SDAccel Programmers Guide

UG1277 (v2018.3) January 24, 2019

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
01/24/2019 Version 2018.3	
Entire document	Editorial Updates.
12/05/2018 Version 2018.3	
Chapter 1: SDAccel Compilation Flow and Execution Model	Updated SDAccel Architecture figure.
Devices	Added sub-device description.
Buffer Transfer to/from the FPGA Device	Updated sections.
Summary	Updated #3 description.
Chapter 3: Programming C/C++ Kernels	Added description.
Memory Data Inputs and Outputs	Updated whole section.
Customization of DDR Bank to Kernel Connection	Added note.
10/02/2018 Version 2018.2.xdf	
	No changes for release.
07/02/2018 Version 2018.2	
Customization of DDR Bank to Kernel Connection	Updated syntax and requirements for `--sp` option.
06/06/2018 Version 2018.2	
Initial Xilinx® release.	N/A

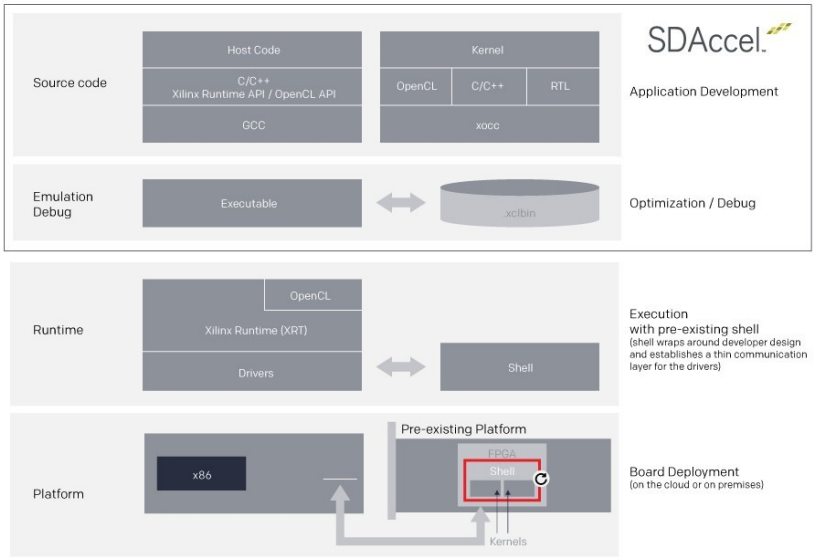
Table of Contents

Chapter 1

SDAccel Compilation Flow and Execution Model

The SDAccel™ development environment is a heterogeneous system architecture platform to accelerate compute intensive tasks using Xilinx® FPGA devices. The SDAccel environment contains a Host x86 machine that is connected to one or more Xilinx FPGA devices through a PCIe® bus, as shown below.

Figure 1: **SDAccel Architecture**



Programming Model

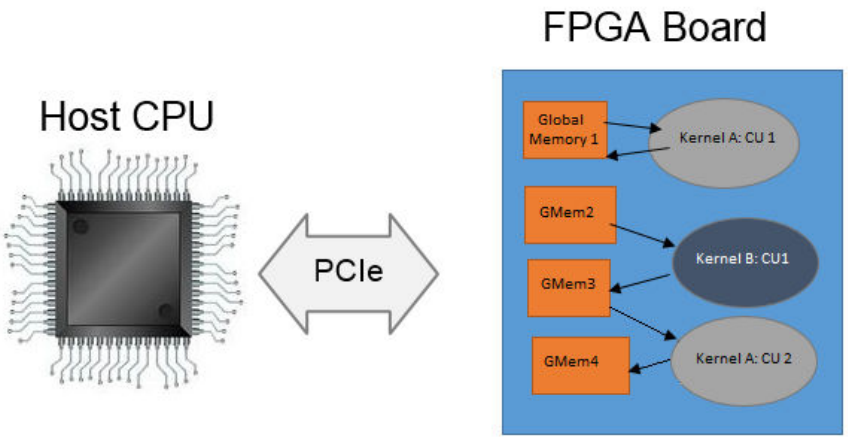
The SDAccel environment supports heterogeneous computing using the industry standard OpenCL protocol (https://www.khronos.org/opencl/). The host program executes on the Host CPU and offloads compute intensive tasks to execute on Xilinx FPGA devices using the OpenCL programming paradigm. Unlike a CPU (or GPU), the FPGA can be thought of as a blank canvas which does not have predefined instruction sets or fixed word size, and is capable of running the same or different instructions in parallel to greatly improve the performance of your applications.

Device Topology

In the SDAccel environment, devices are one or more FPGAs connected to a host x86 machine through a PCIe bus. The FPGA contains a programmable region that implements and executes kernels. The FPGA platform contains one or more global memory banks. The data transfer from the host machine to kernels and from kernels to the host happens through these global memory banks. The kernels running on the FPGA can have one or more memory interfaces. The connection from the global memory banks to those memory interfaces are configurable, as their features are determined by the kernel compilation options.

The programmable logic of Xilinx devices can implement multiple kernels at the same time, allowing for significant task parallelism. A single kernel can also be instantiated multiple times. The number of instances of a kernel is programmable and determined by the kernel compilation options.

Figure 2: **SDAccel Architecture**



The diagram above illustrates the flexible connections from the host application to multiple kernels through the global memory banks. The FPGA board device shown above contains four DDR memory banks. The programmable logic of the FPGA is running two kernels, Kernel A and Kernel B. Each kernel has two memory interfaces one for reading the data and another for writing. Also, note that there are two instances of Kernel A, totaling three simultaneous kernel instances on the FPGA.

In the diagram, the first instance of Kernel A: CU1 uses a single memory interface for reading and writing. Kernel B and the second instance of Kernel A: CU2 use different memory interfaces for reading, and writing, with Kernel B essentially passing data directly to Kernel A: CU2 through the global memory.

RECOMMENDED: *To achieve the best performance, the global memory banks to kernel interface connections should carefully be defined as discussed in* Customization of DDR Bank to Kernel Connection.

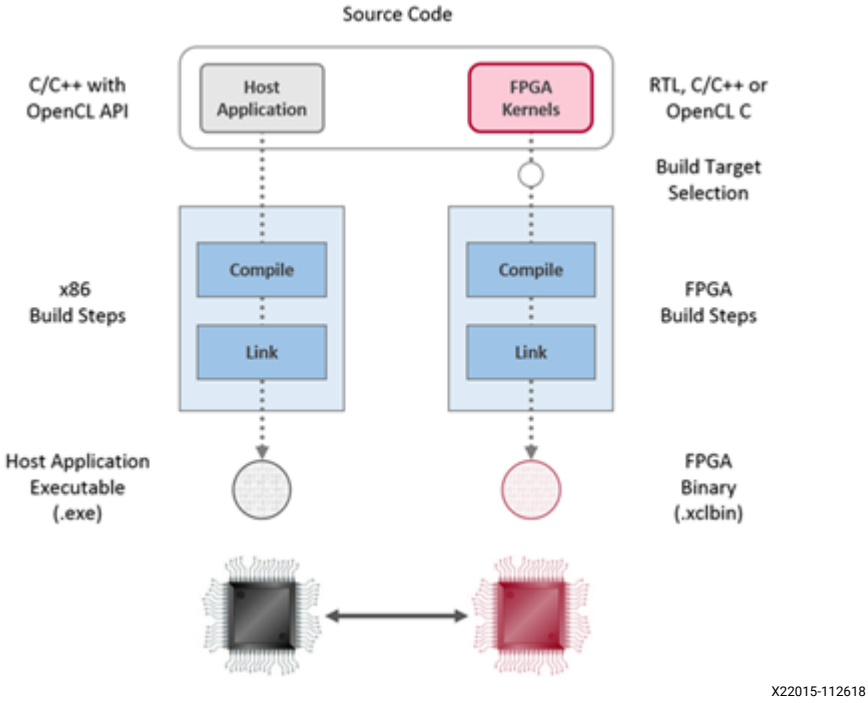
SDAccel Build Process

The SDAccel environment offers all of the features of a standard software development environment:

- Optimized compiler for host applications
- Cross-compilers for the FPGA
- Robust debugging environment to help identify and resolve issues in the code
- Performance profilers to identify bottlenecks and optimize the code

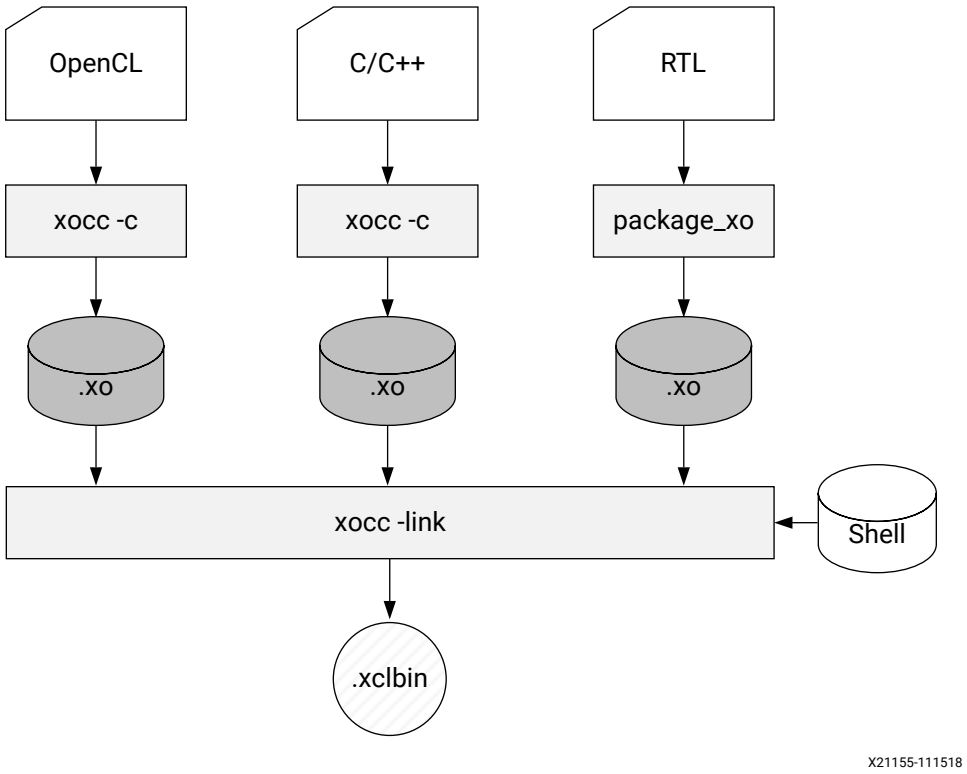
Within this environment, the build process uses a standard compilation and linking process for both the software elements, and the hardware elements of the project. As shown in the following figure, the host application is built through one process using standard GCC compiler, and the FPGA binary is built through a separate process using the Xilinx `xocc` compiler.

