# SDAccel Methodology Guide

UG1346 (v2019.1) May 22, 2019



XILINX

# Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| 05/22/2019 Version 2019.1 | |
| General | Editorial updates.<br>Added additional links and updated images throughout document. |

# Table of Contents

XILINX

Chapter 1

# Introduction

This guide is intended for software developers and RTL designers who want to create FPGA-accelerated applications using the SDAccel™ development environment. It introduces developers to the fundamental concepts of FPGA-based acceleration and provides steps for accelerating applications with the best possible performance.

## FPGA-Based Acceleration: An Industrial Analogy

There are distinct differences between CPUs, GPUs and FPGAs. Understanding these differences is key to efficiently developing for each kind of device and achieving optimal acceleration.

Both CPUs and GPUs have pre-defined architectures, with a fixed number of cores, a fixed-instruction set, and a rigid memory architecture. GPUs scale performance through the number of cores and by employing SIMD/SIMT parallelism. In contrast, FPGAs are fully customizable architectures. The developer creates compute units that are optimized for application needs. Performance is achieved by creating deeply pipelined datapaths, rather than multiplying the number of compute units.

Think of a CPU as a group of workshops, with each one employing a very skilled worker. These workers have access to general purpose tools that let them build almost anything. Each worker crafts one item at a time, successively using different tools to turn raw material into finished goods. This sequential transformation process can require many steps, depending on the nature of the task. The workshops are independent, and the workers can all be doing different tasks without distractions or coordination problems.

A GPU also has workshops and workers, but it has considerably more of them, and the workers are much more specialized. They have access to only specific tools and can do fewer things, but they do them very efficiently. GPU workers function best when they do the same few tasks repeatedly, and when all of them are doing the same thing at the same time. After all, with so many different workers, it is more efficient to give them all the same orders.

FPGAs take this workshop analogy into the industrial age. If CPUs and GPUs are groups of individual workers taking sequential steps to transform inputs into outputs, FPGAs are factories with assembly lines and conveyer belts. Raw materials are progressively transformed into finished goods by groups of workers dispatched along assembly lines. Each worker performs the same task repeatedly and the partially finished product is transferred from worker to worker on the conveyer belt. This results in a much higher production throughput.

Another major difference with FPGAs is that the factories and assembly lines do not exist de facto, unlike the workshops and workers in CPUs and GPUs. To refine our analogy, an FPGA would be like a collection of empty lots waiting to be developed. This means that the FPGA developer gets to build factories, assembly lines, and workstations, and then customizes them for the required task instead of using general purpose tools. And just like lot size, FPGA real-estate is not infinite, which limits the number and size of the factories which can be built in the FPGA. Properly architecting and configuring these factories is therefore a critical part of the FPGA programming process.

Traditional software development is about programming functionality on a pre-defined architecture. FPGA development is about programming an architecture to implement the desired functionality.

# Methodology Overview

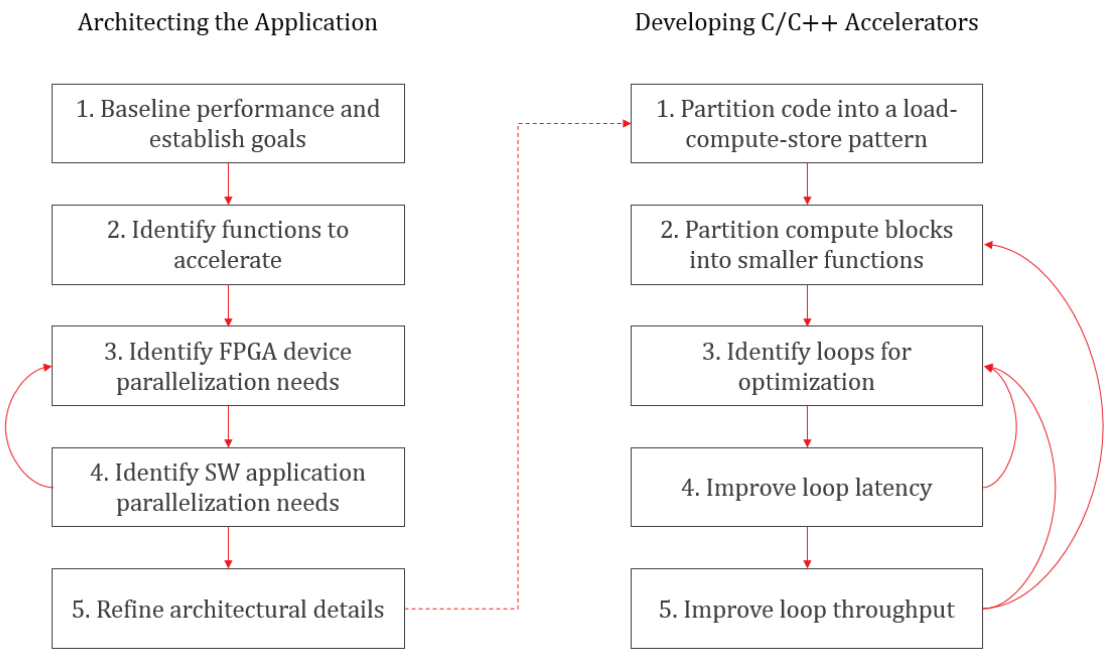
The methodology is comprised of two major phases:

1. Architecting the application
2. Developing the C/C++ kernels

In the first phase, the developer makes key decisions about the application architecture by determining which software functions should be mapped to FPGA kernels, how much parallelism is needed, and how it should be delivered.

In the second phase, the developer implements the kernels. This primarily involves structuring source code and applying the desired compiler pragma to create the desired kernel architecture and meet the performance target.

*Figure 1:* **Methodology Overview**




Performance optimization is an iterative process. The initial version of an accelerated application will likely not produce the best possible results. The methodology described in this guide is a process involving constant performance analysis and repeated changes to all aspects of the implementation.

# Recommendations

A good understanding of the SDAccel programming and execution model is critical to embarking on a project with this methodology. The following resources provide the necessary knowledge to be productive with SDAccel:

- *SDAccel Environment Programmers Guide* (UG1277)
- SDAccel Tutorials (GitHub)

In addition to understanding the key aspects of the SDAccel environment, a good understanding of the following topics will help achieve optimal results with this methodology:

