwidth of the PCIe link. It has been experimentally measured that the PCIe Gen3 x16 link has a total bandwidth of ~22 GB/s (11 GB/s bidirectional). This means that, while the kernel can process data at 26 GB/s, you can only send data to it and receive from it at a rate of 11 GB/s. The system bottleneck is the PCIe link between the CPU and FPGA. It is common for the bottleneck in a heterogenous system to be the inter-device link, rather than any single device.

In this analysis so far, you assumed that the Producer and Consumer had infinite bandwidth. You can now perform a roofline analysis on the entire system by assuming a finite bandwidth for the Producer and Consumer. For the entire Producer-Kernel-Consumer design, the maximum possible throughput is the minimum of the Producer, Kernel, and Consumer throughput, and the bandwidth of the link (in this case, PCIe) connecting the CPU and FPGA (a chain is only as strong as the weakest link).

Optimizing the Design for Throughput

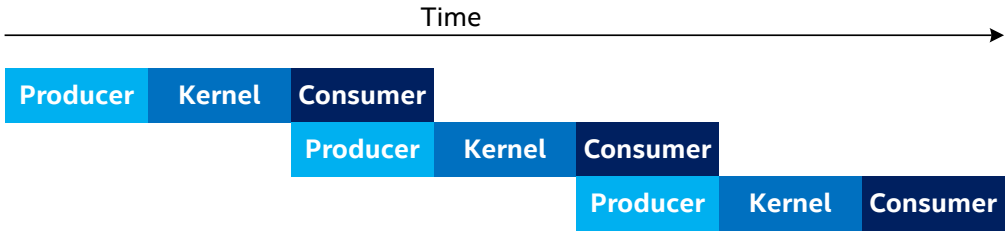
A naive approach to this design is for the Producer, Kernel, and Consumer to run sequentially that is depicted by the following timing diagram. This naive approach underperforms because operations, which could run in parallel, are run sequentially. For example, in the following figure, the Producer could be producing the second set of data, while the Consumer is consuming the first set of data:

Figure 75. Timing Diagram



To improve the performance, you can create a separate CPU thread (called the Producer thread) that produces all of the data for the Kernel to process, and consumes later as shown in the following figure. This allows the Producer and Consumer to run in parallel (that is, the Producer thread produces the next set of data, while the Consumer consumes the current data).

Figure 76. Timing Diagram with Producer and Consumer Running in Parallel

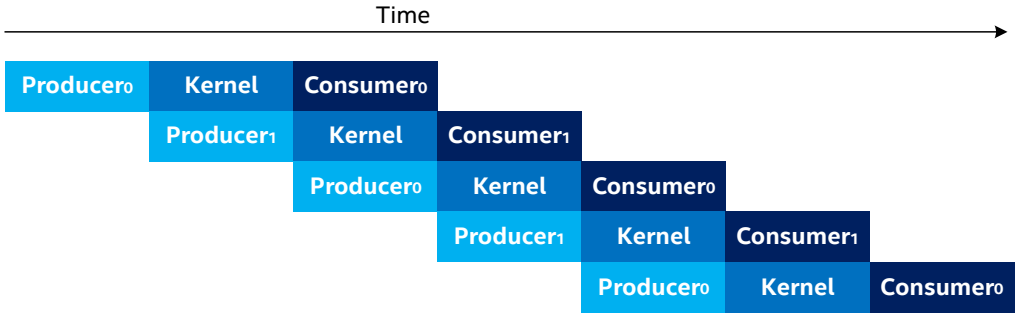


This approach improves the design's throughput, but does require more complicated thread synchronization logic to synchronize between the Producer thread and the main thread that is launching kernels and consuming its output.

Observe that in the Figure 76, the Producer and Consumer are running simultaneously (in separate threads). You must consider the fact that both of these processes are running on the CPU in parallel, which affects their individual throughput capabilities. For example, running these processes in parallel can result in saturating the CPU's memory bandwidth, and therefore lower the overall throughput for the Producer and Consumer processes even if the threads run on different CPU cores.

By putting the Producer into its own thread, you can improve performance by producing and consuming data simultaneously. However, it is ideal if the Producer can produce the next set of data while the kernel is processing the current data. This allows the next kernel to launch as soon as the current kernel finishes, instead of waiting for the Producer to finish, as shown in Figure 76. Unfortunately, you cannot produce data into the buffer that the kernel is currently processing, since it is still in use by the kernel. To address this, use multiple buffers (that is, N-Way buffering) of the input and output buffers, which results in a timing diagram, as shown in the following:

Figure 77. Timing Diagram With Multiple Buffers



The subscript number on the Producer and Consumer (for example, $Producer_0$ and $Consumer_0$) represents the buffer index they are processing. The Figure 77 illustrates the case where you use two sets of buffers (double-buffering). Observe that in Figure 77, the Producer produces into buffer 1 ($Producer_1$) while the kernel is processing the data from buffer 0. Thus, by the time the kernel finishes processing buffer 0, it can start processing buffer 1 right away, so long as the Producer has finished producing the next buffer of data.

In the steady-state for Figure 77, the throughput of the design is bottlenecked by the throughput of the slowest stage (the Producer, Kernel, or Consumer), which matches the Roofline Analysis you did earlier. For example, the Figure 78 shows an example timeline where the Kernel is the throughput bottleneck. The Figure 79 shows an example timeline where the Producer (or Consumer) is the bottleneck.

Figure 78. Timing Diagram With Kernel as the Throughput Bottleneck

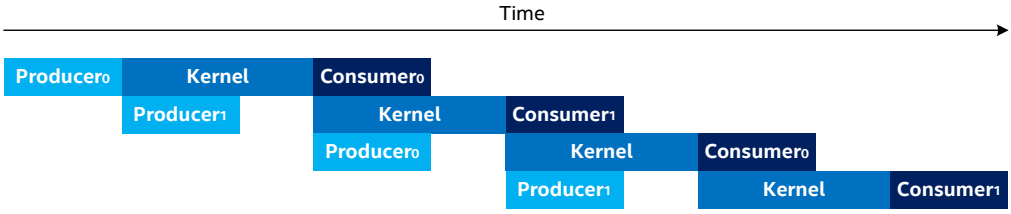
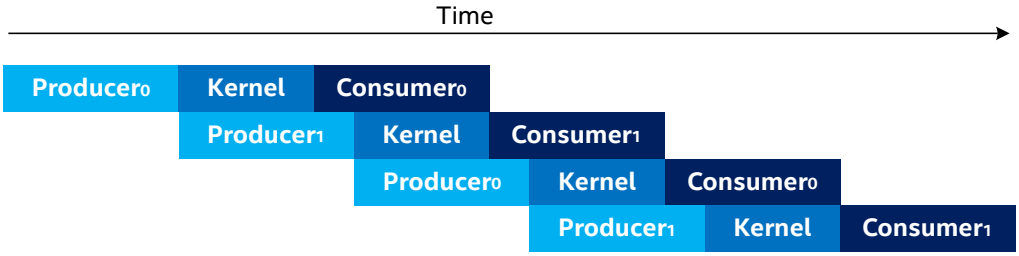


Figure 79. Timing Diagram With Producer/Consumer as the Throughput Bottleneck



Streaming API

In the Buffered Host-Device Streaming example on GitHub, the techniques described in the previous section are implemented in two ways. The design is implemented directly using SYCL USM host allocations and C++ multithreading, and the kernel queue is managed intelligently. The code achieves high performance, but it might be difficult to understand and extend it to different designs. To address this, a convenient

and performant API wrapper (`HostStreamer.hpp`) is created. The same design is implemented in `streaming_with_api.hpp` with similar performance and significantly less code that is much easier to understand.

IMPORTANT

While the code that uses the `HostStreamer` API achieves similar performance to a direct implementation, it uses extra FPGA resources. The direct implementation has a single kernel (Kernel) that does all the processing. Using the API creates a Producer and Consumer kernel that access host allocations and produce/consume data to/from the processing kernel (`APIKernel` in `streaming_with_api.hpp`). These extra kernels (that are transparent) are the mechanism by which the API abstracts the production/consumption of data, but come at the cost of extra FPGA resources. However, when compiled for the Intel FPGA PAC D5005, these extra kernels result in less than 1% increase in FPGA resource utilization. Therefore, given the programming convenience they provide, the tradeoff is often worth it.

NOTE

For additional information, refer to the FPGA tutorial sample "Buffered Host-Device Streaming" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

3.2 Resource Use

You may want to improve throughput first. However, you must also consider the use of resources (ALMs, DSPs, and so on) in your design when targeting FPGAs, for the following reasons:

- To obtain a design to fit on an FPGA image.
- To be able to use multiple copies of the design to increase throughput.
- To make room for other functionality to be implemented in the same FPGA image.

3.2.1 Data Types and Operations

Data Type Selection Considerations

Select the appropriate data type to optimize the FPGA area use by your SYCL* application:

- 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.
- 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® oneAPI DPC++/C++ Compiler to create implicit conversion operators, which can become expensive if they are present in large numbers.
- 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.

Arithmetic Operation Considerations

Select the appropriate arithmetic operation for your SYCL application to avoid excessive FPGA area use.

- Introduce floating-point arithmetic operations only when necessary.
- The Intel® oneAPI DPC++/C++ 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.
- 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:
  - 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.
  - Ensure you are aware of the original and approximated area uses because in some cases, computing a result via approximation might result in excess area use. For example, the `sqrt` function is not resource-intensive. Other than a rough approximation, replacing the `sqrt` function with arithmetic operations that the host must compute at runtime might result in larger area use.
  - If your kernel performs a complex arithmetic operation with a constant that the Intel® oneAPI DPC++/C++ 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.

RESTRICTION

Currently, SYCL implementation of math functions is not supported on FPGAs.

3.2.1.1 Optimize Floating-point Operation

Starting with the oneAPI 2021.2 release, fast math is enabled by default, allowing the Intel® oneAPI DPC++/C++ Compiler to make various out-of-box floating point math (`float` or `double`) optimizations. With these optimizations enabled, you might observe different bitwise results when compared to results from the oneAPI 2021.1 release or from GCC. The tradeoff is done to improve performance and area of your design. Automatic dot product inference and floating-point contraction for double precision math are two key noticeable FPGA optimizations that save a large amount of FPGA area and improve performance/latency. To return to the same precision as the oneAPI 2021.1 release or GCC, use the following compiler options:

- For Linux: `-no-fma -fp-model=precise`
- For Windows: `/Qfma- /fp:precise`

For more information about these options, refer to `-fp-model, fp` and `fma, Qfma` topics in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

For floating-point operations, you can manually direct the Intel® oneAPI DPC++/C++ Compiler to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware use. These optimizations can cause small differences in floating-point results. You can also apply the `fp contract` and `fp reassociate` floating-point pragmas to handle kernel's arithmetic and floating-point operations at finer granularity. For more information about the pragmas, refer to Floating Point Pragmas on page 215.

3.2.1.2 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. For more information about optimizing floating-point operations, refer to the Optimize Floating-point Operation 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 following code example depicts a case of an expensive floating-point operation (division) executed by every work-item in the NDRange:

```
// this function is used in kernel code
void myKernel (accessor<int, access::mode::read, access:: target::global_buffer>
a,
accessor<int, access::mode::read, access:: target::global_buffer> b,
cl::sycl::id<1> wiID,
const float c,
const float d)
{
   //inefficient since each work-item must calculate c divided by d
   b[wiID ] = a[wiID ] * (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 in the following:

```
void myKernel (accessor<int, access::mode::read, access:: target::global_buffer>
a,
accessor<int, access::mode::read, access:: target::global_buffer> b,
```

```
cl::sycl::id<1> wiID,
const float c_divided_by_d)
{
   /*host calculates c divided by d once and passes it into
   kernel to avoid redundant expensive calculations*/
   b[wiID ] = a[wiID ] * c_divided_by_d;
}
```

The Intel® oneAPI DPC++/C++ 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 Intel® oneAPI DPC++/C++ 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.

3.2.1.3 Variable-Precision Integer and Floating-Point Support

The Intel® oneAPI DPC++/C++ Compiler supports a range of FPGA-optimized arbitrary-precision data types that are defined in header files, which you can include in your designs. Some of these header files are based on the Algorithmic C (AC) data types provided by Siemens EDA* *(formerly Mentor Graphics)* under the Apache 2.0 license. For more information about the Algorithmic C data types, refer to https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

Table 15. Algorithmic C Data Types Supported by the Intel® oneAPI DPC++/C++ Compiler

Data Type	Header File	Description
`ac_int`	`<sycl/ext/intel/ac_types/ac_int.hpp>`	Arbitrary-precision integer support
`ac_fixed`	`<sycl/ext/intel/ac_types/ac_fixed.hpp>`	Arbitrary-precision fixed-point number support
`ac_fixed_math`	`<sycl/ext/intel/ac_types/ac_fixed_math.hpp>`	Support for some non-standard math functions for arbitrary-precision fixed-point data types.
`ac_complex`	`<sycl/ext/intel/ac_types/ac_complex.hpp>`	Complex number support
`ap_float`	`<sycl/ext/intel/ac_types/ap_float.hpp>`	Arbitrary-precision floating-point number support
`ap_float_math`	`<sycl/ext/intel/ac_types/ap_float_math.hpp>`	Support for commonly used exponential, logarithmic, power, and trigonometric functions with `ap_float type`.

NOTE

Ensure that the AC data type headers are included after `<CL/sycl.hpp>` and `<sycl/ext/intel/fpga_extensions.hpp>` header files.

DEPRECATION NOTICE:

In the oneAPI 2022.1 release, the `hls_float` data type was renamed as `ap_float`. The header files `<sycl/ext/intel/ac_types/hls_float.hpp>` and `<sycl/ext/intel/ac_types/hls_float_math.hpp>` are still available in this release, so you can access the `hls_float` data type. However, the header files will be removed in a future release. Hence, you must update your design to use the new `ap_float` data type and the header files `<sycl/ext/intel/ac_types/ap_float.hpp>` and `<sycl/ext/intel/ac_types/ap_float_math.hpp>`.

Compilation Flags

Table 16. Compilation Flags for Data Types

Data Type	dpcpp Command Flags	Description
AC type	• **Linux**: `-qactypes` • **Windows**: `/Qactypes`	Use these flags to include `ac_types` header files on the include path and link against AC type libraries required for the host device execution support.
`ap_float` type	• **Linux**: `-fp-model=precise -no-fma` • **Windows**: `/fp:precise /Qfma-`	Use these flags to ensure that floating-point operations are accurate. For more information about these flags, refer to `-fp-model, fp` and `fma, Qfma` topics in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.
	`-DFPGA_EMULATOR`	Use this flag when using the `ap_float` data type and compiling programs for emulation.

3.2.1.3.1 Advantages and Limitations of Arbitrary Precision Data Types

Advantages

The arbitrary precision data types have the following advantages over the use of standard C/C++ data types:

- You can achieve narrower data paths and processing elements for various operations in the circuit.
- The data types ensure that all operations are carried out in a size guaranteed not to lose any data. However, you can still lose data if you store data in a location where the data type is too narrow in size.

Limitations

AC Data Types

The AC data types have the following limitations:

- Multipliers are limited to generating 512-bit results.
- Dividers for `ac_int` data types are limited to a maximum of 64-bit unsigned or 63-bit signed.
- You must initialize an `ac_int` variable before accessing it using the bit-select operator `[]` or by bit-slice operations `slc` and `set_slc`. Using the bit-select operator or bit-slice operations on an uninitialized `ac_int` variable is an undefined

behavior and can give you unexpected results. Assigning each bit explicitly using the `[]` operator or `set_slc` function does not count as initializing the `ac_int` variable.

- Dividers for `ac_fixed` data types are limited to a maximum of 64-bits (unsigned or signed).
- Creation of `ac_fixed` variables larger than 32 bits are supported only with the use of the `bit_fill` utility function.

  For example:

```
// Creating an ac_fixed with value set to 4294967298, which is larger than
2^32.

// Unsupported
ac_fixed<64, 64, false> v1 = ac_fixed<64, 64, false>(4294967298);

// Supported
// 4294967298 is 0b10000000000000000000000000000010 in binary
// Express that as two 32-bit numbers and use the bit_fill utility function.
const int vec_inp[2] = {0x00000001, 0x00000002};
ac_fixed<64, 64, false> bit_fill_res;
bit_fill_res.bit_fill<2>(vec_inp);
```

- The AC data types are not supported on the Red Hat Enterprise Linux* (RHEL) 7 operating system for emulation due to a bug in the `glibc` version bundled with RHEL 7.
- You cannot template the `ac_complex` data type with the `ap_float` data type.
- When using the `bit_fill_hex()` function inside a kernel, pass the input string to the kernel through a `char` buffer and not as a `string` buffer. In addition, hardware and simulation compile flows do not support using a `string` literal or passing the string directly to the function. The following are the supported and unsupported code patterns:

  Supported Patterns

```
// Supported Pattern 1: Passing string as a char sycl::buffer to the kernel
ac_int<140, false> supported_example1(queue &q) {
  ac_int<140, false> a;
  std::string hex_string{"0x177632EE7E265080BD54FF0CE7EF42C12"};
  constexpr int N = 36; // size of hex_string

  buffer<ac_int<140, false>, 1> inp1(&a, 1);
  // Note: the N + 1 ensures that the null byte
  //terminating the char array buffer is copied
  buffer<char, 1> inp2(hex_string.c_str(), range<1>(N + 1));

  q.submit([&](handler &h) {
    accessor x(inp1, h, read_write);
    accessor y(inp2, h, read_only);
    h.single_task<class D>([=] {
      x[0].bit_fill_hex(&y[0]);
    });
  });
  q.wait();
  return a;
}
```

```
// Supported Pattern 2: Create a char array with the string literal.
ac_int<140, false> supported_example2(queue &q) {
  ac_int<140, false> a;
```

```
  buffer<ac_int<140, false>, 1> inp1(&a, 1);

  q.submit([&](handler &h) {
    accessor x(inp1, h, read_write);
    h.single_task<class D>([=] {
      char str[36] = "0x177632EE7E265080BD54FF0CE7EF42C12";
      x[0].bit_fill_hex(str);
    });
  });
  q.wait();
  return a;
}
```

Unsupported Patterns

```
// Unsupported Pattern 1 - Using a string Literal, will result in compilation
error
ac_int<140, false> unsupported_example1(queue& q) {
  {
    ac_int<140, false> a;
    buffer<ac_int<140, false>, 1> a_buff(&a, 1);

    q.submit([&](handler &h) {
      accessor a_acc {a_buff, h, write_only, no_init};
      h.single_task<class A>([=]() {
        a_acc[0].bit_fill_hex("1141e98e8c51b7ac7ad387d7f8ee4f1b9");
      });
    });
    q.wait_and_throw();
    return a;
  }
}
```

```
// Unsupported Pattern 2 - Passing the string to the kernel in a string
sycl::buffer
ac_int<140, false> unsupported_example2(queue& q) {
  {
    std::string str{"1141e98e8c51b7ac7ad387d7f8ee4f1b9"};
    ac_int<140, false> a;
    buffer<std::string, 1> str_buff(&str, 1);
    buffer<ac_int<140, false>, 1> a_buff(&a, 1);

    q.submit([&](handler &h) {
      accessor str_acc {str_buff, h, read_only};
      accessor a_acc {a_buff, h, write_only, no_init};

      h.single_task<class B>([=]() {
        a_acc[0].bit_fill_hex(str_acc[0].c_str());
      });
    });
    q.wait_and_throw();
    return a;
  }
}
```

`ap_float` Data Type

The `ap_float` data type has the following limitations:

- While the floating-point optimization of converting into constants is performed for `float` and `double` data types, it is not performed for the `ap_float` data type.
- A limited set of math functions is supported. For details, see Math Functions Supported by `ap_float` Data Type on page 159.

- Constant initialization works only with the round-towards-zero (`RZERO`) rounding mode.
- For emulation, the `ap_float` math library is not supported on the Red Hat Enterprise Linux* (RHEL) 7 operating system.
- When computing `A^B` using `ap_float`'s `ihc_pown` function, if *B* is an unsigned type *T* of size *N* bits and is equal to the maximum unsigned value, redefine *B* to be of size *N+1* bits. Otherwise, results will be incorrect. For example:

```
// Sample Code:
ap_float<8, 7> a = 2;
ac_int<4, false> b = 15; // max value that this ac_int can hold
… = ihc_pown(a , b); // !!! Will produce incorrect result
```

```
// Workaround:
ap_float<8, 7> a = 2;
ac_int<5, false> b = 15; // Workaround
… = ihc_pown(a , b); // Will produce correct result
```

3.2.1.3.2 Declare and Use the AC Data Types

Refer to the following topics for more information about how to declare and use various AC data types:

- Declare the `ac_int` Data Type on page 154
- Declare the `ac_fixed` Data Type on page 156
- Declare the `ac_complex` Data Type on page 157
- Declare the `ap_float` Data Type on page 157

Declare the `ac_int` Data Type

Perform the following steps to declare the `ac_int` data type:

1. Include the `ac_int.hpp` header file as follows:

```
#include <sycl/ext/intel/ac_types/ac_int.hpp>
```

2. Declare your `ac_int` variables in one of the following ways:
   - Template-based declaration:

```
ac_int<N, true> var_name; //Signed N-bit integer
```

```
ac_int<N, false> var_name; //Unsigned N-bit integer
```

   - Predefined types up to 63 bits:

```
ac_intN::intN var_name; //Signed N-bit integer
```

```
ac_intN::uintN var_name; //Unsigned N-bit integer
```

   Where, *N* is the total length of the integer in bits.

The `ac_int` data type has several API calls. For more information about the Algorithmic C data types, refer to https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

RESTRICTION

If you want to initialize an `ac_int` variable to a value larger than 64 bits, you must use the `bit_fill` or `bit_fill_hex` utility function. For details, refer to the section *Methods to Fill Bits* in the documentation provided in https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

The following code example shows the use of the `bit_fill` or `bit_fill_hex` utility functions:

```
typedef ac_int<80,false> i80_t;
i80_t x;
x.bit_fill_hex("a9876543210fedcba987"); // member funtion
x = ac::bit_fill_hex<i80_t>("a9876543210fedcba987"); // global function
int vec[] = { 0xa987, 0x6543210f, 0xedcba987 };
x.bit_fill(vec); // member function
x = bit_fill<i80_t>(vec); // global function
// inlining the constant array
x.bit_fill( (int [3]) { 0xa987,0x6543210f,0xedcba987 } ); // member function
x = bit_fill<i80_t>( (int [3]) { 0xa987,0x6543210f,0xedcba987 } ); // global
function
```

Debugging the `ac_int` Data Type Use

The `ac_int.hpp` header file provides tools to help you check `ac_int` data type operations and assignments for overflow.

RESTRICTION

Currently, you can debug `ac_int` data type operations and assignments for overflow only in the emulation flow and only for non-kernel code.

NOTES

- When you use `DEBUG_AC_INT_WARNING` and `DEBUG_AC_INT_ERROR` macros, you cannot declare `constexpr ac_int` variables or `constexpr ac_int` arrays.

Macro	Description
`DEBUG_AC_INT_WARNING`	Emits a warning for each detected overflow.
`DEBUG_AC_INT_ERROR`	Emits a message for the first overflow that is detected and then exits the code with an error.

- Within your code, you must declare the macros before you include the `ac_int.hpp` header file.

NOTE

For additional information, refer to the FPGA tutorial sample "Algorithmic C Integer Data Type ac_int" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

Declare the `ac_fixed` Data Type

Perform the following steps to declare the `ac_fixed` data type:

1. Include the `ac_fixed.hpp` header file as follows:

```
#include <sycl/ext/intel/ac_types/ac_fixed.hpp>
```

2. Declare your `ac_fixed` variables as follows:

```
ac_fixed<N, I, true, Q, O> var_name; //Signed fixed-point number
```

```
ac_fixed<N, I, false, Q, O> var_name; //Unsigned fixed-point number
```

   The following table describes the template parameters:

Template Parameter	Description
N	The total length of the fixed-point number in bits.
I	The number of bits used to represent the integer value of the fixed-point number. The difference of *N–I* determines how many bits represent the fractional part of the fixed-point number.
Q	The quantization mode that determines how to handle values where the generated precision (number of decimal places) exceeds the number of bits available in the variable to represent the fractional part of the number. For a list of quantization modes and their descriptions, refer to the *Quantization and Overflow* section in https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.
O	The overflow mode that determines how to handle values where the generated value has more bits than the number of bits available in the variable. For a list of overflow modes and their descriptions, refer to the *Quantization and Overflow* section in https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

For a list of supported operators and their return types, refer to the *Arbitrary-Length Bit-Accurate Integer and Fixed-Point Datatypes* chapter in https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

Math Functions Provided by the `ac_fixed_math.hpp` Header File

Include the header file as follows:

```
#include <sycl/ext/intel/ac_types/ac_fixed_math.hpp>
```

The `ac_fixed_math.hpp` header file provides support for the following arbitrary precision fixed-point (`ac_fixed`) data type functions:

- `sqrt_fixed`
- `reciprocal_fixed`
- `reciprocal_sqrt_fixed`
- `sin_fixed`
- `cos_fixed`
- `sincos_fixed`

- `sinpi_fixed`
- `cospi_fixed`
- `sincospi_fixed`
- `log_fixed`
- `exp_fixed`

For details about input type restrictions, input value limits, and output type propagation rules, review the comments in the `ac_fixed_math.hpp` header file.

IMPORTANT

Due to the differences in the internal math implementations, the results from the `ac_fixed` data type operations might differ between simulation and emulation. The maximum difference is within a few units in the last place (ULPs).

Declare the `ac_complex` Data Type

Perform the following steps to declare the `ac_complex` data type:

1. Include the `ac_complex.hpp` header file as follows:

```
#include <sycl/ext/intel/ac_types/ac_complex.hpp>
```

2. Declare your `ac_complex` variables according to the data type of your complex number.

The underlying data type can be `ac_int`, `ac_fixed`, `ap_float`, and standard-C integer or floating-point data types.

For a list of supported operators and their return types, refer to *Complex Datatype* chapter in https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf.

Declare the `ap_float` Data Type

The `ap_float.hpp` header file provides support for arbitrary-precision floating-point numbers. The floating-point representation for `ap_float` data types adopts the same data layout as the IEEE 754 floating-point representation.

An `ap_float` variable carries an explicit sign bit and an arbitrary number of bits for the exponent and mantissa. Due to the differences in the internal math implementations and rounding errors, the results from `ap_float` operations might not always be bit-accurate compared to those produced by C++ native floating-point types with the same exponent and mantissa bit widths.

Perform the following steps to declare the `ap_float` data type:

1. Include the `ap_float.hpp` header file as follows:

```
#include <sycl/ext/intel/ac_types/ap_float.hpp>
```

2. Declare your `ap_float` variables as follows:

```
ihc::ap_float<exponent_width, mantissa_width[,rounding_mode]>
```

Where, the template attributes are defined as follows:

- ***exponent_width*, *mantissa_width***: The bit-width of the exponent and mantissa of the floating-point variable.

  The `ap_float.hpp` header file also provides aliases to declare `bfloat16` and `bfloat19` data types directly. The `ap_float` data type supports the following *`exponent_width`*, *`mantissa_width`* combinations:

Table 17. Exponent- and Mantissa-Width Combinations Supported by the `ap_float` Data Type

5, 10	8, 7	8, 10	8, 17	8, 23
8, 26	10, 35	11, 44	11, 52	15, 63

Some of these width combinations map to some commonly used floating-point formats, as listed in the following table:

Table 18. Exponent- and Mantissa-Width Setting for Various Floating-point Formats

Floating-point Format	`exponent_width`, `mantissa_width` Setting
IEEE 754 half-precision (*binary16*)	5, 10
`bfloat16`	8, 7
`bfloat19`	8,10
IEEE 754 single-precision (*binary32*)	8. 23
IEEE 754 double-precision (*binary64*)	11, 52
80-bit extended precision	15, 63

- **`rounding_mode`**: Optional parameter to specify the IEEE 754 rounding mode used when converting between data types. Set the rounding mode with one of the following values:

Table 19. Rounding Mode Values

Rounding Mode	Description
`ihc::fp_config::FP_Round::RNE`	Round to the nearest, tie break to even. This rounding mode is more accurate (0.5 ULP) but requires more FPGA area.
`ihc::fp_config::FP_Round::RZERO`	Round towards zero. This rounding mode is less accurate (1 ULP) and requires less FPGA area.

NOTE

If you do not set the `rounding_mode` parameter, the `ihc::FP_Round::RNE` rounding mode is used by default.

Math Functions Supported by `ap_float` Data Type

The `ap_float` data type supports all overloaded math operators and a limited set of the math functions provided by the Intel® oneAPI DPC++/C++ Compiler. For some math operators, you can control the output's precision by using templated versions of the functions.

IMPORTANT

Due to the differences in the internal math implementations and rounding errors, the results from `ap_float` operations might not always be bit-accurate when compared to those produced by C++ native floating-point types with the same exponent and mantissa-bit widths. However, these results are validated against the infinitely accurate results.

The following additional math functions are supported through the `ap_float_math.hpp` header file:

Table 20. Math Functions Provided by the `ap_float_math.hpp` header file

Function Type	Math Function	Comment
Exponential and logarithmic functions	• `ln` • $\log_2$, $\log_{10}$ • e^x, 2^x, 10^x	Supported only for `ap_float` data types with exponent width less than or equal to 15 bits and mantissa width less than or equal to 63 bits.
	• `ln(1+x)` • e^x-1	Supported only for `ap_float` data types with exponent width less than or equal to 11 bits and mantissa width less than or equal to 52 bits.
Advanced functions	• `reciprocal` • `reciprocal_sqrt` • `sqrt` • `cube root` • `hypot` (hypotenuse)	
Power functions	• `pow` • `powr` • `pown`	
Trigonometric functions	• `sin`, `cos`, `sincos` • `sinpi`, `cospi` • `asin`, `asinpi` • `acos`, `acospit` • `atan`, `atanpi`, `atan2`	

Conversion Rules for `ap_float`

You can convert between different sizes of `ap_float` data types through assignment or by using the `convert_to()` function. For example,

```
using namespace ihc;
ap_float<8, 32> myFloat = ...;
ap_float<3, 18> myFloat2 = myFloat; // use rounding rules defined by ap_float
type
```

```
// use rounding rules defined in convert_to() function call
ap_float <3, 18> myFloat3 = myFloat.convert_to<3, 18,
ihc::fp_config::FP_Round::RZERO>();
```

To convert between native types (for example, `float`, `double`) and `ap_float` data types, assign to or from the types. Type conversion in an assignment occurs according to the rules mentioned in Table 21.

For two `ap_float` variables in a binary operation, the `ap_float` variable with the larger exponent bit-width is considered to be the *larger* variable. If two variables have the same exponent bit width, the variable with the larger mantissa bit-width is considered to be the *larger* variable. The operands are then unified to the *larger* type before the binary operation occurs.

Native floating-point data types and `ap_float` data types are converted to `ap_float` data types according to the rules in Table 21.

The Intel® oneAPI DPC++/C++ Compiler also provides some operations that leave the precision of input types untouched and provide control over the output precision. For more details, refer to Operations with Explicit Precision Controls on page 160.

Table 21. Default Conversion Rules for `ap_float` Variables

Data Type	From `ap_float` To *Data Type*	From *Data Type* To `ap_float`
`ap_float` with higher representable range	Keep exponent equivalent. The mantissa is rounded according to the rounding mode of the target `ap_float` (with the higher representable range).	`+-Inf` if the source of the conversion is out of the representable range. Otherwise, keep exponent equivalent. The mantissa is rounded according to the rounding mode of the target `ap_float` (with the smaller representable range).
`float`	Convert original `ap_float` to `ap_float<8, 23>` with the previous `ap_float` rule, and then bit cast to `float`.	Bit-cast `float` to `ap_float<8, 23>`, and then convert to target `ap_float` precision using the `ap_float` to `ap_float` rules described previously.
`double`	Convert original `ap_float` to `ap_float<11, 52>` with earlier `ap_float` rule, and then bit cast to `double`.	Bit-cast `double` to `ap_float<11, 52>`, and then convert to the target `ap_float` precision using the `ap_float` to `ap_float` rules described earlier.
`long double` (emulation only) (Linux only)	Convert the original `ap_float` to `ap_float<15, 63>` with the earlier `ap_float` rule, and then insert a 1-bit 1 to the MSB of fraction bits to get an approximate equivalent of 80-bit representation of a `long double`.	Drop the explicit one fraction bit to convert `long double` to 79-bit `ap_float<15, 63>`.
C++ native integer types	Truncate towards zero. Converting from `ap_float` that is larger than the range of integer type is an undefined behavior.	Round to the nearest, tie breaks to even. If the integer value is too large, the `ap_float` value saturates to plus infinity.

Operations with Explicit Precision Controls

The following operations leave the precision of the input `ap_float`-type variables untouched and allow you to control the output precision:

Rounding Mode Control For `ap_float` to `ap_float` Conversions

Syntax

```
convert_to<output_exponent_width, output_mantissa_width, rounding_mode>
```

Description

Use this method to override the rounding mode set for an `ap_float` variable when converting the variable to a different precision. By default, `ap_float` to `ap_float` conversions use the rounding mode that you specified when you declared the variable.

Multiplication

Syntax

```
ihc::ap_float<output_exponent_width, output_mantissa_width>::mul
<[accuracy_setting], [subnormal_setting]> (ap_float_a, ap_float_b)
```

Description

This math function supplements the basic multiplication operation performed by the multiplication (*) operator. Multiplies *`ap_float_a`* and *`ap_float_b`* without changing the input types and outputs an `ap_float` at the specified precision. The optional parameters are defined as follows:

- ***`subnormal_setting`***: Optional parameter to specify whether input and output numbers are flushed to zero when carrying out basic binary operations explicitly. Set this parameter with one of the following values:

Table 22. `subnormal_setting` Parameter Values

`subnormal_setting` Values	Description
`ihc::fp_config::FP_Subnormal::ON`	Input and output numbers in the subnormal range are preserved. The target FPGA device must have subnormal support. Subnormal support might require more FPGA area.
`ihc::fp_config::FP_Subnormal::OFF`	Input or output numbers in the subnormal range are flushed to zero.
`ihc::fp_config::FP_Subnormal::AUTO`	With this setting, the Intel® oneAPI DPC++/C++ Compiler enables subnormal support only when the target FPGA device directly supports it and it does incur any extra FPGA area overhead.

  If you do not set the *`subnormal_setting`* parameter, the `ihc::fp_config::FP_Subnormal::AUTO` subnormal setting is used by default.

- ***`accuracy_setting`***: Optional parameter that influences trade-offs between the accuracy of the result due to different rounding decisions in the intermediary calculations and the FPGA area utilized by the generated hardware. Floating-point operations with less accurate results typically use fewer logic elements. For example, a divider with a high accuracy might use 20% more FPGA area than a divider with low accuracy. The low accuracy divider has a higher error bound [1 unit of least precision (ULP)] than a high accuracy divider (0.5 ULP).

  Set this parameter with one of the following values:

Table 23. `accuracy_setting` Parameter Values

`accuracy_setting` Values	Description
`ihc::fp_config::FP_Accuracy::HIGH`	Uses the high precision version of the floating-point math operations. This is the default setting.
`ihc::fp_config::FP_Accuracy::LOW`	Allows the compiler to use a higher error bound to save on-chip area.

If you do not set the `accuracy_setting` parameter, `ihc::fp_config::FP_Accuracy::HIGH` accuracy setting is used by default.

Addition/Subtraction/Division

Syntax

```
ihc::ap_float<output_exponent_width, output_mantissa_width>::add <[optional parameters]> (ap_float_a, ap_float_b)
```

```
ihc::ap_float<output_exponent_width, output_mantissa_width>::sub <[optional parameters]> (ap_float_a, ap_float_b)
```

```
ihc::ap_float<output_exponent_width, output_mantissa_width>::div <[optional parameters]> (ap_float_a, ap_float_b)
```

Description

These math functions supplement the basic math operations performed by the addition/subtraction/division (+ / – /) operators. Adds/subtracts/divides *`ap_float_a`* and *`ap_float_b`* by first casting *`ap_float_a`* and *`ap_float_b`* to the specified `ap_float` precision. The operation and output are at the specified precision.

You can also specify the optional parameters *`accuracy_setting`* and *`subnormal_setting`* described earlier.

Comparison Operators

Comparison operators (>, <, ==, !=, >=, <=) are subject to the conversion rules described in Conversion Rules for `ap_float` on page 159.

The == and != operators impose a bit-wise comparison of the casted values.

Comparisons with `NaN` always return `false`.

Additional `ap_float` Functions

The `ap_float` data type also provides the following additional functions:

Table 24. Additional `ap_float` Functions

Function	Description
Getters and Setters	
`ap_float::get_exponent` `ap_float::set_exponent`	Gets/sets the exponent value of the `ap_float` variable.
`ap_float::get_mantissa` `ap_float::set_mantissa`	Gets/sets the mantissa value of the `ap_float` variable.
`ap_float::get_sign` `ap_float::set_sign`	Gets/sets the sign bit of the `ap_float` variable.
Special Constants	
`ap_float<e,m>::nan()`	Assigns the `ap_float` variable a value of `NaN`.
`ap_float<e,m>::pos_inf()`	Assigns the `ap_float` variable a value of $+\infty$.
`ap_float<e,m>::neg_inf()`	Assigns the `ap_float` variable a value of $-\infty$.
Value Queries	
`ap_float::is_nan()`	Returns `true` if the value of the `ap_float` variable is `NaN`.
`ap_float::is_inf()`	Returns `true` if the value of the `ap_float` variable is $\pm\infty$.
`ap_float::is_zero()`	Returns `true` if the value of the `ap_float` variable is zero.
Special Functions	
`ap_float::next_after(next_val)`	Returns the next representable value towards *`next_val`*.

Additional Data Types Provided by the `ap_float.hpp` Header File

The `ap_float.hpp` header file provides some aliases for certain data types that you can use instead of explicitly declaring an `ap_float` data type.

Data Type	Description
`bfloat16`	A 16-bit floating-point number with an 8-bit exponent and a 7-bit mantissa (equivalent to declaring `ap_float<8, 7>`). On Intel® Agilex™ devices, dot product operations that involve the `bfloat16` (or `ap_float<8,7>`) data type are mapped to FP16 digital signaling blocks (DSPs). On other device families, dot product operations are mapped to adaptive logic modules (ALMs) and fixed-point 18-bit DSPs. On all device families, all other math functions are mapped to ALMs and fixed-point 18-bit DSPs.
`bfloat19`	A 19-bit floating-point number with an 8-bit exponent and a 10-bit mantissa (equivalent to declaring `ap_float<8,10>`). On Intel® Agilex™ devices, dot product operations that involve the `bfloat19` (or `ap_float<8,10>`) data type are mapped to FP19 digital signaling blocks (DSPs). On other device families, dot product operations are mapped to adaptive logic modules (ALMs) and fixed-point 18-bit DSPs. On all device families, all other math functions are mapped to ALMs and fixed-point 18-bit DSPs.

3.2.2 Kernel Variable Accesses

This section shows techniques you can use to optimize local and private variables in kernels.

Inferring a Shift Register

The shift register design pattern is a very important design pattern for efficient implementation of many applications on the FPGA. However, the implementation of a shift register design pattern might seem counter-intuitive at first.

Consider the following code example:

```
using InPipe = ext::intel::pipe<class PipeIn, int, 4>;
using OutPipe = ext::intel::pipe<class PipeOut, int, 4>;

#define SIZE 512
//Shift register size must be statically determinable
// this function is used in kernel
void foo()
{
  int shift_reg[SIZE];
  //The key is that the array size is a compile time constant
  // Initialization loop
  #pragma unroll
  for (int i = 0; i < SIZE; i++)
  {
    //All elements of the array should be initialized to the same value
    shift_reg[i] = 0;
  }
  while(1)     {
    // Fully unrolling the shifting loop produces constant accesses
    #pragma unroll
    for (int j = 0; j < SIZE-1; j++)
    {
      shift_reg[j] = shift_reg[j + 1];
    }

    shift_reg[SIZE - 1] = InPipe::read();
    // Using fixed access points of the shift register
    int res = (shift_reg[0] + shift_reg[1]) / 2;

    // 'out' pipe will have running average of the input pipe
    OutPipe::write(res);
  }
}
```

In each clock cycle, the kernel shifts a new value into the array. By placing this shift register into a block RAM, the Intel® oneAPI DPC++/C++ Compiler can efficiently handle multiple access points into the array. The shift register design pattern is ideal for implementing filters (for example, image filters like a Sobel filter or time-delay filters like a finite impulse response (FIR) filter).

When implementing a shift register in your kernel code, remember the following key points:

- Unroll the shifting loop so that it can access every element of the array.
- All access points must have constant data accesses. For example, if you write a calculation in nested loops using multiple access points, unroll these loops to establish the constant access points.
- Initialize all elements of the array to the same value. Alternatively, you may leave the elements uninitialized if you do not require a specific initial value.

Memory Access Considerations

Intel® recommends the following kernel programming strategies that can improve memory access efficiency and reduce area use of your kernel:

- Minimize the number of access points to external memory to reduce area. The compiler infers an LSU for each access point in your kernel, which consumes area.

  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.
- Instead of relying on local or global memory accesses, structure your kernel as a single work-item with shift register inference whenever possible.

4.0 FPGA Optimization Flags, Attributes, Pragmas, and Extensions

This chapter describes a list of compiler optimization flags, attributes, pragma, and extensions that allow you to customize the kernel compilation process.

4.1 Optimization Flags

This section describes the FPGA optimization flags supported by the Intel® oneAPI DPC++/C++ Compiler. Refer to the sub-topics for detailed information about the supported flags.

4.1.1 Specify Schedule F_{MAX} Target for Kernels (`-Xsclock=<clock target>`)

The schedule f_{MAX} target determines the pipelining effort the scheduler attempts during the scheduling process.

You can direct the Intel® oneAPI DPC++/C++ Compiler to globally compile all kernels with `-Xsclock=<clock target in Hz/KHz/MHz/GHz or s/ms/us/ns/ps>` option in the `dpcpp` command.

Example 5. Example

```
dpcpp -fintelfpga -Xshardware -Xsclock=<clock target> <source_file>.cpp
```

NOTE

The schedule target f_{MAX} determines the pipelining effort during compilation. Compile to hardware to get the actual f_{MAX} value.

4.1.2 Disable Burst-Interleaving of Global Memory (`-Xsno-interleaving=<global_memory_type>`)

The Intel® oneAPI DPC++/C++ Compiler cannot burst-interleave global memory across different memory types. You can disable burst-interleaving for all global memory banks of the same type and manage them manually by including the `-Xsno-interleaving=<global_memory_type>` option in your `dpcpp` command.

Manual partitioning of memory buffers overrides the default burst-interleaved configuration of global memory.

CAUTION

The `-Xsno-interleaving` option requires a global memory type parameter. If you do not specify a memory type, the Intel® oneAPI DPC++/C++ Compiler issues an error message.

- To direct the Intel® oneAPI DPC++/C++ Compiler to disable burst-interleaving for the default global memory, invoke the following command:

```
dpcpp -fintelfpga -Xshardware <source_file>.cpp -Xsno-interleaving=default
```

- Your accelerator board might include multiple global memory types. To identify the default global memory type, refer to board vendor's documentation for your Custom Platform.
- For a heterogeneous memory system, to direct the Intel® oneAPI DPC++/C++ Compiler to disable burst-interleaving of a specific global memory type, perform the following tasks:
  1. Consult the `board_spec.xml` file of your Custom Platform for the names of the available global memory types (for example, DDR and quad data rate (QDR)).
  2. To disable burst-interleaving for one of the memory types (for example, DDR), invoke:

     ```
     dpcpp -fintelfpga -Xshardware <source_file>.cpp -Xsno-interleaving=DDR
     ```

     The Intel® oneAPI DPC++/C++ Compiler enables manual partitioning for the DDR memory bank and configures the other memory bank in a burst-interleaved fashion.
  3. To disable burst-interleaving for more than one type of global memory buffers, include a `-Xsno-interleaving=<global_memory_type>` option for each global memory type. For example, to disable burst-interleaving for both DDR and QDR, invoke the following command:

     ```
     dpcpp -fintelfpga -Xshardware <source_file>.cpp -Xsno-interleaving=DDR -
     Xsno-interleaving=QDR
     ```

CAUTION

Do not pass a buffer as a kernel argument that associates it with multiple memory technologies.

4.1.3 Force Ring Interconnect for Global Memory (`-Xsglobal-ring`)

The Intel® oneAPI DPC++/C++ Compiler attempts to choose an optimal global memory interconnect topology based on various characteristics of the design.

To override the compiler's choice and force a ring topology, use the `-Xsglobal-ring` option in your `dpcpp` command. This can improve your kernel f_{MAX}. In particular, designs that target board support packages with four or more banks of global memory may see an f_{MAX} benefit from this option.

Example 6. Example

```
dpcpp -fintelfpga -Xshardware -Xsglobal-ring <source_file>.cpp
```

4.1.4 Force a Single Store Ring to Reduce Area (`-Xsforce-single-store-ring`)

When the Intel® oneAPI DPC++/C++ Compiler implements a ring topology for the global memory interconnect (either by automatic choice or by forcing the ring through `-Xsglobal-ring`), it widens the interconnect by default to allow more writes to occur in parallel. This allows for saturation of the global memory throughput using write-only traffic. The `-Xsforce-single-store-ring` option allows you to save area if you do not require that much write bandwidth.

To narrow the interconnect in order to save area while limiting write-only throughput to one bank's worth, use the `-Xsforce-single-store-ring` option in your `dpcpp` command.

Example 7. Example