Figure 3: **Software/Hardware Build Process**



1. Host application build process using GCC:
   - Each host application source file is compiled to an object file (`.o`).
   - The object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`).
2. FPGA build process is highlighted in the following figure:
   - Each kernel is independently compiled to a Xilinx object (`.xo`) file.
     - C/C++ and OpenCL C kernels are compiled for implementation on an FPGA using the `xocc` compiler. This step leverages the Vivado® HLS compiler. Pragmas and attributes supported by Vivado HLS can be used in C/C++ and OpenCL C kernel source code to specify the desired kernel micro-architecture and control the result of the compilation process.
     - RTL kernels are compiled using the `package_xo` utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.
   - The kernel `.xo` files are linked with the hardware platform (shell) to create the FPGA binary (`.xclbin`). Important architectural aspects are determined during the link step. In particular, this is where connections from kernel ports to global memory banks are established and where the number of instances for each kernel is specified.
     - When the build target is software or hardware emulation, as described below, `xocc` generates simulation models of the device contents.
     - When the build target is the system (actual hardware), `xocc` generates the FPGA binary for the device leveraging the Vivado Design Suite to run synthesis and implementation.

Figure 4: **FPGA Build Process**



***Note*:** The `xocc` compiler automatically uses the Vivado HLS and Vivado Design Suite tools to build the kernels to run on the FPGA platform. It uses these tools with predefined settings which have proven to provide good quality of results. Using the SDAccel environment and the `xocc` compiler does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all their available features to implement kernels.

Build Targets

The SDAccel tool build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel build target defines the nature of FPGA binary generated by the build process.

The SDAccel tool provides three different build targets, two emulation targets used for debug and validation purposes, and the default hardware target used to generate the actual FPGA binary:

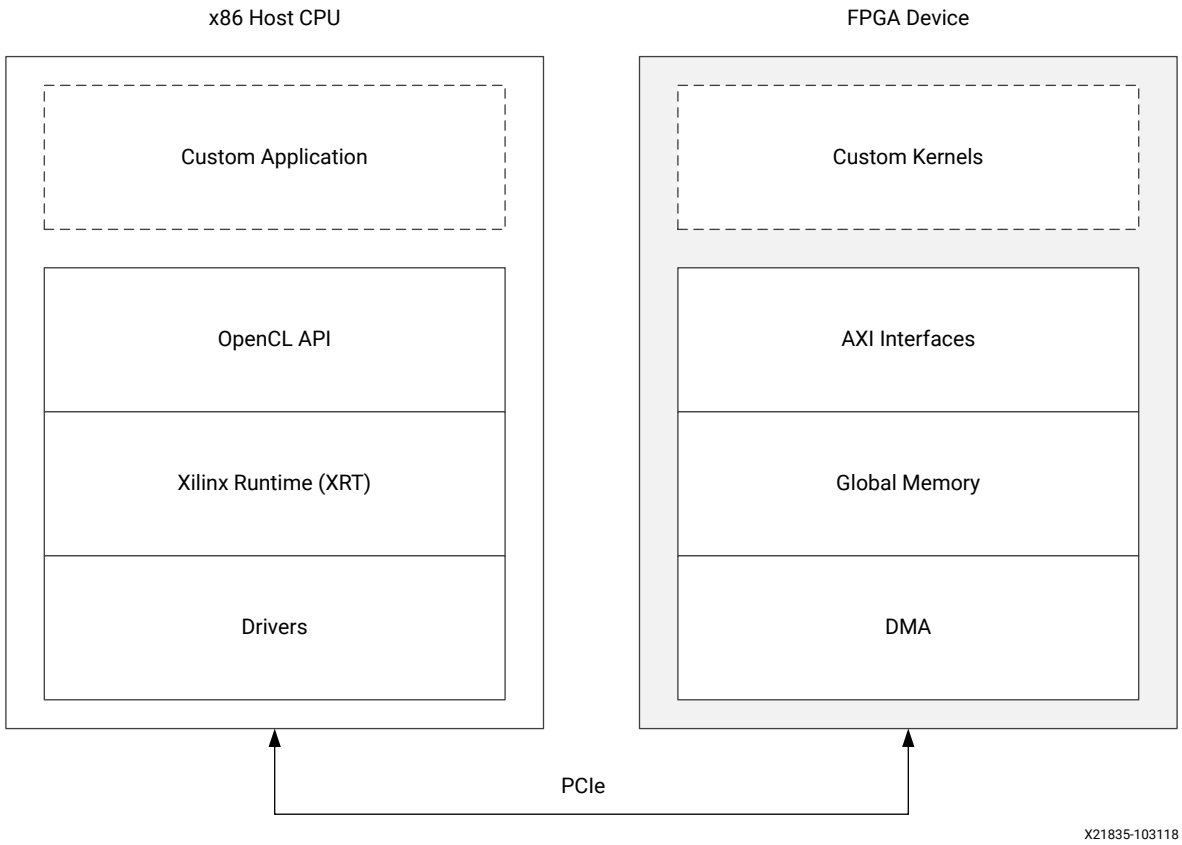
- **Software Emulation (`sw_emu`):** Both the host application code and the kernel code are compiled to run on the x86 processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.

- **Hardware Emulation (`hw_emu`):** The kernel code is compiled into a hardware model (RTL) which is run in a dedicated simulator. This build and run loop takes longer but provides a detailed, cycle-accurate, view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and for getting initial performance estimates.
- **System (`hw`):** The kernel code is compiled into a hardware model (RTL) and is then implemented on the FPGA device, resulting in a binary that will run on the actual FPGA.

Execution Model of an SDAccel Application

The SDAccel environment is designed to provide a simplified development experience for FPGA-based software acceleration platforms. The general structure of the acceleration platform is shown in the following figure.

Figure 5: **Architecture of an SDAccel Application**



The custom application is running on the host x86 server and uses OpenCL API calls to interact with the FPGA accelerators. The Xilinx runtime (XRT) manages those interactions. The application is written in C/C++ using OpenCL APIs. The custom kernels are running within a Xilinx FPGA with the XRT managing interactions between the host application and the accelerator. Communication between the host x86 machine and the accelerator board occurs across the PCIe bus.

The SDAccel hardware platform contains global memory banks. The data transfer between the host machine and kernels, in either direction, occurs through these global memory banks. The kernels running on the FPGA can have one or more memory interfaces. The connection from the memory banks to those memory interfaces is programmable and determined by linking options of the compiler.

The SDAccel execution model follows these steps:

1. The host application writes the data needed by a kernel into the global memory of the attached device through the PCIe interface.
2. The host application programs the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading and writing data from global memory, as necessary.
5. The kernels write data back to the memory banks, and notify the host that it has completed its task.
6. The host application reads data back from global memory into the host memory space, and continues processing as needed.

The FPGA can accommodate multiple kernel instances at one time; this can occur between different types of kernels or multiple instances of the same kernel. The XRT transparently orchestrates the communication between the host application and the kernels in the accelerator. The number of instances of a kernel is determined by compilation options.

SDAccel Emulation Flows

The SDAccel environment development flow can be divided into two distinct steps. The first step is to compile the host and kernel code to generate the executables. The second step is to run the executables in a heterogeneous system comprised of the Host CPU and SDAccel environment accelerator platform. However, the kernel compilation process is long and can take several hours depending on the size of the kernels and the architecture of the target FPGA. Therefore, to save time and shorten the debug cycle before the kernel compilation process, the SDAccel environment provides two other build targets for testing purposes: software emulation and hardware emulation. The compilation and execution of these emulation targets are significantly

faster, and do not require the actual FPGA board to be run. These emulation flows abstract the FPGA board, and its connection to the host machine, into software models to validate the combined functionality of the host and kernel code, as well as providing some performance estimates early in the design process. These performance estimates are just estimates, but they can greatly help debugging and identifying performance bottlenecks. Refer to the *SDAccel Environment Debugging Guide* (UG1281) for more information on debugging using software and hardware emulation flows.

Software Emulation Flow

Compilation of the software emulation target is the fastest. It is mainly used for checking the functional correctness when the host and kernel code are running together. The `xocc` compiler does the minimum transformation of the kernel code in order to run it in conjunction with the host code, so this software emulation flow helps to check functional correctness at the very early stage of the final binary creation. The software emulation flow can be used for algorithm refinement, debugging functional issues, and letting developers iterate quickly through the code to make improvements.

Hardware Emulation Flow

In the hardware emulation flow, the `xocc` compiler generates a model of the kernel in a hardware description language (RTL Verilog). The hardware emulation flow helps to check the functional correctness of the final binary creation after synthesis of the RTL from the C, C++, or OpenCL kernel code. The hardware emulation flow offers great debug capability with the waveform viewer if the system does not work as expected.

Table 1: **Comparison of Emulation Flows with Hardware Execution**

Software Emulation	Hardware Emulation	Hardware Execution
Host application runs with a C/C++ or OpenCL model of the kernels.	Host application runs with a simulated RTL model of the kernels.	Host application runs with actual hardware implementation of the kernels.
Confirm functional correctness of the system.	Test the host / kernel integration, get performance estimates.	Confirm that the system runs correctly and with desired performance.
Fastest turnaround time.	Best debug capabilities, moderate compilation time.	Final FPGA implementation and run provides accurate performance result with long build time.

SDAccel Example Designs

SDAccel Examples on GitHub

Xilinx provides many examples for programming with the SDAccel environment in the GitHub repository to help beginning users get familiar with the coding style of host and kernel code, and for more experienced users to use as a source of coding examples. All examples are available with host code, kernel code, and Makefile associated with the compilation flow and runtime flow. The following is one such example to get a basic understanding of the file structure of a standard example.

Inverse Discrete Cosine Transform (IDCT) Example

Look at the IDCT example design that demonstrates the key coding organization required for the SDAccel environment.

The `Readme.md` file discusses in detail how to run this example in both emulation and FPGA execution flows using the provided `Makefile`.

Inside the `./src` directory, you can find host code `idct.cpp`, and kernel code `krnl_idct.cpp`.

In the following chapters you will learn the basic required knowledge to program the host code and kernel code for the SDAccel environment. During this process you might refer to the above design as an example.