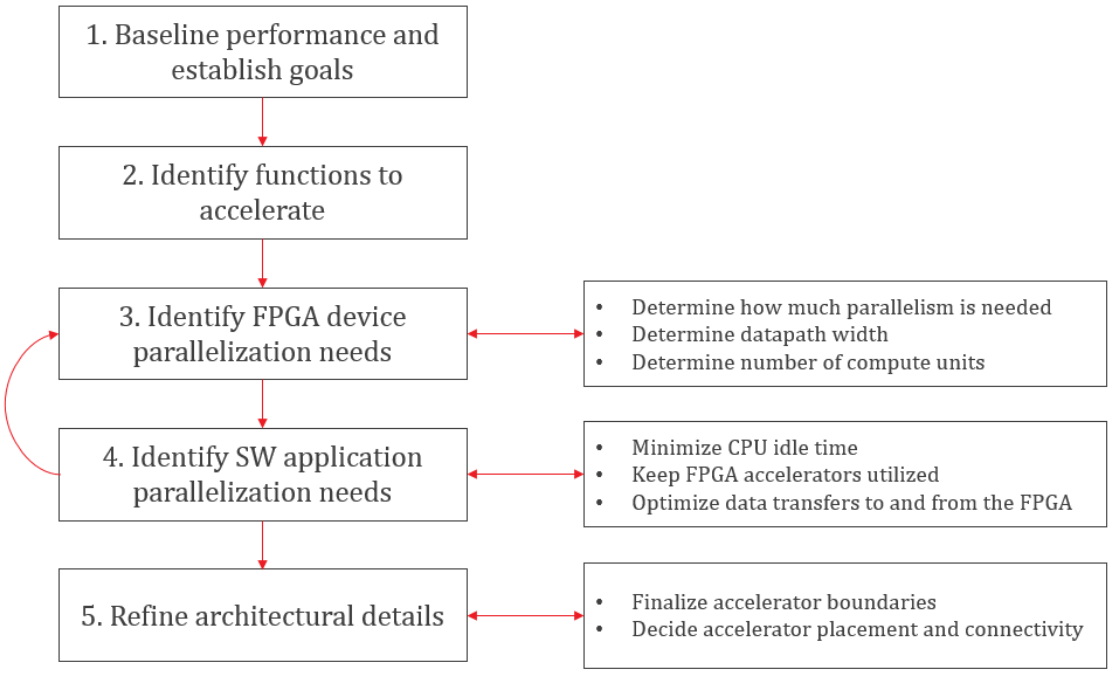
- Application domain
- Software acceleration principles
- Concepts, features and architecture of FPGA
- Features of the targeted FPGA accelerator card and corresponding shell

- Parallelism in hardware implementations (http://kastner.ucsd.edu/hlsbook/)

# Methodology for Architecting an FPGA Accelerated Application

Before beginning the development of an accelerated application, it is important to architect it properly. In this phase, the developer makes key decisions about the architecture of the application and determines factors such as what software functions should be mapped to FPGA kernels, how much parallelism is needed, and how it should be delivered.

*Figure 2:* **Architecting the Application Methodology**




This section walks through the various steps involved in this process. Taking an iterative approach through this process helps refine the analysis and leads to better design decisions.

# Step 1: Baseline Application Performance and Establish Goals

Start by measuring the runtime and throughput performance of the target application. These performance numbers should be generated for the entire application (end-to-end) as well as for each major function in the application. These numbers provide the baseline for most of the subsequent analysis process.

## Measure Running Time

Measuring running time is a standard practice in software development. This can be done using common software profiling tools such as gprof, or by instrumenting the code with timers and performance counters.

The following figure shows an example profiling report generated with gprof. Such reports conveniently show the number of times a function is called, and the amount of time spent (runtime).

*Figure 3:* **Gprof Output Example**

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
70.45     25.54    25.54      256    99.76    99.76  F4(int*, int*, int*)
12.44     30.05     4.51      256    17.61    17.61  F2(int*, int*)
 9.91     33.64     3.59      256    14.03    14.03  F1(int*, int*, int*)
 7.83     36.48     2.84      256    11.08    11.08  F3(int*, int*)
 0.00     36.48     0.00      256     0.00   142.48  F(int*, int*)
```

## Measure Throughput

Throughput is the rate at which data is being processed. To compute the throughput of a given function, divide the volume of data the function processed by the running time of the function.

$$T_{SW} = \max(V_{INPUT}, V_{OUTPUT}) / \text{Running Time}$$

Some functions process a pre-determined volume of data. In this case, simple code inspection can be used to determine this volume. In some other cases, the volume of data is variable. In this case, it is useful to instrument the application code with counters to dynamically measure the volume.

Measuring throughput is as important as measuring running time. While FPGA kernels can improve overall running time, they have an even greater impact on application throughput. As such, it is important to look at throughput as the main optimization target.

## Determine the Maximum Achievable Throughput

In most FPGA-accelerated systems, the maximum achievable throughput is limited by the PCIe® bus. PCIe performance is influenced by many different aspects, such as motherboard, drivers, targeted shell, and transfer sizes. Run DMA tests upfront to measure the effective throughput of PCIe transfers and thereby determine the upper bound of the acceleration potential, such as the xbutil dma test.

*Figure 4:* **Sample Result of dmatest on Alveo U200**