```
dpcpp -fintelfpga -Xshardware -Xsforce-single-store-ring <source_file>.cpp
```

4.1.5 Force Fewer Read Data Reorder Units to Reduce Area (`-Xsnum-reorder`)

When the Intel® oneAPI DPC++/C++ Compiler implements a ring topology for the global memory interconnect (either by automatic choice or by forcing the ring through `-Xsglobal-ring`), it widens the interconnect by default to allow more reads to occur in parallel. This allows for saturation of the global memory throughput using read-only traffic.

To narrow the interconnect in order to save area while reducing read-only throughput, use the `-Xsnum-reorder=N` option in your `dpcpp` command, where `N` is the number of bank's worth of read bandwidth you desire. For example, if on a two-bank BSP, you require only one bank's worth of read bandwidth, set `-Xsnum-reorder=1`.

Example 8. Example

```
dpcpp -fintelfpga -Xshardware -Xsnum-reorder=1 <source_file>.cpp
```

4.1.6 Disable Hardware Kernel Invocation Queue (`-Xsno-hardware-kernel-invocation-queue`)

To direct the Intel® oneAPI DPC++/C++ Compiler to reduce kernel area use by removing kernel invocation queue in SYCL* kernel, include the `-Xsno-hardware-kernel-invocation-queue` option in your `dpcpp` command.

Example 9. Example

```
dpcpp -fintelfpga -Xshardware -Xsno-hardware-kernel-invocation-queue
<source_file>.cpp
```

CAUTION

Using this option may result in longer kernel execution time as the kernel invocation queue allows SYCL runtime environment to queue kernel launches in accelerator so that the accelerator can start execution on the next invocation as soon as previous invocation of the same kernel is complete.

NOTE

Use the `-Xsno-hardware-kernel-invocation-queue` option only when your kernel execution time is much greater than the system and SYCL runtime environment overhead hidden by the kernel invocation queue (20-100us), or if the Intel® oneAPI DPC++/C++ Compiler has a difficulty fitting your kernel.

Refer to Utilizing Hardware Kernel Invocation Queue on page 129 for more information about how to utilize the kernel invocation queue.

4.1.7 Modify the Handshaking Protocol (`-Xshyper-optimized-handshaking`)

To modify the handshaking protocol used in certain areas of your design, use the `-Xshyper-optimized-handshaking=<auto|off>` option in your `dpcpp` command. The `-Xshyper-optimized-handshaking` option can be set to one of the following values:

- **`auto`**: The default behavior without the option specified. The Intel® oneAPI DPC++/C++ Compiler enables the optimization if it is possible to do so, else it sets to off. Use this value when you want to achieve a higher f_{MAX}. When you enable the optimization, the Intel® oneAPI DPC++/C++ Compiler adds pipeline registers to the handshaking paths of the stallable nodes. As a result, you observe higher f_{MAX} at the cost of increased area and latency.
- **`off`**: The Intel® oneAPI DPC++/C++ Compiler attempts to optimize for lower latency at the potential cost of lower f_{MAX}. Disabling hyper-optimized handshaking might also decrease area. This is useful for smaller designs where you are willing to give up f_{MAX} for lower latency and area.

Example 10. Examples

```
dpcpp -fintelfpga -Xshardware -Xshyper-optimized-handshaking=auto
<source_file>.cpp
```

```
dpcpp -fintelfpga -Xshardware -Xshyper-optimized-handshaking=off <source_file>.cpp
```

NOTE

The `-Xshyper-optimized-handshaking` option applies only to designs targeting Intel® Stratix® 10 and Intel® Agilex™ devices. If you use this option on other target devices, the compiler fails and produces an error. This option applies only when running the report or hardware flow.

4.1.8 Disable Automatic Fusion of Loops (`-Xsdisable-auto-loop-fusion`)

Include the `-Xsdisable-auto-loop-fusion` flag in your `dpcpp` command to direct the Intel® oneAPI DPC++/C++ Compiler to disable automatic loop fusion when compiling your design.

Example 11. Example

```
dpcpp -fintelfpga -Xshardware -Xsdisable-auto-loop-fusion <source_file>.cpp
```

For more information about automatic loop fusion, refer to Automatic Loop Fusion on page 92

4.1.9 Fusing Adjacent Loops With Unequal Trip Counts (`-Xsenable-unequal-tc-fusion`)

Use the `-Xsenable-unequal-tc-fusion` flag in your `dpcpp` command to direct the Intel® oneAPI DPC++/C++ Compiler to fuse adjacent loops with different trip counts into a single loop without affecting either loop's functionality.

Example 12. Example

```
dpcpp -fintelfpga -Xshardware -Xsenable-unequal-tc-fusion <source_file>.cpp
```

For more information about fusing loops, refer to Fuse Loops to Reduce Overhead and Improve Performance on page 91.

4.1.10 Pipelining Loops in Non-task Kernels (`-Xsauto-pipeline`)

To direct the Intel® oneAPI DPC++/C++ Compiler to compile your design and pipeline loops in non-task (`parallel_for`) kernels, include the `-Xsauto-pipeline` option in your `dpcpp` command. The host program invokes non-task kernels through the kernel execution function `parallel_for`, `parallel_for_work_item`, or `parallel_for_work_group`.

Example 13. Example

```
dpcpp -fintelfpga -Xshardware -Xsauto-pipeline <source_file>.cpp
```

With the `-Xsauto-pipeline` option, the compiler attempts to pipeline the loops in your design, but the pipelining is not guaranteed. If you do not include the `-Xsauto-pipeline` option, the compiler does not pipeline the loops in `parallel_for` kernels. However, it executes different work items in parallel.

NOTE

The `-Xsauto-pipeline` option might improve or degrade performance depending on the memory access pattern in your design.

- If the auto-pipelining is successful, the Loop Analysis report displays the message `Auto-pipelined parallel_for` and `parallel_for rewritten as a pipelined single_task` (Details pane) . The compiler-generated loops appear marked as `Compiler generated auto-pipeline loop` in the report.
- If the compiler chooses not to auto-pipeline the loops, the Loop Analysis report displays a message for the kernel. The reasons for not auto-pipelining a loop can be one of the following:
  - A barrier in the function is not at the top-level function scope.
  - Kernel uses a local or private memory.
  - Kernel uses a volatile or atomic memory, or channels.

TIP

If you do not want the compiler to pipeline some infrequently used loops while allowing other loops to be auto-pipelined, use the `[[intel::disable_loop_pipelining]]` loop directive on specific loops when using the `-Xsauto-pipeline` option. This loop directive disables the loop pipelining.

4.1.11 Controlling Floating-Point Rounding Operations (`-fp-model=<value>`)

Include the `-fp-model=<value>` option in your `dpcpp` command to direct the Intel® oneAPI DPC++/C++ Compiler to control the semantics of floating-point operations.

NOTE

You can also use the Floating Point Pragmas on page 215 to influence the floating-point operations.

Example 14. Example

```
dpcpp -fintelfpga -Xshardware -fp-model=<value> <source_file>.cpp
```

where, the *<value>* can be one of the following values:

Value	Description
fast[=1|2]	Default value. Enables more aggressive optimizations on floating-point operations. *Note:* There is currently no difference between `fast=1` and `fast=2`.
precise	Disables optimizations that are not value-safe on floating-point data.

For additional information about this flag, refer to `fp-model, fp` topic in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

4.1.12 Modify the Rounding Mode of Floating-point Operations (-Xsrounding=<rounding_type>)

To modify the rounding mode of floating-point elementary operations in your design, use the `-Xsrounding=<rounding_type>` option in your `dpcpp` command. You can set the `-Xsrounding` option to one of the following values:

- `ieee`: All elementary operations (`+`, `-`, `*`, `/`) of both single-precision and double-precision floating-point use IEEE-754 round nearest ties to even (RNE) mode, which has a 0.5 unit of least precision (ULP) error at most.
- `faithful`: All elementary operations of double-precision floating-point and multiplication and division of single-precision floating-point use faithful rounding mode, which has a 1 ULP error at most. This rounding mode leads to more efficient hardware at the expense of numerical variation in results. Addition and subtraction of single-precision floating-point still have to use IEEE-754 RNE rounding mode.

The following tables summarize the rounding modes:

Single-precision Floating-point

	Addition	Subtraction	Multiplication	Division
Default	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE	Faithful
`-Xsrounding=ieee`	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE
`-Xsrounding=faithful`	IEEE-754 RNE	IEEE-754 RNE	Faithful	Faithful

Double-precision Floating-point

	Addition	Substraction	Multiplication	Division
Default	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE
`-Xsrounding=ieee`	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE	IEEE-754 RNE
`-Xsrounding=faithful`	Faithful	Faithful	Faithful	Faithful

Example 15. Examples

```
dpcpp -fintelfpga -Xshardware -Xsrounding=ieee <source_file>.cpp
```

```
dpcpp -fintelfpga -Xshardware -Xsrounding=faithful <source_file>.cpp
```

4.1.13 Global Control of Exit FIFO Latency of Stall-free Clusters (-Xssfc-exit-fifo-type=<value>)

Use the `-Xssfc-exit-fifo-type=<value>` flag in the `dpcpp` command to direct the Intel® oneAPI DPC++/C++ Compiler to globally compile all stall-free clusters in kernels with a specified exit FIFO type. This flag supports the following arguments:

- **`default`**: Infers the mid-speed FIFO (implemented with MLABs or M20Ks) for a minimum latency of three cycles.
- **`zero-latency`**: Combinational path around the default FIFO for a minimum latency of zero cycles.
- **`low-latency`**: Registered path around the default FIFO for a minimum latency of one cycle.

CAUTION

Depending on the specified exit FIFO type and resulting hardware implementation, f_{MAX} or FPGA area use might be affected negatively.

Example 16. Example

```
dpcpp -fintelfpga -Xshardware -Xssfc-exit-fifo-type=zero-latency <source_file>.cpp
```

Related Links

Clustering the Datapath on page 19

4.1.14 Enable the Read-Only Cache for Read-Only Accessors (`-Xsread-only-cache-size=<N>`)

If your kernel accesses a read-only accessor that is guaranteed not to alias with other accessors and USM pointers, consider enabling the read-only cache using the `-Xsread-only-cache-size=<N>` flag in your `dpcpp` command. You should use a read-only cache for high-bandwidth table lookups that is constant throughout the kernel execution. The read-only cache is optimized for high cache-hit performance.

Example 17. Example

```
dpcpp -fintelfpga -Xshardware -Xsread-only-cache-size=<N> <source_file>.cpp
```

The compiler implements the read-only cache using on-chip memory blocks and privatizes it per kernel. Each kernel receives a version of the cache that serves all reads in the kernel from read-only no-alias accessors. The compiler replicates each private cache as many times as necessary to expose extra read ports. The size of each replicate is `<N>` bytes as specified by the `-Xsread-only-cache-size=<N>` flag.

NOTES

- Unlike global memory accesses that have extra hardware for tolerating long memory latencies, the read-only cache suffers significant performance penalties for cache misses. If the buffer being accessed in your kernel code cannot fit in the cache, you might achieve better performance without enabling the cache. The cached data is discarded (invalidated) from the read-only cache every time the kernel is launched.
- Currently, omitting the read-only cache for only a subset of your read-only accessors in your design is unsupported. If your design has multiple read-only no-alias accessors, you can either enable caching for all of them using the global `-Xsread-only-cache-size=<N>` flag or disable caching for all of them by removing the flag.

Consider the following example code snippet:

```
q.submit([&](handler &h) {
  accessor sqrt_lut(sqrt_lut_buf, h, read_only,
                    ext::oneapi::accessor_property_list{no_alias});
  accessor indices(indices_buf, h, read_write,
                   ext::oneapi::accessor_property_list{no_alias, no_init});
  accessor output(output_buf, h, write_only,
                  ext::oneapi::accessor_property_list{no_alias, no_init});

  h.single_task<class Test>([=]() {
    for (int i = 0; i < kNumInputs; ++i) {
      output[i] = sqrt_lut[indices[i]];
    }
  });
});
```

Compile the above code using the following command:

```
dpcpp -fintelfpga -Xshardware -Xsread-only-cache-size=2048 <source_file>.cpp
```

The compiler creates a read-only cache of size 2048 bytes that serves the single read from `sqrt_lut`. If the cache is sized correctly to match the size of `sqrt_lut_buf`, then the cache improves the design throughput, especially because the read accesses are random.

4.1.15 Control Hardware Implementation of the Supported Data Types and Math Operations

The Intel® oneAPI DPC++/C++ Compiler allows you to control (at global and local scope) whether the implementation of the supported data type and math functions uses DSPs or soft logic using ALMs. You can find this implementation in the Details pane of the corresponding math instruction nodes in the System Viewer of the FPGA optimization report (`report.html`).

Global-scope Control

Include the `-Xsdsp-mode=[default|prefer-dsp|prefer-softlogic]` flag in your `dpcpp` command to control the hardware implementation of the supported data types and math operations of all kernels in your source code.

Example 18. Example

```
dpcpp -fintelfpga -Xshardware -Xsdsp-mode=<option> <source_file>.cpp
```

where the *<option>* is one of the following values:

Option	Description
`default`	Default option if you do not pass this command-line flag manually. The compiler determines the implementation based on the data type and math operation.
`prefer-dsp`	Prefer math operations to be implemented in DSPs. If a math operation is implemented by DSPs by default, you see no difference in resource utilization or area. Otherwise, you will notice a decrease in the use of soft-logic resources and an increase in the use of DSPs.
`prefer-softlogic`	Prefer math operations to be implemented in soft-logic. If a math operation is implemented without DSPs by default, you see no difference in resource utilization or area. Otherwise, you will notice a decrease in the use of DSPs and an increase in the use of soft-logic resources.

Local-scope Control

You can control DSP usage of the supported math operations on a local scope by using the library function `sycl::ext::intel::math_dsp_control(<Preference::<enum>, Propagate::<enum>>)` defined in the `fpga_dsp_control.hpp` header file, which is included in the `fpga_extensions.hpp` header file. This library function provides control at the block level within a single kernel. A reference-capturing lambda expression is passed as the argument to this library function. Inside the lambda expression, the implementation preference of math operations that support DSP control is determined by two template arguments.

The *<Preference::<enum>>* argument takes up an `enum` data type with one of the following values:

Value	Description
`cl::sycl::ext::intel::Preference::DSP`	Prefer math operations to be implemented in DSPs. Its behavior on a math operation is equivalent to the global control `-Xsdsp-mode=prefer-dsp`. *Note:* The `cl::sycl::ext::intel::Preference::DSP` option is automatically applied if you do not specify the template argument `Preference` manually.
`cl::sycl::ext::intel::Preference::Softlogic`	Prefer math operations to be implemented in soft logic. Its behavior on a math operation is equivalent to the global control `-Xsdsp-mode=prefer-softlogic`.
`cl::sycl::ext::intel::Preference::Compiler_default`	Compiler determines the implementation based on the data type and math operation. Its behavior on a math operation is equivalent to the global control `-Xsdsp-mode=default`.

NOTE

Local control overrides global control on a controlled math operation. For example:

```
// dpcpp -Xsdsp-mode=prefer-softlogic ...
using namespace sycl;
cgh.single_task<class LocalControl>([=]() {
  out_acc[0] = in_acc[0] + inp_acc[1]; // Addition implemented in soft-logic.
  ext::intel::math_dsp_control<ext::intel::Preference::DSP>([&] {
    out_acc[1] = in_acc[0] + in_acc[1]; // Addition implemented in DSP.
  });
});
```

The *<Propagate::<enum>>* argument is a boolean value that determines the propagation of the *Preference::<enum>* value to function calls in the lambda expression as follows:

Value	Description
`cl::sycl::ext::intel::Propagate::On`	DSP control recursively applies to controllable math operations in all function calls in the lambda expression function. This value is the default and it is applied if you do not specify the argument *<Propagate::<enum>>*.
`cl::sycl::ext::intel::Propagate::Off`	DSP control applies only to the controllable math operations directly inside the lambda expression. Math operations in function calls inside the lambda expression are not affected by this DSP control.

Example 19. Example

```
sycl::ext::intel::math_dsp_control<sycl::ext::intel::Preference::Softlogic,
                                   sycl::ext::intel::Propagate::On>([&]{
  a += 1.23f;
});
```

NOTE

A nested `math_dsp_control<>()` call is controlled only by its own `Preference` argument. The `Preference` of the parent `math_dsp_control<>()` function does not affect the nested `math_dsp_control<>()` function, even if the parent has the argument `Propagate::On`. For example:

```
using namespace sycl;
cgh.single_task<class LocalControl>([=]() {
  ext::intel::math_dsp_control<ext::intel::Preference::DSP,
                               ext::intel::Propagate::On>([&] {
    out_acc[0] = in_acc[0] + in_acc[1]; // Addition implemented in DSP.
    ext::intel::math_dsp_control<ext::intel::Preference::Softlogic>([&] {
      // Addition implemented in soft-logic. Preference::DSP on the parent
      // math_dsp_control<>() call does not affect this math_dsp_control<>().
      out_acc[1] = in_acc[0] + in_acc[1];
    });
  });
});
```

Supported Data Type and Math Operations

Data types	Math Operations
`float`	Addition, subtraction, multiplication by a constant
`ap_float<8, 23>`	Addition, subtraction, multiplication by a constant
`int`	Multiplication by a constant
`ac_int`	Multiplication by a constant
`ac_fixed`	Multiplication by a constant

NOTE

For additional information, refer to the FPGA tutorial sample `DSP Control` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

4.2 Kernel Attributes

This section describes the FPGA kernel attributes supported by the Intel® oneAPI DPC++/C++ Compiler. Refer to the sub-topics for detailed information about the supported attributes.

4.2.1 Specify Schedule F_{MAX} Target for Kernels

The schedule f_{max} target determines the pipelining effort the scheduler attempts during the scheduling process.

You can use one or both of the following options to specify the kernel specific f_{max} target:

- By using the `[[intel::scheduler_target_fmax_mhz(`*N*`)]]` source-level attribute.
- By directing the Intel® oneAPI DPC++/C++ Compiler to globally compile all kernels `-Xsclock=<clock target in Hz/KHz/MHz/GHz or s/ms/us/ns/ps>` option in the `dpcpp` command.

If you use both the command-line option and source-level attribute, the kernel attribute takes the priority. Consider the following example:

```
cgh.single_task<class mykernel1>([=] {
  ...
});

cgh.single_task<class mykernel2>([=
]() [[intel::scheduler_target_fmax_mhz(200)]] {
  ...
});
```

In you direct the compiler to compile the above code with `-Xsclock=300MHz` in the `dpcpp` command, the compiler schedules kernel `mykernel1` at 300 MHz and kernel `mykernel2` at 200 MHz.

The schedule target f_{max} determines the pipelining effort during compilation. Refer to the Quartus compilation summary in the `report.html` file to get the actual f_{max} value.

4.2.2 Specify a Work-Group Size

Specify a maximum or the required work-group size whenever possible. The Intel® oneAPI DPC++/C++ Compiler relies on this specification to optimize hardware use of the SYCL* kernel without involving excess logic.

- If you do not specify the `[[intel::max_work_group_size(Z, Y, X)]]` or `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute in your kernel, the work-group size assumes a default value depending on compilation time and runtime constraints.
- If your kernel contains a barrier, the Intel® oneAPI DPC++/C++ Compiler sets a default maximum scalarized work-group size of 128 work-items.
- If your kernel does not query any SYCL intrinsic that allow different threads to behave differently (that is, local or global thread IDs, or work-group ID), the Intel® oneAPI DPC++/C++ Compiler infers a single-threaded execution mode and sets the maximum work-group size to `(1, 1, 1)`. In this case, the SYCL runtime also enforces a global enqueue size of `(1, 1, 1)`, and loop pipelining optimizations are enabled within the Intel® oneAPI DPC++/C++ Compiler.

DEPRECATION NOTICE:

The `[[cl::reqd_work_group_size(Z, Y, X)]]` attribute is deprecated. Use the `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute.

To specify the work-group size, modify your kernel code in the following manner:

- To specify the maximum number of work-items that the compiler provisions for a work-group in a kernel, insert the `[[intel::max_work_group_size(Z, Y, X)]]` attribute in your kernel source code.

  For example:

```
constexpr unsigned MAX_WG_SIZE = 4;
...
cgh.parallel_for<class kernelCompute>(
  nd_range<1>(range<1>(N), range<1>(wg_size)),
  [=] (nd_item<id> it)
  [[intel::max_work_group_size(1, 1, MAX_WG_SIZE)]] {
    auto gid = it.get_global_id(0);
    accessorRes[gid] = accessorIdx[gid] * 2;
});
```

- To specify the required number of work-items that the Intel® oneAPI DPC++/C++ Compiler provisions for a work-group in a kernel, insert the `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute in your kernel source code.

  For example:

```
constexpr unsigned REQD_WG_SIZE = 4;
...
cgh.parallel_for<class kernelCompute>(
  nd_range<1>(range<1>(N), range<1>(wg_size)),
```

```
  [=] (nd_item<id> it)
  [[sycl::reqd_work_group_size(1, 1, REQD_WG_SIZE)]] {
    auto gid = it.get_global_id(0);
    accessorRes[gid] = accessorIdx[gid] * 2;
  });
```

4.2.3 Specify Number of SIMD Work-Items

You have the option to increase the data-processing efficiency of a SYCL kernel by executing multiple work-items in a single instruction multiple data (SIMD) manner without manually vectorizing your kernel code.

Specify the number of work-items within a work-group that the Intel® oneAPI DPC++/C++ Compiler should execute in a SIMD or vectorized manner.

DEPRECATION NOTICE:

The `[[cl::reqd_work_group_size(Z, Y, X)]]` attribute is deprecated. Use the `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute.

Important: Introduce the `[[intel::num_simd_work_items(N)]]` attribute in conjunction with the `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute. The `[[intel::num_simd_work_items(N)]]` attribute you specify must evenly divide the last argument that you specify to the `req_work_group_size` attribute.

To specify the number of SIMD work-items in a work-group, insert the `[[intel::num_simd_work_items(N)]]` attribute in the kernel source code.

Consider the following example:

```
cgh.parallel_for<class kernelComputeSIMD>(
  nd_range<1>(range<1>(N), range<1>(REQD_WORK_GROUP_SIZE)),
  [=] (nd_item<id> it)
    [[intel::num_simd_work_items(NUM_SIMD_WORK_ITEMS),
    sycl::reqd_work_group_size(1, 1, REQD_WORK_GROUP_SIZE)]] {
      auto gid = it.get_global_id(0);
      accessorRes[gid] = cl::sycl::sqrt(accessorIdx[gid]);
    }
```

NOTE

Always use the `[[intel::num_simd_work_items(N)]]` attribute with `[[sycl::reqd_work_group_size(Z, Y, X)]]`, and `REQD_WORK_GROUP_SIZE` % `NUM_SIMD_WORK_ITEMS` must be 0.

For additional information about `[[sycl::reqd_work_group_size(Z, Y, X)]]` attribute, refer to Specify a Work-Group Size on page 178.

4.2.4 Omit Hardware that Generates and Dispatches Kernel IDs

The `[[intel::max_global_work_dim(0)]]` kernel attribute instructs the Intel® oneAPI DPC++/C++ Compiler to omit logic that generates and dispatches global, local, and group IDs into the compiled kernel.

Semantically, the `[[intel::max_global_work_dim(0)]]` kernel attribute specifies that the global work dimension of the kernel is zero. Setting this kernel attribute means that the kernel does not use any global, local, or group IDs. The presence of this attribute in the kernel code serves as a guarantee to the compiler that the kernel is a single work-item kernel.

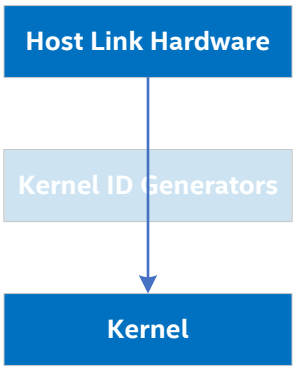
When compiling the following kernel, the compiler generates interface hardware as illustrated in Figure 82 on page 180:

```
cgh.single_task<class kernelComputeAsTask>(
  [=]()
    [[intel::max_global_work_dim(0)]] {
      for (unsigned i = 0; i < SIZE; i++) {
        accessorRes[i] = accessorIdx[i] * 2;
      }
    });
```

NOTE

The `[[intel::max_global_work_dim(0)]]` attribute must be run as a task and not as a `parallel_for` function.

Figure 82. Compiler-generated Interface Hardware for a Kernel with the `[[intel::max_global_work_dim(0)]]` Attribute



If your current kernel implementation has multiple work-items but does not use global, local, or group IDs, you can use the `[[intel::max_global_work_dim(0)]]` kernel attribute if you modify the kernel code accordingly:

1. Wrap the kernel body in a `for` loop that iterates as many times as the number of work-items.
2. Use `cgh.single_task<kernelName>` to invoke the device code.

4.2.5 Omit Hardware to Support the `no_global_work_offset` Attribute in `parallel_for` Kernels

The `[[intel::no_global_work_offset(1)]]` kernel attribute instructs the Intel® oneAPI DPC++/C++ Compiler to omit generating hardware required to support global work offsets. This may improve area and/or throughput for all device kernels that are invoked using `parallel_for` without the `workItemOffset` parameter and that do not use `ndrange` objects with offsets.

4.2.6 Reduce Kernel Area and Latency

The `[[intel::use_stall_enable_clusters]]` attribute enables you to direct the Intel® oneAPI DPC++/C++ Compiler to reduce the area and latency of your kernel. Reducing the latency does not have a large effect on loops that are pipelined, unless the number of iterations of the loop is very small.

Computations in an FPGA kernel are normally grouped into the following cluster types:

- **Stall-Free Clusters (SFC)**: Allows simplification of signals within a cluster, but the FIFO queue at the end of the cluster is used to save intermediate results if the computation must stall. For more information about SFCs, refer to Clustering the Datapath.
- **Stall-Enable Clusters (SEC)**: Saves area and cycles by removing the FIFO queue and passing the stall signals to each part of the computation. These extra signals may cause the f_{MAX} to reduce. For more information, refer to Clustering the Datapath.

CAUTION

If you specify the `[[intel::use_stall_enable_clusters]]` attribute on one or more kernels, the compiler might reduce the f_{MAX} of the generated FPGA bitstream, which may reduce performance on all kernels.

INTEL STRATIX 10 RESTRICTION:

The `[[intel::use_stall_enable_clusters]]` attribute is not applicable for designs that target the Intel® Stratix® 10 architecture unless the `-Xshyper-optimized-handshaking=off` option is passed to your `dpcpp` command.

Example 20. Example

```
h.single_task<class KernelComputeStallFree>( [=]()
[[intel::use_stall_enable_clusters]] {
  // The computations in this device kernel uses Stall Enable Clusters
  Work(accessor_vec_a, accessor_vec_b, accessor_res);
});
```

The compiler uses Stall-Enable Clusters for the kernel when possible. Some computations might not be stallable, so the compiler places them in a Stall-Free Cluster even if a Stall-Enable Cluster was requested.

NOTE

For more information, refer to the FPGA tutorial sample "Stall Enable Clusters" listed in the Intel® oneAPI Samples Browser on Linux*/Windows* or GitHub.

4.3 Kernel Controls

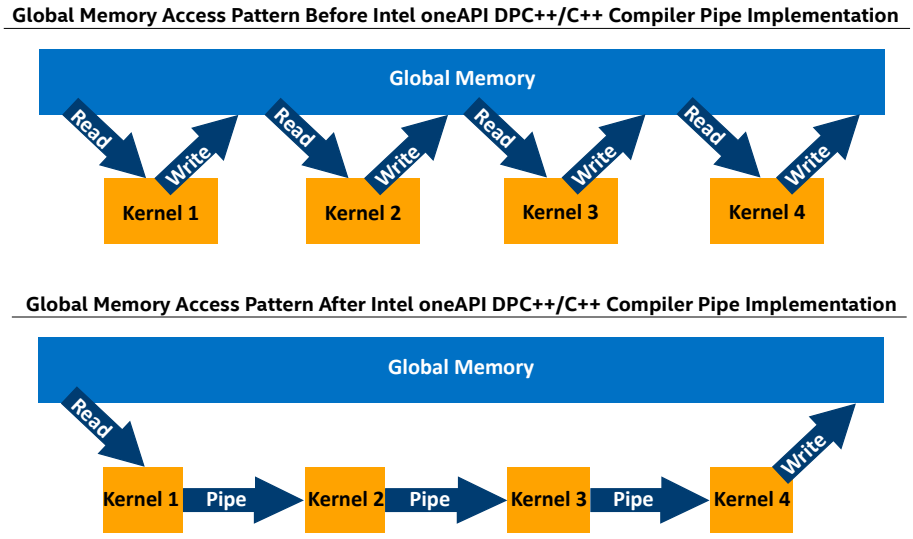
Kernel controls provide mechanism for passing data to or from host to kernel or kernel to kernel and help in synchronizing kernels.

This section describes the Pipes Extension on page 182 kernel control in detail.

4.3.1 Pipes Extension

Using global memory to communicate data between your kernels can constrain the performance of your design. DPC++ pipes provide a mechanism for passing data between kernels and synchronizing kernels with high efficiency and low latency. DPC++ pipes allow kernels to use on-device FIFO buffers to communicate directly with each other. The memory model of pipes allows them to be used for inter-kernel communication without waiting for kernel completion or involvement of the host processor, as shown in the following figure:

Figure 83. Using SYCL* Pipes to Decouple Data Movement between Concurrently Executing Kernels



4.3.1.1 Key Properties of a Pipe

The following are the key properties of a pipe:

Table 25. Key Properties

Property	Description
FIFO ordering	Data is only accessible (readable) in FIFO order. *Note:* There is no concept of a memory address or pointer (that is, there is no random access).
Capacity	Number of outstanding words or packets that can be written to an initially empty pipe before reading anything from it.

4.3.1.2 Pipe Accessors

Data is written to a pipe through an API that commits a single word or packet (of data type contained in the pipe), and that word or packet is later returned by an API reading data from the pipe. The API accessing the pipe can be a blocking or non-blocking type. Blocking calls wait until there is available a capacity to commit data, or until data is available to be read. Non-blocking calls do not wait. They return with a status to indicate whether their operation is successful or not.

4.3.1.3 The `pipe` Class and its Use

The `pipe` Class and its Use

The `pipe` API exposed by the FPGA implementation is equivalent to the following class declaration:

```
template <class name,
          class dataT,
          size_t min_capacity = 0>
class pipe {
public:
  // Blocking
  static dataT read();
  static void write(dataT data);
  // Non-blocking
  static dataT read(bool &success_code);
  static void write(dataT data, bool &success_code);
}
```

The following table describes the `template` parameters:

Table 26. Template Parameters

Parameter	Description
`name`	The type that is the basis of a pipe identification. It is typically a user-defined class, in a user namespace. Forward declaration of the type is enough, and the type need not be defined.
`dataT`	The type of data packet contained within a pipe. This is the data type that is read during a successful pipe `read()` operation, or written during a successful pipe `write()` operation. The type must have a standard layout and be trivially copyable.
`min_capacity`	User-defined minimum number of words (in units of `dataT`) that the pipe must be able to store without any being read out. The compiler may create a pipe with a larger capacity due to performance considerations.

The `pipe` class exposes static methods for writing a data word to a pipe and reading a data word from a pipe. The reads and writes can be blocking or non-blocking depending on the parameters you pass to the `read()` and/or `write()` function.

NOTE

A data word in this context is the data type that the pipe contains (`dataT pipe template` argument).

Example 21. Example Code Using Blocking Inter-Kernel Pipes

When writing code with SYCL* pipes, use of the C++ type alias mechanism (`using`) is highly encouraged to avoid errors where slightly different pipe types inadvertently lead to unique pipes. The following code sample shows how to use pipes with blocking accessors to transfer data between two kernels:

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
constexpr int N = 3;
// Specialize a pipe type
using my_pipe = ext::intel::pipe<class some_pipe, int, 8>;
void producer(const std::array<int, N> &src) {
  queue q;
  // Launch the producer kernel
  buffer<int> src_buf = {std::begin(src), std::end(src)};
  q.submit([&](handler &cgh) {
    // Get read access to src array
    accessor rd_src_buf(src_buf, cgh, read_only);
    cgh.single_task<class producer>([=]() {
      for (int i = 0; i < N; i++) {
        // Blocking write an int to the pipe
        my_pipe::write(rd_src_buf[i]);
      }
    });
  });
}
void consumer(std::array<int, N> &dst) {
  queue q;
  // Launch the consumer kernel
  buffer<int> dst_buf = {std::begin(dst), std::end(dst)};
  q.submit([&](handler &cgh) {
    // Get write access to dst array
    accessor wr_dst_buf(dst_buf, cgh, write_only);
    cgh.single_task<class consumer>([=]() {
      for (int i = 0; i < N; i++) {
        // Blocking read an int from the pipe
        wr_dst_buf[i] = my_pipe::read();
      }
    });
  });
}
```

The pipe data packet is of type `int` and the pipe has a depth of 8, as specified by the template parameters of `my_pipe` type. The pipe `read()` call blocks only when the pipe is empty, and the pipe `write()` call blocks only when the pipe is full.

NOTE

The SYCL specification does not guarantee concurrent kernel execution. However, the Intel® oneAPI DPC++/C++ Compiler supports concurrent execution of kernels. You can execute multiple SYCL kernels concurrently by launching them using separate command queues (as shown in the above **Example Code Using Blocking Inter-Kernel Pipes** on page 184). Hence, you can modify your host application and kernel program to take advantage of this capability. The modifications increase the throughput of your application.

Example 22. Example Code Using Non-Blocking Inter-Kernel Pipes

The code samples (Sample 1 and Sample 2) in this section illustrate how to use pipes with non-blocking writes and reads to transfer data between two concurrently running kernels:

```
//Sample 1
#include <CL/sycl.hpp>
using namespace cl::sycl;
constexpr size_t N = 16;
// Specialize the two pipe types, differentiated based on their
// first template parameter
using pipe1 = ext::intel::pipe<class some_pipe, int>;
using pipe2 = ext::intel::pipe<class other_pipe, int>;
void producer(const std::array<int, N> &src) {
  queue q;
  // Launch the producer kernel
  // Get read access to src array
  buffer<int> src_buf = {std::begin(src), std::end(src)};
  q.submit([&](handler &cgh) {
    accessor rd_src_buf(src_buf, cgh, read_only);
    cgh.single_task<class producer>([=]() {
      for (int i = 0; i < N; i++) {
        bool success = false;
        do {
          pipe1::write(rd_src_buf[i], success);
          if (!success) {
            pipe2::write(rd_src_buf[i], success);
          }
        }
        while (!success);
      }
    });
  });
}
// the consumer kernels are not shown here
```

For both pipes, the data packet is of type `int`. The pipes are different because the first `template` parameter is different. The non-blocking pipe `write()` and `read()` calls do not block. They respectively return a boolean value that indicates whether data is written successfully to the pipe (that is, the pipe is not full) or if the data is read successfully from the pipe (that is, the pipe is not empty).

Perform non-blocking pipe writes to facilitate applications where writes to a full FIFO buffer should not cause the kernel to stall until a slot in the FIFO buffer becomes free. Consider a scenario where your application has one data producer with two identical workers that consume the data. Assume the time each worker takes to process a message varies depending on the contents of the data. In this case, there might be a situation where one worker is busy while the other is free. A non-blocking write can

facilitate work distribution such that both workers are busy. Like a non-blocking write, perform non-blocking reads to facilitate applications where data is not always available, and other operations need not wait for the data to become available.

NOTE

You can mix blocking and non-blocking accessors for writing or reading data to or from pipes. For example, you can write data to a pipe using a blocking pipe `write()` call and read it from the other end using a non-blocking pipe `read()` call, and vice versa.

```
//Sample 2
#include <CL/sycl.hpp>
using namespace cl::sycl;
constexpr size_t N = 16;
// Specialize the two pipe types, differentiated based on their first template
// parameter
using pipe1 = ext::intel::pipe<class some_pipe, int>;
using pipe2 = ext::intel::pipe<class other_pipe, int>;
// the producer kernels are not shown
void consumer(const std::array<int, N> &dst) {
  queue q;
  // Launch the consumer kernel
  buffer<int> dst_buf = {std::begin(dst), std::end(dst)};
  q.submit([&](handler &cgh) {
    // Get write access to src array
    accessor wr_dst_buf(dst_buf, cgh, write_only);
    cgh.single_task<class consumer>([=]() {
      int = 0;
      while (i < N) {
        bool valid0 = false, valid1 = false;
        auto data0 = pipe1::read(valid0);
        auto data1 = pipe2::read(valid1);
        if (valid0) {
          wr_dst_buf[i++] = process(data0);
        }
        if (valid1) {
          wr_dst_buf[i++] = process(data1);
        }
      }
    });
  });
}
```

NOTE

For additional information, refer to FPGA tutorial samples "Pipe Array" and "Pipes" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code samples in GitHub.

4.3.1.4 I/O Pipes

An I/O pipe is a unidirectional (source or sink) connection to the hardware that may be connected to input or output features of an FPGA board. These features might include network interfaces, PCIe®, cameras, or other data capture or processing devices or protocols.

A source I/O device provides data that a SYCL kernel can read whereas a sink I/O device accepts data written by a SYCL kernel and sends it to the hardware device. A common use of I/O pipes is to interface to ethernet connections that interface directly

with the FPGA. The source reads from the network and the sink writes to the network. This allows a SYCL kernel to process data from the network and resend it back to the network.

For testing purposes, you can use files on the disk as input or output devices, allowing you to debug using emulation or simulation for faster compilation.

I/O Pipes Implementation

To implement I/O pipes, follow these steps:

1. Consult your board vendor documentation before you implement I/O pipes in your kernel program.
2. Use a `struct` with a numeric `id` to define a `kernel_readable_io_pipe` or `kernel_writable_io_pipe` type to declare an I/O pipe to interface with hardware peripherals. These definitions are typically provided by a board vendor.
   - The numeric `id` value is the 0-origin index into the interfaces in the channels section of the `board_spec.xml` for the device.
   - The `chan_id` argument is necessary for simulation and it is the name of the I/O interface listed in the `board_spec.xml` file as shown in the following:

```
 <channels>
   <interface name="board" port="c1"  type="streamsource" width="32"
chan_id="c1"/>
   <interface name="board" port="c2"  type="streamsink" width="32"
chan_id="c2"/>
 </channels>
```

NOTE

Only channels marked type `streamsource` or `streamsink` are used for indexing.

3. Implement the interface to hardware I/O pipes or files (emulator or simulator) by mapping the `id` variable, as shown in the following:

```
// Specialize a pipe type
struct read_io_pipe {
  static constexpr unsigned id = 0;
};

struct write_io_pipe {
  static constexpr unsigned id = 1;
};

// id 0 -> file name or channel name: "c1" for hardware, "0" for emulator,
"c1" for simulation.
using read_iopipe = ext::intel::kernel_readable_io_pipe<read_io_pipe,
unsigned, 4>;

// id 1 -> file name or channel name: "c2" for hardware, "1" for emulator,
"c2" for simulation.
using write_iopipe = ext::intel::kernel_writeable_io_pipe<write_io_pipe,
unsigned, 3>;
```

where:

Interface	Description
For Hardware	`id N` is mapped to the channel (not file) in the associated hardware. For example: • `id 0` is mapped to the `chan_id` in the first interface defined. • `id 1` is mapped to the `chan_id` in the second interface defined, and so on. If there is no matching channel, an error is generated.
For Emulator	`id N` is mapped to a file named `N`, which means `id 0` is file "0". This file is read or written by reading or writing to the associated I/O pipe.
For Simulator	`id N` is mapped to `chan_id` names in the `board_spec.xml` file supplied with the BSP (See channels). For example: • `id 0` is mapped to the `chan_id` in the first interface defined. • `id 1` is mapped to the `chan_id` in the second interface defined, and so on. If there is no matching channel, an error is generated.

The I/O Pipe Classes and Their Use

The I/O pipe APIs exposed by the FPGA implementations is equivalent to the following class declarations:

```
template <class name,
          class dataT,
          size_t min_capacity = 0>
class kernel_readable_io_pipe {
  public:
    static dataT read();  // Blocking
    static dataT read(bool &success_code);  // Non-blocking
};

template <class name,
          class dataT,
          size_t min_capacity = 0>
class kernel_writeable_io_pipe {
  public:
    static void write(dataT data);  // Blocking
    static void write(dataT data, bool &success_code);  // Non-blocking
}
```

The following table describes the `template` parameters:

Table 27. Template Parameters

Parameter	Description
`name`	The type that is the basis of an I/O pipe identification. It may be provided by the device vendor or user defined. The type must contain a `static constexpr unsigned` expression with name `id`, which is used to determine the hardware device referenced by the I/O pipe.
`dataT`	The type of data packet contained within an I/O pipe. This is the data type that is read during a successful pipe `read()` operation, or written during a successful pipe `write()` operation. The type must have a standard layout and be trivially copyable.
`min_capacity`	User-defined minimum number of words (in units of `dataT`) that an I/O pipe must be able to store without any being read out. The compiler may create an I/O pipe with a larger capacity due to performance considerations.

NOTE

A data word in this context is the data type that the pipe contains (`dataT pipe template` argument).

Example 23. Example Code for I/O Pipes

Here is an example that includes the definitions above:

```
// "Built-in pipes" provide interfaces with hardware peripherals
// These definitions are typically provided by a device vendor and
// made available to developers for use.
namespace example_platform {
  template <unsigned ID>
  struct ethernet_pipe_id {
    static constexpr unsigned id = ID;
  };

  using ethernet_read_pipe = kernel_readable_io_pipe<ethernet_pipe_id<0>, int, 0>;
  using ethernet_write_pipe = kernel_writeable_io_pipe<ethernet_pipe_id<1>, int,
0>;
}
```

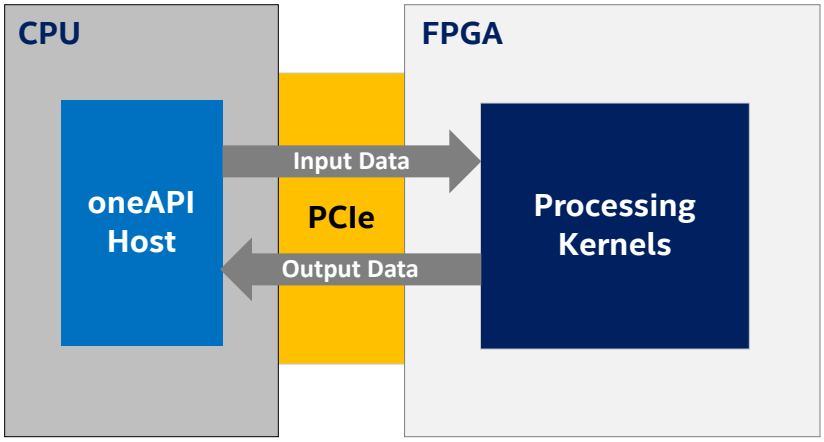
NOTE

For additional information, refer to the FPGA tutorial sample "IO Streaming with IO Pipes" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

Using I/O Pipes to Stream Data

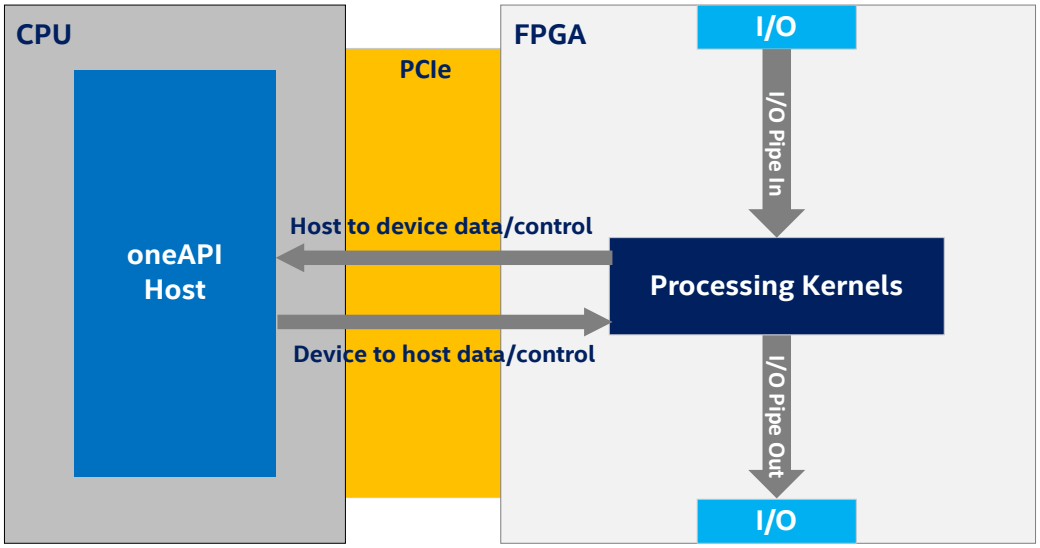
In many of the design examples, an FPGA device is treated as an accelerator, as illustrated in the following diagram, where the main computation (and therefore the data to be computed) resides on the host (CPU) and you accelerate some compute-intensive task using kernels on the device. The host moves the data to the device, performs the calculation, and moves the data back:

Figure 84. FPGA Device Used as an Accelerator



However, a key feature of FPGAs is their rich input-output (I/O) capabilities (for example, Ethernet). Taking advantage of these capabilities within the oneAPI programming environment requires a different programming model than the accelerator model as described above. In the model illustrated in the following figure, consider a kernel (or kernels) where some of the kernels are connected to the FPGA's I/O via I/O pipes and the main data flow is through the FPGA device rather than from CPU to FPGA and back:

Figure 85. Data Flow Through the FPGA Device Using I/O Pipes



In the Figure 85, there are four possible directions of data flow:

- I/O to device
- Device to I/O
- Device to host
- Host to device

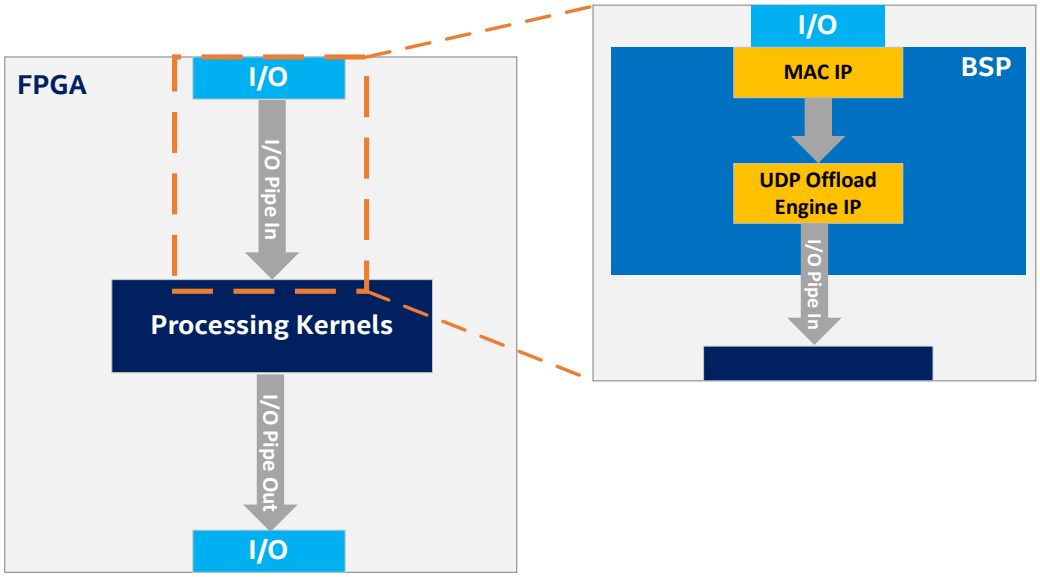
The direction and amount of data flowing in the system is application dependent. The main source of data is I/O. Data streams into the device from I/O (I/O to device) and out of the device to I/O (device to I/O). However, the host and device exchange low-bandwidth control signals using a host-to-device and device-to-host connection (or side channel).