Organization of this Guide

The remaining chapters are organized as follows:

- Chapter 2: Programming the Host Application: Describes writing host code in C or C++ using the OpenCL API targeted for Xilinx FPGA devices. This chapter assumes the user has prior knowledge of OpenCL. It discusses coding practices to follow when writing an effective host application interfacing with acceleration kernels running on Xilinx FPGA devices.
- Chapter 3: Programming C/C++ Kernels: Describes different elements of writing high-performance, compute-intensive kernel code to implement on an FPGA device.
- Chapter 4: Configuring the System Architecture: Discusses integrating and connecting the host application to one or more kernel instances during the linking process.

Programming the Host Application

In the SDAccel™ environment, host code is written in C or C++ language using the industry standard OpenCL™ API. The SDAccel environment provides an OpenCL 1.2 embedded profile conformant runtime API.

IMPORTANT! *The SDAccel environment supports the OpenCL Installable Client Driver (ICD) extension (`cl_khr_icd`). This extension allows multiple implementations of OpenCL to co-exist on the same system. Refer to Appendix A: OpenCL Installable Client Driver Loader for details and installation instructions.*

The SDAccel environment consists of a host x86 CPU and compute devices running on a Xilinx® FPGA.

In general, the structure of the host code can be divided into three sections:

1. Setting up the environment.
2. Core command execution including executing one or more kernels.
3. Post processing and FPGA release.

The following sections discuss each of the above topics in detail.

Note: For multithreading the host program, exercise caution when calling a `fork()` system call from an SDAccel environment application. The `fork()` does not duplicate all the runtime threads. Hence the child process cannot run as a complete application in the SDAccel environment. It is advisable to use the `posix_spawn()` system call to launch another process from the SDAccel environment application.

Setting Up the OpenCL Environment

The host code in the SDAccel environment follows OpenCL programming paradigm. To set the environment properly, the host application should identify the standard OpenCL models. They are: platform, devices, context, command queue, and program.

TIP: *The host code examples and API commands used in this document follow the OpenCL C API. The IDCT example referred to in SDAccel Example Designs is also written with the C API. However, the SDAccel runtime environment also supports the OpenCL C++ wrapper API, and many of the examples in the GitHub repository are written using the C++ API. Refer to https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf for more information on this C++ wrapper API.*

Platform

From the very beginning the host code should identify the platform composed of Xilinx FPGA as one or more devices. The host code segment below is standard coding to identify the Xilinx device based platform.

```
cl_platform_id platform_id;         // platform id

err = clGetPlatformIDs(16, platforms, &platform_count);

// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
  err = clGetPlatformInfo(platforms[iplat],
    CL_PLATFORM_VENDOR,
    1000,
    (void *)cl_platform_vendor,
    NULL);

  if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
  // Xilinx Platform found
  platform_id = platforms[iplat];
  }
}
```

The OpenCL API call `clGetPlatformIDs` is used to discover the set of available OpenCL platforms for a given system. Thereafter, `clGetPlatformInfo` is used to identify the Xilinx device based platform by matching `cl_platform_vendor` with the string `"Xilinx"`.

RECOMMENDED: *Though it is not explicitly shown in the preceding code, or in other host code examples used throughout this chapter, it is always a good coding practice to use error checking after each of the OpenCL API calls. This can help debugging and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution. Below is an error checking code example for `clGetPlatformIDs` command:*

```
err = clGetPlatformIDs(16, platforms, &platform_count);
if (err != CL_SUCCESS) {
  printf("Error: Failed to find an OpenCL platform!\n");
  printf("Test failed\n");
  exit(1);
}
```

Devices

After the platform detection, the Xilinx FPGA devices attached to the platform are identified. The SDAccel environment supports one or more Xilinx FPGA devices working together.

The following code demonstrates finding all the Xilinx devices (with a upper limit of 16) by using API `clGetDeviceIDs` and printing their names.

```
cl_device_id devices[16];  // compute device id
char cl_device_name[1001];

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
  16, devices, &num_devices);

printf("INFO: Found %d devices\n", num_devices);

//iterate all devices to select the target device.
for (uint i=0; i<num_devices; i++) {
  err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name,
0);
  printf("CL_DEVICE_NAME %s\n", cl_device_name);
}
```

IMPORTANT! *The `clGetDeviceIDs` API is called with the `device_type` `CL_DEVICE_TYPE_ACCELERATOR` to get all the available Xilinx devices.*

Sub-devices

In the SDAccel environment, sometimes devices contain multiple kernel instances, of a single kernel or of different kernels. The OpenCL API `clCreateSubDevices` allows the host code to divide the device into multiple sub-devices containing one kernel instance per sub-device. Currently SDAccel environment supports equally divided sub-devices each containing only one kernel instance.

The following example shows:

1. The sub-devices are created by equal partition to execute one kernel instance per sub-device.
2. Iterating over the sub-device list and using a separate context and command queue to execute the kernel on each of them.
3. The API related to kernel execution (and corresponding buffer related) code is not shown for the sake of simplicity, but would be described inside the function `run_cu`.

```
cl_uint num_devices = 0;
cl_device_partition_property props[3] = {CL_DEVICE_PARTITION_EQUALLY,1,0};

// Get the number of sub-devices
clCreateSubDevices(device,props,0,nullptr,&num_devices);

// Container to hold the sub-devices
std::vector<cl_device_id> devices(num_devices);

// Second call of clCreateSubDevices
// We get sub-device handles in devices.data()
clCreateSubDevices(device,props,num_devices,devices.data(),nullptr);

// Iterating over sub-devices
std::for_each(devices.begin(),devices.end(),[kernel](cl_device_id sdev) {

    // Context for sub-device
    auto context = clCreateContext(0,1,&sdev,nullptr,nullptr,&err);
```

```
    // Command-queue for sub-device
    auto queue = clCreateCommandQueue(context,sdev,
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,&err);

    // Execute the kernel on the sub-device using local context and
  queue run_cu(context,queue,kernel); // Function not shown
});
```

Currently, if a kernel has multiple hardware instances (can be specified during the kernel compilation phase), the SDAccel environment execution model assumes all those hardware instances have the same global memory connectivity. If not, then you need to use sub-devices to allocate separate `cl_kernel` for each of those hardware instances.

Context

The OpenCL context creation process is straightforward. The API `clCreateContext` is used to create a context that contains one or more Xilinx devices that will communicate with the host machine.

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

In the code example above, the API `clCreateContext` is used to create a context that contains one Xilinx device. You can create only one context for a device from a host program. However, the host program should use multiple contexts if sub-devices are used; one context for each sub-device.

Command Queues

One or more command queues for each device is created using the `clCreateCommandQueue` API. The FPGA device can contain multiple kernels. When developing the host application, there are two main programming approaches to execute kernels on a device:

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. The SDAccel runtime environment dispatches those kernels as soon as possible in any order allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queue: Each kernel execution will be requested from different in-order command queues. In such cases, the SDAccel runtime environment can dispatch kernels from any command queue with the intention of improving performance by running them concurrently on the FPGA.

The following is an example of standard API calls to create in-order and out-of-order command queues.

```
// Out-of-order Command queue
commands = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Program

As described in the SDAccel Build Process, the host and kernel code are compiled separately to create separate executable files: the host application (`.exe`) and the FPGA binary (`.xclbin`). When the host application is executed it must load the `.xclbin` using the `clCreateProgramWithBinary` API.

The following code example shows how the standard OpenCL API is used to build the program from the `.xclbin` file:

```
unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
&size_var,
                       (const unsigned char **) &kernelbinary, &status, &err);

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Function
int load_file_to_memory(const char *filename, char **result)
{
  uint size = 0;
  FILE *f = fopen(filename, "rb");
  if (f == NULL) {
    *result = NULL;
    return -1; // -1 means file opening fail
  }
  fseek(f, 0, SEEK_END);
  size = ftell(f);
  fseek(f, 0, SEEK_SET);
  *result = (char *)malloc(size+1);
  if (size != fread(*result, sizeof(char), size, f)) {
    free(*result);
    return -2; // -2 means file reading fail
  }
  fclose(f);
  (*result)[size] = 0;
  return size;
}
```

The above example performs the following steps:

1. The kernel binary file, `.xclbin`, is passed in from the command line argument, `argv[1]`.

TIP: *Passing the `.xclbin` through a command line argument is specific to this example. You can also hardcode the kernel binary file in the application.*

2. The `load_file_to_memory` function is used to load the file contents in the host machine memory space.

3. The API `clCreateProgramWithBinary` and `clBuildProgram` are used to complete the program creation process.

Executing Commands in the FPGA Device

Once the OpenCL environment is initialized, the host application is ready to issue commands to the device and interact with the kernels. Such commands include:

1. Memory data transfer to and from the FPGA device.
2. Kernel execution on FPGA.
3. Event synchronization.

Buffer Transfer to/from the FPGA Device

Interactions between the host application and kernels rely on transferring data to and from global memory in the device. The simplest way to send data back and forth from the FPGA is using `clCreateBuffer`, `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` commands. The following code example demonstrates this:

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector
// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
  host_mem_ptr[i] = <... >
}
cl_mem dev_mem_ptr = clCreateBuffer(context,  CL_MEM_READ_WRITE,
                     sizeof(int) * number_of_words, NULL, NULL);

err = clEnqueueWriteBuffer(commands, dev_mem_ptr, CL_TRUE, 0,
      sizeof(int) * number_of_words, host_mem_ptr, 0, NULL, NULL);
```

IMPORTANT! *A single buffer cannot be bigger than 4 GB.*

For the majority of applications the example code above would be sufficient to transfer data from the host to the device memory. However, there are a number of coding practices you should adopt in order to maximize performance and fine-grain control.

Using `clEnqueueMigrateMemObjects`

Another consideration when transferring data is using `clEnqueueMigrateMemObjects` instead of `clEnqueueWriteBuffer` or `clEnqueueReadBuffer` to improve the performance. Typically, memory objects are implicitly migrated to a device for enqueued kernels. Using this API call results in data transfer ahead of kernel execution to reduce latency, particularly when a kernel is called multiple times.

The following code example is modified to use `clEnqueueMigrateMemObjects`:

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
  host_mem_ptr[i] = <... >
}

cl_mem_ext_ptr_t d_bank0_ext;
  d_bank0_ext.flags = XCL_MEM_DDR_BANK0;
  d_bank0_ext.obj = host_mem_ptr;
  d_bank0_ext.param = 0;

cl_mem dev_mem_ptr = clCreateBuffer(context,
                     CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX,
                     sizeof(int) * number_of_words, &d_bank0_ext, NULL);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
      NULL, NULL);
```

Using `posix_memalign` for Host Memory Space

SDAccel runtime allocates the memory space in 4K boundary for internal memory management. If the host memory pointer is not aligned to a 4K word boundary, the runtime performs extra `memcpy` to make it aligned. It does not significantly impact performance, but you should align the host memory pointer with the 4K boundary to follow the SDAccel runtime memory management.

The following is an example of how `posix_memalign` is used instead of `malloc` for the host memory space pointer.