```
$ xbutil dmatest
INFO: Found total 1 card(s), 1 are usable
Total DDR size: 65536 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 11381.7 MB/s
Host <- PCIe <- FPGA read bandwidth  =  8358.9 MB/s
Data Validity & DMA Test on bank1
Host -> PCIe -> FPGA write bandwidth = 11235.3 MB/s
Host <- PCIe <- FPGA read bandwidth  =  7485.3 MB/s
INFO: xbutil dmatest succeeded.
```

An acceleration goal that exceeds this upper bound throughput cannot be met as the system will be I/O bound. Similarly, when defining kernel performance and I/O requirements, keep this upper bound in mind.

## Establish Overall Acceleration Goals

Determining acceleration goals early in the development is necessary as the ratio between the acceleration goal and the baseline performance will drive the analysis and decision-making process.

Acceleration goals can be hard or soft. For example, a real-time video application could have the hard requirement to process 60 frames per second. A data science application could have the soft goal to run 10 times faster than an alternative implementation.

Either way, domain expertise is important for setting obtainable and meaningful acceleration goals.

# Step 2: Identify Functions to Accelerate

After establishing the performance baseline, the next step is to determine which functions should be accelerated in the FPGA device. To this extent, there are two aspects to consider:

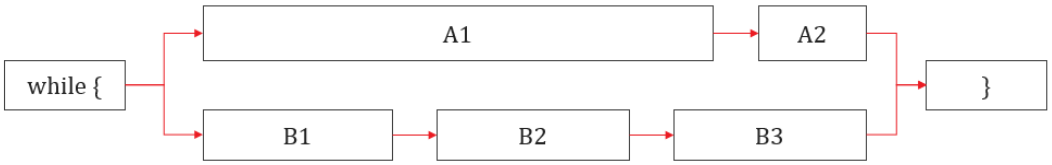
- **Performance bottlenecks:** Which functions are in application hot spots?
- **Acceleration potential:** Do these functions have the potential for acceleration?

## Identify Performance Bottlenecks

In a purely sequential application, performance bottlenecks can be easily identified by looking at profiling reports. However, most real-life applications are multi-threaded and it is important to the take the effects of parallelism in consideration when looking for performance bottlenecks.

Figure 5 represents the performance profile of an application with two parallel paths. The width of each rectangle is proportional to the performance of each function.

*Figure 5:* **Application with Two Parallel Paths**




The above performance visualization in the context of parallelism shows that accelerating only one of the two paths will not improve the application's overall performance. Because paths A and B re-converge, they are dependent upon each other to finish. Likewise, accelerating A2, even by 100x, will not have a significant impact on the performance of the upper path. Therefore, the performance bottlenecks in this example are functions A1, B1, B2 and B3.

When looking for acceleration candidates, consider the performance of the entire application, not just of individual functions.

## Identify Acceleration Potential

A function that is a bottleneck in the software application does not necessarily have the potential to run faster in an FPGA. A detailed analysis is usually required to accurately determine the real acceleration potential of a given function. However, some simple guidelines can be used to assess if a function has potential for hardware acceleration:

- What is the computational complexity of the function?

In FPGAs, acceleration is achieved by creating highly parallel and deeply pipelined data paths. These would be the assembly lines in the earlier analogy. The longer the assembly line and the more stations it has, the more efficient it will be compared to a worker taking sequential steps in his workshop.

Good candidates for acceleration are functions where a deep sequence of operations needs to be performed on each input sample to produce an output sample.

- How does the throughput of the function compare to the maximum achievable in FPGA?

  The throughput of the application cannot exceed the throughput of the PCIe bus. This constitutes the upper bound. Therefore, the developer can determine the maximum acceleration potential by dividing the throughput of the PCIe by the throughput of the selected function.

  $$\text{Maximum Acceleration Potential} = T_{PCIe} / T_{SW}$$

  For example, considering a PCIe throughput of 10GB/sec and a software throughput of 50MB/sec, the maximum acceleration factor for this function is 200x.

These two criteria are not guarantees of acceleration, but they are reliable tools to identify the right functions to accelerate on an FPGA.

# Step 3: Identify FPGA Device Parallelization Needs

Once the functions to be accelerated have been identified and the overall acceleration goals have been established, the next step is to determine what level of parallelization is needed to meet the goals.

The factory analogy is once again helpful to understand what parallelism is possible within kernels.

As described, the assembly line allows the progressive and simultaneous processing of inputs. In hardware, this kind of parallelism is called pipelining. The number of stations on the assembly line corresponds to the number of stages in the hardware pipeline.

Another dimension of parallelism within kernels is the ability to process multiple samples at the same time. This is like putting not just one, but multiple samples on the conveyer belt at the same time. To accommodate this, the assembly line stations are customized to process multiple samples in parallel. This is effectively defining the width of the datapath within the kernel.

Performance can be further scaled by increasing the number of assembly lines. This can be accomplished by putting multiple assembly lines in a factory, and also by building multiple identical factories with one or more assembly lines in each of them.

The developer will need to determine which combination of parallelization techniques will be most effective at meeting the acceleration goals.

## Determine the Likely Achievable Throughput of the Kernel

The throughput of the kernel can be approximated as:

$$T_{HW} = \text{Frequency} / \text{Sample Rate}$$

*Frequency* is the clock frequency of the kernel. This value is determined by the targeted acceleration platform, or shell. For instance, the maximum kernel clock on an Alveo™ U200 card is currently 300 MHz.

The *Sample Rate* is the time interval, measured in clock cycles, between new inputs or new outputs. A Sample Rate of 1 means that the kernel can process one sample each clock cycle. This is the best possible scenario and it can easily be achieved when there are no feedback paths or loop carried dependencies in the function to be accelerated. When this is not the case, a cursory review of the code can help assess the complexity of the feedback paths and estimate an achievable Sample Rate.

Estimating the likely achievable Sample Rate is very useful in determining if the performance goal can be achieved or if additional parallelism is required.

## Determine How Much Parallelism is Needed

The ratio between the desired throughput and the estimated throughput gives a sense how much additional parallelism is needed.

$$\text{Parallelism Needed} = T_{Goal} / T_{HW}$$

