Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)

Overlapping Data Transfers with Kernel Computation

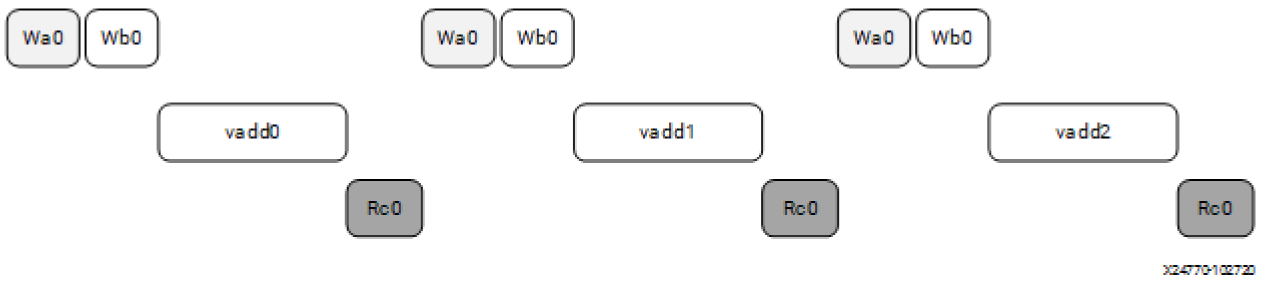
Overlapping Data Transfers with Kernel Computation

Applications, such as database analytics, have a much larger data set than can be stored in the available global device memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.
An example can be found in the `vadd` kernel from the overlap example in the host category of Vitis Accelerated Examples on GitHub. This examples demonstrates techniques to overlap Host (CPU) and FPGA computation in the application. In this example, the kernel processes two arrays by adding them together and writing to output. From the host perspective, there are four tasks to perform in this example:

1. Write buffer a (`Wa`)
2. Write buffer b (`Wb`)
3. Execute `vadd` kernel
4. Read buffer c (`Rc`)

Using a simple in-order command queue without data transfer optimization, the overall execution timeline trace should look similar to the one shown below:

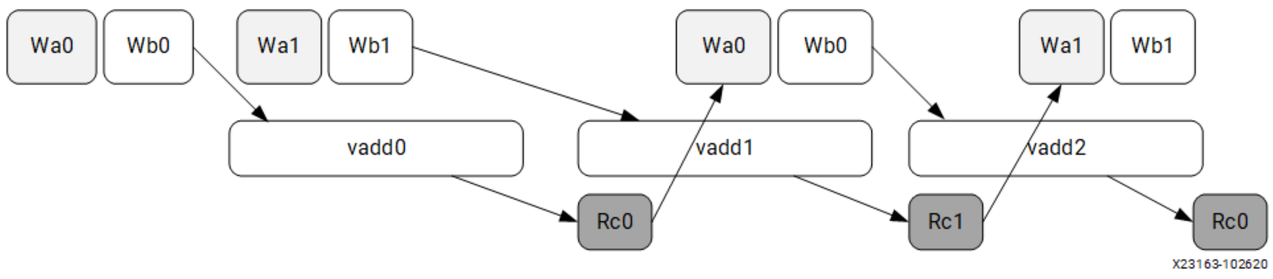
Figure: Host View of Tasks



Using an out-of-order command queue, data transfer and kernel execution can overlap as illustrated in the figure below. In the host code for this example, double buffering is used for all buffers so that the kernel can process one set of buffers while the host can operate on the other set of buffers.
The OpenCL `event` object provides an easy method to set up complex operation dependencies and synchronize host threads and device operations. Events are OpenCL objects that track the status of operations. Event objects are created by

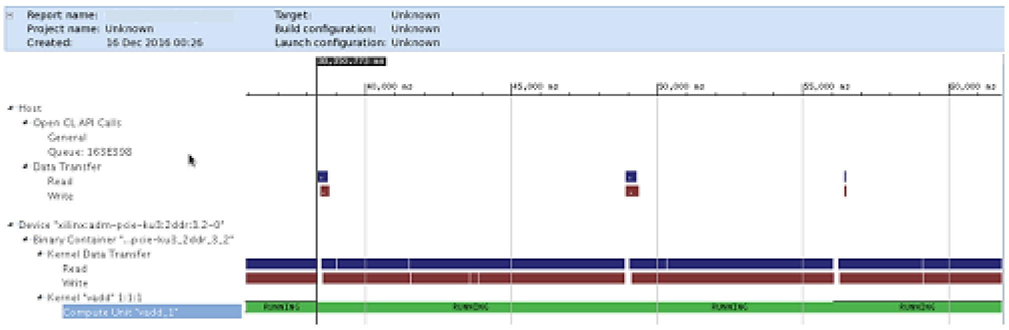
kernel execution commands, `read`, `write`, and `copy` commands on memory objects, or user events created using `clCreateUserEvent`.
You can ensure an operation has completed by querying the events returned by these commands. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

Figure: Event Triggering Setup



In the example, the host code (host.cpp) enqueues the four tasks in a loop to process the complete data set. It also sets up event synchronization between different tasks to ensure that data dependencies are met for each task. The double buffering is set up by passing different memory objects values to `clEnqueueMigrateMemObjects` API. The event synchronization is achieved by having each API call wait for other event as well as trigger its own event when the API completes.
The Timeline Trace view below clearly shows that the data transfer time is completely hidden, while the compute unit `vadd_1` is running constantly.

Figure: Data Transfer Time Hidden in Timeline Trace View



Report name:
Project name: Unknown
Created: 16 Dec 2016 08:26
Target: Unknown
Build configuration: Unknown
Launch configuration: Unknown