I/O pipes have the same interface API as inter-kernel pipes. This abstraction is implemented in the Board Support Package (BSP) and provides a much simpler programming model for accessing the FPGA I/O. See The I/O Pipe Classes and Their Use on page 188.

Consider an example scenario where you want the kernel to be able to receive UDP packets through the FPGA's Ethernet interface. Implementing the necessary hardware to process the data coming from the Ethernet pins in oneAPI would be both extremely difficult and inefficient. Moreover, there are already many RTL solutions for doing this. So, instead, you can implement this low-level logic with RTL in the BSP. This example is illustrated in the following figure, where the BSP connects the Ethernet I/O pins to a MAC IP, the MAC IP to a UDP offload engine IP, and finally the UDP offload engine IP to an I/O pipe. This I/O pipe can then be simply connected to the processing kernels. The

details of these connections are abstracted from the oneAPI kernel developer. The following figure shows only an input I/O pipe. The process is similar for an output I/O pipe, but the data flows in the opposite direction.

Figure 86. I/O Input Pipe



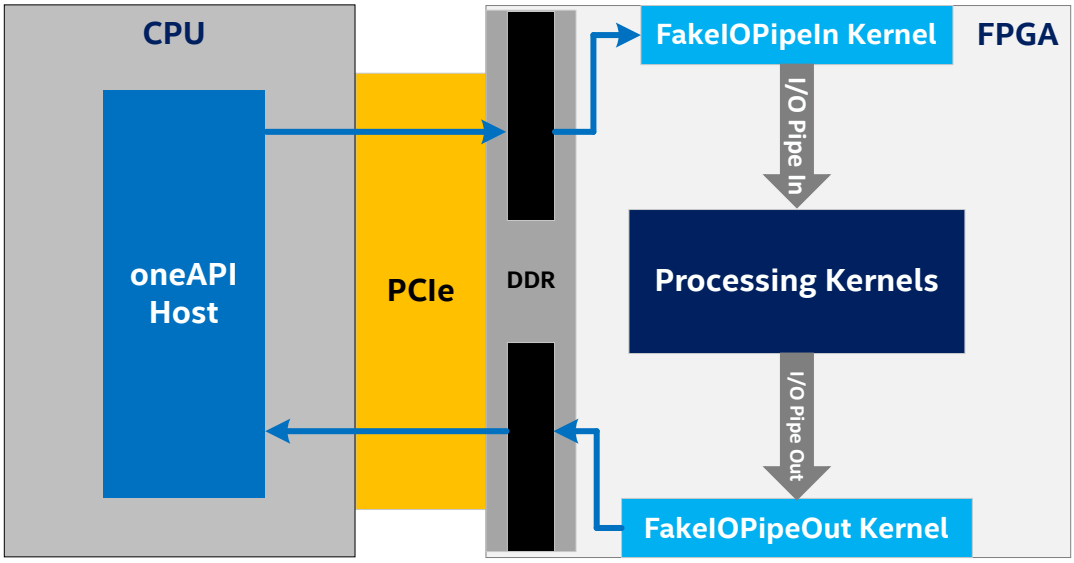
Faking I/O Pipes

Unfortunately, designs that use these I/O pipes are confined to BSPs that support that specific I/O pipe, which makes prototyping and testing the processing kernels difficult. To address these concerns, a fake I/O pipes library is available. With this library, you can create kernels that behave like I/O pipes without having a BSP that actually supports them. This allows you to start prototyping and testing your processing kernels without a BSP that supports I/O pipes.

There are currently two options for faking IO pipes:

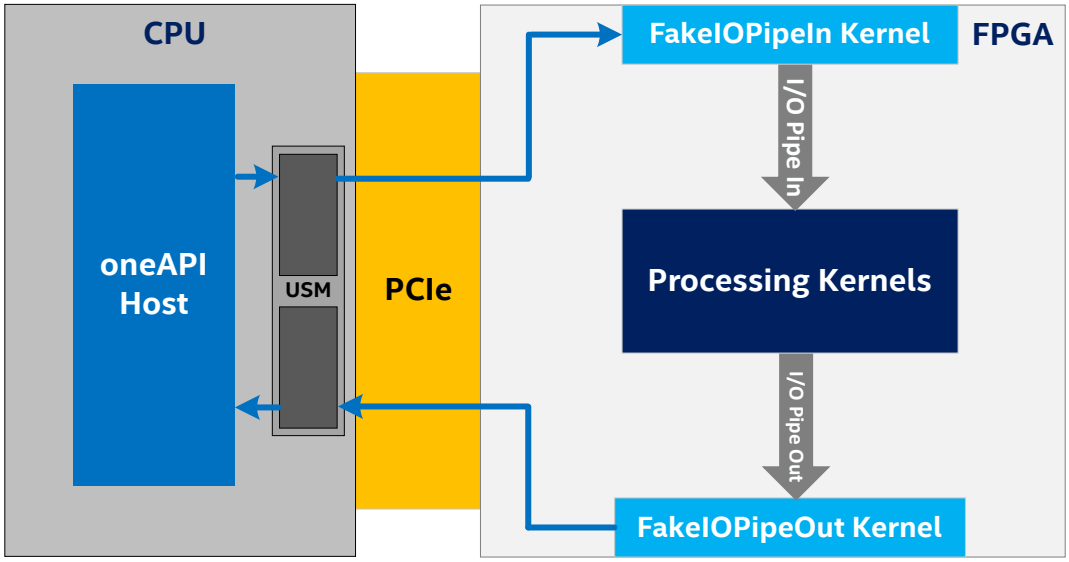
- **Device memory allocations**: This option requires no special BSP support, as shown in the following figure:

Figure 87. Device Memory Allocations



- **Unified Shared Memory (USM) host allocations**: This option requires USM support in the BSP as shown in the following figure. See also Unified Shared Memory topic in the open-access book *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*.

Figure 88. Unified Shared Memory (USM) Host Allocations



4.3.1.5 Characteristics of Pipes

Pipes in your FPGA program have characteristics as described in this section.

Data Persistence in Pipes

Data written to a pipe using a pipe `write()` call remains in the pipe as long as the kernel program remains loaded on the FPGA device. In other words, the data written to a pipe is persistent across work-groups and NDRange invocations. Data written by a work item to a pipe remains in that pipe until another work item reads from it. In addition, the sequence of data in a pipe always follows FIFO ordering, and the order is independent of the work item that performs the write or read operation.

Data in pipes is not persistent across multiple or different invocations of kernel programs that lead to FPGA device reprogramming, or between context, program, device, or platform releases, even if the compiler performs optimizations that avoid reprogramming operations on a device. For example, if you run a host program twice using the same FPGA image, or if a host program releases and reacquires a context, the data in the pipe may or may not persist across the operation. FPGA device reset operations might happen behind the scenes on object releases that may purge data in any pipes.

Work-Item Order in NDRange kernels

In an NDRange kernel, the order in which work items and work groups access a pipe is not deterministic. Kernels should not make assumptions about the order in which the data from different work items is written to or read from a pipe.

4.3.1.6 Restrictions of Pipes

The following table summarizes restrictions of pipes:

Table 28. Restrictions of Pipes

Restriction	Description
Multiple Pipe Call Sites	A kernel can read from the same pipe multiple times. However, multiple kernels cannot read from the same pipe. Similarly, a kernel can write to the same pipe multiple times, but multiple kernels cannot write to the same pipe.
Feedback and Feed-Forward Pipes	Within a single kernel, you should either read from a pipe or write to a pipe. Writing and reading to the same pipe within a single kernel may lead to poor performance.
Emulation Support	The FPGA emulator supports emulation of kernels that contain pipes. For better conformity between emulation and hardware, provide a non-zero pipe capacity while specializing a pipe type. For more information about the Emulator, refer to the Intel® oneAPI Programming Guide. *Note:* If the same kernel is invoked more than once, the FPGA emulator may attempt to execute kernels concurrently, resulting in a data race if both invocations attempt to read or write from the same pipe. This issue affects only emulation. In the hardware flow, multiple invocations of the same kernel are executed serially. To work around this issue, add a call to the `cl::sycl::queue::wait()` function between executions of the same kernel.

4.3.1.7 Guidelines for Designing Pipes

Consider the following best practices when designing pipes:

- Determine whether you should split the design into multiple kernels connected by pipes.
- Aggregate data on pipes only when all the data is used at the same point in the kernel.
- Do not use non-blocking pipes if you are using a looping structure waiting for the data, that is, avoid the following coding pattern for non-blocking pipe accessors:

```
bool success = false;
while (!success) {
  my_pipe::write(rd_src_buf[i], success); // can be a non-blocking read too
}
```

NOTE

Whenever you use the above code pattern, use the corresponding blocking accessor instead because it is more efficient in hardware.

4.3.1.8 Pipe and Atomic Fence

ATTENTION

This topic assumes that you already have an understanding of the `atomic_fence` function described in the SYCL specification. If you are new to it, then before you proceed, read about the `atomic_fence` function in the Khronos* SYCL Specification.

When running kernels in parallel, you might want multiple kernels to collaboratively access a shared memory. SYCL* provides the `atomic_fence` function as a synchronization construct to reason about the order of memory instructions accessing the shared memory. The `atomic_fence` function controls the reordering of memory load and store operations (subject to the associated memory order and memory scope) when paired with synchronization through an atomic object. Pipe read and write operations behave as if they are SYCL-relaxed atomic load and store operations. When paired with `atomic_fence` functions to establish a synchronizes-with relationship, pipe operations can provide guarantee on side-effect visibility in memory, as defined by the SYCL memory model. For additional information about the `atomic_fence` function, refer to the Khronos* SYCL Specification.

CAUTION

The current `atomic_fence` function for FPGA uses an overly conservative implementation and is still preliminary.

- The implementation guarantees only functional correctness and not the maximum performance because the `atomic_fence` function currently enforces more memory ordering than it requires. If you do not use the `atomic_fence` function with a correct `memory_order` parameter, then you might see unexpected behavior in your program when the `atomic_fence` function handles memory ordering properly in a future release.
- The implementation does not support the `memory_scope::system` constraint. The broadest scope supported for FPGA is the `memory_scope::device` constraint.

Example 24. Example Code for Using the `atomic_fence` Function and Blocking Inter-Kernel Pipes

The following code sample shows how to use the `atomic_fence` function with a blocking inter-kernel pipe to synchronize the load and store to a shared device memory between a producer and a consumer:

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
using my_pipe = ext::intel::pipe<class some_pipe, int>;
constexpr int READY = 1;

int produce_data(int data);
int consume_data(int data);

event Producer(queue&q, int *shared_ptr, size_t size) {
  return q.submit([&](handler& h) {
    h.single_task<class ProducerKernel>([=]() [[intel::kernel_args_restrict]] {
      // create a device pointer to explicitly inform the compiler the
      // pointer resides in the device's address space
      device_ptr<int> shared_ptr_d(shared_ptr);

      // produce data
      for (size_t i = 0; i < size; i++) {
        shared_ptr_d[i] = produce_data(i);
      }
      // use atomic_fence to ensure memory ordering
      atomic_fence(memory_order::seq_cst, memory_scope::device);
      // notify the consumer to start data processing
      my_pipe::write(READY);
    });
  }
}

event Consumer(queue & q, int* shared_ptr, size_t size, int *output_ptr) {
  return q.submit([&](handler& h) {
    h.single_task<class ConsumerKernel>([=]() [[intel::kernel_args_restrict]] {
      // create device pointers to explicitly inform the compiler these
      // pointer reside in the device's address space
      device_ptr<int> shared_ptr_d(shared_ptr);
      device_ptr<int> out_ptr_d(output_ptr);

      // wait on the blocking pipe_read until notified by the producer
      int ready = my_pipe::read();

      // use atomic_fence to ensure memory ordering
      atomic_fence(memory_order::seq_cst, memory_scope::device);
```

```
      // consume data and write to output memory address
      for(int i = 0; i < size; i++) {
        out_ptr_d[i] = consume_data(shared_ptr_d[i]);
      }
    });
  });
}
```

In the above example, the consumer loads data produced by the producer. To prevent a scenario where the consumer loads the shared device memory before the producer finishes storing to it, a blocking pipe is used to synchronize between the two kernels. The consumer's pipe read does not return until it sees the `READY` written by the producer. In this example, the `atomic_fence` functions in the producer and consumer prevent the shared memory read and write from being reordered with the pipe instructions. They also form a release-acquire ordering, which ensures that by the time the consumer sees the pipe read returns, the producer's write operation to the shared device memory is also visible to the consumer.

NOTE

The shared device memory is created using a USM device allocation that allows the two kernels to be running in parallel even though they both access the shared device memory simultaneously.

4.4 Kernel Variables

This section describes mechanisms available to manipulate datapath for kernel variables.

fpga_reg

The Intel® oneAPI DPC++/C++ Compiler provides FPGA extension `fpga_reg()` that you can include in your kernel code.

The `fpga_reg()` function directs the compiler to insert at least one register between the operand and the return value of the function call. In general, it is not necessary to include the `fpga_reg()` function in your kernel code to achieve desired performance.

NOTE

Intel® strongly recommends that you use the `fpga_reg()` function only if you are experienced in using the Intel® Quartus® Prime Pro Edition software performing advanced optimization for a specific target device. You must have sufficient knowledge about the placement of portions of the datapath on an FPGA device.

Syntax

```
T fpga_reg(T op)
```

Where, `T` may be any sized type, such as standard SYCL* device data types, or a user-defined `struct` containing SYCL types.

Example

Consider the following example:

```
#include <sycl/ext/intel/fpga_extensions.hpp>
...
r[k] = ext::intel::fpga_reg(a[k]) + b[k];
...
```

Use the `fpga_reg()` function to perform the following:

- Break critical paths between spatially distant portions of a datapath, such as between processing elements of a large systolic array.
- Reduce the pressure on placement and routing efforts caused by spatially distinct portions of the kernel implementation.

The `fpga_reg()` function directs the compiler to insert at least one hardware pipelining register on the signal path that assigns the operand to the return value. This built-in function operates as an assignment in the SYCL programming language where the operand is assigned to the return value. The assignment has no implicit semantic or functional meaning beyond a standard-C assignment. Functionally, you can consider the `fpga_reg()` function being always optimized away by the compiler.

TIP

For additional information, refer to the FPGA tutorial sample "FPGA register" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

NOTE

The compiler does not provide feedback on where you should insert the `fpga_reg()` function calls in your code. Use the Intel® Quartus® Prime Pro Edition software to determine where you should insert the calls to address specific aspects of performance.

You may introduce nested `fpga_reg()` function calls in your kernel code to increase the minimum number of registers that the compiler inserts on the assignment path. Because each function call guarantees the insertion of at least one register stage, the number of calls provides a lower limit on the number of registers.

4.5 Memory Attributes

The following table lists attributes that allow fine-grained control on how you can implement a variable (usually an array) in on-chip memory. The attribute immediately precedes the variable declaration.

Table 29. FPGA Memory Attributes

Attribute	Syntax	Description
`bank_bits`	`[[intel::bank_bits(`b_0`,`b_1`,...,`b_n`)]]`	Specifies that the local memory addresses should use bits (b_0,b_1,...,b_n) for bank selection, where, (b_0,b_1,...,b_n) are indicated in terms of word-addressing and not byte-addressing. As a result, the

continued...

Attribute	Syntax	Description
		number of banks is equal to $2^{<\text{number of bank bits}>}$. The bits of the local memory address not included in ($b_0, b_1, \ldots, b_n$) are used for word selection in each bank.
bankwidth	[[intel::bankwidth(N)]]	Specifies that the memory system implementing the local variable must have banks that are N bytes wide, where N is a power-of-2 integer value greater than zero.
doublepump	[[intel::doublepump]]	Specifies that the memory system implementing the local variable must operate at twice the clock frequency of the kernel accessing it. This allows twice as many memory accesses per kernel clock cycle but may reduce the maximum kernel clock frequency.
force_pow2_depth	[[intel::force_pow2_depth(N)]]	Specifies that the memory implementing the variable or array has a power-of-2 depth. This attribute is enabled if N is 1, and disabled if N is 0.
max_replicates	[[intel::max_replicates(N)]]	Specifies that the memory implementing the local variable or array has no more than the specified number of replicates to enable simultaneous reads from the datapath.
fpga_memory	[[intel::fpga_memory("impl_type")]]	Specifies that the compiler must implement the local variable in a memory system. You may pass an optional string argument to specify the memory implementation type. Specify impl_type as either a BLOCK_RAM or MLAB to implement the memory using memory blocks (for example, M20K) or memory logic array blocks (MLABs), respectively.
merge	[[intel::merge("key", "direction")]]	Merges of two or more variables or arrays defined in the same scope in a width-wise or depth-wise manner. All variables with the same key string are merged into the same memory system. The string direction can be either width or depth.
numbanks	[[intel::numbanks(N)]]	Specifies that the memory system implementing the local variable must have N banks, where N is a power-of-2 integer value greater than zero.
private_copies	[[intel::private_copies(N)]]	Specifies that the memory has a defined number of copies to allow simultaneous iterations of a loop at any given time. When you declare an array in the scope of a loop body, the array is private to that iteration of the loop. However, concurrency is limited if all iterations of the loop must share the same physical memory. Specifying private_copies(N) allows N iterations of the loop to execute

continued...

Attribute	Syntax	Description
		concurrently, each with its own private copy of the array. A larger value of `N` may expose more opportunities for parallel execution, at a cost of higher on-chip memory utilization. `private_copies(N)` interacts with `max_concurrency` attribute applied to the loop. For more information, refer to the `max_concurrency` Attribute on page 205 and FPGA tutorial sample "Private Copies" listed in the Intel® oneAPI Samples Browser on Linux*/Windows* or GitHub.
`fpga_register`	`[[intel::fpga_register]]`	Specifies that the variable must be carried through the pipeline in registers. The compiler may implement a register variable either exclusively in flip-flops (FFs), or in a combination of FFs and RAM-based FIFOs.
`simple_dual_port`	`[[intel::simple_dual_port]]`	Specifies that the memory implementing the variable or array should have read-only and write-only ports rather than read or write ports. Specifying `simple_dual_port` forces the compiler to configure the memory's underlying hardware resources in simple dual port mode, which can have area benefits in some corner cases.
`singlepump`	`[[intel::singlepump]]`	Specifies that the memory system implementing the local variable must operate at the same clock frequency as the kernel accessing it.

TIP

For additional information, refer to the FPGA tutorial sample "Memory Attributes" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

Struct Data Types and Memory Attributes

You can apply memory attributes to the member variables in a `struct` variable within the `struct` declaration. If you also apply memory attributes to the object instantiation of a `struct` variable, the attributes on the instantiation override the attributes from the declaration.

Consider the following code example where memory attributes are applied to both a declaration and instantiation:

```
struct State {
 [[intel::fpga_memory]] int array[100];
 [[intel::fpga_register]] int reg[4];
};

cgh.single_task<class test>([=] {
  struct State S1;
```

```
  [[intel::fpga_memory]] struct State S2;
  // some uses
});
```

In this example code, the compiler splits `S1` into two variables, `S1.array[100]` (implemented in the memory) and `S1.reg[4]` (implemented in registers). However, the compiler ignores attributes applied at the `struct` declaration for object `S2` and does not split it since the `S2` object has the `[[intel::fpga_memory]]` attribute applied to it.

4.6 Loop Directives

The following directives apply to a loop that immediately follows the pragma or an attribute directive. No statements can be placed between the pragma or attribute directive and the loop (for, while, do) to which the directive applies, with the exception of other loop directives.

4.6.1 `disable_loop_pipelining` Attribute

If loop-carried dependencies result in an initiation interval (II) that is equal or close to the latency of a given iteration (effectively inducing serial execution of the pipelined loop), disable pipelining of the loop to generate a simpler datapath and reduce area utilization. Use the `disable_loop_pipelining` attribute to direct the Intel® oneAPI DPC++/C++ Compiler to disable pipelining of a loop. When you apply this attribute, the compiler generates a simple sequential loop datapath. This attribute applies to single work-item kernels (that is, single-threaded kernels) in which loops are pipelined. For more information about single work-item kernels and associated concepts, refer to Single Work-item Kernels on page 80.

Syntax

```
[[intel::disable_loop_pipelining]]
```

Unless otherwise specified, the compiler always attempts to generate a pipelined loop datapath where possible. When generating a pipelined circuit, resources of the loop must be duplicated to execute multiple iterations simultaneously, leading to an increased silicon area utilization. In cases where loop pipelining does not result in an improvement in throughput, avoid the area overhead by applying the `disable_loop_pipelining` attribute to the loop, as shown in the following example code snippet:

Example 25. Example

```
[[intel::disable_loop_pipelining]]
for (int i = 1; i < N; i++) {
  int j = a[i-1];
  // Memory dependency induces a high-latency loop feedback path
  a[i] = foo(j)
}
```

In the above example, the compiler fails to schedule the loop with a small II due to memory dependency (as reported in the Details pane of the Loop Analysis report). In such cases, loop pipelining is unlikely to be beneficial.

4.6.2 `initiation_interval` Attribute

The initiation interval, or II, is the number of clock cycles between the launch of successive loop iterations.

Use the `initiation_interval` attribute to direct the Intel® oneAPI DPC++/C++ Compiler to attempt to set the II for the loop that follows the attribute declaration. If the Intel® oneAPI DPC++/C++ Compiler cannot achieve the specified II for the loop, then the compilation errors out.

Syntax

```
[[intel::initiation_interval(n)]]
```

The `initiation_interval` attribute applies to pipelined loops in single task kernels. Refer to Pipelining on page 27 for information about loop pipelining.

The attribute parameter `n` is required and must be a positive constant expression of integer type. The parameter specifies a minimum number of clock cycles to wait between the beginnings of execution of successive loop iterations.

The higher the II value, the longer the wait before the subsequent loop iteration starts executing. Refer to Loop Analysis on page 42 for information about II and compiler reports that provide you with details on the performance implications of II on a specific loop.

NOTE

The `initiation_interval` attribute should only be applied to a pipelined loop in a single work-item (task) kernel.

If the throughput of a loop is important to the overall throughput of your kernel, you can use the `initiation_interval` attribute to force an II value of 1 even though this may result in lower f_{MAX}.

For some loops in your kernel, specifying a higher II value with the `initiation_interval` attribute than the value the compiler chooses by default can increase the maximum operating frequency (f_{MAX}) of your kernel without a decrease in throughput.

A loop is a good candidate to have a higher `initiation_interval` than the default if the loop meets the following conditions:

- The loop is not critical to the throughput of your kernel.
- The running time of the loop is small compared to other loops it might contain.

Example 26. Example

Consider a case where your kernel has two distinct pipelineable loops:

- A short-running initialization loop that has a loop-carried dependence
- A long-running loop that does the bulk of your processing.

In this case, the compiler does not know that the initialization loop has a much smaller impact on the overall throughput of your design. If possible, the compiler attempts to pipeline both loops with an II of 1.

Because the initialization loop has a loop-carried dependence, it does have a feedback path in the generated hardware. To achieve an II with such a feedback path, some clock frequency might be forfeited. Depending on the feedback path in the main loop, the rest of your design could have run at a higher operating frequency.

If you specify `[[intel::initiation_interval(2)]]` on the initialization loop, then you are informing the compiler that it can be less aggressive in optimizing II for this loop. Less aggressive optimization allows the compiler to pipeline the path limiting the f_{MAX} and allow your overall kernel design to achieve a higher f_{MAX}.

The initialization loop takes longer to run with its new II. However, the decrease in the running time of the long-running loop due to higher f_{MAX} compensates for the increased length in running time of the initialization loop.

4.6.3 `ivdep` Attribute

Include the `ivdep` attribute in your single task kernel to direct the Intel® oneAPI DPC++/C++ Compiler to ignore memory dependencies carried by the loop that the attribute is applied to. This attribute applies only to the loop it is applied to, and not to any of the future loops that might appear as a result of the [[intel::loop_coalesce(N)]] attribute.

Syntax

```
[[intel::ivdep]]
```

```
[[intel::ivdep(safelen)]]
```

```
[[intel::ivdep(array)]]
```

```
[[intel::ivdep(array, safelen)]]
```

```
[[intel::ivdep(safelen, array)]]
```

CAUTION

Applying the `ivdep` attribute incorrectly results in functionally incorrect hardware and potential functional differences between the hardware run and emulation. The `ivdep` attribute is in fact ignored in emulation.

During compilation, the Intel® oneAPI DPC++/C++ 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` attribute instructs the Intel® oneAPI DPC++/C++ Compiler to remove the extra hardware between load and store

instructions in the loop that immediately follows the attribute declaration in the kernel code. Removing the extra hardware may reduce logic utilization and lower the II value.

You can provide more information about loop dependencies by specifying a `safelen` parameter to the attribute by adding an integer type C++ constant expression argument to the attribute. The `safelen` parameter specifies the maximum number of consecutive loop iterations without loop-carried dependencies. For example, `[[intel::ivdep(32)]]` indicates to the compiler that there are at least 32 iterations of the loop before loop-carried dependencies might be introduced. That is, while the `[[intel::ivdep]]` attribute guarantees to the compiler that there are no implicit memory dependencies between any iteration of this loop, `[[intel::ivdep(32)]]` guarantees that there does not exist a loop-carried dependence with a dependence distance less than 32. For example, if an iteration reads from memory, the preceding 31 iterations and succeeding 31 iterations are guaranteed not to write to the same memory location.

To specify that accesses to a particular memory array inside a loop do not cause loop-carried dependencies, add the array parameter to the attribute by specifying the array variable name as an argument to the attribute. The array specified by the `ivdep` attribute must be a local or private memory array, or a pointer variable that points to a global, local, or private memory storage. The array specified by the `ivdep` attribute can also be an array or a pointer member of a `struct`.

Example 27. Examples

```
// No loop-carried dependencies for accesses to arrays A and B
[[intel::ivdep]]
for (int i = 0; i < N; i++) {
  A[i] = A[i - X[i]];
  B[i] = B[i - Y[i]];
}

// No loop-carried dependencies for accesses to array A
// Compiler inserts hardware that reinforces dependency constraints for B
[[intel::ivdep(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
[[intel::ivdep(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
[[intel::ivdep(S->X[2][3].A)]]
for (int i = 0; i < N; i++) {
  S->X[2][3].A[i] = S.A[i - X[i]];
}
```

NOTE

For additional information, refer to the FPGA tutorial sample "Loop IVDep" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

4.6.4 `loop_coalesce` Attribute

Use the `loop_coalesce` attribute to direct the Intel® oneAPI DPC++/C++ Compiler to coalesce nested loops into a single loop without affecting the loop functionality. Coalescing loops can help reduce your kernel area usage by directing the compiler to reduce the overhead needed for loop control.

NOTE

If you want to use the `ivdep` attribute to ignore loop-carried dependencies, apply it to the loop that causes dependencies and not to any of the future loops that might appear as a result of the `[[intel::loop_coalesce(N)]]` attribute.

Syntax

```
[[intel::loop_coalesce(N)]]
```

where, the integer argument `N` specifies the nested loop levels you want the compiler to attempt to coalesce.

For example, consider the following set of nested loops:

```
for (A)
  for (B)
    for (C)
      for (D)
    for (E)
```

If you place the `loop_coalesce` attribute before loop (A), then the loop nesting level for these loops is defined as:

- Loop (A) has a loop nesting level of 1.
- Loop (B) has a loop nesting level of 2.
- Loop (C) has a loop nesting level of 3.
- Loop (D) has a loop nesting level of 4.
- Loop (E) has a loop nesting level of 3.

Depending on the loop nesting level that you specify, the compiler attempts to coalesce loops differently:

- If you specify `[[intel::loop_coalesce(1)]]` on loop (A), the compiler does not attempt to coalesce any of the nested loops.
- If you specify `[[intel::loop_coalesce(2)]]` on loop (A), the compiler attempts to coalesce loops (A) and (B).
- If you specify `[[intel::loop_coalesce(3)]]` on loop (A), the compiler attempts to coalesce loops (A), (B), (C), and (E).
- If you specify `[[intel::loop_coalesce(4)]]` on loop (A), the compiler attempts to coalesce all of the loops [loop (A) - loop (E)].

Coalescing nested loops also reduce the latency of the component, which could further reduce your kernel area usage. However, in some cases, coalescing loops might lengthen the critical loop initiation interval path, so coalescing loops might not be suitable for all kernels.

For `parallel_for` kernels, the compiler automatically attempts to coalesce loops even if they are not annotated by the `[[intel::loop_coalesce(N)]]` attribute. Coalescing loops in `parallel_for` kernels usually improves throughput as well as reducing kernel area use. You can use the `[[intel::loop_coalesce(N)]]` attribute to prevent the automatic coalescing of loops in `parallel_for` kernels.

NOTE

If you specify `[[intel::loop_coalesce(1)]]` for a loop in an `parallel_for` kernel, you prevent automatic loop coalescing for that loop.

Example 28. Example

The following simple example shows how the compiler coalesces two loops into a single loop.
Consider a simple nested loop written as follows:

```
[[intel::loop_coalesce(2)]]
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

The compiler coalesces the two loops together so that they run as if they were a single loop written as follows:

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if (j == M){
    j = 0;
    i++;
  }
}
```

NOTE

For additional information, refer to the FPGA tutorial sample `loop_coalesce` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

4.6.5 `max_concurrency` Attribute

Use the `max_concurrency` attribute to limit the concurrency of a loop in your kernel. The concurrency of a loop is how many iterations of that loop can be in progress at one time. By default, the Intel® oneAPI DPC++/C++ Compiler tries to maximize the concurrency of loops so that your kernel runs at peak throughput.

Syntax

```
[[intel::max_concurrency(n)]]
```

The `max_concurrency` attribute applies to pipelined loops in single task kernels. Refer to Pipelining on page 12 for information about loop pipelining.

The `max_concurrency` attribute enables you to control the on-chip memory resources required to pipeline your loop. To achieve simultaneous execution of loop iterations, the Intel® oneAPI DPC++/C++ Compiler must create copies of any memory that is private to a single iteration. These copies are called private copies. The greater the permitted concurrency, the more private copies the compiler must create.

The attribute parameter `n` is required and must be a non-negative constant expression of integer type. The parameter directs the compiler to restrict the loop's concurrency to n simultaneous iterations.

The kernel's `report.html` (Review the `report.html` File on page 42) provides the following information pertaining to loop concurrency:

- **Maximum concurrency that the Intel® oneAPI DPC++/C++ Compiler has chosen:** This information is available in the Loop Analysis report and Kernel Memory Viewer.
  - In the Loops Analysis report, a message in the **Details** pane reports as the maximum number of simultaneous executions has been limited to `n`.

    NOTE

    The value of unsigned `N` can be greater than or equal to zero. A value of `N = 0` indicates unlimited concurrency.

  - In the Memory Viewer, the bank view of your local memory graphically shows the number of private copies.
- **Impact to memory usage**: This information is available in the Area Analysis of System report. A message in the Details pane reports that the Intel® oneAPI DPC++/C++ Compiler has created `N` independent copies of the memory to enable simultaneous execution of `N` loop iterations.

  If you want to exchange some performance for physical memory savings, apply `[[intel::max_concurrency(n)]]` to the loop, as shown in the following code snippet:

```
[[intel::max_concurrency(1)]]
for (int i = 0; i < N; i++) {
  int arr[M];
  // Doing work on arr
}
```

  When you apply this attribute, the Intel® oneAPI DPC++/C++ Compiler limits the number of simultaneously-executed loop iterations to `n`. The number of private copies of loop-scooped memories is also restricted to `n`.

  You can also control the number of private copies (created for a local memory and accessed within a loop) by using `[[intel::private_copies(N)]]`. If a local memory with `[[intel::private_copies(N)]]` is accessed with a loop that has `[[intel::max_concurrency(M)]]` attribute, the Intel® oneAPI DPC++/C++ Compiler limits the number of simultaneously-executed loop iterations to `min(M,N)`. For more information about `[[intel::private_copies(N)]]`, refer to FPGA Memory Attributes on page

222. For additional information about using `[[intel::private_copies(N)]]`, refer to the FPGA tutorial sample "Private Copies" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

4.6.6 `max_interleaving` Attribute

Use the `max_interleaving` attribute to maximize the throughput and hardware resource occupancy of pipelined inner loops in a loop nest by issuing new inner loop iterations as frequently as possible (minimizing the loop initiation interval). When the compiler cannot achieve a loop II of 1 for an inner loop, the compiler configures the loop nest to interleave iterations of one invocation of the inner loop with iterations of other invocations of the inner loop.

Syntax

```
[[intel::max_interleaving(n)]]
```

The Intel® oneAPI DPC++/C++ Compiler restricts the annotated (inner) loop to be invoked at most `n` times per outer loop iteration. When this attribute is specified with `n=0`, the compiler allows the pipeline to contain a number of simultaneous invocations of the annotated loop equal to the loop initiation interval (II) of that loop. For example, an annotated inner loop with an II of 2 can have iterations from two invocations in the pipeline at a time. This behavior is the default behavior for the compiler if you do not specify the `max_interleaving` attribute.

As an example, consider the loop nest in the following code snippet:

```
// Loop j is pipelined with ii=1
for (int j = 0; j < M; j++) {
  int a[N];
  // Loop i is pipelined with ii=2
  for (int i = 1; i < N; i++) {
    a[i] = foo(i)
  }
}
```

In this example, the inner `i` loop is pipelined with a loop II of 2. Under normal pipelining, this means that the inner loop hardware only achieves 50% utilization since one `i` iteration is initiated every other cycle. To take advantage of these idle cycles, the compiler interleaves a second invocation of the `i` loop from the next iteration of the outer `j` loop. Here, a loop invocation means to start pipelined execution of a loop body. In this example, since the `i` loop resides inside the `j` loop, and the `j` loop has a trip count of `M`, the `i` loop is invoked `M` times. Since the `j` loop is an outermost loop, it is invoked once. The following table illustrates the difference between normal pipelined execution of the `i` loop and interleaved execution for this example where `N=5`:

Table 30. Difference Between Normal Pipelined Execution and Interleaved Execution

Cycle	Pipelined	Interleaved
0	(0,0)	(0,0)
1	---	(1,0)
2	(0,1)	(0,1)

continued...

Cycle	Pipelined	Interleaved
3	---	(1,1)
4	(0,2)	(0,2)
5	---	(1,2)
6	(0,3)	(0,3)
7	---	(1,3)
8	(0,4)	(0,4)
9	---	(1,4)
10	(1,0)	(2,0)
11	---	(3,0)
12	(1,1)	(2,1)
13	---	(3,1)
14	(1,2)	(2,2)
15	---	(3,2)
16	(1,3)	(2,3)
17	---	(3,3)
18	(1,4)	(2,4)
19	---	(3,4)

The table shows the values (`j,i`) for each inner loop iteration that is initiated at each cycle. At cycle 0, both modes of execution initiate the (0,0)th iteration of the `i` loop. Under normal pipelined execution, no `i` loop iteration is initiated at cycle 1. Under interleaved execution, the (1,0)th iteration of the innermost loop, that is, the first iteration of the next (`j=1`) invocation of the `i` loop is initiated. By cycle 10, interleaved execution has initiated all of the iterations of both the `j=0` invocation of the `i` loop and the `j=1` invocation of the `i` loop. This represents twice the efficiency of the normal pipelined execution.

In some cases, you may decide that the performance benefit from interleaving is not equal to the area cost associated with enabling interleaving. In these cases, you may want to limit or restrict the amount of interleaving to reduce FPGA area utilization. To limit the number of interleaved invocations of an inner loop that can be executed simultaneously, annotate the inner loop with the `[[intel::max_interleaving(n)]]` attribute. The annotated loop must be contained inside another pipelined loop. The required parameter (`n`) specifies an upper bound on the degree of interleaving allowed, that is, how many invocations of the containing loop can execute the annotated loop at a given time.

Specify the `[[intel::max_interleaving(n)]]` attribute in one of the following ways:

- `[[intel::max_interleaving(1)]]`

  The compiler restricts the annotated (inner) loop to be invoked only once per outer loop iteration. That is, all iterations of the inner loop travel the pipeline before the next invocation of the inner loop can occur.
- `[[intel::max_interleaving(0)]]`

The compiler allows the pipeline to contain a number of simultaneous invocations of the inner loop equal to the loop initiation interval (II) of the inner loop. For example, an inner loop with an II of 2 can have iterations from two invocations in the pipeline at a time. This behavior is the default behavior for the compiler if you do not specify the `[[intel::max_interleaving(n)]]` attribute.

Example

In the following code snippet, the compiler restricts the pipelined execution of the `i` loop. A new invocation of the `i` loop corresponds only to the subsequent iteration of the `j` loop.

```
// Loop j is pipelined with ii=1
for (int j = 0; j < M; j++) {
  int a[N];
  // Loop i is pipelined with ii=2
  [[intel::max_interleaving(1)]]
  for (int i = 1; i < N; i++) {
    a[i] = foo(i)
  }
  …
}
```

NOTE

For additional information, refer to the FPGA tutorial sample `max_interleaving` listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

4.6.7 `speculated_iterations` Attribute

Use the `speculated_iterations` attribute to direct the Intel® oneAPI DPC++/C++ Compiler to improve the performance of pipelined loops. The `speculated_iterations` attribute is applied to loops and hence, it must appear directly before the loop (the same place as other loop attributes). For more information, refer to Optimize Loops With Loop Speculation on page 92.

Syntax

```
[[intel::speculated_iterations(N)]]
```

Where, the integer argument *N* specifies the permissible number of iterations to speculate.

The Intel® oneAPI DPC++/C++ Compiler generates hardware to run *N* extra iterations of the loop while ensuring the extra iterations do not affect anything. This allows either reducing the II of the loop or increasing the f_{max}. The deciding factor is how quickly the exit condition of the loop is calculated. If the calculation takes many cycles, it is better to have larger `speculated_iterations`.

NOTE

Extra iterations increase the time before the next invocation of the loop can begin. This may be a factor if the actual number of iterations of the loop is very small (less than 5 to 10 or similar). In this case, specify the *N* value as 0 to allow subsequent loop iterations to start immediately but at the cost of a larger II to allow more time to evaluate the exit condition. Refer to the Loop Analysis report to identify whether the exit condition is a bottleneck for II.

Example 29. Example

```
[[intel::speculated_iterations(1)]]
  while (m*m*m < N) {
    m += 1;
  }
  dst[0] = m;
```

The loop in this example will have one speculated iteration.

NOTE

For additional information, refer to the FPGA tutorial sample "Speculated Iterations" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample in GitHub.

4.6.8 `unroll` Pragma

Loop unrolling involves replicating a loop body multiple times and reducing the trip count of a loop. Unroll loops to reduce or eliminate loop control overhead on the FPGA. In cases where there are no loop-carried dependencies and the Intel® oneAPI DPC++/C++ Compiler can perform loop iterations in parallel, unrolling loops can also reduce latency and overhead.

IMPORTANT

Unrolling of nested loops with large bounds might generate huge number of instructions that could lead to very long compile times.

The compiler might unroll simple loops even if a pragma does not annotate them. To direct the compiler to unroll a loop, or to explicitly not unroll a loop, insert an `unroll` kernel pragma in the kernel code preceding a loop you want to unroll. To specify an unroll factor `N`, use the optional unroll factor specifier `#pragma unroll <N>`. For more information, see *Determining the Correct Unroll Factor* section in Unrolling Loops FPGA tutorial.

Syntax

```
#pragma unroll

#pragma unroll N
```

If you specify the unroll factor `N`, the factor must be a positive constant expression of integer type. If you omit the unroll factor `N`, the loop is unrolled fully.

Examples

The following is an example of full loop unrolling:

```
// Before unrolling loop
#pragma unroll
for(i = 0 ; i < 5; i++){
  a[i] += 1;
}
```

```
// After fully unrolling the loop by a factor of 5,
// the loop is flattened. There is no loop after unrolling.
a[0] += 1;
a[1] += 1;
a[2] += 1;
a[3] += 1;
a[4] += 1;
```

You can observe that a full unroll is a special case where the unroll factor is equal to the number of loop iterations.

The following is an example of partial loop unrolling:

```
// Before unrolling loop
#pragma unroll 4
for(i = 0 ; i < 20; i++){
  a[i] += 1;
}
```

```
// After the loop is unrolled by a factor of 4,
// the loop has five (20 / 4) iterations.
for(i = 0 ; i < 5; i++){
  a[i * 4] += 1;
  a[i * 4 + 1] += 1;
  a[i * 4 + 2] += 1;
  a[i * 4 + 3] += 1;
}
```

In the partial unroll example, each loop iteration in the unrolled loop is equivalent to four iterations. The Intel® oneAPI DPC++/C++ Compiler instantiates four adders instead of one adder. Because there is no data dependency between iterations in the loop (which is true in this case), the compiler executes four adds in parallel.

TIP

For additional information, refer to the FPGA tutorial sample "Loop Unroll" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on GitHub.

Notes

- Provide an `unroll` factor whenever possible. To specify an unroll factor `N`, insert the `#pragma unroll <N>` directive before a loop in your kernel code. The Intel® oneAPI DPC++/C++ Compiler attempts to unroll the loop at most `<N>` times. Consider the following code fragment. By assigning a value of 2 as the unroll factor, you direct the compiler to unroll the loop twice.

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

  For more information, see *Determining the Correct Unroll Factor* in Unrolling Loops FPGA tutorial.
- To unroll a loop fully, you may omit the unroll factor by simply inserting the `#pragma unroll` directive before a loop in your kernel code. The compiler attempts to unroll the loop fully if it understands the trip count and issues a warning if it cannot execute the unroll request.

4.6.9 Loop Fuse Functions and `nofusion` Attribute

This topic describes the loop fuse functions and `nofusion` attribute that the Intel® oneAPI DPC++/C++ Compiler supports.

Loop Fuse Functions

The loop fuse functions are declared in the `sycl/ext/intel/fpga_loop_fuse.hpp` header file, which is invoked by the `sycl/ext/intel/fpga_extensions.hpp` header file. Apply these loop functions to a block of code to indicate to the compiler that it must fuse adjacent loops in the code block overriding the compiler profitability or safety analysis of the fusion. Fusing adjacent loops reduces the amount of loop control overhead in your kernel that reduces the FPGA area used and increases the performance by executing both loops as one (fused) loop. For additional information, see Fuse Loops to Reduce Overhead and Improve Performance on page 91

The compiler supports the following loop fuse functions:

- **`sycl::ext::intel::fpga_loop_fuse<v>(f)`**: Directs the compiler to fuse loops within the function `f` and up to a depth of `v >= 1` without affecting the functionality of either loop, overriding the compiler profitability analysis of fusing the loops. By default, `v = 1`, which is equivalent to indicating that the compiler should consider only the adjacent top-level loops for fusing. For example:

```
[=]() { //Kernel
  sycl::ext::intel::fpga_loop_fuse<1>([&] {
    L1: for(...) {}
    L2: for(...) {
      L3: for(...) {}
      L4: for(...) {
        L5: for(...) {}
        L6: for(...) {}
      }
    }
  });
}
```

  By default (`v = 1`), only loops `L1` and `L2` are initially considered for fusing. At a depth of `v = 2`, the compiler considers `L1-L2` and `L3-L4` loop pairs for fusing.

The compiler automatically considers fusing adjacent loops with equal trip counts when the loops meet the Automatic Loop Fusion criteria or when fusion is deemed safe and profitable. You can use the `sycl::ext::intel::fpga_loop_fuse<v>(f)` function to inform the compiler to consider fusing adjacent loops with different trip counts, which is considered to be unprofitable by default.. With the loop fuse function applied to a block of code, the compiler always attempts to fuse adjacent loops (with equal or different trip counts) in the block whenever the compiler determines that it is safe to fuse the loops. Two loops are considered safe to merge if they meet the Fusion Criteria. The following example shows the effects of fusing loops with unequal trip counts:

Unfused Loops

```
[=]() { //Kernel
  sycl::ext::intel::fpga_loop_fuse([&] {
    for (int i = 0; i < N; i++) {
      // Loop Body 1
    }
    for (int j = 0; j < M; j++) {
      // Loop Body 2
    }
  });
}
```

Fused Loops

```
for (int f = 0; f < max(M,N); f++) {
  if (f < N) {
    // Loop Body 1
  }
  if (f < M) {
    // Loop Body 2
  }
}
```

A fused loop can itself be considered for fusing with other loops. For example, in the following code, L1 and L2 are initially considered for fusing. That resulting fused loop can then be considered for fusing with L4.