If the estimated throughput is insufficient to meet the target performance, then additional parallelism is required. This parallelism can be implemented in various ways: by widening the datapath, by using multiple engines, and by using multiple kernel instances. The developer should then determine the best combination given his needs and the characteristics of his application.

## Determine How Many Samples Can and Should the Datapath be Processing in Parallel

One possibility is to accelerate the computation by creating a wider datapath and processing more samples in parallel. Some algorithms lend themselves well to this approach, whereas others do not. It is important to understand the nature of the algorithm to determine if this approach will work and if so, how many samples should be processed in parallel to meet the performance goal.

Processing more samples in parallel using a wider datapath improves performance by reducing the latency (running time) of the accelerated function.

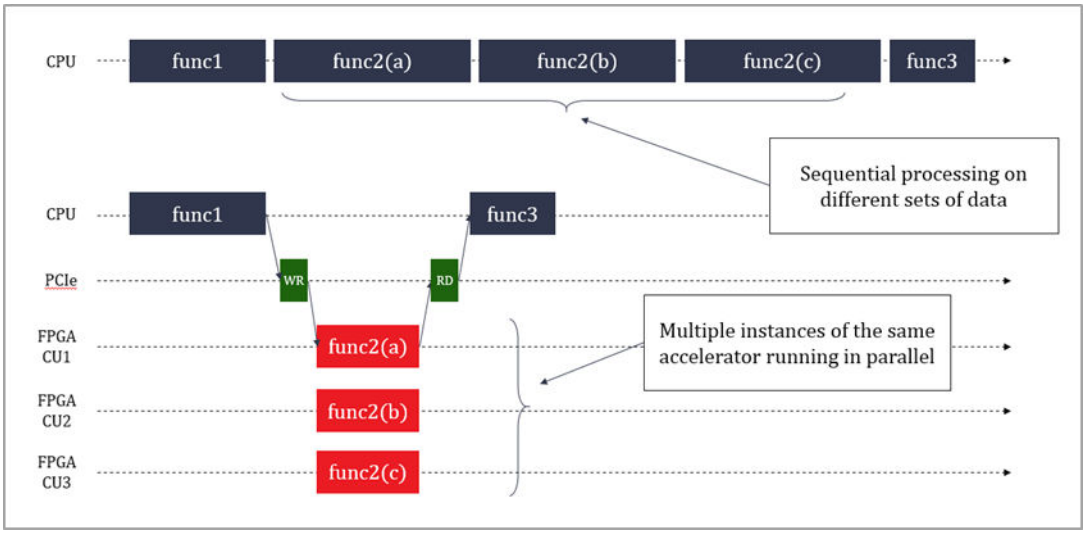
## Determine How Many Kernels Can and Should be Instantiated in the Device

If the datapath cannot be parallelized (or not sufficiently), then look at adding more kernel instances. This is usually referred to as using multiple compute units (CUs).

Adding more kernel instances improves the performance of the application by allowing the execution of more invocations of the targeted function in parallel, as shown below. Multiple data sets are processed concurrently by the different instances. Application performance scales linearly with the number of instances, provided that the host application can keep the kernels busy.

As illustrated in this tutorial and in this link in *SDAccel Environment Programmers Guide* (UG1277), SDAccel™ makes it easy to scale performance by adding additional instances.

*Figure 6:* **Improving Performance with Multiple Compute Units**




At this point, the developer should have a good understanding of the amount of parallelism necessary in the hardware to meet performance goals and, through a combination of datapath width and kernel instances, how that parallelism will be achieved

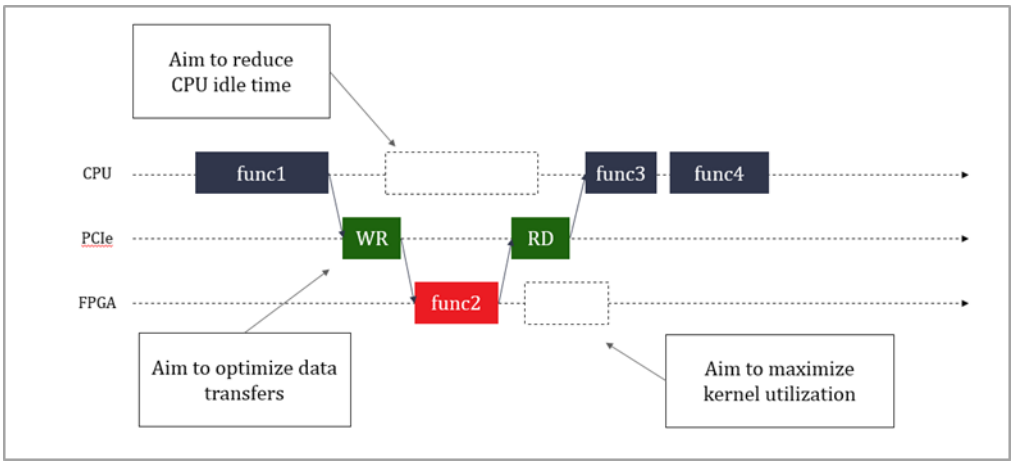
# Step 4: Identify Software Application Parallelization Needs

While the hardware device and its kernels are architected to offer potential parallelism, the software application must be engineered to take advantage of this potential parallelism.

Parallelism in the software application is the ability for the host application to:

- Minimize idle time and do other tasks while the FPGA kernels are running.
- Keep the FPGA kernels active performing new computations as early and often as possible
- Optimize data transfers to and from the FPGA.

*Figure 7:* **Software Optimization Goals**




In the world of factories and assembly lines, the host application would be the headquarters keeping busy and planning the next generation of products while the factories manufacture the current generation.

Similarly, headquarters must orchestrate the transport of goods to and from the factories and send them requests. What is the point of building many factories if the logistics department doesn't send them raw material or blueprints of what to create?

## Minimize CPU Idle Time While the FPGA Kernels are Running

FPGA-acceleration is about offloading certain computations from the host processor to the kernels in the FPGA device. In a purely sequential model, the application would be waiting idly for the results to be ready and resume processing, as shown in the above figure.

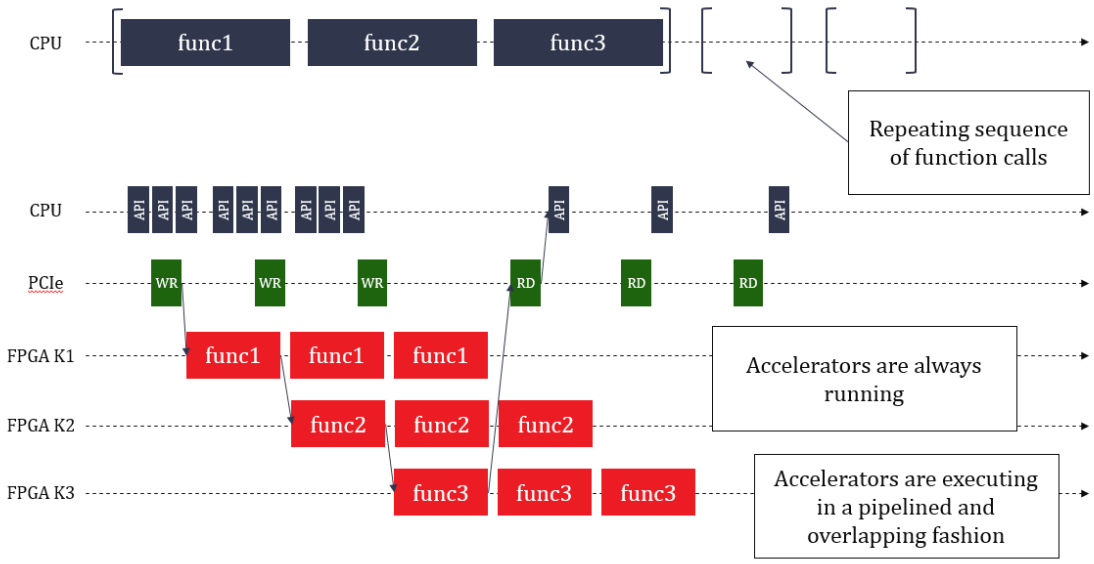
Instead, engineer the software application to avoid such idle cycles. Begin by identifying parts of the application that do not depend on the results of the kernel. Then structure the application so that these functions can be executed on the host in parallel to the kernel running in the device.

## Keep the FPGA Kernels Utilized

Kernels might be present in the FPGA device, but they will only run when the application requests them. To maximize performance, engineer the application so that it will keep the kernels busy.

Conceptually, this is achieved by issuing the next requests before the current ones have completed. This results in pipelined and overlapping execution, leading to kernels being optimally utilized, as shown in the figure below.

*Figure 8:* **Pipelined Execution of Accelerators**




In this example, the original application repeatedly calls func1, func2 and func3. Corresponding kernels (K1, K2, K3) have been created for the three functions. A naïve implementation would have the three kernels running sequentially, like the original software application does. But this means that each kernel is active only a third of the time. A better approach is to structure the software application so that it can issue pipelined requests to the kernels. This allows K1 to start processing a new data set at the same time K2 starts processing the first output of K1. With this approach, the three kernels are constantly running with maximized utilization.

More information on software pipelining can be found in this link in *SDAccel Environment Profiling and Optimization Guide* (UG1207), the Concurrent Kernel Execution (C) example, and the Host Code Optimization tutorial.

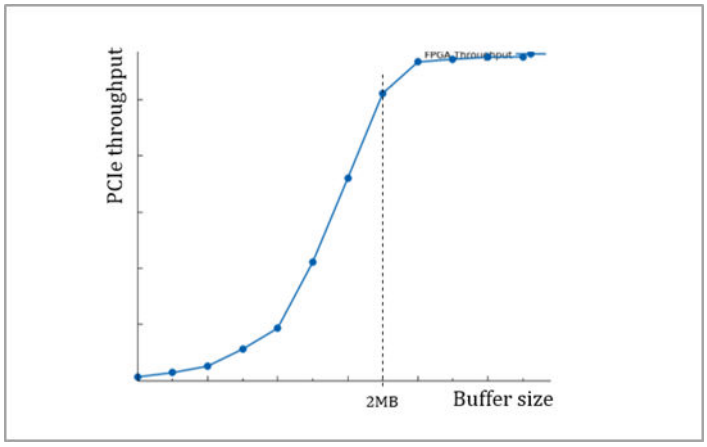
# Optimize Data Transfers To and From the FPGA

In an accelerated application, data must be transferred from the host to the device. This introduces latency which can be very costly to the overall performance of the application.

Data needs to be transferred at the right time, otherwise the application performance is negatively impacted if the kernel must wait for data to be available. It is therefore important to transfer data ahead of when the kernel needs it. This is achieved by overlapping data transfers and kernel execution, as described in the previous section. As shown in the sequence in the above figure and further detailed in this link in *SDAccel Environment Profiling and Optimization Guide* (UG1207), this technique enables hiding the latency overhead of the PCIe transfers and avoids the kernel having to wait for data to be ready.

Another method of optimizing data transfers is to transfer optimally sized buffers. As shown in the following figure, the effective PCIe throughput varies greatly based on the transferred buffer size. The larger the buffer, the better the throughput, ensuring the accelerators always have data to operate on ant are not wasting cycles. It is usually better to make data transfers of 1MB or more. Running DMA tests upfront can be useful for finding the optimal buffer sizes.

*Figure 9:* **Performance of PCIe Transfers as a Function of Buffer Size**



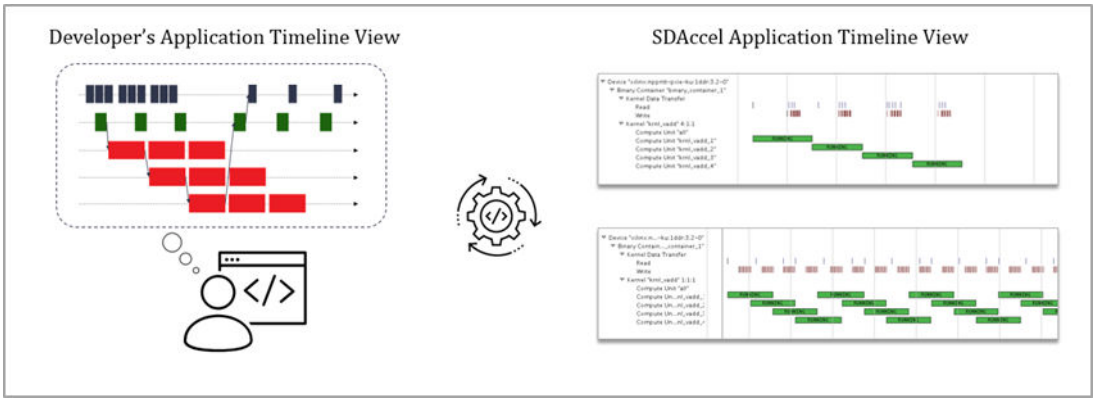

With recommendation, group multiple sets of data in a common buffer to achieve the highest possible throughput.