```
int *host_mem_ptr; // = (int*) malloc(MAX_LENGTH*sizeof(int));
// Aligning memory in 4K boundary
posix_memalign(&host_mem_ptr,4096,MAX_LENGTH*sizeof(int));

// Fill the memory input
for(int i=0; i<MAX_LENGTH; i++) {
  host_mem_ptr[i] = <... >
}

cl_mem_ext_ptr_t d_bank0_ext;
  d_bank0_ext.flags = XCL_MEM_DDR_BANK0;
  d_bank0_ext.obj = host_mem_ptr;
  d_bank0_ext.param = 0;

cl_mem dev_mem_ptr = clCreateBuffer(context,
                     CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX,
                     sizeof(int) * number_of_words, &d_bank0_ext, NULL);

err = clEnqueueMigrateMemObjects(commands, 1, dev_mem_ptr, 0, 0,
      NULL, NULL);
```

Enhanced Buffer Allocation

By default, all the memory interfaces from all the kernels are connected to a single global memory bank when kernels are linked. As a result, only one memory interface can transfer data to and from the global memory bank at a time, limiting the overall performance of the application. If the FPGA device contains only one global memory bank, this is the only option. However, if the device contains multiple global memory banks, you can customize the global memory bank connections by modifying the default connection. This topic is discussed in greater detail in Customization of DDR Bank to Kernel Connection. This improves overall performance by enabling multiple kernel memory interfaces to concurrently read and write data from separate global memory banks.

When kernel ports are mapped to memory banks other than the default one, it is necessary to use the enhanced buffer allocation pattern when creating the OpenCL buffers.

The enhanced buffer allocation pattern uses a Xilinx vendor extension, `cl_mem_ext_ptr_t`, pointer to help the Xilinx runtime determine which global memory bank the buffer should be allocated.

The `cl_mem_ext_ptr_t` type is a struct as defined below:

```
typedef struct{
      unsigned flags;
      void *obj;
      void *param;
  }
  cl_mem_ext_ptr_t;
```

Use the explicit bank name method to operate `cl_mem_ext_ptr_t` for enhanced buffer allocation.

Explicit Bank Name Method

In this approach, the struct field `flags` is used to denote the DDR bank (`XCL_MEM_DDR_BANK1`, `XCL_MEM_DDR_BANK2`, etc.). The struct field `param` should not be used and set to `NULL`.

The following code example uses `cl_mem_ext_ptr_t` to assign the device buffer to DDR Bank 2.

```
int host_mem_ptr[MAX_LENGTH]; // host memory for input vector
  // Fill the memory input
  for(int i=0; i<MAX_LENGTH; i++) {
    host_mem_ptr[i] = <... >    }

cl_mem_ext_ptr_t d_bank0_ext;
  d_bank0_ext.flags = XCL_MEM_DDR_BANK2;
  d_bank0_ext.obj = NULL;
  d_bank0_ext.param = 0;

cl_mem dev_mem_ptr = clCreateBuffer(context,
```

```
                    CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX,
                     sizeof(int) * number_of_words, &d_bank0_ext, NULL);

err = clEnqueueWriteBuffer(commands, dev_mem_ptr, CL_TRUE, 0,
      sizeof(int) * number_of_words, host_mem_ptr, 0, NULL, NULL);
```

IMPORTANT! *Starting from the 2018.3 release, the new method of specifying a bank name is:*

```
var_ext.flags =  <some integer> | XCL_MEM_TOPOLOGY
```

Where 0, 1, 2, and 3 stand for different DDR banks. However, the older naming style of `XCL_MEM_DDR_BANK0`, etc. would still work for the existing platform.

Kernel Setup and Execution

This section focuses on how a typical host application performs the following kernel related tasks in the SDAccel environment:

1. Identifying the kernels.
2. Setting kernel arguments.
3. Executing kernels on the FPGA.

Identifying the kernels

At the beginning, the individual kernels present in the `.xclbin` file should be mapped to the kernel handles (denoted by `cl_kernel` type) in the host code. This is done by the `clCreateKernel` command with the kernel name as an argument:

```
kernel1 = clCreateKernel(program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err);  // etc
```

Setting Kernel Arguments

In the SDAccelenvironment framework two types of kernel arguments can be set.

1. The scalar arguments are used for small data transfer, such as for constant, or configuration type data. These are write-only arguments.
2. The buffer arguments are used for large data transfer as discussed in Buffer Transfer to/from the FPGA Device.

The kernel arguments can be set using the `clSetKernelArg` command as shown below. The following example shows setting kernel arguments for two scalar arguments, and three buffer arguments.

```
int err = 0;
// Setting up scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scaler_arg_image_width);
cl_uint scaler_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint),
&scaler_arg_image_height);

// Setting up buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_mem_ptr0);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_mem_ptr1);
err |= clSetKernelArg(kernel, 4, sizeof(cl_mem), &dev_mem_ptr2);
```

Enqueing the Kernels

The kernel is enqueued to run on FPGA either by the `clEnqueueTask` or `clEnqueueNDRangeKernel` commands. Xilinx recommends using the `clEnqueueTask` command to execute the kernel over the entire range of input data set using the maximum number of work group items:

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

TIP: *`clEnqueueTask` is the same as calling `clEnqueueNDRangeKernel` with `work_dim` set to 1, `global_work_offset` set to NULL, `global_work_size[0]` set to 1, and `local_work_size[0]` set to 1.*

Just like all the *enqueue* commands, the `clEnqueueTask` and `clEnqueueNDRangeKernel` are asynchronous in nature. The host code continues executing without waiting for the kernel computation to complete on the FPGA device. This allows the host program to execute more kernels, either the same kernel multiple times over a different set of data, or different kernel. After finishing its work, the kernel writes the result data to the global memory bank. This data is read back to the host memory space by using `clEnqueueReadBuffer` or the `clEnqueueMigrateMemObjects` command.

Event Synchronization

All OpenCL `clEnqueueXXX` API calls are asynchronous. In other words, these commands will return immediately after the command is enqueued in the command queue. To resolve the dependencies among the commands, an API call such as `clWaitForEvents` or `clFinish` can be used to pause or block execution of the host program.

Example usage of `clWaitForEvents` and `clFinish` commands are shown below:

```
err = clEnqueueTask(command_queue, kernel, 0, NULL, NULL);

// Execution will wait here until all commands in the command queue are
finished
clFinish(command_queue);

// Read back the results from the device to verify the output
cl_event readevent;
int host_mem_output_ptr[MAX_LENGTH]; // host memory for output vector

clEnqueueReadBuffer(command_queue, dev_mem_ptr, CL_TRUE, 0, sizeof(int) *
number_of_words,
  host_mem_output_ptr, 0, NULL, &readevent );

clWaitForEvents(1, &readevent); // Wait for clEnqueueReadBuffer event to
finish

// Check Results
// Compare Golden values with host_mem_output_ptr
```

Note how the synchronization APIs have been added in the above example.

1. The `clFinish` API has been explicitly used to block the host execution until the Kernel execution is finished. This is necessary otherwise the host can attempt to read back from the FPGA buffer too early and may read garbage data.
2. The data transfer from FPGA memory to the local host machine is done through `clEnqueueReadBuffer`. Here the last argument of `clEnqueueReadBuffer` returns an event object that identifies this particular read command and can be used to query the event, or wait for this particular command to complete. The `clWaitForEvents` specifies that one event, and waits to ensure the data transfer is finished before checking the data from the host side memory.

Post Processing and FPGA Cleanup

At the final stage of the host program, it is good practice to confirm the FPGA functionality by comparing the output data from the FPGA with golden data. This action greatly helps identify and debug any issues with the kernel.

```
bool failed = false;
for (i=0; i<N; i++)
  if (Res[i] != GOLD_RES[i])
        failed=true;
```

At the end of the host code, all the allocated resources should be released by using proper release functions. The SDAccel environment may not able to generate a correct performance related profile and analysis report if resources are not properly released.

```
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```

Summary

As discussed in earlier topics, the recommended coding style for the host application in the SDAccel environment includes the following points:

1. Add error checking after each OpenCL API call for debugging purpose, if required.
2. In the SDAccel environment, one or more kernels are separately pre-compiled to the `.xclbin` file. The API `clCreateProgramWithBinary` is used to build the program from the kernel binary.
3. Ensure using `cl_mem_ext_ptr_t` to match custom kernel memory interface to the DDR bank connection that has been used to build the kernel binary.
4. Transfer data back and forth from the host code to the FPGAs by using `clEnqueueMigrateMemObjects`.
5. Use `posix_memalign` to align the host memory pointer at 4K boundary.
6. Use the out-of-order command queue, or multiple in-order command queues, for concurrent kernel execution on the FPGA.
7. Execute the whole workload with `clEnqueTask`, rather than splitting the workload by using `clEnqueueNDRangeKernel`.
8. Use synchronization commands to resolve dependencies of the asynchronous OpenCL API calls.

Programming C/C++ Kernels

In the SDAccel™ environment, the kernel code is generally a compute-intensive part of the algorithm and meant to be accelerated on the FPGA. The SDAccel environment supports the kernel code written in C, OpenCL™, and also in RTL. This guide mainly focuses on the C kernel coding style.

During the runtime, the C/C++ kernel executable is called through the host code executable. As the host code and the kernel code are developed and compiled independently, there could be a name mangling issue if one of the code is written in C and another in C++. To avoid this issue, it is good practice to put the kernel function declaration inside a header file wrapped around the `extern "C"` linkage.

```
extern "C" {
          void kernel_function(int *in, int *out, int size);
        }
```

Data Types

As it is faster to write and verify the code by using native C data types such as `int`, `float`, or `double`, it is a common practice to use these data types when coding for the first time. However, the code is implemented in hardware, and all the operator sizes used in the hardware are dependent on the data types used in the accelerator code. The default native C/C++ data types can result in larger and slower hardware resources that can limit the performance of the kernel. Instead, consider using bit-accurate data types to ensure the code is optimized for implementation in hardware. Using bit-accurate, or arbitrary precision data types, results in hardware operators which are smaller and faster. This allows more logic to be placed into the programmable logic and also allows the logic to execute at higher clock frequencies while using less power.

Consider using bit-accurate data types instead of native C/C++ data types in your code.

RECOMMENDED: *Consider using bit-accurate data types instead of native C/C++ data types in your code.*

In the following sections, the two most common arbitrary precision data types (arbitrary precision integer type and arbitrary precision fixed-point type) supported by the `xocc` compiler are discussed. Note that these data types should be used for C/C++ kernels only, not for OpenCL kernel (or inside the host code).