```
[=]() { //Kernel
  sycl::ext::intel::fpga_loop_fuse([&] {
    L1: for(...) {}
    L2: for(...) {
      L3: for(...) {}
    }
    L4: for(...) {
    }
  });
}
```

- **`sycl::ext::intel::fpga_loop_fuse_independent<v>(f)`**: Directs the compiler to fuse loops within the function `f` up to a depth `v >= 1` while overriding fusion-safety checks. Here, `v = 1` by default. When you use this function, you are guaranteeing to the compiler that fusing pairs of loops affected by the loop fuse function is safe. That is, there are no negative distance dependencies between the pairs of loops. If it is not safe, you might get functional errors in your kernel. Besides this difference, the `sycl::ext::intel::fpga_loop_fuse<v>(f)` and `sycl::ext::intel::fpga_loop_fuse_independent<v>(f)` functions behave identically.

Function Calls in Loop Fuse Code Blocks

If a function call occurs in a code block annotated with a loop fuse function and inlining that function call contains a loop, the resulting loop can be a candidate for loop fusion.

Nested Loop Fusion Functions

When you nest loop fusion functions, you might create overlapping sets of candidate loops. Consider the following example:

```
[=]() { //Kernel
  sycl::ext::intel::fpga_loop_fuse_independent<2>([&] {
    L1: for(...) {}
    L2: for(...) {
      sycl::ext::intel::fpga_loop_fuse<2>([&] {
        L3: for(...) {}
        L4: for(...) {
          L5: for(...) {}
          L6: for(...) {}
        }
      });
    }
  });
}
```

In this example, the compiler considers the following loop pairs for fusion: `L1-L2`, `L3-L4`, and `L5-L6`. In addition, the compiler overrides the compiler negative distance dependency analysis of `L1-L2` and `L3-L4` loop pairs.

`nofusion` Attribute

You can exempt a loop from being fused with an adjacent loop by annotating the loop with the `nofusion` loop attribute. This attribute prevents the annotated loop from being automatically or fused when it is subject to the loop fusion functions.

Syntax

```
[[intel::nofusion]]
```

Example 30. Example

For example, the following code samples have the same effect. If one loop in a pair is annotated with the `nofusion` attribute, the other loop has no other loop to fuse with.

```
[[intel::nofusion]]
L1: for (int j = 0; j < N; ++j){
  data[j] += Q;
}
L2: for (int i = 0; i < N; ++1) {
  output[i] = Q * data[i];
}
```

```
L1: for (int j = 0; j < N; ++j){
  data[j] += Q;
}
[[intel::nofusion]]
L2: for (int i = 0; i < N; ++1) {
  output[i] = Q * data[i];
}
```

In the following example, the compiler does not apply the loop fusion transformation to the loops:

```
for (int x = 0; x < N; x++) { // loop 1
  arr1_acc[x] = x;
}

[[intel::nofusion]]
for (int x = 0; x < N; x++) { // loop 2
  arr2_acc[x] = x;
}
```

Because you have applied the `nofusion` attribute to loop 2, the compiler cannot fuse loop 1 with loop 2.

Related Links

Fuse Loops to Reduce Overhead and Improve Performance on page 91

Fusing Adjacent Loops With Unequal Trip Counts (`-Xsenable-unequal-tc-fusion`) on page 170

4.7 Floating Point Pragmas

Use `fp contract` and `fp reassociate` pragmas to influence the intermediate rounding and conversions of floating-point operations and the ordering of arithmetic operations in your kernel at finer granularity than the Intel® oneAPI DPC++/C++ Compiler flags.

NOTICE

Starting with the oneAPI 2021.2 release, fast math is enabled by default, allowing the Intel® oneAPI DPC++/C++ Compiler to make various out-of-box floating point math (`float` or `double`) optimizations. With these optimizations enabled, you might observe different bitwise results when compared to results from the oneAPI 2021.1 release or from GCC. The tradeoff is done to improve performance and area of your design. Automatic dot product inference and floating-point contraction for double precision math are two key noticeable FPGA optimizations that save a large amount of FPGA area and improve performance/latency. To return to the same precision as the oneAPI 2021.1 release or GCC, use the following compiler options:

- For Linux: `-no-fma -fp-model=precise`
- For Windows: `/Qfma- /fp:precise`

For more information about these options, refer to `-fp-model, fp` and `fma, Qfma` topics in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

NOTE

You can place `fp contract` and `fp reassociate` pragma statements inside any compound statement (brace-enclosed sequences of statements), outside of all functions (file scope), or at the start of a function within the curly braces. For example:

```
{
  #pragma clang fp reassociate(on)
  float temp1 = 0.0f, temp2 = 0.0f;
  temp1 = accessorA[0] + accessorB[0];
  temp2 = accessorC[0] + accessorD[0];
  accessorRES[0] += temp1 * temp2;
}
```

`fp contract` Pragma

The `fp contract` pragma controls whether the compiler can skip intermediate rounding and conversions mainly between double precision arithmetic operations. If multiple occurrences of this pragma affect the same scope of your code, the pragma with the narrowest scope takes precedence.

Syntax

```
#pragma clang fp contract(fast|off|on)
```

where:

State	Description
`fast`	Allows the fusing of multiply and add instructions into an FMA, but it might violate the language standard. With fast math enabled by default, the `fp contract` pragma is set to `fast` by default. This allows the compiler to skip intermediate rounding and conversions mainly between double-precision arithmetic operations.
`off`	Prohibits the compiler from fusing multiple floating-point operations.
`on`	Fuses floating-point operations (multiply and add) within the same statement to form FMAs.

`fp reassociate` Pragma

The `fp reassociate` pragma controls the relaxing of the order of floating-point arithmetic operations within the code block that this pragma is applied to. With reordering, the compiler can optimize the hardware structure, which improves the performance of your kernel. If multiple occurrences of this pragma affect the same scope of your code, the pragma with the narrowest scope takes precedence.

Syntax

```
#pragma clang fp reassociate(on|off)
```

Where:

State	Description
`on`	Enables the compiler to reorder floating-point operations to improve performance and on-chip area. With fast math enabled by default, the `fp reassociate` pragma is set to on by default.
`off`	Prohibits the compiler from reordering floating-point operations to improve performance and on-chip area.

4.8 Latency Controls (Beta)

The Intel® oneAPI DPC++/C++ Compiler allows you to set latency constraints between operations with side effects, such as pipes and LSUs, which are visible outside the kernel. Specifically, you can apply latency controls to pipe read/write and LSU load/store.

For stallable operations, the scheduler considers only the inherent latency of the operation without making any assumption about the actual stall time. The compiler strives to achieve the latency constraints. If it cannot achieve the latency controls, the compiler errors out.

NOTE

Latency controls is a Beta feature currently. In a future release, its API will change.

Syntax

While latency controls is a Beta feature, you must declare the side effects (pipes and LSUs) that latency controls apply to in the `sycl::ext::intel::experimental` namespace. For example:

```
#include <sycl/ext/intel/fpga_extensions.hpp>
using Pipe = sycl::ext::intel::experimental::pipe<class PipeClass, int, 8>;
```

You must set a latency constraint between an anchor and a non-anchor side-effect operation. You can specify an anchor and a constraint on a side-effect operation with the following two template arguments, which are also in the `sycl::ext::intel::experimental` namespace:

Template argument	Description	Example
`sycl::ext::intel::experimental::latency_anchor_id<N>`	Specifies the ID of the current side-effect operation where it behaves as an anchor. *N* is an integer and its default value is -1.	`// This pipe read() performs as anchor 0 in latency control.` `Pipe::read<ext::intel::experimental::latency_anchor_id<0>>();`

`sycl::ext::intel::experimental::latency_constraint<A, B, C>`	Specifies the latency constraint when the current side-effect operation behaves as a non-anchor, where:
	• *A* is an integer that specifies the ID of the target anchor defined on a different operation through the `latency_anchor_id` argument.
	• *B* is an `enum` value that specifies one of the control type from the set `{type::exact, type::max, type::min}`.
	• *C* is an integer that specifies the relative clock-cycle difference between the target anchor and the current side-effect operation that the constraint must infer subject to the control type (`exact`, `max`, or `min`). This relative cycle can be both positive and negative. A positive relative cycle means the anchor occurs before the current side effect.
		`// Set a latency constraint between anchor 0 and this pipe write().`
		`// This pipe write() starts exactly 2 cycles after anchor 0 is done.`
		`Pipe::write<ext::intel::experimental::latency_constraint<0, ext::intel::experimental::type::exact, 2>>(...);`

NOTES

- You can specify two template arguments in an arbitrary order. You need not always specify both template arguments together.
- If you do not specify either of the template arguments, the compiler does not apply latency control.

Example 31. Example

The following is an example of applying latency controls between side-effect operations:

```
#include <sycl/ext/intel/fpga_extensions.hpp>
...
using namespace sycl;
using Pipe1 = ext::intel::experimental::pipe<class PipeClass1, int, 8>;
using Pipe2 = ext::intel::experimental::pipe<class PipeClass2, int, 8>;
using BurstCoalescedLSU = ext::intel::experimental::lsu<
                          ext::intel::experimental::burst_coalesce<false>,
                          ext::intel::experimental::statically_coalesce<false>>;
...
// Set read() as anchor 0.
Pipe1::read<ext::intel::experimental::latency_anchor_id<0>>();

// write() starts exactly 2 cycles after anchor 0 read() is done.
// Set write() as anchor 1.
Pipe2::write<ext::intel::experimental::latency_constraint<
             0, ext::intel::experimental::type::exact, 2>,
             ext::intel::experimental::latency_anchor_id<1>>(...);