# Conceptualize the Desired Application Timeline

The developer now should have a good understanding of what functions needs to be accelerated, what parallelism is needed to meet performance goals, and how it will be delivered.

At this point, it is very useful to summarize this information in the form of an expected application timeline. Application timeline sequences, such as the ones shown in the previous chapter, are very effective ways of representing performance and parallelization in action as the application runs. It represents how the potential parallelism built into the architecture is mobilized by the application.

*Figure 10:* **Application Timelines**




The SDAccel development environment generates timeline views from actual application runs. If the developer has a desired timeline in mind, he can then compare it to the actual results, identify potential issues, iterate and converge on the optimal results, as shown in the above figure.

# Step 5: Refine Architectural Details

Before proceeding with the development of the application and its kernels, the final step consists of refining and deriving second order architectural details from the top-level decisions made up to this point.

## Finalize Kernel Boundaries

As discussed earlier, performance can be improved by creating multiple instances of kernels (compute units). However, adding CUs has a cost in terms of I/O ports, bandwidth, and resources.

In the SDAccel flow, kernel ports have a maximum width of 512 bits (64 bytes) and have a fixed cost in terms of FPGA resources. Most importantly, the targeted platform sets a limit on the maximum number of ports which can be used. Be mindful of these constraints and use these ports and their bandwidth optimally.

An alternative to scaling with multiple compute units, is to scale by adding multiple engines within a kernel. This approach allows increasing performance in the same way as adding more CUs: multiple data sets are processed concurrently by the different engines within the kernel.

Placing multiple engines in the same kernel takes the fullest advantage of the bandwidth of the kernel's I/O ports. If the datapath engine doesn't require the full 512 bits of the port, it can be more efficient to add additional engines in the kernel than to create multiple CUs with single engines in them.

Putting multiple engines in a kernel also reduces the number of ports and the number of transactions to global memory that require arbitration, improving the effective bandwidth.

On the other hand, this transformation requires coding explicit I/O multiplexing behavior in the kernel. This is a trade-off the developer needs to make.

## Decide Kernel Placement and Connectivity

Once the kernel boundaries have been finalized, the developer knows exactly how many kernels will be instantiated and therefore how many ports will need to be connected to global memory resources.

At this point, it is important to understand the features of the targeted platform (shell) and what global memory resources are available. For instance, the Alveo U200 platform has 4x16GB banks of DDR4 and 3x128KB banks of PLRAM distributed across 3 super-logic regions (SLRs). For more information, see this link in *SDAccel Environments Release Notes, Installation, and Licensing Guide* (UG1238).

If kernels are factories, then global memory banks are the warehouses through which goods transit to and from the factories. The SLRs are like distinct industrial zones where warehouses preexist and factories can be built. While it is possible to transfer goods from a warehouse in one zone to a factory in another zone, this can add delay and complexity.

Using multiple DDRs helps balance the data transfer loads and improves performance. This comes with a cost, however, as each DDR controller consumes FPGA resources. Balance these considerations when deciding how to connect kernel ports to memory banks.

As explained in this link in *SDAccel Environment Programmers Guide* (UG1277), establishing these connections is done through a simple compiler switch, making it easy to change configurations if necessary.

After refining the architectural details, the developer should have all the information necessary to start implementing the kernels and ultimately, assembling the entire application.

XILINX

*Chapter 3*

# Methodology for Developing C/C++ Kernels

SDAccel™ supports kernels modeled in either C/C++ or RTL (Verilog, VHDL, System Verilog). This methodology guide applies to C/C++ kernels. For details on developing RTL kernels, see this link in *SDAccel Environment User Guide* (UG1023).

The following key kernel requirements for optimal application performance should have already been identified during the architecture definition phase:

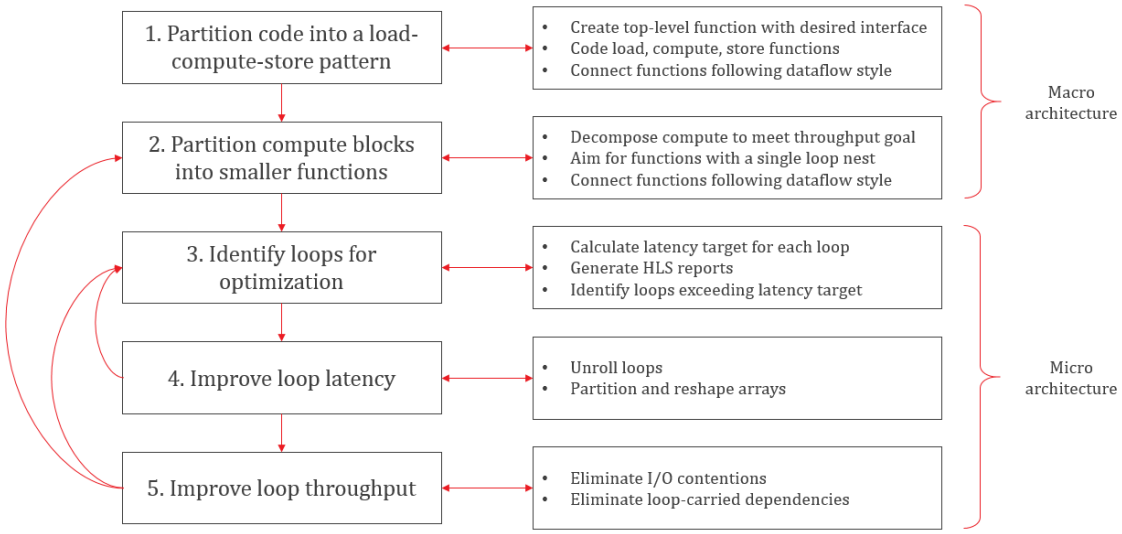
- Throughput goal
- Latency goal
- Datapath width
- Number of engines
- Interface bandwidth

These requirements drive the kernel development and optimization process. Achieving the kernel throughput goal is the primary objective, as overall application performance is predicated on each kernel meeting the specified throughput.

The kernel development methodology therefore follows a throughput-driven approach and works from the outside-in. This approach has two phases, as also described in the following figure:

1. Defining and implementing the macro-architecture of the kernel
2. Coding and optimizing the micro-architecture of the kernel

*Figure 11:* **Kernel Development Methodology**




# About the High-Level Synthesis Compiler

Before starting the kernel development process, the developer should have familiarity with high-level synthesis (HLS) concepts. The HLS compiler turns C/C++ code into RTL designs which will further map onto the FPGA fabric.

The HLS compiler is more restrictive than standard software compilers. For example, there are unsupported constructs including: system function calls, dynamic memory allocation and recursive functions. See this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on unsupported constructs.

More importantly, always keep in mind that the structure of the C/C++ source code has a strong impact on the performance of the generated hardware implementation. This methodology guide will help you structure the code to meet the application throughput goals. For more information on HLS programming concepts, see this link in *SDAccel Environment Programmers Guide* (UG1277) .

# Verification Considerations

This methodology described in this guide is iterative in nature and involves successive code modifications. Xilinx® recommends verifying the code after each modification. This can be done using standard software verification methods or with the SDAccel emulation flows. In either case, make sure your testing provides sufficient coverage and verification quality.

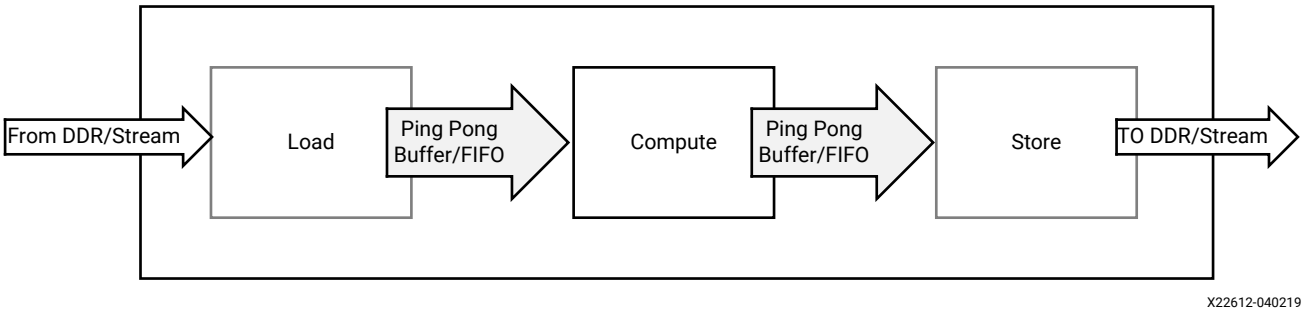
# Step 1: Partition the Code into a Load-Compute-Store Pattern

The first step of the kernel development methodology requires structuring the kernel code into the load-compute-store pattern.