Arbitrary Precision Integer Types

Arbitrary precision integer data types are defined by `ap_int` or `ap_uint` for signed and unsigned integer respectively inside the header file `ap_int.h`. To use arbitrary precision integer data type:

- Add header file `ap_int.h` to the source code.
- Change the bit types to `ap_int<N>` or `ap_uint<N>`, where N is a bit-size from 1 to 1024.

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer.

```
#include “ap_int.h”
ap_int<9> var1 // 9 bit signed integer
ap_uint<10> var2 // 10 bit unsigned integer
```

Arbitrary Precision Fixed-Point Data Types

Some existing applications use floating point data types as they are written for other hardware architectures. However, fixed-point data types are a useful replacement for floating point types which require many clock cycles to complete. Carefully evaluate trade-offs in power, cost, productivity, and precision when choosing to implement floating-point vs. fixed-point arithmetic for your application and accelerators.

As discussed in *Deep Learning with INT8 Optimization on Xilinx Devices* (WP486), using fixed-point arithmetic instead of floating point for applications like machine learning can increase power efficiency, and lower the total power required. Unless the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type resulting in the same accuracy with smaller and faster hardware. The paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* (WP491) provides some examples of this conversion.

Fixed-point data types model the data as an integer and fraction bits. The fixed-point data type requires the `ap_fixed` header, and supports both a signed and unsigned form as follows:

- Header file: `ap_fixed.h`
- Signed fixed point: `ap_fixed<W,I,Q,O,N>`
- Unsigned fixed point: `ap_ufixed<W,I,Q,O,N>`
  - W = Total width < 1024 bits

- I = Integer bit width. The value of I must be less than or equal to the width (W). The number of bits to represent the fractional part is W minus I. Only a constant integer expression can be used to specify the integer width.
- Q = Quantization mode. Only predefined enumerated values can be used to specify Q. The accepted values are:
  - `AP_RND`: Rounding to plus infinity.
  - `AP_RND_ZERO`: Rounding to zero.
  - `AP_RND_MIN_INF`: Rounding to minus infinity.
  - `AP_RND_INF`: Rounding to infinity.
  - `AP_RND_CONV`: Convergent rounding.
  - `AP_TRN`: Truncation. This is the default value when Q is not specified.
  - `AP_TRN_ZERO`: Truncation to zero.
- O = Overflow mode. Only predefined enumerated values can be used to specify O. The accepted values are:
  - `AP_SAT`: Saturation.
  - `AP_SAT_ZERO`: Saturation to zero.
  - `AP_SAT_SYM`: Symmetrical saturation.
  - `AP_WRAP`: Wrap-around. This is the default value when O is not specified.
  - `AP_WRAP_SM`: Sign magnitude wrap-around.
- N = The number of saturation bits in the overflow WRAP modes. Only a constant integer expression can be used as the parameter value. The default value is zero.

TIP: *The `ap_fixed` and `ap_ufixed` data types permit shorthand definition, with only W and I being required, and other parameters assigned default values. However, to define Q or N, you must also specify the parameters before those, even if you just specify the default values.*

In the example code below, the `ap_fixed` type is used to define a signed 18-bit variable with 6 bits representing the integer value above the binary point, and by implication, 12 bits representing the fractional value below the binary point. The quantization mode is set to round to plus infinity (`AP_RND`). Because the overflow mode and saturation bits are not specified, the defaults `AP_WRAP` and 0 are used.

```
#include <ap_fixed.h>
...
  ap_fixed<18,6,AP_RND> my_type;
...
```

When performing calculations where the variables have different numbers of bits (W), or different precision (I), the binary point is automatically aligned. See the "C++ Arbitrary Precision Fixed-Point Types" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on using fixed-point data types.

Interfaces

Two types of data transfer occur from the host machine to and from the kernels on the FPGA device: data transferred through the global memory memory banks, and scalar data.

Memory Data Inputs and Outputs

The main data processed by the kernel computation, often in a large volume, should be transferred through the global memory banks on the FPGA board. The host machine transfers a large chunk of data to one or more global memory bank(s). The kernel accesses the data from those global memory banks, preferably in burst. After the kernel finishes the computation, the resulting data is transferred back to the host machine through the global memory banks.

When writing the kernel interface description, pragmas are used to denote the interfaces coming to and from the global memory banks.

Memory Data from One Global Memory Bank

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

In the example above, there are three large data interfaces. The two inputs are `pixel` and `weights` and one output, `out`. These inputs and outputs connected to the global memory bank are specified in C code by using `HLS INTERFACE m_axi` pragmas as shown above.

The `bundle` keyword specifies the same name `gmem` to indicate that all these memory interfaces should be connected to the same global memory bank (denoted by an arbitrary string `gmem`).

Memory Data from Multiple Global Memory Banks

If you want to connect multiple global memory bank interfaces, then different bundle names should be used.

```
void cnn( int *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  int *out, // Output pixel
  ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

In the above example, you want to read the two inputs, `pixel` and `weights`, in parallel. As discussed in Buffer Transfer to/from the FPGA Device, it would prevent the data from being read concurrently if the inputs are attached to the same global memory bank. To connect `weights` to a different global memory bank, a different bundle name is needed, and `gmem1` is used.

However, the pragma is only used for the kernel compilation. At the linking stage the same memory connection information should be provided so that actual hardware connection is made from kernel to global memory bank. This is done by the `--sp` compiler linking switch which is described in detail in the Configuring the System Architecture.

Memory Interface Data Width Considerations

In the SDAccel environment, the maximum data width from the global memory to and from the kernel is 512 bits. To maximize the data transfer rate, using this full data width is recommended. The kernel code should be modified to take advantage of the full bit width.

Because a native integer type is used in the prior example, the full data transfer bandwidth is not used. As discussed previously in Data Types, arbitrary precision data types `ap_int` or `ap_uint` can be used to achieve bit-accurate data width for this purpose.

```
void cnn( ap_uint<512> *pixel, // Input pixel
  int *weights, // Input Weight Matrix
  ap_uint<512> *out, // Output pixel
  ... // Other input or output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

The example above shows the output (`out`) interface using the `ap_uint` data type to make use of the full transfer width of 512 bits.

The data width of the memory interfaces should be a power of 2. For data width less than 32, use native C data types. Use `ap_int/ap_uint` for data widths greater than 32 and with power of 2 increment.

Reading and Writing by Burst

Accessing the global memory bank interface from the kernel has a large latency. So global memory data transfer should be done in burst. To infer the burst, a pipelined loop coding style is recommended as shown below:

```
hls::stream<datatype_t> str;

INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
  #pragma HLS PIPELINE
  str.write(inp[i]); // Reading from Input interface
}
```

In the above code example, a pipelined `for` loop is used to read data from the input memory interface, and writes to an internal `hls::stream` variable. The above coding style reads from the global memory bank in burst.

It is a recommended coding style to implement the `for` loop operation in the example above inside a separate function, and apply the `dataflow` optimization from the top level as shown below:

```
top_function(datatype_t * m_in, // Memory data Input
  datatype_t * m_out, // Memory data Output
  int inp1,     // Other Input
  int inp2) {   // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1;   // Internal stream to transfer
hls::stream<datatype_t> out_var1;  // data through the dataflow region

read_function(m_in, inp1); // Read function contains pipelined for loop
                           // to infer burst

execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

write_function(out_var1, m_out); // Write function contains pipelined for
loop
                                 // to infer burst
}
```

TIP: *The Dataflow Optimization is discussed in a later topic.*

Scalar Data Inputs

Scalar data are typically small control variables that are directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side. These interfaces are specified in the kernel code as shown below:

```
void process_image(int *input, int *output, int width, int height) {
  #pragma HLS INTERFACE s_axilite port=width bundle=control
  #pragma HLS INTERFACE s_axilite port=height bundle=control
```

In the example above, there are two scalar inputs specify the image `width` and `height`. These inputs are specified using the `#pragma HLS INTERFACE s_axilite`. These data inputs come to the kernel directly from the host machine and not using global memory bank.

IMPORTANT! *Currently, the SDAccel environment supports one and only one control interface bundle for each kernel. Hence the `bundle` name should be same for all scalar data inputs. In the preceding example the same `bundle` name, `control`, is used for all control inputs.*

Loops

Loops are an important aspect for a high performance accelerator. Generally, loops are either pipelined or unrolled to take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU.

By default, loops are neither pipelined nor unrolled. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit *wait until clock* for the loop body. The next iteration of a loop only starts when the previous iteration is finished.

Loop Pipelining

By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables and stores the result in a third variable. Assume that in hardware this loop takes three cycles to finish one iteration. Also, assume that the loop variable `len` is 20, that is, the `vadd` loop runs for 20 iterations in the kernel. Therefore, it requires a total of 60 clock cycles (20 iterations * 3 cycles) to complete all the operations of this loop.

```
vadd: for(int i = 0; i < len; i++) {
  c[i] = a[i] + b[i];
}
```

TIP: *It is good practice to always label a loop as shown in the above code example (`vadd:...`). This practice helps with debugging when working in the SDAccel environment. Note that the labels generate warnings during compilation, which can be safely ignored.*

Pipelining the loop executes subsequent iterations in a pipelined manner. This means that subsequent iterations of the loop overlap and run concurrently, executing at different sections of the loop-body. Pipelining a loop can be enabled by the pragma `HLS PIPELINE`. Note that the pragma is placed inside the body of the loop.

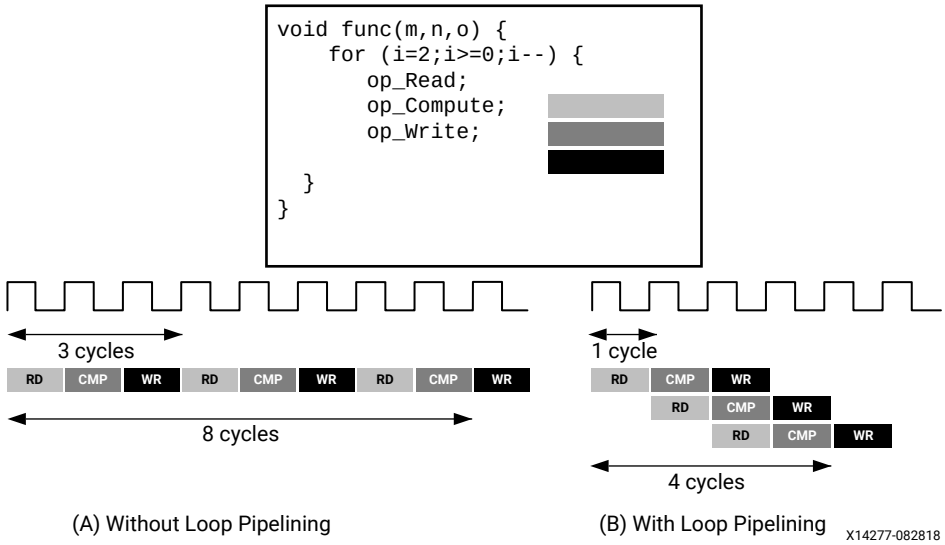
```
vadd: for(int i = 0; i < len; i++) {
  #pragma HLS PIPELINE
  c[i] = a[i] + b[i];
}
```

In the example above, it is assumed that every iteration of the loop takes three cycles: read, add, and write. Without pipelining, each successive iteration of the loop starts in every third cycle. With pipelining the loop can start subsequent iterations of the loop in fewer than three cycles, such as in every second cycle, or in every cycle.