// store() starts at most 5 cycles after anchor 1 write() is done.
BurstCoalescedLSU::store<ext::intel::experimental::latency_constraint<
                         1, ext::intel::experimental::type::max, 5>>(...);
```

Rules and Limitations

- Anchor ID must be a non-negative number.
- Anchor ID must be a unique number within the whole design.
- Two endpoints of a constraint must meet one of the following conditions:
  - Both endpoints are not in any cluster.
  - Both endpoints are in the same cluster.

Appendix A Quick Reference

This section provides a quick reference list of all FPGA-specific attributes, pragmas, and variables.

A.1 FPGA Optimization Flags

The following table summarizes FPGA optimization flags:

Table 31. FPGA Optimization Flags

Flags	Description	Example
`-Xsclock=<clock target in Hz/KHz/MHz/GHz or s/ms/us/ns/ps>`	Schedules f_{MAX} target for kernels.	`dpcpp -fintelfpga -Xshardware -Xsclock=<clock target> <source_file>.cpp`
`-Xsno-interleaving=<global_memory_type>`	Disables burst-interleaving for all global memory banks of the same type and manages them manually	`dpcpp -fintelfpga -Xshardware <source_file>.cpp -Xsno-interleaving=DDR`
`-Xsglobal-ring`	Forces ring interconnect for global memory.	`dpcpp -fintelfpga -Xshardware -Xsglobal-ring <source_file>.cpp`
`-Xsforce-single-store-ring`	Narrows the interconnect to save area while limiting write-only throughput to one bank's worth.	`dpcpp -fintelfpga -Xshardware -Xsforce-single-store-ring <source_file>.cpp`
`-Xsnum-reorder`	Narrows the interconnect to save area while reducing read-only throughput.	`dpcpp -fintelfpga -Xshardware -Xsnum-reorder=1 <source_file>.cpp`

continued...

Flags	Description	Example
`-Xsno-hardware-kernel-invocation-queue`	Reduces kernel area use by removing kernel invocation queue in SYCL* kernel.	`dpcpp -fintelfpga -Xshardware -Xsno-hardware-kernel-invocation-queue <source_file>.cpp`
`-Xshyper-optimized-handshaking=<auto|off>`	Modifies the handshaking protocol used in certain areas of the design	`dpcpp -fintelfpga -Xshardware -Xshyper-optimized-handshaking=auto <source_file>.cpp` `dpcpp -fintelfpga -Xshardware -Xshyper-optimized-handshaking=off <source_file>.cpp`
`-Xsdisable-auto-loop-fusion`	Disables the automatic fusion of loops when compiling the design.	`dpcpp -fintelfpga -Xshardware -Xsdisable-auto-loop-fusion <source_file>.cpp`
`-Xsenable-unequal-tc-fusion`	Fuses adjacent loops with unequal trip counts into a single loop without affecting either loop's functionality.	`dpcpp -fintelfpga -Xshardware -Xsenable-unequal-tc-fusion <source_file>.cpp`
`-Xsauto-pipeline`	Pipelines loops in non-task (`parallel_for`) kernels.	`dpcpp -fintelfpga -Xshardware -Xsauto-pipeline <source_file>.cpp`
`-fp-model=<value>`	Controls the semantics of floating-point operations.	`dpcpp -fintelfpga -Xshardware -fp-model=<value> <source_file>.cpp`
`-Xsrounding=<rounding_type>`	Modifies the rounding mode of floating-point elementary operations in your design.	`dpcpp -fintelfpga -Xshardware -Xsrounding=ieee <source_file>.cpp` `dpcpp -fintelfpga -Xshardware -Xsrounding=faithful <source_file>.cpp`
`-Xssfc-exit-fifo-type=<default|zero-latency|low-latency>`	Globally controls exit FIFO latency of stall-free clusters using the specified exit FIFO type.	`dpcpp -fintelfpga -Xshardware -Xssfc-exit-fifo-type=zero-latency <source_file>.cpp`
`-Xsread-only-cache-size=<N>`	Enables read-only cache and sets its size to `<N>` bytes.	`dpcpp -fintelfpga -Xshardware -Xsread-only-cache-size=<N> <source_file>.cpp`
`-Xsdsp-mode=[default|prefer-dsp|prefer-softlogic]`	Controls the hardware implementation of the supported data types and math functions of all kernels in your source code.	`dpcpp -fintelfpga -Xshardware -Xsdsp-mode=<option> <source_file>.cpp`

A.2 FPGA Loop Directives

The following table summarizes loop directives:

Table 32. FPGA Loop Directives

Directive (Pragma, Attribute, or Function)	Description	Example

`disable_loop_pipelining`	Directs the Intel® oneAPI DPC++/C++ Compiler to disable pipelining of a loop.	`[[intel::disable_loop_pipelining]]`
		`for (int i = 1; i < N; i++) {`
		`  int j = a[i-1];`
		`  // Memory dependency induces a high-latency loop`

continued...

Directive (Pragma, Attribute, or Function)	Description	Example
		feedback path a[i] = foo(j) }

initiation_interval	Forces a loop to have a loop initialization interval (II) of a specified value.	// ii set to 5
		[[intel::initiation_interval(5)]]
		  for (int i = 0; i < N; ++i){
		}

ivdep	Ignores memory dependencies between iterations of this loop	// ivdep loop
		[[intel::ivdep]] for (…) {}
		
		//ivdep safelen
		[[intel::ivdep(safelen)]] for (;;) {}
		
		// ivdep accessor
		[[intel::ivdep(accessorA)]] for (;;) {}
		
		//ivdep array safelen
		[[intel::ivdep(accessorA, safelen)]]
		for (;;){}

loop_coalesce	Coalesces nested loops into a single loop without affecting the loop functionality.	[[intel::loop_coalesce(2)]]
		for (int i = 0; i < N; i++)
		  for (int j = 0; j < M; j++)
		  sum[i][j] += i+j;

max_concurrency	Limits the number of iterations of a loop that can simultaneously execute at any time.	//max concurrency set to 1
		[[intel::max_concurrency(1)]]
		  for (int i = 0; i < c; ++i){
		}

max_interleaving	Maximizes the throughput and hardware resource occupancy of pipelined inner loops in a loop nest.	// Loop j is pipelined with ii=1
		for (int j = 0; j < M; j++) {
		  int a[N];
		  // Loop i is pipelined with ii=2
		  [[intel::max_interleaving(1)]]
		  for (int i = 1; i < N; i++) {
		  a[i] = foo(i)
		  }
		  …
		}

speculated_iterations	Improves the performance of pipelined loops.	[[intel::speculated_iterations(1)]]
		  while (m*m*m < N) {
		  m += 1;
		  }
		  dst[0] = m;

unroll	Unrolls a loop in the kernel code.	// unroll factor N set to 2
		#pragma unroll 2
		for(size_t k = 0; k < 4; k++){
		  mac += data_in[(gid * 4) + k] * coeff[k];
		}

continued...

Directive (Pragma, Attribute, or Function)	Description	Example

`nofusion`	Prevents the compiler from fusing the annotated loop with any of the adjacent loops.	`for (int x = 0; x < N; x++) { a1_acc[x] = x; }`
		`[[intel::nofusion]]`
		`for (int x = 0; x < N; x++) { a2_acc[x] = x; }`

`sycl::ext::intel::fpga_loop_fuse<v>(f)`	Fuses loops within the function `f` up to a depth of `v >= 1`, where `v = 1` by default.	`[=]() [[intel::kernel_args_restrict]] { sycl::ext::intel::fpga_loop_fuse<v>{ for (int x = 0; x < N; x++) { for (int y = 0; y < N; y++) { for (int z = 0; z < N; z++) { a1_acc[x][y][z] = 0; } } } for (int x = 0; x < N + 1; x++) { for (int y = 0; y < N + 1; y++) { for (int z = 0; z < N + 1; z++) { a2_acc[x][y][z] = 0; } } } } }`
`sycl::ext::intel::fpga_loop_fuse_independent<v>(f)`	Fuses loops within the function `f` up to a depth `v >= 1` while overriding fusion-safety checks. Here, `v = 1` by default.	`[=]() { //Kernel sycl::ext::intel::fpga_loop_fuse_independent([&] { for(int x = 0; x < N; x++){ a3_acc[x] = x; } for(int x = 0; x < N + 1; x++){ a4_acc[x] = x; } }); }`

A.3 Floating Point Pragmas

The following table summarizes the floating point pragmas:

Pragma	Description	Example
`fp contract(off|fast)`	Controls whether the compiler can skip intermediate rounding and conversions mainly between double precision arithmetic operations.	`{ #pragma clang fp contract(fast) float temp1 = 0.0f, temp2 = 0.0f; temp1 = accessorA[0] + accessorB[0]; temp2 = accessorC[0] + accessorD[0]; accessorRES[0] += temp1 * temp2; }`
`fp reassociate(on|off)`	Controls the relaxing of the order of floating-point arithmetic operations within the code block that this pragma is applied to.	`{ #pragma clang fp reassociate(on) float temp1 = 0.0f, temp2 = 0.0f; temp1 = accessorA[0] + accessorB[0]; temp2 = accessorC[0] + accessorD[0]; accessorRES[0] += temp1 * temp2; }`

A.4 FPGA Memory Attributes

The following table summarizes memory attributes:

Table 33. FPGA Memory Attributes

Attribute	Description	Example

`bank_bits`	Specifies that the local memory addresses should use bits for bank selection.	`// Array is implemented with 4 banks where`
		`// bits 6 and 5 of the memory word address`
		`// are used to select between the banks`
		`[[intel::bank_bits(6,5)]] int array[128];`

`bankwidth`	Specifies that the memory implementing the variable or array must have memory banks of a defined width.	`// Each  memory bank  is 8 bytes (64-bits) wide` `[[intel::bankwidth(8)]] int array[128];`

`doublepump`	Specifies that the memory implementing the variable, or an array must be clocked at twice the rate as the kernel accessing it.	`// Array is implemented in a memory that operates at`
		`// twice the clock frequency of the kernel`
		`[[intel::doublepump, bankwidth(128)]] int array[128];`

`force_pow2_depth`	Specifies that the memory implementing the variable or array has a power-of-2 depth.	`// array1 is implemented in a memory with depth 1536` `[[intel::force_pow2_depth(0)]] int array1[1536];`

`max_replicates`	Specifies that the memory implementing the variable, or an array has no more than the specified number of replicates to enable simultaneous accesses from the datapath.	`// Array is implemented in a memory with maximum four`
		`// replicates`
		`[[intel::max_replicates(4)]] int array[128];`

`fpga_memory`	Forces a variable or an array to be implemented as an embedded memory.	`// Array is implemented in memory (MLAB/M20K),`
		`// the actual implementation is automatically decided`
		`// by the compiler`
		`[[intel::fpga_memory]] int array1[128];`
		
		`// Array is implemented in M20K`
		`[[intel::fpga_memory("BLOCK_RAM")]] int array2[64];`
		
		`// Array is implemented in MLAB`
		`[[intel::fpga_memory("MLAB")]] int array3[64];`

`merge`	Allows merging of two or more variables or arrays defined in the same scope with respect to width or depth.	`// Both arrays are merged width-wise and implemented`
		`// in the same memory system`
		`[[intel::merge("mem", "width")]] short arrayA[128];`
		`[[intel::merge("mem", "width")]] short arrayB[128];`

`numbanks`	Specifies that the memory implementing the variable or array must have a defined number of memory banks.	`// Array is implemented with 2 banks` `[[intel::numbanks(2)]] int array[128];`

`private_copies`	Specifies that the memory implementing the variable, or an array has no more than the specified number of independent	`// Array is implemented in a memory with two`
		`// private copies`
		`[[intel::private_copies(2)]] int array[128];`

continued...

Attribute	Description	Example
	copies to enable concurrent thread or loop iteration accesses.	
`fpga_register`	Forces a variable or an array to be carried through the pipeline in registers.	`// Array is implemented in register` `[[intel::fpga_register]] int array[128];`

`simple_dual_port`	Specifies that the memory implementing the variable or array should have no port that serves both reads and writes.	`// Array is implemented in a memory such that no`
		`// single port serves both a read and a write`
		`[[intel::simple_dual_port]] int array[128];`

`singlepump`	Specifies that the memory implementing the variable or array must be clocked at the same rate as the kernel accessing it.	`// Array is implemented in a memory that operates`
		`// at the same clock frequency as the kernel`
		`[[intel::singlepump]] int array[128];`

A.5 FPGA Local Memory Function

The following table summarizes the local memory function:

Table 34. Local Memory Function

Function	Description
`template <typename T, typename Group> multi_ptr<T, Group::address_space> group_local_memory_for_overwrite(Group g)`	Constructs an object of type `T` in an address space accessible by all work items in the workgroup `g`, using the default initialization. The object is initialized upon or before the first call to the `group_local_memory_for_overwrite` function. The storage for the object is allocated upon or before the first call to the `group_local_memory_for_overwrite` function, and deallocated when all work items in the workgroup have completed executing the kernel. All arguments in `args` must be the same for all work items in the group. Group must be `sycl::group`, and `T` must be trivially destructible.

A.6 FPGA LSU Controls

The following table summarizes the load-store unit (LSU) control:

Table 35. LSU Controls

Syntax	Description	Example

`ext::intel::lsu<…>;`	Allows you to specify LSU attributes to control the LSU inferred by the compiler.
	No attributes attempt to infer a pipelined LSU.
	For example:
	• `ext::intel::burst_coalesce<true>`
	• `ext::intel::statically_coalesce<false>`
		`#include <sycl/ext/intel/fpga_extensions.hpp>`
		
		`using namespace sycl`
		`…`
		`using BurstCoalescedLSU =`
		`ext::intel::lsu<ext::intel::burst_coalesce<true>,`
		
		`ext::intel::statically_coalesce<false>>;`
		
		`BurstCoalescedLSU::store(output_ptr, X);`

Syntax	Description	Example
	• `ext::intel::prefetch <true>` • `ext::intel::cache<1024>`	

A.7 FPGA Kernel Attributes

The following table summarizes kernel attributes:

Table 36. FPGA Kernel Attributes

Attribute	Description	Example

`[[intel::scheduler_target_fmax_mhz(N)]]`	Determines the pipelining effort the scheduler attempts during the scheduling process.	`[[intel::scheduler_target_fmax_mhz(SCHEDULER_TARGET_FMAX)]] {`
		`  for (unsigned i = 0; i < SIZE; i++) {`
		`  accessorRes[0] += accessorIdx[i] * 2;`
		`  }`
		`});`

`[[intel::max_work_group_size(Z, Y, X)]]`	Specifies a maximum or the required work-group size for optimizing hardware use of the SYCL kernel without involving excess logic.	`[[intel::max_work_group_size(1,1,MAX_WG_SIZE)]] {`
		`  accessorRes[wiID] = accessorIdx[wiID] * 2;`
		`});`

`[[intel::max_global_work_dim(0)]]`	Omits logic that generates and dispatches global, local, and group IDs into the compiled kernel.	`[[intel::max_global_work_dim(0)]] {`
		`  for (unsigned i = 0; i < SIZE; i++) {`
		`  accessorRes[i] = accessorIdx[i] * 2;`
		`  }`
		`}`

`[[intel::num_simd_work_items(N)]]`	Specifies the number of work items within a work group that the compiler executes in a SIMD or vectorized manner.	`[[intel::num_simd_work_items(NUM_SIMD_WORK_ITEMS),`
		`cl::reqd_work_group_size(1,1,REQD_WORK_GROUP_SIZE)]] {`
		`  accessorRes[wiID] = sqrt(accessorIdx[wiID]);`
		`});`

`[[intel::no_global_work_offset(1)]]`	Omits generating hardware required to support global work offsets.	`[[intel::no_global_work_offset(1))]] {`
		`  accessorRes[wiID] = accessorIdx[wiID] * 2;`
		`}`

`[[intel::kernel_args_restrict]]`	Ignores the dependencies between accessor arguments in a SYCL* kernel.	`[[intel::kernel_args_restrict]] {`
		`  for (unsigned i = 0; i < size; i++) {`
		`  out_accessor[i] = in_accessor[i];`
		`  }`
		`});`

`[[intel::use_stall_enable_clusters]]`	Reduces the area and latency of your kernel.	`h.single_task<class KernelComputeStallFree>( [=]()`
		`[[intel::use_stall_enable_clusters]] {`
		`  // The computations in this device kernel uses Stall Enable Clusters`

continued...

Attribute	Description	Example
		Work(accessor_vec_a, accessor_vec_b, accessor_res); }); });

A.8 FPGA Accessor Properties

The following table summarizes FPGA accessor properties:

Table 37. FPGA Accessor Properties

Property	Description	Example
`buffer_location<index>`	Instructs the host to allocate a buffer to a specific global memory type. It identifies the index of the global memory type in the `board_spec.xml` file of your Custom Platform. The index starts at 0 and follows the order in which the global memory appears in the `board_spec.xml`. If you do not specify the `buffer_location` property, the host allocates the buffer to the default memory type automatically.	`ext::oneapi::accessor_property_list PL{ext::intel::buffer_location<2>}; accessor accessor(buffer, cgh, read_only, PL);`
`no_alias`	Notifies the compiler that all modifications to the memory locations accessed (directly or indirectly) by an accessor during kernel execution is done through the same accessor (directly or indirectly) and not by any other accessor or USM pointer in the kernel. This is an unchecked assertion by the programmer and results in an undefined behavior if it is violated.	`ext::oneapi::accessor_property_list PL{ext::oneapi::no_alias}; accessor accessor(buffer, cgh, read_only, PL);`

A.9 FPGA Extensions

The following table summarizes FPGA extensions supported:

Table 38. FPGA Extensions

FPGA Extension	Description	Example
`ext::intel::fpga_reg()`	Helps the compiler infer at least one pipelining register in the datapath.	`#include <sycl/ext/intel/fpga_extensions.hpp> r[k] = ext::intel::fpga_reg(a[k]) + b[k];`

A.10 Pipe API

The following table summarizes the `pipe` API:

Table 39. Pipe API

API	Description	Example

`ext::intel::pipe`	Use pipes to transfer data between kernels directly using on-device FIFO buffers. Pipes can either be non-blocking or blocking.	`using my_pipe = ext::intel::pipe<class some_pipe, int>;`
		`...`
		`my_pipe::write(rd_src_buf[i]);`
		`...`
		`auto data = my_pipe::read();`
		`...`

A.11 Algorithmic C Data Types

The following table summarizes the algorithmic C (AC) data types:

Table 40. Algorithmic C Data Types Supported by the Intel® oneAPI DPC++/C++ Compiler

Data Type	Header File	Variable Declaration	Description

`ac_int`	`<sycl/ext/intel/ac_types/ac_int.hpp>`	• Template-based declaration:
		`ac_int<N, true> var_name; //Signed N-bit integer`
		`ac_int<N, false> var_name; //Unsigned N-bit integer`
		• Predefined types up to 63 bits:
		`ac_intN::intN var_name; //Signed N-bit integer`
		`ac_intN::uintN var_name; //Unsigned N-bit integer`
			Arbitrary-precision integer support

`ac_fixed`	`<sycl/ext/intel/ac_types/ac_fixed.hpp>`	`ac_fixed<N, I, true, Q, O> var_name; //Signed fixed-point number`
		`ac_fixed<N, I, false, Q, O> var_name; //Unsigned fixed-point number`
			Arbitrary-precision fixed-point number support. For math functions supported by this data type, refer to Math Functions Provided by the `ac_fixed_math.hpp` Header File on page 228.

`ac_complex`	`<sycl/ext/intel/ac_types/ac_complex.hpp>`	Declare according to the data type of your complex number.	Complex number support
`ap_float`	`<sycl/ext/intel/ac_types/ap_float.hpp>`	`ihc::ap_float<exponent_width, mantissa_width[,rounding_mode]>`	Arbitrary-precision floating-point number support. For math functions supported by this data type, refer to Math Functions Provided by `ap_float_math.hpp` Header File on page 228

Compilation Flags

Table 41. Compilation Flags for Data Types

Data Type	dpcpp Command Flags	Description
AC type	• **Linux**: `-qactypes` • **Windows**: `/Qactypes`	Use these flags to include `ac_types` header files on the include path and link against AC type libraries required for the host device execution support.
`ap_float` type	• **Linux**: `-fp-model=precise -no-fma` • **Windows**: `/fp:precise /Qfma-`	Use these flags to ensure that floating-point operations are accurate.
	`-DFPGA_EMULATOR`	Compiles programs with `ap_float` data type for emulation.

Math Functions Provided by the `ac_fixed_math.hpp` Header File

The `ac_fixed_math.hpp` header file adds support for the following non-standard math functions for the arbitrary precision fixed-point (`ac_fixed`) data type:

- `sqrt_fixed`
- `reciprocal_fixed`
- `reciprocal_sqrt_fixed`
- `sin_fixed`
- `cos_fixed`
- `sincos_fixed`
- `sinpi_fixed`
- `cospi_fixed`
- `sincospi_fixed`
- `log_fixed`
- `exp_fixed`

Math Functions Provided by `ap_float_math.hpp` Header File

The `ap_float_math.hpp` header file adds support for the following arbitrary precision fixed-point (`ap_float`) data type functions:

Table 42. Math Functions Provided by `ap_float_math.hpp` Header File

Function Type	Math Functions	Comment
Exponential and logarithmic functions	• `ln` • $\log_2$, $\log_{10}$ • e^x, 2^x, 10^x	Supported only for `ap_float` data types with exponent width less than or equal to 15 bits and mantissa width less than or equal to 63 bits.
	• `ln(1+x)` • e^x-1	Supported only for `ap_float` data types with exponent width less than or equal to 11 bits and mantissa width less than or equal to 52 bits.
Advanced functions	• `reciprocal` • `reciprocal_sqrt`	

continued...

Function Type	Math Functions	Comment
	• `sqrt` • `cube root` • `hypot` (hypotenuse)	
Power functions	• `pow` • `powr` • `pown`	
Trigonometric functions	• `sin`, `cos`, `sincos` • `sinpi`, `cospi` • `asin`, `asinpi` • `acos`, `acospit` • `atan`, `atanpi`, `atan2`	

A.12 Latency Controls (Beta)

The following table summarizes the latency controls:

Table 43. Latency Controls

Argument	Description	Example
`sycl::ext::intel::experimental::latency_anchor_id<N>`	Specifies the ID of the current side-effect operation where it behaves as an anchor.	`// This pipe read() performs as anchor 0 in latency control.` `Pipe::read<ext::intel::experimental::latency_anchor_id<0>>();`

`sycl::ext::intel::experimental::latency_constraint<A, B, C>`	Specifies the latency constraint when the current side-effect operation behaves as a non-anchor.	`// Set a latency constraint between anchor 0 and this pipe write().`
		`// This pipe write() starts exactly 2 cycles after anchor 0 is done.`
		`Pipe::write<ext::intel::experimental::latency_constraint<0, ext::intel::experimental::type::exact, 2>>(...);`

For detailed information about the variables, refer to Latency Controls (Beta) on page 216.

Appendix B Additional Information

For additional information, refer to the following resources:

Resource	Description
Intel® oneAPI Base Toolkit	Main landing page of the Intel® oneAPI Base Toolkit, which includes the Intel® oneAPI DPC++/C++ Compiler and provides tools and libraries for developing high-performance, data-centric applications across diverse architectures.
Intel® FPGA Add-on for oneAPI Base Toolkit	Main landing page of the Intel® FPGA Add-on for oneAPI Base Toolkit, which is a specialized toolkit for programming FPGAs. See also Intel® FPGA Add-On for oneAPI Base Toolkit Release Notes and Intel® FPGA Add-On for oneAPI Base Toolkit System Requirements.
FPGA Tutorials on Github*	Refer to these tutorials for more in-depth instructions about how to use the FPGA tutorials.
FPGA oneAPI Training	Training site with webinars and quick videos.
FPGA-specific Jupiter Notebooks on DevCloud	Training site to sign-up with the Intel® DevCloud and get hands-on practice with code samples in Jupyter Notebooks* running live on Intel® DevCloud.
Get Started with the Intel® oneAPI Base Toolkit on the DevCloud.	Provides instructions to get started with the Intel® DevCloud, which is a cloud-based development sandbox to actively prototype and experiment with workloads on Intel hardware.
Installation Guide for Intel® oneAPI Toolkits	Provides instructions for installing oneAPI toolkits, including the Intel® FPGA Add-On for oneAPI Base Toolkit.
Explore SYCL* Through Intel® FPGA Code Samples	Helps you understand how to navigate the Intel® FPGA SYCL code samples in a coherent manner that builds on complexity and use-case and get your first oneAPI application on the FPGA with the help of six essential FPGA code samples.
Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL	Third-party open-access book to learn how to accelerate C++ programs using data parallelism. This book enables you to be at the forefront of this exciting and important new development that is helping to push computing to new levels. It is full of practical advice, detailed explanations, and code examples to illustrate key topics.
FPGA Workflows on Third-Party IDEs for Intel® oneAPI Toolkits (Eclipse and Visual Studio)	Provides instructions for using Intel® oneAPI tools via third-party integrated development environments (IDEs) on Linux* and Windows* for FPGA development.
Compiling SYCL* FPGA Designs on Red Hat Enterprise Linux (RHEL)* 7.4 OS (Kernel 3.10)	Provides instructions to compile your FPGA designs on Red Hat Enterprise Linux (RHEL) 7.4 OS. *Note:* The Intel® FPGA Add-on for oneAPI Base Toolkit does not officially support the RHEL 7.4 OS (kernel 3.10).

continued...

Resource	Description
Migrating OpenCL FPGA Designs to SYCL*	Provides guidelines to migrate your OpenCL FPGA designs to SYCL.
FPGA Development for Intel® oneAPI Toolkits with Visual Studio Code on Linux	Provides instructions for using Visual Studio Code on Linux* for FPGA development.
Get Started with Intel® Distribution for GDB* on Linux* OS Host	Provides instructions for using Intel® Distribution for GDB* for debugging SYCL and OpenCL™ applications.
Get Started with the Intel® oneAPI Base Toolkit for Linux*	Provides Linux-specific getting started instructions.
Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference	Provides information about the Intel® oneAPI DPC++/C++ Compiler (`icx`/`icpx` or `dpcpp`) and runtime environment.
Intel® oneAPI Programming Guide	Describes the oneAPI programming model in detail, including the FPGA flows.
Intel® VTune™ Profiler User Guide	Provides a comprehensive overview of the product functionality, tuning methodologies, workflows, and instructions to use Intel® VTune Profiler performance analysis tool.
Analyzing CPU and FPGA (Intel® Arria® 10 GX) Interaction	Provides instructions for configuring your platform to analyze an interaction of your CPU and FPGA using Intel® Arria® 10 GX FPGA as an example.
Profiling an FPGA-driven SYCL Application	Provides instructions for profiling an FPGA-driven SYCL application.
Intel® FPGA SDK for OpenCL™ Pro Edition: Custom Platform Toolkit User Guide	Outlines the procedure for creating an Intel® FPGA Software Development Kit (SDK) for OpenCL™ Pro Edition Custom Platform.
Intel® Quartus® Prime Software User Guides	Provides links to various Intel® Quartus® Prime user guides, which cover specific topics to help you see your design through to completion.
Intel® Acceleration Stack Quick Start Guide for Intel® Programmable Acceleration Card with Intel® Arria® 10 GX FPGA	Serves as a high-level quick start guide to help you with installing key software packages, updating the flash image, running diagnostics, and managing security for Intel® PAC with Intel® Arria® 10 GX FPGA.
Intel® Acceleration Stack Quick Start Guide: Intel® FPGA Programmable Acceleration Card D5005	Serves as a high-level quick start guide for Intel® FPGA PAC D5005 to help you with installing OPAE on the host Intel® Xeon® Processor to manage and access the Intel FPGA PAC, managing flash image, running an example, and handling graceful thermal shutdown.

Appendix C Document Revision History for the FPGA Optimization Guide for Intel® oneAPI Toolkits

Date	Release Version	Changes
April 2022	2022.2	• Changed the document title *Intel® oneAPI DPC++ FPGA Optimization Guide* to *FPGA Optimization Guide for Intel® oneAPI Toolkits*. • Replaced all general references to DPC++ with SYCL. • Added new loop functions to the existing list in *FPGA Loop Directives*. • Added a new limitation and removed some existing limitations in *Advantages and Limitations of Arbitrary Precision Data Types*. • Replaced all references to https://hlslibs.org/ with a reference to the documentation at https://github.com/hlslibs/ac_types/blob/v3.7/pdfdocs/ac_datatypes_ref.pdf. • Made a minor update to the description in *The `pipe` Class and its Use*. • Modified all occurrences of `[[cl::reqd_work_group_size(Z, Y, X)]]` to `[[sycl::reqd_work_group_size(Z, Y, X)]]`. `[[cl::reqd_work_group_size(Z, Y, X)]]` is now deprecated. • Added the following new topics: — *Control Hardware Implementation of the Supported Data Types and Math Functions (`-Xsdsp-mode=<option>`)* — *Latency Controls*
December 2021	2022.1	• Added new attributes to the existing list in *FPGA Loop Directives* and *FPGA Optimization Flags*. • Added the following new topics: — *Global Control of Exit FIFO Latency of Stall-free Clusters (`-Xssfc-exit-fifo-type=<value>`)* — *`nofusion` Attribute* — *Enable the Read-Only Cache for Read-Only Accessors (`-Xsread-only-cache-size=<N>`)* — Renamed `hls_float` data type to `ap_float`. Added support for the header files `<sycl/ext/intel/ac_types/ap_float.hpp>` and `<sycl/ext/intel/ac_types/ap_float_math.hpp>`. In a future release, the header files `<sycl/ext/intel/ac_types/hls_float.hpp>` and `<sycl/ext/intel/ac_types/hls_float_math.hpp>` will be removed. — Updated few limitations in *Advantages and Limitations of Arbitrary Precision Data Types*.
September 2021	2021.4	• Changed all occurrences of the namespace `sycl::ONEAPI::` to `sycl::ext::oneapi::`. • Changed all occurrences of the namespace `sycl::INTEL::` to `sycl::ext::intel::`. • Changed all occurrences of the header `#include <CL/sycl/INTEL/...>` to `#include <sycl/ext/intel/...>`. • Changed `clang++` to `dpcpp`. • Changed all occurrences of `[[intel::ii(n)]]` to `[[intel::initiation_interval(n)]]`. • Minor update about the types of FIFOs reported in the *System Viewer*. • Added a note about multi-process execution in the installed FPGAs in *Multi-Threaded Host Application*.

continued...

Date	Release Version	Changes
		• Added an additional value `on` to `#pragma fp contract(fast/off)`. • Updated the code snippets in *Specify Schedule F_{MAX} Target for Kernels* and *Memory Attributes*. • Modified the flag description for AC type in *Variable-Precision Integer and Floating-Point Support*. • Modified the list of limitations in *Advantages and Limitations of Arbitrary Precision Data Types*. • Added information about debugging in *Declare the `ac_int` Data Type*. • Added support for simulation flow. • Changed all occurrences of Mentor Graphics* to Siemens EDA* *(formerly Mentor Graphics)* . • Changed the `noinit` attribute to `no_init`. • Modified the description of `[[intel::kernel_args_restrict]]` attribute in *Global Memory Accesses Optimization*. • Replaced the flag `-Xsffpcontract=<value>` with `-fp-model=<value>` and updated its description in *Reducing Floating-Point Rounding Operations*. • Removed few limitations of shift register in *Kernel Variable Accesses*. • Added the following new topics: — *Modify the Rounding Mode of Floating-point Operations (`-Xsrounding=<rounding_type>`)* — *Reduce Kernel Area and Latency* (`[[intel::use_stall_enable_clusters]]` attribute) — *Timing Failures*
July 2021	2021.3	Bug fixes.
June 2021	2021.3	• Added more information about streaming data using I/O pipes and faking I/O pipes in *I/O Pipes*. • Added description about the local memory function in *Kernel Memory*. • Added a new section to describe the global memory bandwidth use calculation in *Global Memory Accesses Optimization*. • In *System Viewer*, added description about the global memory view, changed the Graph Viewer report name to System Viewer, updated all images and made minor updates in the remaining part of the topic. • Updated the description and image in *Schedule Viewer*. • Updated most of the code snippets to reflect the latest coding practices. • Made minor updates to the description of *`ivdep` Attribute* and *`loop_coalesce` Attribute*. • Added MLABs to the description in *Perform Kernel Computations Using Local or Private Memory*. • Modified the description in *Strategies for Inferring the Accumulator* to remove references to the `-Xsfp-relaxed` flag. • Updated the compilation options in *Variable-Precision Integer and Floating-Point Support*. • Updated the limitations in *Advantages and Limitations of Arbitrary Precision Data Types*. • Merged *Ignoring Dependencies Between Accessor Arguments* topic with *Global Memory Accesses Optimization*. • Updated the list of attributes in *FPGA Kernel Attributes*. • Added the following new topics: — *Shannonization to Improve F_{MAX}/II* — *Simple Host-device Streaming* — *Buffered Host-device Streaming* — *N-Way Buffering* — *Optimize Inner Loop Throughput* — *Improve Loop Performance by Caching On-Chip Memory*

continued...

Date	Release Version	Changes
		— *FPGA Local Memory Function* — *Pipelining Loops in Non-task Kernels (`-Xsauto-pipeline`)* — *Pipe and Atomic Fence* — *Reducing Floating-Point Rounding Operations (`-Xsffp-contract=fast`)* — *FPGA Accessor Properties* • Removed the following topics because the respective flags are now deprecated: — *Relax the Order of Floating-Point Operations (`-Xsfp-relaxed`)* — *Reduce Floating-Point Rounding Operations (`-Xsfpc`)* • Removed *Tree Balancing* and *Rounding Operations* topic from *Optimize Floating-point Operation*.
March 2021	2021.2	• Updated the topic *Cluster the Datapath* completely including the diagrams. • Updated the dependency graph in *Mapping Source Code Instructions to Hardware*. • Updated *Hyper-Optimized Handshaking Data Flow* diagram in *Handshaking Between Clusters* topic. • Made minor updates in *Executing Independent Operations Simultaneously*. • Updated the topic *Pipelining* completed including the diagrams. • Added a note about the enablement of fast math operations for floating point operations in *Data Types and Operations*. • Updated the topic *Access HLD FPGA Reports in JSON Format* to include details about the Bottlenecks viewer. • Made minor update to a note in *Specify Number of SIMD Work-Items* • Added the following new topics: — *Floating Point Optimizations* — *Floating Point Pragmas* — *Specify Schedule F_{MAX} Target for Kernels* — *Bottlenecks Viewer* — *Loop Bottlenecks* — *Variable-Precision Integer and Floating-Point Support* — *Advantages and Limitations of Arbitrary Precision Data Types* — *Declare and Use the AC Data Types* — *Declare the `ac_int` Data Type* — *Declare the `ac_fixed` Data Type* — *Declare the `ac_complex` Data Type* — *Declare the `hls_float` Data Type* — *Conversion Rules for `hls_float`* — *Operations with Explicit Precision Controls* — *Comparison Operators* — *Additional `hls_float` Functions* — *Additional Data Types Provided by `hls_float.hpp`*
December 2020	2021.1	First major release.