This means creating a top-level function with:

- Interface parameters matching the desired kernel interface.
- Three sub-functions: load, compute, and store.
- Local arrays or hls::stream variables to pass data between these functions.

*Figure 12:* **Load-Compute-Store Pattern**




Structuring the kernel code this way enables task-level pipelining, also known as HLS dataflow. This compiler optimization results in a design where each function can run simultaneously, creating a pipeline of concurrently running tasks. This is the premise of the assembly line in our factory, and this structure is key to achieving and sustaining the desired throughput. For more information about HLS dataflow, see this link in *SDAccel Environment Programmers Guide* (UG1277).

The load function is responsible for moving data external to the kernel (i.e. global memory) to the compute function inside the kernel. This function doesn't do any data processing but focuses on efficient data transfers, including buffering and caching if necessary.

The compute function, as its name suggests, is where all the processing is done. At this stage of the development flow, the internal structure of the compute function isn't important.

The store function mirrors the load function. It is responsible for moving data out of the kernel, taking the results of the compute function and transferring them to global memory outside the kernel.

Creating a load-compute-store structure that meets the performance goals starts by engineering the flow of data within the kernel. Some factors to consider are:

- How does the data flow from outside the kernel into the kernel?
- How fast does the kernel need to process this data?
- How is the processed data written to the output of the kernel?

Understanding and visualizing the data movement as a block diagram will help to partition and structure the different functions within the kernel.

A working example featuring the load-compute-store pattern can be found on the SDAccel Examples GitHub repository.

## Create a Top-Level Function with the Desired Interface

SDAccel infers kernel interfaces from the parameters of the top-level function. Therefore, start by writing a kernel top-level function with parameters matching the desired interface.

Input parameters should be passed as scalars. Blocks of input and output data should be passed as pointers. Compiler pragmas should be used to finalize the interface definition. For complete details, see this link in *SDAccel Environment Programmers Guide* (UG1277) and this link in *SDAccel Environment User Guide* (UG1023).

## Code the Load and Store Functions

Data transfers between the kernel and global memories have a very big influence on overall system performance. If not properly done, they will throttle the kernel. It is therefore important to optimize the load and store functions to efficiently move data in and out of the kernel and optimally feed the compute function.

The layout of data in global memory matches the layout of data in the software application. This layout must be known when writing the load and store functions. Conversely, if a certain data layout is more favorable for moving data in and out of the kernel, it is possible to adapt buffer layout in the software application. Either way, the kernel developer and application developer need to agree on how data is organized in buffers and global memory.

The following are guidelines for improving the efficiency of data transfers in and out of the kernel.

## *Match Port Width to Datapath Width*

In SDAccel, the port of a kernel can be up to 512 bits wide, which means that a kernel can read or write up to 64 bytes per clock cycle per port.

It is recommended to match the width of the kernel ports to width of the datapath in the compute function. For instance, if the datapath needs to process 16 bytes in parallel to meet the desired throughput, then ports should be made 128 bit wide to allow reading and writing 16 bytes in parallel.

In some case, it might be useful to access the full 512 bits of the interface even if the datapath doesn't need them. This can help reduce contention when many kernels are trying to access the same global memory bank. However, this will usually lead to additional buffering and internal memory resources in the kernel.

## *Use Burst Transfers*

The first read or write request to global memory is expensive, but subsequent contiguous operations are not. Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller.

Atomic accesses to global memory should always be avoided unless absolutely required. The load and store functions should be coded to always infer bursting transaction. This can be done using a `memcpy` operation as shown in this GitHub example, or by creating a tight for loop accessing all the required values sequentially, as explained in this link in *SDAccel Environment Programmers Guide* (UG1277).

## *Minimize the Number of Data Transfers from Global Memory*

Since accesses to global memory can add significant latency to the application, only make necessary transfers.

The guideline is to only read and write the necessary values, and only do so once. In situations where the same value must be used several times by the compute function, buffer this value locally instead of reading it from global memory again. Coding the proper buffering and caching structure can be key to achieving the throughput goal.

## Code the Compute Functions

The compute function is where all the actual processing is done. This first step of the methodology is focused on getting the top-level structure right and optimizing data movement. The priority is to have a function with the right interfaces and make sure the functionality is correct. The following sections focus on the internal structure of the compute function.

## Connect the Load, Compute, and Store Functions

Use standard C/C++ variables and arrays to connect the top-level interfaces and the load, compute and store functions. It can also be useful to use the hls::stream class, which models a streaming behavior.

Streaming is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management and can be implemented with FIFOs. For more information about the hls::stream class, see this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

When connecting the functions, use the canonical form required by the HLS compiler. See this link in *SDAccel Environment Programmers Guide* (UG1277) for more information. This helps the compiler build a high-throughput set of tasks using the dataflow optimization. Key recommendations include:

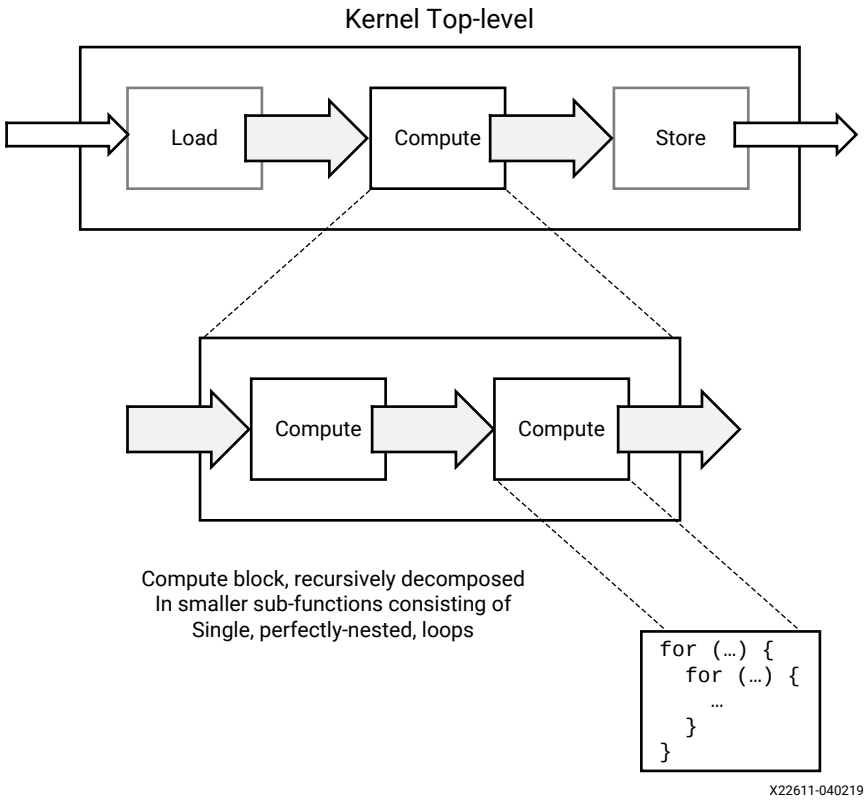
- Data should be transferred in the forward direction only, avoiding feedback whenever possible.
- Each connection should have a single producer and a single consumer.
- Only the load and store functions should access the primary interface of the kernel.

At this point, the developer has created the top-level function of the kernel, coded the interfaces and the load/store functions with the objective of moving data through the kernel at the desired throughput.