The number of cycles it takes to start the next iteration of a loop is called the initiation interval (II) of the pipelined loop. So II = 2 means each successive iteration of the loop starts every two cycles. An II = 1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use the `pragma HLS PIPELINE` the compiler always tries to achieve II = 1 performance.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II = 3), and it requires eight clock cycles before the last output write is performed.

Figure 6: **Loop Pipelining**



In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II = 1) and the final output is written after only four clock cycles: substantially improving both the II and latency while using the same hardware resources.

IMPORTANT! *Pipelining a loop causes any loops nested inside the pipelined loop to get unrolled.*

If there are data dependencies inside a loop it might not be possible to achieve II = 1, and a larger initiation interval might be the result. Loop dependencies are discussed in Loop Dependencies.

Loop Unrolling

The compiler can also unroll a loop, either partially or completely to perform multiple loop iterations in parallel. This is done using the `HLS UNROLL` pragma. Unrolling a loop can lead to a very fast design, with significant parallelism. However, because all the operations of the loop iterations are executed in parallel, a large amount of programmable logic resource are required to implement the hardware. As a result, the compiler can face challenges dealing with such a large number of resources and can face capacity problems that slow down the kernel compilation process. It is a good guideline to unroll loops that have a small loop body, or a small number of iterations.

```
vadd: for(int i = 0; i < 20; i++) {
  #pragma HLS UNROLL
  c[i] = a[i] + b[i];
}
```

In the preceding example, you can see `pragma HLS UNROLL` has been inserted into the body of the loop to instruct the compiler to unroll the loop completely. In other words, all 20 iterations of the loop are executed in parallel if that is permitted by any data dependency.

Completely unrolling a loop can consume significant device resources, while partially unrolling the loop provides some performance improvement without causing a significant impact on hardware resources.

Partially Unrolled Loop

To completely unroll a loop, the loop must have a constant bound (20 in the example above). However, partial unrolling is possible for loops with a variable bound. A partially unrolled loop means that only a certain number of loop iterations can be executed in parallel.

The following code examples illustrates how partially unrolled loops work:

```
array_sum:for(int i=0;i<4;i++){
  #pragma HLS UNROLL factor=2
  sum += arr[i];
}
```

In the above example the `UNROLL` pragma is given a factor of 2. This is the equivalent of manually duplicating the loop body and running the two loops concurrently for half as many iterations. The following code shows how this would be written. This transformation allows two iterations of the above loop to execute in parallel.

```
array_sum_unrolled:for(int i=0;i<2;i+=2){
  // Manual unroll by a factor 2
  sum += arr[i];
  sum += arr[i+1];
}
```

Just like data dependencies inside a loop impact the initiation interval of a pipelined loop, an unrolled loop performs operations in parallel only if data dependencies allow it. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel, but execute as soon as the data from one iteration is available to the next.

RECOMMENDED: *A good methodology is to `PIPELINE` loops first, and then `UNROLL` loops with small loop bodies and limited iterations to improve performance further.*

Loop Dependencies

Data dependencies in loops can impact the results of loop pipelining or unrolling. These loop dependencies can be within a single iteration of a loop or between different iterations of a loop. The straightforward method to understand loop dependencies is to examine an extreme example. In the following code example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

```
Minim_Loop: while (a != b) {
  if (a > b)
    a -= b;
  else
    b -= a;
}
```

This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.

Dealing with various types of dependencies with the `xocc` compiler is an extensive topic requiring a detailed understanding of the high-level synthesis procedures underlying the compiler. Refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on "Dependencies with Vivado HLS."

Nested Loops

Coding with nested loops is a common practice. Understanding how loops are pipelined in a nested loop structure is key to achieving the desired performance.

If the pragma `HLS PIPELINE` is applied to a loop nested inside another loop, the xocc compiler attempts to flatten the loops to create a single loop, and apply the `PIPELINE` pragma to the constructed loop. The loop flattening helps in improving the performance of the kernel.

The compiler is able to flatten the following types of nested loops:

1. Perfect nested loop:
   - Only the inner loop has a loop body.
   - There is no logic or operations specified between the loop declarations.
   - All the loop bounds are constant.
2. Semi-perfect nested loop:
   - Only the inner loop has a loop body.
   - There is no logic or operations specified between the loop declarations.
   - The inner loop bound must constant, but the outer loop bound can be variable.

The following code example illustrates the structure of a perfect nested loop:

```
ROW_LOOP: for(int i=0; i< MAX_HEIGHT; i++) {
  COL_LOOP: For(int j=0; j< MAX_WIDTH; j++) {
    #pragma HLS PIPELINE
    // Main computation per pixel
  }
}
```

The above example shows a nested loop structure with two loops that performs some computation on incoming pixel data. In most cases, you want to process a pixel in every cycle, hence `PIPELINE` is applied to the nested loop body structure. The compiler is able to flatten the nested loop structure in the example because it is a perfect nested loop.

The nested loop in the preceding example contains no logic between the two loop declarations. No logic is placed between the `ROW_LOOP` and `COL_LOOP`; all of the processing logic is inside the `COL_LOOP`. Also, both the loops have a fixed number of iterations. These two criteria help the xocc compiler flatten the loops and apply the `PIPELINE` constraint.

RECOMMENDED: *If the outer loop has a variable boundary, then the compiler can still flatten the loop. You should always try to have a constant boundary for the inner loop.*

Sequential Loops

If there are multiple loops in the design, by default they do not overlap, and execute sequentially. This section introduces the concept of dataflow optimization for sequential loops. Consider the following code example:

```
void adder(unsigned int *in, unsigned int *out, int inc, int size) {

  unsigned int in_internal[MAX_SIZE];
  unsigned int out_internal[MAX_SIZE];
  mem_rd: for (int i = 0 ; i < size ; i++){
    #pragma HLS PIPELINE
    // Reading from the input vector "in" and saving to internal variable
    in_internal[i] = in[i];
  }
  compute: for (int i=0; i<size; i++) {
  #pragma HLS PIPELINE
    out_internal[i] = in_internal[i] + inc;
  }

  mem_wr: for(int i=0; i<size; i++) {
  #pragma HLS PIPELINE
    out[i] = out_internal[i];
  }
}
```

In the previous example, three sequential loops are shown: `mem_rd`, `compute`, and `mem_wr`.

- The `mem_rd` loop reads input vector data from the memory interface and stores it in internal storage.
- The main `compute` loop reads from the internal storage and performs an increment operation and saves the result to another internal storage.
- The `mem_wr` loop writes the data back to memory from the internal storage.

As described in , this code example is using two separate loops for reading and writing from/to the memory input/output interfaces to infer burst read/write.

By default, these loops are executed sequentially without any overlap. First, the `mem_rd` loop finishes reading all the input data before the `compute` loop starts its operation. Similarly, the `compute` loop finishes processing the data before the `mem_wr` loop starts to write the data. However, the execution of these loops can be overlapped, allowing the `compute` (or `mem_wr`) loop to start as soon as there is enough data available to feed its operation, before the `mem_rd` (or `compute`) loop has finished processing its data.

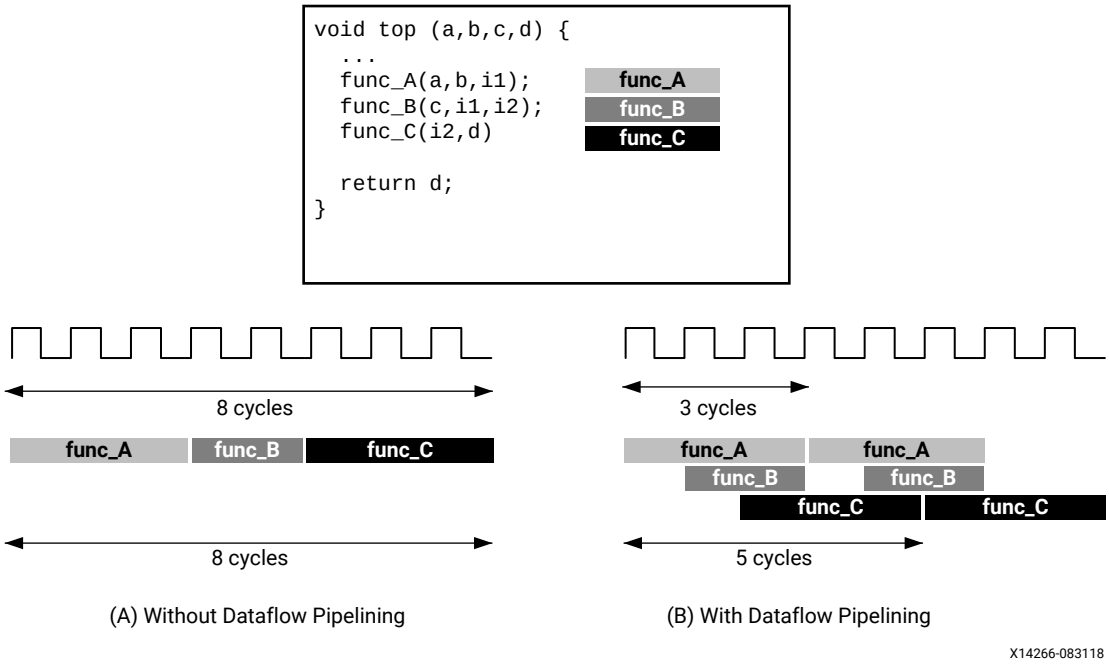
The loop execution can be overlapped using dataflow optimization as described in Dataflow Optimization.

Dataflow Optimization

Dataflow optimization is a powerful technique to improve the kernel performance by enabling tasklevel pipelining and parallelism inside the kernel. It allows the xocc compiler to schedule multiple functions of the kernel to run concurrently to achieve higher throughput and lower latency. This is also known as task-level parallelism.

The following figure shows a conceptual view of dataflow pipelining. The default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. With the `HLS DATAFLOW` pragma enabled, the compiler can schedule each function to execute as soon as data is available. In this example, the original `top` function has a latency and interval of eight clock cycles. With `DATAFLOW` optimization, the interval is reduced to only three clock cycles.

Figure 7: **Dataflow Optimization**



Dataflow Coding Example

In the dataflow coding example you should notice the following:

1. The `HLS DATAFLOW` pragma is applied to instruct the compiler to enable dataflow optimization. This is not a data mover, which deals with interfacing between the PS and PL, but how the data flows through the accelerator.
2. The `stream` class is used as a data transferring channel between each of the functions in the dataflow region.

TIP: *The `stream` class infers a first-in first-out (FIFO) memory circuit in the programmable logic. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the functions and achieves better performance. For additional details on the `hls::stream` class, see the Vivado Design Suite User Guide: High-Level Synthesis (UG902).*

```
void compute_kernel(ap_int<256> *inx, ap_int<256> *outx, DTYPE alpha) {
  hls::stream<unsigned int>inFifo;
  #pragma HLS STREAM variable=inFifo depth=32
  hls::stream<unsigned int>outFifo;
  #pragma HLS STREAM variable=outFifo depth=32

  #pragma HLS DATAFLOW
  read_data(inx, inFifo);
  // Do computation with the acquired data
  compute(inFifo, outFifo, alpha);
  write_data(outx, outFifo);
  return;
}
```

Canonical Forms of Dataflow Optimization

Xilinx recommends writing the code inside a dataflow region using canonical forms. There are canonical forms for dataflow optimizations for both functions and loops.

- Functions: The canonical form coding guideline for dataflow inside a function specifies:
  1. Use only the following types of variables inside the dataflow region:
     a. Local non-static scalar/array/pointer variables.
     b. Local static `hls::stream` variables.
  2. Function calls transfer data only in the forward direction.
  3. Array or `hls::stream` should have only one producer function and one consumer function.
  4. The function arguments (variables coming from outside the dataflow region) should only be read, or written, not both. If performing both read and write on the same function argument then read should happen before write.
  5. The local variables (those that are transferring data in forward direction) should be written before being read.

The following code example illustrates the canonical form for dataflow within a function. Note that the first function (`func1`) reads the inputs and the last function (`func3`) writes the outputs. Also note that one function creates output values that are passed to the next function as input parameters.

```
void dataflow(Input0, Input1, Output0, Output1) {
  UserDataType C0, C1, C2;
  #pragma HLS DATAFLOW
  func1(read Input0, read Input1, write C0, write C1);
  func2(read C0, read C1, write C2);
  func3(read C2, write Output0, write Output1);
}
```

- Loop: The canonical form coding guideline for dataflow inside a loop body includes the coding guidelines for a function defined above, and also specifies the following:

  1. Initial value 0.
  2. The loop condition is formed by a comparison of the loop variable with a numerical constant or variable that does not vary inside the loop body.
  3. Increment by 1.

  The following code example illustrates the canonical form for dataflow within a loop.

```
void dataflow(Input0, Input1, Output0, Output1) {
             UserDataType C0, C1, C2;
             for (int i = 0; i < N; ++i) {
                  #pragma HLS DATAFLOW
                 func1(read Input0, read Input1, write C0, write C1);
                 func2(read C0, read C0, read C1, write C2);
                 func3(read C2, write Output0, write Output1);
               }
}
```

Troubleshooting Dataflow

The following behaviors can prevent the xocc compiler from performing `DATAFLOW` optimizations:

1. Single producer-consumer violations.
2. Bypassing tasks.
3. Feedback between tasks.
4. Conditional execution of tasks.
5. Loops with multiple exit conditions or conditions defined within the loop.

If any of the above conditions occur inside the dataflow region, you might need to re-architect the code to successfully achieve dataflow optimization.

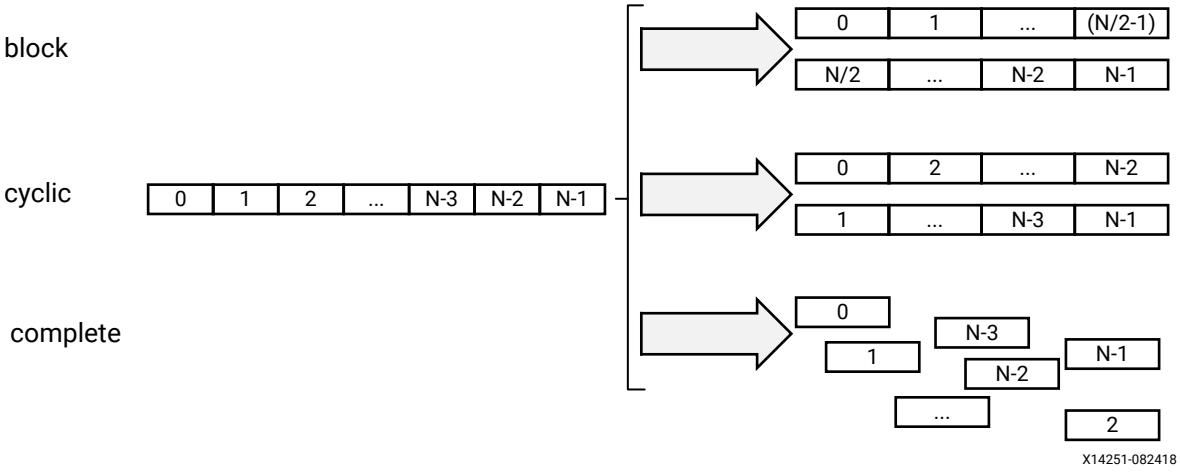
Array Configuration

The SDAccelcompiler maps large arrays to the memory (known as block RAM or BRAM) in the PL region. These BRAM can have a maximum of two access points or ports. This can limit the performance of the application as all the elements of an array cannot be accessed in parallel when implemented in hardware.

Depending on the performance requirements, you might need to access some or all of the elements of an array in the same clock cycle. To achieve this, the `#pragma HLS ARRAY_PARTITION` can be used to instruct the compiler to split the elements of an array and map it to smaller arrays, or to individual registers. The compiler provides three types of array partitioning, as shown in the following figure. The three types of partitioning are:

- `block`: The original array is split into equally sized blocks of consecutive elements of the original array.
- `cyclic`: The original array is split into equally sized blocks interleaving the elements of the original array.
- `complete`: Split the array into its individual elements. This corresponds to resolving a memory into individual registers. This is the default for the `ARRAY_PARTITION` pragma.

Figure 8: **Partitioning Arrays**

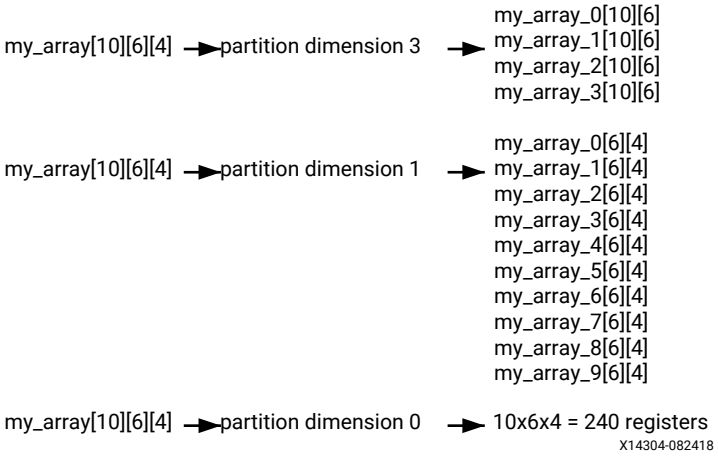


For block and cyclic partitioning, the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used to split the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the later arrays will have fewer elements.

When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code in three different ways:

```
void foo (...) {
  // my_array[dim=1][dim=2][dim=3]
  // The following three pragma results are shown in the figure below
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=3 <block|cyclic>
factor=2
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=1 <block|cyclic>
factor=2
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=0 complete
  int  my_array[10][6][4];
  ...
}
```

Figure 9: **Partitioning the Dimensions of an Array**



The examples in the figure demonstrate how partitioning dimension 3 results in four separate arrays and partitioning dimension 1 results in 10 separate arrays. If 0 is specified as the dimension, all dimensions are partitioned.

The Importance of Careful Partitioning

A complete partition of the array maps all the array elements to the individual registers. This helps in improving the kernel performance because all of these registers can be accessed concurrently in a same cycle.

CAUTION! *Complete partitioning of the large arrays consumes a lot of PL region. It could even cause the compilation process to slow down and face capacity issue. Partition the array only when it is needed. Consider selectively partitioning a particular dimension or performing a block or cycle partitioning.*

Choosing a Specific Dimension to Partition

Suppose A and B are two-dimensional arrays representing two matrices. Consider the following Matrix Multiplication algorithm:

```
int A[64][64];
int B[64][64];

ROW_WISE: for (int i = 0; i < 64; i++) {
  COL_WISE : for (int j = 0; j < 64; j++) {
    #pragma HLS PIPELINE
    int result = 0;
    COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
      result += A[i ][ k] * B[k ][ j];
    }
    C[i][ j] = result;
  }
}
```

Due to the PIPELINE pragma, the `ROW_WISE` and `COL_WISE` loop is flattened together and `COMPUTE_LOOP` is fully unrolled. To concurrently execute each iteration (k) of the `COMPUTE_LOOP`, the code must access each column of matrix A and each row of matrix B in parallel. Therefore, the matrix A should be split in the second dimension, and matrix B should be split in the first dimension.

```
#pragma HLS ARRAY_PARTITION variable=A dim=2 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
```

Choosing Between Cyclic and Block Partitions

Here the same matrix multiplication algorithm is used to demonstrate choosing between cyclic and block partitioning and determining the appropriate factor, by understanding the array access pattern of the underlying algorithm.

```
int A[64 * 64];
int B[64 * 64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64

ROW_WISE: for (int i = 0; i < 64; i++) {
  COL_WISE : for (int j = 0; j < 64; j++) {
    #pragma HLS PIPELINE
    int result = 0;
    COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
      result += A[i * 64 +  k] * B[k * 64 + j];
    }
    C[i* 64 + j] = result;
  }
}
```

In this version of the code, A and B are now one-dimensional arrays. To access each column of matrix A and each row of matrix B in parallel, cyclic and block partitions are used as shown in the above example. To access each column of matrix A in parallel, `cyclic` partitioning is applied with the `factor` specified as the row size, in this case 64. Similarly, to access each row of matrix B in parallel, `block` partitioning is applied with the `factor` specified as the column size, or 64.