# Step 2: Partition the Compute Blocks into Smaller Functions

The next step is to refine the main compute function, decomposing it into a sequence of smaller sub-functions, as shown in the following figure.

*Figure 13:* **Compute Block Sub-Functions**




## Decompose to Identify Throughput Goals

In a dataflow system like the one created with this approach, the slowest task will be the bottleneck.

$$\text{Throughput(Kernel)} = \text{°min(Throughput(Task}_1\text{), Throughput(Task}_2\text{), …, Throughput(Task}_N\text{))}$$

Therefore, during the decomposition process, always have the kernel throughput goal in mind and assess whether each sub-function will be able to satisfy this throughput goal.

In the following steps of this methodology, the developer will get actual throughput numbers from running the SDAccel HLS compiler. If these results cannot be improved, the developer will have to iterate and further decompose the compute stages.

## Aim for Functions with a Single Loop Nest

As a general rule, if a function has sequential loops in it, these loops execute sequentially in the hardware implementation generated by the HLS compiler. This is usually not desirable, as sequential execution hampers throughput.

However, if these sequential loops are pushed into sequential functions, then the HLS compiler can apply the dataflow optimization and generate an implementation that allows the pipelined and overlapping execution of each task. For more information on the dataflow optimization, see this link in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

During this partitioning and refining process, put sequential loops into individual functions. Ideally, the lowest-level compute block should only contain a single perfectly-nested loop. For more information on loops, see this link in *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

### Connect Compute Functions Using the Dataflow 'Canonical Form'

The same rules regarding connectivity within the top-level function apply when decomposing the compute function. Aim for feed-forward connections and having a single producer and consumer for each connecting variable. If a variable must be consumed by more than one function, then it should be explicitly duplicated.

When moving blocks of data from one compute block to another, the developer can choose to use arrays or hls::stream objects.

Using arrays requires fewer code changes and is usually the fastest way to make progress during the decomposition process. However, using hls::stream objects can lead to designs using less memory resources and having shorter latency. It also helps the developer reason about how data moves through the kernel, which is always an important thing to understand when optimizing for throughput.

Using hls::stream objects is usually a good thing to do, but it is up to the developer to determine the most appropriate moment to convert arrays to streams. Some developers will do this very early on while others will do this at the very end, as a final optimization step. This can also be done using a pragma. For more information, see this link in *SDx Pragma Reference Guide* (UG1253).

At this stage, maintaining a graphical representation of the architecture of the kernel can be very useful to reason through data dependencies, data movement, control flows, and concurrency.

## Step 3: Identify Loops Requiring Optimization

At this point, the developer has created a dataflow architecture with data motion and processing functions intended to sustain the throughput goal of the kernel. The next step is to make sure that each of the processing functions are implemented in a way that deliver the expected throughput.

As explained before, the throughput of a function is measured by dividing the volume of data processed by the latency, or running time, of the function.

$$T = \max(V_{INPUT}, V_{OUTPUT}) / \text{Latency}$$

Both the target throughput and the volume of data consumed and produced by the function should be known at this stage of the 'outside-in' decomposition process described in this methodology. The developer can therefore easily derive the latency target for each function.

The SDAccel HLS compiler generates detailed reports on the throughput and latency of functions and loops. Once the target latencies have been determined, use the HLS reports to identify which functions and loops do not meet their latency target and require attention.

The latency of a loop can be calculated as follows:

$$\text{Latency}_{Loop} = (\text{Steps} + \text{II} \times (\text{TripCount} - 1)) \times \text{ClockPeriod}$$

Where:

- **Steps:** Duration of a single loop iteration, measured in number of clock cycles
- **TripCount:** Number of iterations in the loop.
- **II:** Initiation Interval, the number of clock cycles between the start of two consecutive iterations. When a loop is not pipelined, its II is equal to the number of Steps.

Assuming a given clock period, there are three ways to reduce the latency of a loop, and thereby improve the throughput of a function:

- Reduce the number of Steps in the loop (take less time to perform one iteration).
- Reduce the Trip Count, so that the loop performs fewer iterations.
- Reduce the Initiation Interval, so that loop iterations can start more often.

Assuming a trip count much larger than the number of steps, halving either the II or the trip count can be sufficient to double the throughput of the loop.

Understanding this information is key to optimizing loops with latencies exceeding their target. By default, the SDAccel HLS compiler will try to generate loop implementations with the lowest possible II. Start by looking at how to improve latency by reducing the trip count or the number of steps. For additional information about how loops are handled by the HLS compiler, see the this link in *SDAccel Environment Programmers Guide* (UG1277).

# Step 4: Improve Loop Latencies

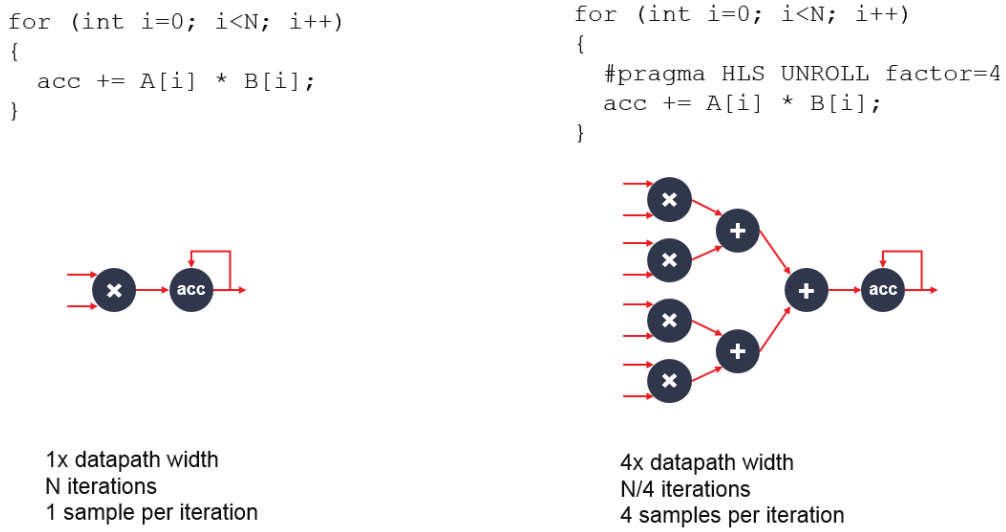
After identifying loops latencies that exceed their target, the first optimization to consider is loop unrolling.

## Apply Loop Unrolling

Loop unrolling unwinds the loop, allowing multiple iterations of the loop to be executed together, reducing the loop's overall trip count.

In the industrial analogy, factories are kernels, assembly lines are dataflow pipelines, and stations are compute functions. Unrolling creates stations which can process multiple objects arriving at the same time on the conveyer belt, which results in higher performance.

*Figure 14:* **Loop Unrolling**




Loop unrolling can widen the resulting datapath by the corresponding factor. This usually increases the bandwidth requirements as more samples are processed in parallel. This has two implications:

- The width of the function I/Os must match the width of the datapath and vice versa.
- No additional benefit is gained by loop unrolling and widening the datapath to the point where I/O requirements exceed the maximum size of a kernel port (512 bits / 64 bytes).

The following guidelines will help optimize the use of loop unrolling:

- Start from the innermost loop within a loop nest.

- Assess which unroll factor would eliminate all loop-carried dependencies.
- For more efficient results, unroll loops with fixed trip counts.
- If there are function calls within the unrolled loop, in-lining these functions can improve results through better resource sharing, although at the expense of longer synthesis times. Note also that the interconnect may become increasingly complex and lead to routing problems later on..
- Do not blindly unroll loops. Always unroll loops with a specific outcome in mind.

### Apply Array Partitioning

Unrolling loops changes the I/O requirements and data access patterns of the function. If a loop makes array accesses, as is almost always the case, make sure that the resulting datapath can access all the data it needs in parallel.

If unrolling a loop doesn't result in the expected performance improvement, this is almost always because of memory access bottlenecks.

By default, the SDAccel HLS compiler maps large arrays to memory resources with a word width equal to the size of one array element. In most cases, this default mapping needs to be changed when loop unrolling is applied.

As explained in the this link in *SDAccel Environment Programmers Guide* (UG1277), the HLS compiler supports various pragmas to partition and reshape arrays. Consider using these pragmas when loop unrolling to create a memory structure that allows the desired level of parallel accesses.

Unrolling and partitioning arrays can be sufficient to meet the latency and throughput goals for the targeted loop. If so, shift to the next loop of interest. Otherwise, look at additional optimizations to improve throughput.

## Step 5: Improve Loop Throughput

If improving loop latency by reducing the trip count wasn't sufficient, look at ways to reduce the Initiation Interval (II).

The loop II is the count of clock cycles between the start of two loop iterations. The SDAccel HLS compiler will always try to pipeline loops, minimize the II, and start loop iterations as early as possible, ideally starting a new iteration each clock cycle (II=1).

There are two main factors that can limit the II:

- I/O contentions

- Loop-carried dependencies

The HLS Schedule Viewer automatically highlights loop dependencies limiting the II. It is a very useful visualization tool to use when working to improve the II of a loop.

## Eliminate I/O Contentions

I/O contentions appear when a given I/O port of internal memory resources must be accessed more than once per loop iteration. A loop cannot be pipelined with an II lower than the number of times an I/O resource is accessed per loop iteration. If port A must be accessed four times in a loop iteration, then the lowest possible II will be 4.

The developer needs to assess whether these I/O accesses are necessary or if they can be eliminated. The most common techniques for reducing I/O contentions are:

- Creating internal cache structures

  If some of the problematic I/O accesses involve accessing data already accessed in prior loop iterations, then a possibility is to modify the code to make local copies of the values accessed in those earlier iterations. Maintaining a local data cache can help reduce the need for external I/O accesses, thereby improving the potential II of the loop.

  This example on the SDAccel Examples GitHub repository illustrates how a shift register can be used locally, cache previously read values, and improve the throughput of a filter.

- Reconfiguring I/Os and memories

  As explained earlier in the section about improving latency, the HLS compiler maps arrays to memories, and the default memory configuration can offer sufficient bandwidth for the required throughput. The array partitioning and reshaping pragmas can also be used in this context to create memory structure with higher bandwidth, thereby improving the potential II of the loop.

## Eliminate Loop-Carried Dependencies

The most common case for loop-carried dependencies is when a loop iteration relies on a value computed in a prior iteration. There are differences whether the dependencies are on arrays or on scalar variables. For more information, see this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

- Eliminating dependencies on arrays

  The HLS compiler performs index analysis to determine whether array dependencies exist (read-after-write, write-after-read, write-after-write). The tool may not always be able to statically resolve potential dependencies and will in this case report false dependencies.

  Special compiler pragmas can overwrite these dependencies and improve the II of the design. In this situation, be cautious and do not overwrite a valid dependency.

- Eliminating dependencies on scalars

  In the case of scalar dependencies, there is usually a feedback path with a computation scheduled over multiple clock cycles. Complex arithmetic operations such as multiplications, divisions, or modulus are often found on these feedback paths. The number of cycles in the feedback path directly limits the potential II and should be reduced to improve II and throughput. To do so, analyze the feedback path to determine if and how it can be shortened. This can potentially be done using HLS scheduling constraints or code modifications such as reducing bit widths.

## Advanced Techniques

If an II of 1 is usually the best scenario, it is rarely the only *sufficient* scenario. The goal is to meet the latency and throughput goal. To this extent, various combinations of II and unroll factor are often sufficient.

The optimization methodology and techniques presented in this guide should help meet most goals. The HLS compiler also supports many more optimization options which can be useful under specific circumstances. A complete reference of these optimizations is available in this link in *SDx Pragma Reference Guide* (UG1253).

XILINX

*Appendix A*

# 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. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- 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™ Development Environment web page

2. Vivado® Design Suite Documentation
3. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
4. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
5. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
6. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
7. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
8. *Vivado Design Suite Properties Reference Guide* (UG912)
9. Khronos Group web page: Documentation for the OpenCL standard
10. Xilinx® Virtex® UltraScale+™ FPGA VCU1525 Acceleration Development Kit
11. Xilinx® Kintex® UltraScale™ FPGA KCU1500 Acceleration Development Kit
12. Xilinx® Alveo™ web page

# Please Read: Important Legal Notices

### 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