Minimizing Array Accesses with Caching

As arrays are mapped to BRAM with limited number of access ports, repeated array accesses can limit the performance of the accelerator. You should have a good understanding of the array access pattern of the algorithm, and limit the array accesses by locally caching the data to improve the performance of the kernel.

The following code example shows a case in which accesses to an array can limit performance in the final implementation. In this example, there are three accesses to the array `mem[N]` to create a summed result.

```
#include "array_mem_bottleneck.h"
dout_t array_mem_bottleneck(din_t mem[N]) {
  dout_t sum=0;
  int i;
  SUM_LOOP:for(i=2;i<N;++i)
    sum += mem[i] + mem[i-1] + mem[i-2];
  return sum;
}
```

The code in the preceding example can be rewritten as shown in the following example to allow the code to be pipelined with a II = 1. By performing pre-reads and manually pipelining the data accesses, there is only one array read specified inside each iteration of the loop. This ensures that only a single-port BRAM is needed to achieve the performance.

```
#include "array_mem_perform.h"
dout_t array_mem_perform(din_t mem[N]) {
  din_t tmp0, tmp1, tmp2;
  dout_t sum=0;
  int i;
  tmp0 = mem[0];
  tmp1 = mem[1];
  SUM_LOOP:for (i = 2; i < N; i++) {
    tmp2 = mem[i];
    sum += tmp2 + tmp1 + tmp0;
    tmp0 = tmp1;
    tmp1 = tmp2;
  }
  return sum;
}
```

RECOMMENDED: *Consider minimizing the array access by caching to local registers to improve the pipelining performance depending on the algorithm.*

For more detailed information related to the configuration of arrays, see the "Arrays" section in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Function Inlining

C code generally consists of several functions. By default, each function is compiled, and optimized separately by the `xocc` compiler. A unique hardware module will be generated for the function body and reused as needed.

From performance perspective, in general it is better to inline the function, or in other words dissolve the function hierarchy. This helps `xocc` compiler to do optimization more globally across the function boundary. For example, if a function is called inside a pipelined loop, then inlining the function helps the compiler to do more aggressive optimization and results in a better pipeline performance of the loop (lower initiation interval or II number).

The following `INLINE` pragma placed inside the function body instruct the compiler to inline the function.

```
foo_sub (p, q) {
  #pragma HLS INLINE
  ....
  ...
}
```

However, if the function body is very big and called several times inside the main kernel function, then inlining the function may cause capacity issues due to too many resources. In cases like that you might not want to inline such functions, and let the `xocc` compiler optimize the function separately in its local context.

Summary

As discussed in earlier topics, several important aspects of coding the kernel for FPGA acceleration using C/C++ include the following points:

1. Consider using arbitrary precision datatypes, `ap_int`, and `ap_fixed`.
2. Understand kernel interfaces to determine scaler and memory interfaces. Use `bundle` switch with different names if separate DDR memory banks will be specified in the linking stage.
3. Use Burst read and write coding style from and to the memory interface.
4. Consider exploiting the full width of DDR banks during the data transfer when selecting width of memory data inputs and outputs.
5. Get the greatest performance boost using pipelining and dataflow.
6. Write perfect or semi-perfect nested loop structure so that the `xocc` compiler can flatten and apply pipeline effectively.
7. Unroll loops with a small number of iterations and low operation count inside the loop body.

8. Consider understanding the array access pattern and apply `complete` partition to specific dimensions or apply `block` or `cyclic` partitioning instead of a `complete` partition of the whole array.
9. Minimize the array access by using local cache to improve kernel performance.
10. Consider inlining the function, specifically inside the pipelined region. Functions inside the dataflow should not be inlined.

Configuring the System Architecture

In Chapter 1: SDAccel Compilation Flow and Execution Model, you learned of the two distinct phases in the SDAccel™ environment kernel build process:

1. Compilation stage: The compilation process is controlled by the `xocc -c` option. At the end of the compilation stage one or more kernel functions are compiled into separate `.xo` files. At this stage, the `xocc` compiler extracts the hardware intent from the C/C++ code and associated pragmas. Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information on the `xocc` compiler.
2. Linking stage: The linking stage is controlled by the `xocc -l` option. During the linking process all the `.xo` files are integrated into the FPGA hardware.

If needed, the kernel linking process can be customized to improve the SDAccel environment runtime performance. This chapter introduces a few such techniques.

Multiple Instances of a Kernel

By default, a single hardware instance is implemented from a kernel. If the host intends to execute the same kernel multiple times, then multiple such kernel executions take place on the same hardware instance sequentially. However, you can customize the kernel compilation (linking stage) to create multiple hardware instances from a single kernel. This can improve execution performance as the multiple kernel calls can now run concurrently, overlapping their execution while running on separate hardware instances.

Multiple instances of the kernel can be created by using the `xocc --nk` switch during linking.

For example, for a kernel name `foo`, two hardware instances can be implemented as follows:

```
# xocc --nk <kernel name>:<number of instance>
xocc --nk foo:2
```

By default, the implemented instance names are `<kernel_name>_1` and `<kernel_name>_2`. However, you can optionally change the default instance names as shown below:

```
# xocc --nk <kernel name>:<no of instance>:<name 1>.<name 2>…<name N>
xocc --nk foo:3:fooA.fooB.fooC
```

This example implements three identical copies, or hardware instances of kernel `foo`, named `fooA`, `fooB`, and `fooC` on the FPGA programmable logic.

Customization of DDR Bank to Kernel Connection

By default, all the memory interfaces from all the kernels are connected to a single global memory bank. As a result, only one memory interface at a time can transfer data to and from the memory bank, limiting the performance of the kernel. If the FPGA contains only one DDR (or global) memory bank, this is the only option.

However, some FPGA devices contain multiple DDR memory banks. You can customize the connections among the kernel memory interfaces and the DDR memory bank of such a device by altering the default connection.

The above approach can even improve the performance of a single kernel.

Consider the following example:

```
void cnn( int *image, // Read-Only Image
  int *weights, // Read-Only Weight Matrix
  int *out, // Output Filters/Images
  ... // Other input or Output ports

  #pragma HLS INTERFACE m_axi port=image offset=slave bundle=gmem
  #pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
  #pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

The example shows two memory interface inputs for the kernel: `image` and `weights`. If both are connected to the same DDR bank, a concurrent transfer of both of these inputs into the kernel is not possible.

The following steps are needed to implement separate DDR bank connections for the `image` and `weights` inputs:

1. Specify separate bundle names for these inputs. This is discussed in Memory Data Inputs and Outputs. However, for reference the code is shown here again.

```
void cnn( int *image, // Read-Only Image
 int *weights, // Read-Only Weight Matrix
 int *out, // Output Filters/Images
 ... // Other input or Output ports

 #pragma HLS INTERFACE m_axi port=image offset=slave bundle=gmem
 #pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
 #pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

IMPORTANT! *When specifying a `bundle=` name, you should use all lowercase characters to be able to assign it to a specific memory bank using the `--sp` option.*

   The memory interface inputs `image` and `weights` are assigned different bundle names in the example above.

2. Alter the XOCC link process to create custom DDR bank connections. This is done using `--sp` switch:

```
--sp <kernel_instance_name>.<interface_name>:<bank name>
```

   Where:

   - `<kernel_instance_name>` is the instance name of the kernel as specified by the `--nk` option, described in Multiple Instances of a Kernel.
   - `<interface_name>` is the name of the interface bundle defined by the HLS INTERFACE pragma, including `m_axi_` as a prefix, and the `bundle=` name when specified.

TIP: *If the port is not specified as part of a bundle, then the `<interface_name>` is the specified `port=` name, without the `m_axi_` prefix.*

   - `<bank_name>` is denoted as `bank0`, `bank1`, etc. For a device with four DDR banks, the bank names are `bank0`, `bank1`, `bank2`, and `bank3`.

   For the above example, considering a single instance of the `cnn` kernel, the `--sp` switch can be specified as follows:

```
--sp cnn_1.m_axi_gmem:bank0  \
        --sp cnn_1.m_axi_gmem1:bank1
```

The customized bank connection needs to be reflected in the host code as well. This was previously discussed in Specifying Exact Memory from the Host Code.

Note: You can choose not to customize the automatic memory mapping by the tool as long as the total number of memory interfaces is less than 15. If there are more than 15 memory interfaces, then you must explicitly perform the memory mapping as described in this chapter.

If `-nk` and `-sp` switches are used together for a kernel, each hardware instance should have identical memory connectivity. If not, you should use the OpenCL™ sub-device to allocate each kernel instances separately in the host code.

Summary

This section discusses two powerful ways to customize the kernel compilation to improve the system performance during execution.

1. Consider creating multiple instances of a kernel on the fabric of the FPGA by specifying the `xocc --nk` if the kernel is called multiple times from the host code.
2. Consider using the `xocc --sp` switch to customize the DDR bank connection to kernel memory interfaces to achieve concurrent access.

Depending on the host and kernel design, these options can be exploited to improve the kernel acceleration on Xilinx® FPGAs.

XILINX

Appendix A

OpenCL Installable Client Driver Loader

A system can have multiple OpenCL™ platforms, each with its own driver and OpenCL version. The SDAccel™ environment supports the OpenCL Installable Client Driver (ICD) extension (`cl_khr_icd`). This extension allows multiple implementations of OpenCL to co-exist on the same system. The ICD Loader acts as a supervisor for all installed platforms, and provides a standard handler for all API calls.

Applications can choose an OpenCL platform from the list of installed platforms. Based on the platform ID specified by the application, the ICD dispatches the OpenCL host calls to the right runtime.

Xilinx does not provide the OpenCL ICD library, so the following should be used to install the library on your preferred system.

Ubuntu

On Ubuntu the ICD library is packaged with the distribution. Install the following packages:

- ocl-icd-libopencl1
- opencl-headers
- ocl-icd-opencl-dev

Linux

For RHEL/CentOS 7.X use EPEL 7, install the following packages:

- ocl-icd
- ocl-icd-devel
- opencl-headers

XILINX

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoC™ and SDAccel™ development environments. To open it:

- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

1. *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
4. *SDAccel Environment Getting Started Tutorial* (UG1021)
5. SDAccel™ Development Environment web page
6. Vivado® Design Suite Documentation
7. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
8. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
9. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
10. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
11. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
12. *Vivado Design Suite Properties Reference Guide* (UG912)
13. Khronos Group web page: Documentation for the OpenCL standard
14. Xilinx® Virtex® UltraScale+™ FPGA VCU1525 Acceleration Development Kit
15. Xilinx® Kintex® UltraScale™ FPGA KCU1500 Acceleration Development Kit

Please Read: Important Legal Notices

Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright