Vitis Application Acceleration Development Flow Tutorials (https://github.com/Xilinx/Vitis-Tutorials)
Provides a number of tutorials that can be worked through to teach specific concepts regarding the tool flow and application development, including the use of Vitis HLS as a standalone application, and in the Vitis bottom up design flow.

HLS Programmers Guide

Introduction

This Programmers Guide is intended to provide real world design techniques, and details of hardware design which will help you get the most out of the AMD Vitis™ HLS tool. This guide provides details on programming techniques you should apply when writing C/C++ code for high-level synthesis into RTL code, and a checklist of best practices to follow when creating IP that uses the AXI4 interfaces. Finally, it details various optimization techniques that can improve the performance of your code, improving both the fit and function of the resulting hardware design.
This section contains the following chapters:

Design Principles

Introduction

You might be working with the HLS tool to take advantage of productivity gains from writing C/C++ code to generate RTL for hardware; or you might be looking to accelerate portions of a C/C++ algorithm to run on custom hardware implemented in programmable logic. This chapter is intended to help you understand the process of synthesizing hardware from a software algorithm written in C/C++. This document introduces the fundamental concepts used to design and create good synthesizable software in such a way that it can be successfully converted to hardware using high-level synthesis (HLS) tools. The discussion in this document will be tool-agnostic and the concepts introduced are common to most HLS tools. For experienced designers, reviewing this material can provide a useful reinforcement of the importance of these concepts; help you understand how to approach HLS, and in particular how to structure HLS code to achieve high-performance designs.

Throughput and Performance

C/C++ functions implemented as custom hardware in programmable logic can run at a significantly faster rate than what is achievable on traditional CPU/GPU architectures, and achieve higher processing rates and/or performance. Let us first establish what these terms mean in the context of hardware acceleration. *Throughput* is defined as the number of specific actions executed per unit of time or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. For example, the term "memory bandwidth" is sometimes used to specify the throughput of the memory systems. Similarly, *performance* is defined as not just higher throughput but higher throughput with low power consumption. Lower power consumption is as important as higher throughput in today's world.

Architecture Matters

In order to better understand how custom hardware can accelerate portions of your program, you will first need to understand how your program runs on a traditional computer. The von Neumann architecture is the basis of almost all computing done today even though it was designed more than 7 decades ago. This architecture was deemed optimal for a large class of applications and has tended to be very flexible and programmable.
However, as application demands started to stress the system, CPUs began supporting the execution of multiple processes. Multithreading and/or Multiprocessing can include multiple *system processes* (For example: executing two or more programs at the same time), or it can consist of one process that has multiple *threads* within it. Multi-threaded programming using a shared memory system became very popular as it allowed the software developer to design applications with parallelism in mind but with a fixed CPU architecture. But when multi-threading and the ever-increasing CPU speeds could no longer handle the data processing rates, multiple CPU cores and hyperthreading were used to improve throughput as shown in the figure on the right.
This general purpose flexibility comes at a cost in terms of power and peak throughput. In today's world of ubiquitous smart phones, gaming, and online video conferencing, the nature of the data being processed has changed. To achieve higher throughput, you must move the workload closer to memory, and/or into specialized functional units. So the new challenge is to design a new programmable architecture in such a way that you can maintain just enough programmability while achieving higher performance and lower power costs.
A field-programmable gate array (FPGA) provides for this kind of programmability and offers enough memory bandwidth to make this a high-performance and lower power cost solution. Unlike a CPU that executes a program, an FPGA can be configured into a custom hardware circuit that will respond to inputs in the same way that a dedicated piece of hardware would behave. Reconfigurable devices such as FPGAs contain computing elements of extremely flexible granularities, ranging from elementary logic gates to complete arithmetic-logic units such as digital signal processing (DSP) blocks. At higher granularities, user-specified composable units of logic called kernels can then be strategically placed on the FPGA device to perform various roles. This characteristic of reconfigurable FPGA devices allows the creation of custom macro-architectures and gives FPGAs a big advantage over traditional CPUs/GPUs in utilizing application-specific parallelism. Computation can be spatially mapped to the device, enabling much higher operational throughput than processor-centric platforms. Today's latest FPGA devices can also contain processor cores (Arm-based) and other hardened IP blocks that can be used without having to program them into the programmable fabric.

Three Paradigms for Programming FPGAs

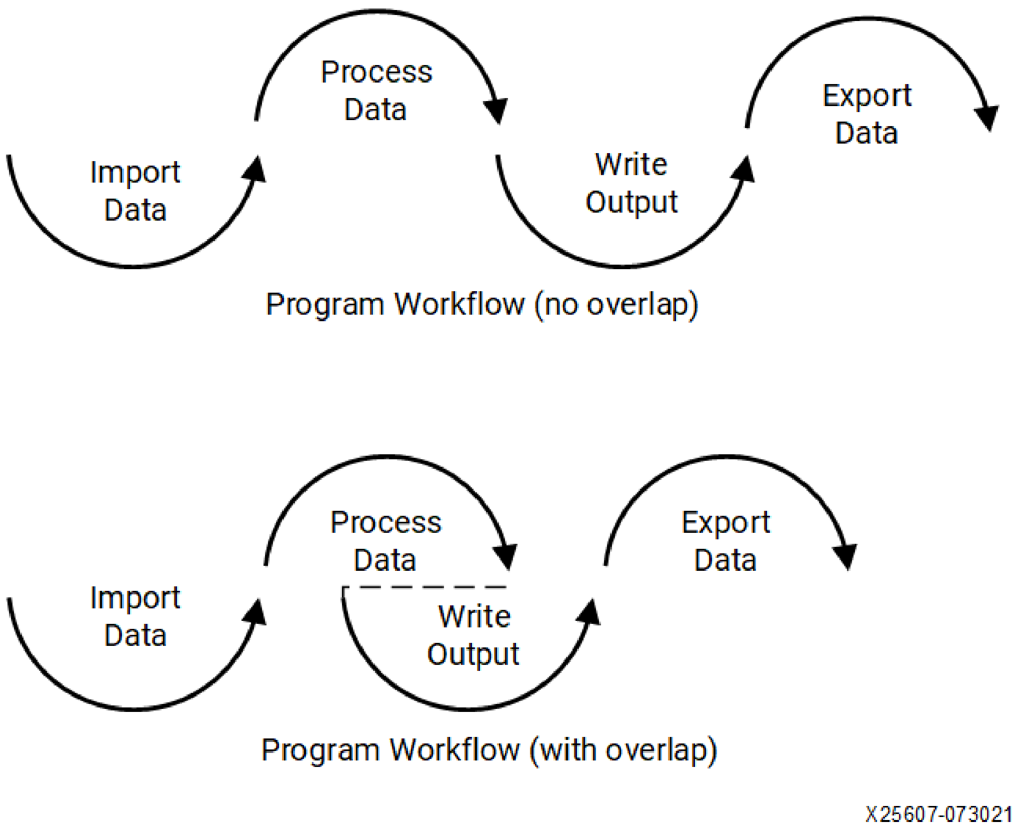
While FPGAs can be programmed using lower-level Hardware Description Languages (HDLs) such as Verilog or VHDL, there are now several High-Level Synthesis (HLS) tools that can take an algorithmic description written in a higher-level language like C/C++ and convert it into lower-level hardware description languages such as Verilog or VHDL. This can then be processed by downstream tools to program the FPGA device.
The main benefit of this type of flow is that you can retain the advantages of the programming language like C/C++ to write efficient code that can then be translated into hardware. Additionally, writing good code is the software designer's forte and is easier than learning a new hardware description language. However, achieving acceptable quality of results (QoR), will require additional work such as rewriting the software to help the HLS tool achieve the desired performance goals. The next few sections will discuss how you can first identify some

macro-level architectural optimizations to structure your program and then focus on some fine-grained micro-level architectural optimizations to boost your performance goals.

Producer-Consumer Paradigm

Consider how software designers write a multithreaded program - there is usually a master thread that performs some initialization steps and then forks off a number of child threads to do some parallel computation and when all the parallel computation is done, the main thread collates the results and writes to the output. The programmer has to figure out what parts can be forked off for parallel computation and what parts need to be executed sequentially. This fork/join type of parallelism applies as well to FPGAs as it does to CPUs, but a key pattern for throughput on FPGAs is the producer-consumer paradigm. You need to apply the producer-consumer paradigm to a sequential program and convert it to extract functionality that can be executed in parallel to improve performance.
You can better understand this decomposition process with the help of a simple problem statement. Assume that you have a datasheet from which you will import items into a list. You will then process each item in the list. The processing of each item takes around 2 seconds. After processing, you will write the result in another datasheet and this action will take an additional 1 second per item. So if you have a total of 100 items in the input Excel sheet then it will take a total of 300 seconds to generate output. The goal is to decompose this problem in such a way that you can identify tasks that can potentially execute in parallel and therefore increase the throughput of the system.

Figure: Program Workflows



The first step is to understand the program workflow and identify the independent tasks or functions. The four-step workflow is something like the Program Workflow (no overlap) shown in the above diagram. In the example, the "Write Output" (step 3) task is independent of the "Process Data" (step 2) processing task. Although step 3 depends on the output of step 2, as soon as any of the items are processed in Step 2, you can immediately write that item to the output file. You don't have to wait for all the data to be processed before starting to write data to the output file. This type of interleaving/overlapping the execution of tasks is a very common principle.

This is illustrated in the above diagram (For example: the program workflow with overlap). As can be seen, the work gets done faster than with no overlap. You can now recognize that step 2 is the producer, and step 3 is the consumer. The producer-consumer pattern has a limited impact on performance on a CPU. You can interleave the execution of the steps of each thread but this requires careful analysis to exploit the underlying multi-threading and L1 cache architecture and therefore a time consuming activity. On FPGAs however, due to the custom architecture, the producer and consumer threads can be executed simultaneously with little or no overhead leading to a considerable improvement in throughput.

The simplest case to first consider is the single producer and single consumer, who communicate via a finite-size buffer. If the buffer is full, the producer has a choice of either blocking/stalling or discarding the data. Once the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can stall if it finds the buffer empty. Once the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be achieved by means of inter-process communication, typically using monitors or semaphores. An inadequate solution could result in a deadlock where both processes are stalled waiting to be woken up. However, in the case of a single producer and consumer, the communication pattern strongly maps to a first-in-first-out (FIFO) or a Ping-Pong buffer (PIPO) implementation. This type of channel provides highly efficient data communication without relying on semaphores, mutexes, or monitors for data transfer. The use of such locking primitives can be expensive in terms of performance and difficult to use and debug. PIPOs and FIFOs are popular choices because they avoid the need for end-to-end atomic synchronization.

This type of macro-level architectural optimization, where the communication is encapsulated by a buffer, frees the programmer from worrying about memory models and other non-deterministic behavior (like race conditions etc). The type of network that is achieved in this type of design is purely a "dataflow network" that accepts a stream of data on the input side and essentially does some processing on this stream of data and sends it out as a stream of data. The complexities of a parallel program are abstracted away. Note that the "Import Data" (Step 1) and "Export Data" (Step 4) also have a role to play in maximizing the available parallelism. In order to allow computation to successfully overlap with I/O, it is important to encapsulate reading from inputs as the first step and writing to outputs as the last step. This will allow for a maximal overlap of I/O with computation. Reading or writing to input/output ports in the middle of the computation step will limit the available concurrency in the design. It is another thing to keep in mind while designing the workflow of your design.

Finally, the performance of such a "dataflow network" relies on the designer being able to continually feed data to the network such that data keeps streaming through the system. Having interruptions in the dataflow can result in lower performance. A good analogy for this is video streaming applications like online gaming where the real-time high definition (HD) video is constantly streamed through the system and the frame processing rate is constantly monitored to ensure that it meets the expected quality of results. Any slowdown in the frame processing rate can be immediately seen by the gamers on their screens. Now imagine being able to support consistent frame rates for a whole bunch of gamers all the while consuming much less power than with traditional CPU or GPU architectures - this is the sweet spot for hardware acceleration. Keeping the data flowing between the producer and consumer is of paramount importance. Next, you will delve a little deeper into this *streaming* paradigm that was introduced in this section.

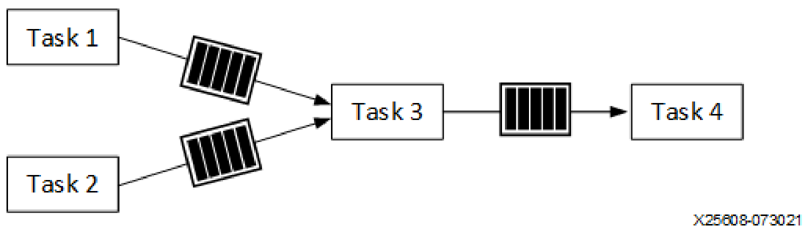
Streaming Data Paradigm

A *stream* is an important abstraction: it represents an unbounded, continuously updating data set, where unbounded means “of unknown or of unlimited size”. A stream can be a sequence of data (scalars or buffers) flowing unidirectionally between a source (producer) process and a destination (consumer) process. The streaming paradigm forces you to think in terms of data access patterns (or sequences). In software, random memory accesses to data are virtually free (ignoring the caching costs), but in hardware, it is really advantageous to make sequential accesses, which can be converted into streams. Decomposing your algorithm into producer-consumer relationships that communicate by streaming data through the network has several advantages. It lets the programmer define the algorithm in a sequential manner and the parallelism is extracted through other means (such as by the compiler). Complexities like synchronization between the tasks etc are abstracted away. It allows the producer and the consumer tasks to process data simultaneously, which is key for achieving higher throughput. Another benefit is cleaner and simpler code.

As was mentioned before, in the case of the producer and consumer paradigm, the data transfer pattern strongly maps to a FIFO or a PIPO buffer implementation. A FIFO buffer is simply a queue with a predetermined size/depth where the first element that gets inserted into the queue also becomes the first element that can be popped from the queue. The main advantage of using a FIFO buffer is that the consumer process can start accessing the data inside the FIFO buffer as soon as the producer inserts the data into the buffer. The only issue with using FIFO buffers is that due to varying rates of production/consumption between the producers and consumers, it is possible for improperly sized FIFO buffers to cause a deadlock. This typically happens in a design that has several producers and consumers. A Ping Pong Buffer is a double buffer that is used to speed up a process that can overlap the I/O operation with the data processing operation. One buffer is used to hold a block of data so that a consumer process will see a complete (but old) version of the data, while in the other buffer a producer process is creating a new (partial) version of data. When the new block of data is complete and valid, the consumer and the producer processes will alternate access to the two buffers. As a result, the usage of a ping-pong buffer increases the overall throughput of a device and helps to prevent eventual bottlenecks. The key advantage of PIPOs is that the tool automatically matches the rate of production vs the rate of consumption and creates a channel of communication that is both high performance and is deadlock free. It is important to note here that regardless of whether FIFOs/PIPOs are used, the key characteristic is the same: the producer sends or streams a block of data to the consumer. A block of data can be a single value or a group of N values. The bigger the block size, the more memory resources you will need.

The following is a simple sum application to illustrate the classic streaming/dataflow network. In this case, the goal of the application is to pair-wise add a stream of random numbers then print them. The first two tasks (Task 1 and 2) provide a stream of random numbers to add. These are sent over a FIFO channel to the sum task (Task 3) which reads the values from the FIFO channels. The sum task then sends the output to the print task (Task 4) to publish the result. The FIFO channels provide asynchronous buffering between these independent threads of execution.

Figure: Streaming/Dataflow Network



The streams that connect each “task” are usually implemented as FIFO queues. The FIFO abstracts away the parallel behavior from the programmer, leaving them to reason about a “snapshot” of time when the task is active (scheduled). FIFOs make parallelization easier to implement. This largely results from the reduced variable space that programmers must contend with when implementing parallelization frameworks or fault-tolerant solutions. The FIFO between two independent kernels (see example above) exhibits classic queueing behavior. With purely streaming systems, these can be modeled using queueing or network flow models. Another big advantage of this dataflow type network and streaming optimization is that it can be applied at different levels of granularity. A programmer can design such a network inside each task as well as for a system of tasks or kernel. In fact, you can have a streaming network that instantiates and connects multiple streaming networks or tasks, hierarchically. Another optimization that allows for finer-grained parallelism is *pipelining*.

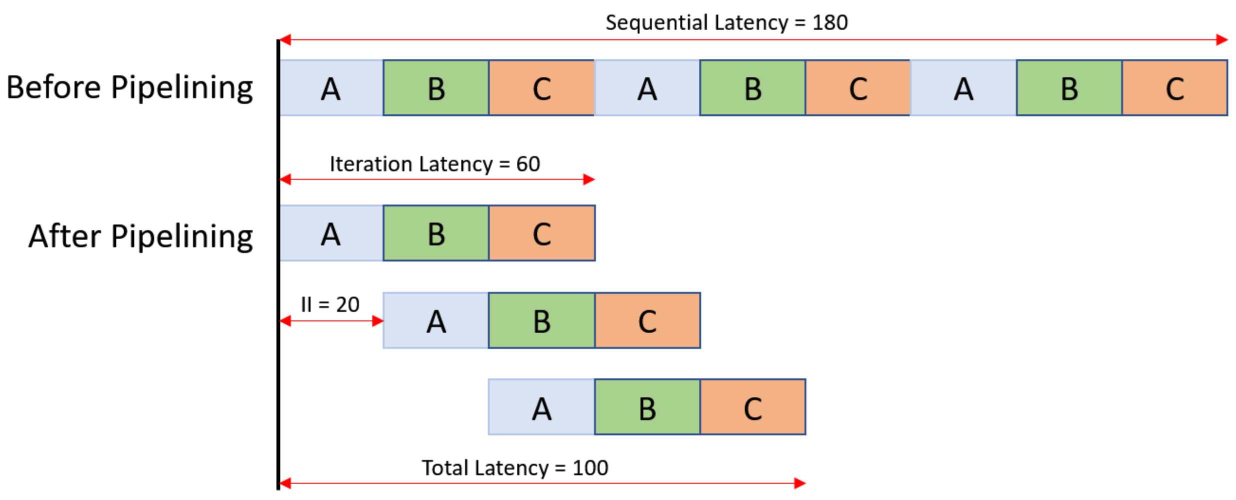
Pipelining Paradigm

Pipelining is a commonly used concept that you will encounter in everyday life. A good example is the production line of a car factory, where each specific task such as installing the engine, installing the doors, and installing the wheels, is often done by a separate and unique workstation. The stations carry out their tasks in parallel, each on a different car. Once a car has had one task performed, it moves to the next station. Variations in the time needed to complete the tasks can be accommodated by *buffering* (holding one or more cars in a space between the stations) and/or by *stalling* (temporarily halting the upstream stations) until the next station becomes available.

Suppose that assembling one car requires three tasks A, B, and C that takes 20, 10, and 30 minutes, respectively. Then, if all three tasks were performed by a single station, the factory would output one car every 60 minutes. By using a pipeline of three stations, the factory would output the first car in 60 minutes, and then a new one every 30 minutes. As this example shows, pipelining does not decrease the *latency*, that is, the total time for one item to go through the whole system. It does however increase the system's throughput, that is, the rate at which new items are processed after the first one.

Since the throughput of a pipeline cannot be better than that of its slowest element, the programmer should try to divide the work and resources among the stages so that they all take the same time to complete their tasks. In the car assembly example above, if the three tasks A. B and C took 20 minutes each, instead of 20, 10, and 30 minutes, the latency would still be 60 minutes, but a new car would then be finished every 20 minutes, instead of 30. The diagram below shows a hypothetical manufacturing line tasked with the production of three cars. Assuming each of the tasks A, B and C takes 20 minutes, a sequential production line would take 180 minutes to produce three cars. A pipelined production line would take only 100 minutes to produce three cars. The time taken to produce the first car is 60 minutes and is called the *iteration latency* of the pipeline. After the first car is produced, the next two cars only take 20 minutes each and this is known as the *initiation interval* (II) of the pipeline. The overall time taken to produce the three cars is 100 minutes and is referred to as the *total latency* of the pipeline, for example, total latency = iteration latency + II * (number of items - 1). Therefore, improving II improves total latency, but not the iteration latency. From the programmer's point of view, the pipelining paradigm can be applied to functions and loops in the design. After an initial setup cost, the ideal throughput goal will be to achieve an II of 1 - for example, after the initial setup delay, the output will be available at every cycle of the pipeline. In the example above, after an initial setup delay of 60 minutes, a car is then available every 20 minutes.

Figure: Pipelining



Pipelining is a classical micro-level architectural optimization that can be applied to multiple levels of abstraction. Task-level pipelining with the producer-consumer paradigm was covered earlier. This same concept applies to

the instruction-level. This is in fact key to keeping the producer-consumer pipelines (and streams) filled and busy. The producer-consumer pipeline will only be efficient if each task produces/consumes data at a high rate, and hence the need for the instruction-level pipelining (ILP).
Due to the way pipelining uses the same resources to execute the same function over time, it is considered a static optimization since it requires complete knowledge about the latency of each task. Due to this, the low level instruction pipelining technique cannot be applied to dataflow type networks where the latency of the tasks can be unknown as it is a function of the input data. The next section details how to leverage the three basic paradigms that have been introduced to model different types of task parallelism.

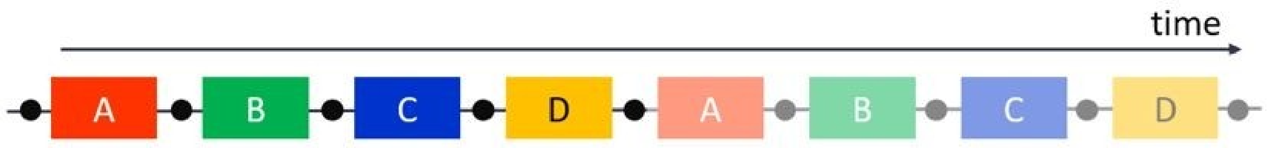
Combining the Three Paradigms

Functions and loops are the main focus of most optimizations in the user's program. Today's optimization tools typically operate at the function/procedure level. Each function can be converted into a specific hardware component. Each such hardware component is like a class definition and many objects (or instances) of this component can be created and instantiated in the eventual hardware design. Each hardware component will in turn be composed of many smaller predefined components that typically implement basic functions such as add, sub, and multiply. Functions may call other functions although recursion is not supported. Functions that are small and called less often can be also inlined into their callers just like how software functions can be inlined. In this case, the resources needed to implement the function are subsumed into the caller function's component which can potentially allow for better sharing of common resources. Constructing your design as a set of communicating functions lends to inferring parallelism when executing these functions.
Loops are one of the most important constructs in your program. Since the body of a loop is iterated over a number of times, this property can be easily exploited to achieve better parallelism. There are several transformations (such as pipelining and unrolling) that can be made to loops and loop nests in order to achieve efficient parallel execution. These transformations enable both memory-system optimizations as well as mapping to multi-core and SIMD execution resources. Many programs in science and engineering applications are expressed as operations over large data structures. These may be simple element-wise operations on arrays or matrices or more complex loop nests with loop-carried dependencies - i.e. data dependencies across the iterations of the loop. Such data dependencies impact the parallelism achievable in the loop. In many such cases, the code must be restructured such that loop iterations can be executed efficiently and in parallel on modern parallel platforms.
The following diagrams illustrate different overlapping executions for a simple example of 4 consecutive tasks (i.e., C/C++ functions) A, B, C, and D, where A produces data for B and C, in two different arrays, and D consumes data from two different arrays produced by B and C. Let us assume that this “diamond” communication pattern is to be run twice (two invocations) and that these two runs are independent.

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
   data_t c1[N], c2[N], c3[N], c4[N];
   #pragma HLS dataflow
   A(vecIn, c1, c2);
   B(c1, c3);
   C(c2, c4);
   D(c3, c4, vecOut);
}
```

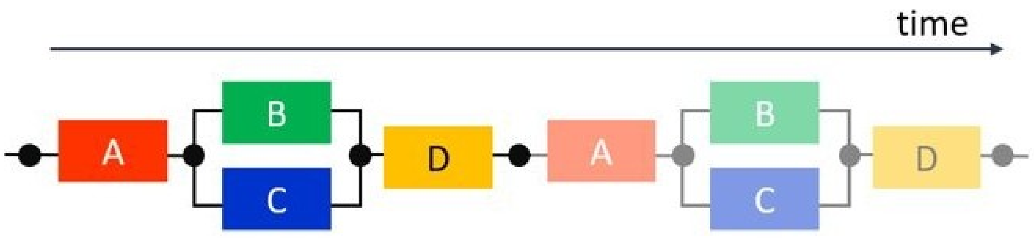
The code example above shows the C/C++ source snippet for how these functions are invoked. Note that tasks B and C have no mutual data dependencies. A fully-sequential execution corresponds to the figure below where the black circles represent some form of synchronization used to implement the serialization.

Figure: Sequential Execution - Two Runs



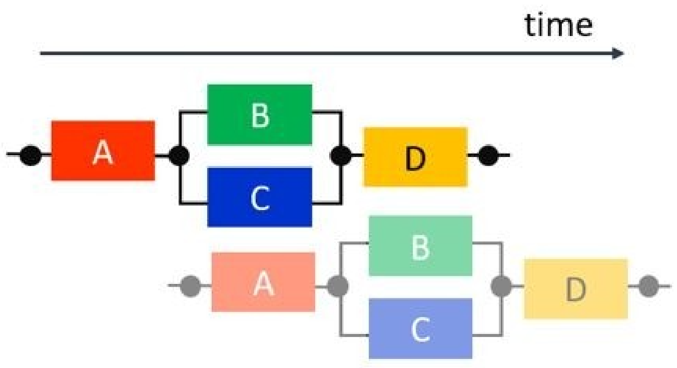
In the diamond example, B and C are fully-independent. They do not communicate nor do they access any shared memory resource, and so if no sharing of computation resource is required, they can be executed in parallel. This leads to the diagram in the figure below, with a form of fork-join parallelism within a run. B and C are executed in parallel after A ends while D waits for both B and C, but the next run is still executed in series.

Figure: Task Parallelism within a Run



Such an execution can be summarized as (A; (B || C); D); (A; (B || C); D) where ";" represents serialization and "||" represents full parallelism. This form of nested fork-join parallelism corresponds to a subclass of dependent tasks, namely series-parallel task graphs. More generally, any DAG (directed acyclic graph) of dependent tasks can be implemented with separate fork-and-join-type synchronization. Also, it is important to note that this is exactly like how a multithreaded program would run on a CPU with multiple threads and using shared memory. On FPGAs, you can explore what other forms of parallelism are available. The previous execution pattern exploited task-level parallelism within an invocation. What about overlapping successive runs? If they are truly independent, but if each function (i.e., A, B, C, or D) reuses the same computation hardware as for its previous run, you may still want to execute, for example, the second invocation of A in parallel with the first invocations of B and C. This is a form of task-level pipelining across invocations, leading to a diagram as depicted in the following figure. The throughput is now improved because it is limited by the maximum latency among all tasks, rather than by the sum of their latencies. The latency of each run is unchanged but the overall latency for multiple runs is reduced.

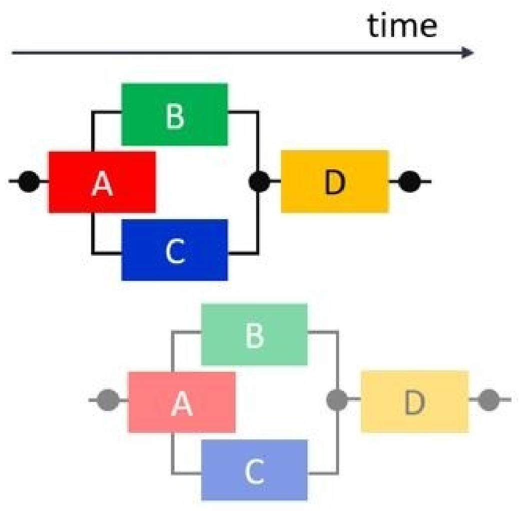
Figure: Task Parallelism with Pipelining



Now, however, when the first run of B reads from the memory where A placed its first result, the second run of A is possibly already writing in the same memory. To avoid overwriting the data before it is consumed, you can rely on a form of memory expansion, namely double buffering or PIPOs to allow for this interleaving. This is represented by the black circles between the tasks.

An efficient technique to improve throughput and reuse computational resources is to pipeline operators, loops, and/or functions. If each task can now overlap with itself, you can achieve simultaneously task parallelism within a run and task pipelining across runs, both of which are examples of macro-level parallelism. Pipelining within the tasks is an example of micro-level parallelism. The overall throughput of a run is further improved because it now depends on the minimum throughput among the tasks, rather than their maximum latency. Finally, depending on how the communicated data are synchronized, only after all are produced (PIPOs) or in a more element-wise manner (FIFOs), some additional overlapping within a run can be expected. For example, in the following figure, both B and C start earlier and are executed in a pipelined fashion with respect to A, while D is assumed to still have to wait for the completion of B and C. This last type of overlap within a run can be achieved if A communicates to B and C through FIFO streaming accesses (represented as lines without circles). Similarily, D can also be overlapped with B and C, if the channels are FIFOs instead of PIPOs. However, unlike all previous execution patterns, using FIFOs can lead to deadlocks and so these streaming FIFOs need to be sized correctly.

Figure: Task Parallelism and Pipelining within a Run, Pipelining of Runs, and Pipelining within a Task



In summary, the three paradigms presented in the earlier section show how parallelism can be achieved in your design without needing the complexities of multi-threading and/or parallel programming languages. The producer-consumer paradigm coupled with streaming channels allows for the composition of small to large scale systems easily. As mentioned before, streaming interfaces allow for easy coupling of parallel tasks or even hierarchical dataflow networks. This is in part due to the flexibility in the programming language (C/C++) to support such specifications and the tools to implement them on the heterogeneous computing platform available on today's FPGA devices.

Conclusion - A Prescription for Performance

The design concepts presented in this document have one main central principle - a model of parallel computation that favors encapsulation of state and sequential execution within modular units or tasks to facilitate a simpler programming model for parallel programming. Tasks are then connected together with streams (for synchronization and communication). A stream can be different types of channels such as FIFOs or PIPOs. The state/logic compartmentalization makes it much easier for tools (such as a compiler and a scheduler) to figure out where to run which pieces of an application and when. The second reason why stream-based processing is becoming popular is that it breaks the traditional multi-threading based “fork/join” view on parallel execution. By enabling task-level pipelining and instruction-level pipelining, the run-time can do many more concurrent actions than what is possible today with the fork/join model. This extra parallelism is critical to taking advantage of the hardware available on today's FPGA devices. In the same vein as enabling pipeline parallelism, streaming also enables designers to build parallel applications without having to worry about locks, race conditions, etc. that make parallel programming hard in the first place.

Finally, the following checklist of high-level actions is recommended as a prescription for achieving performance on reconfigurable FPGA platforms:

- Software written for CPUs and software written for FPGAs is fundamentally different. You cannot write code that is portable between CPU and FPGA platforms without sacrificing performance. Therefore, embrace and do not resist the fact that you will have to write significantly different software for FPGAs.
- Right from the start of your project, establish a flow that can functionally verify the source code changes that are being made. Testing the software against a reference model or using golden vectors are common practices.
- Focus first on the macro-architecture of your design. Consider modeling your solution using the producer-consumer paradigm.
- Once you have identified the macro-architecture of your design, draw the desired activity timeline where the horizontal axis represents time, and show when you expect each function to execute relative to each other over multiple iterations (or invocations). This will give you a sense of the expected parallelism in the design and can then be used to compare with the final achieved results. Often the HLS GUIs can be used to visualize this achieved parallelism.
- Only start coding or refactoring your program once you have the macro-architecture and the activity timeline well established
- As a general rule, the HLS compiler will only infer task-level parallelism from function calls. Therefore, sequential code blocks (such as loops) which need to run concurrently in hardware should be put into dedicated functions.
- Decompose/partition the original algorithm into smaller components that talk to each other via streams. This will give you some ideas of how the data flows in your design.
    - Smaller modular components have the advantage that they can be replicated when needed to improve parallelism.
    - Avoid having communication channels with very wide bit-widths. Decomposing such wide channels into several smaller ones will help implementation on FPGA devices.
    - Large functions (written by hand or generated by inlining smaller functions) can have non-trivial control paths that can be hard for tools to process. Smaller functions with simpler control paths will aid implementation on FPGA devices.
    - Aim to have a single loop nest (with either fixed loop bounds that can be inferred by HLS tool, or by providing loop trip count information by hand to the HLS tool) within each function. This greatly facilitates the measurement and optimization of throughput. While this may not be applicable for all designs, it is a good approach for a large majority of cases.
- Throughput - Having an overall vision about what rates of processing will be required during each phase of your design is important. Knowing this will influence how you write your application for FPGAs.
    - Think about the critical path (i.e., critical task level paths such as ABD or ACD) in your design and study what part of this critical path is potentially a bottleneck. Look at how individual tasks are pipelined and if different branches of a path are unaligned in terms of throughput by simulating the design. HLS GUI tools and/or the simulation waveform viewer can then be used to visualize such throughput issues.
    - Stream-based communication allows consumers to start processing as soon as producers start producing which allows for overlapped execution (which in turn increases parallelism and throughput).
    - In order to keep the producer and consumer tasks running constantly without any hiccups, optimize the execution of each task to run as fast as possible using techniques such as pipelining and the appropriate sizing of streams.
- Think about the granularity (and overhead) of the streaming channels with respect to synchronization. The usage of PIPO channels allows you to overlap task execution without the fear of deadlock while explicit manual streaming FIFO channels allow you to start the overlapped execution sooner (than PIPOs) but require careful adjustment of FIFO sizes to avoid deadlocks.
- Learn about synthesizable C/C++ coding styles.
- Use the reports generated by the HLS compiler to guide the optimization process.

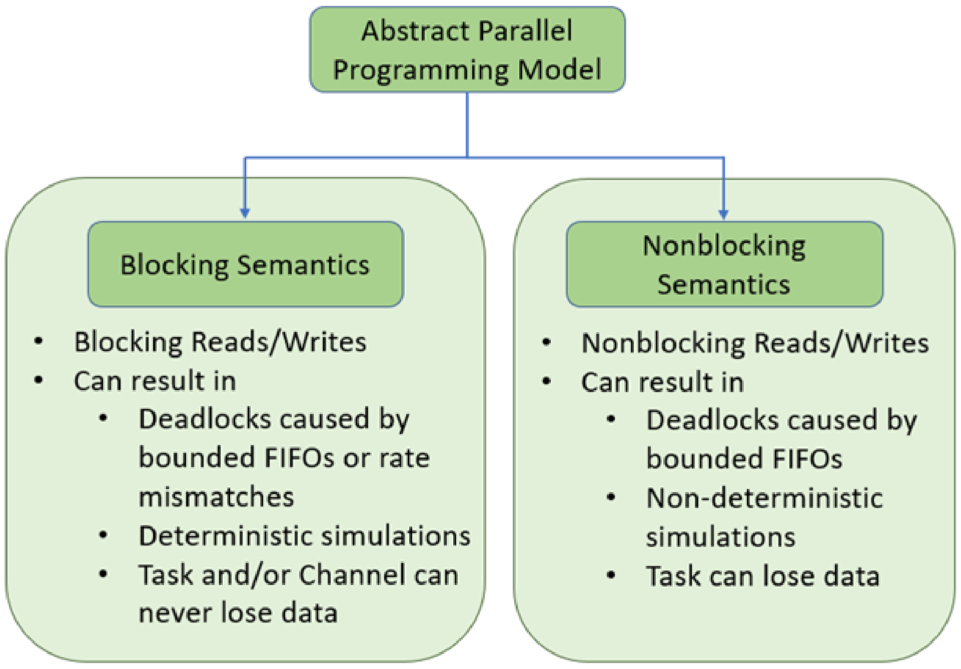
Keep the above checklist nearby so that you can refer to it from time to time. It summarizes the whole design activity needed to build a design that meets your performance goals.
Another important aspect of your design to consider next is the interface of your accelerated function or kernel. The interface of your kernel to the outside world is an important element of your eventual system design. Your kernel may need to plug into a bigger design, or to communicate with other kernels in a large system of kernels, or to communicate with memory or devices outside of the system. Best Practices for Designing with M_AXI Interfaces provides another checklist of items to consider when designing the external interfaces of your acceleration kernel.

Abstract Parallel Programming Model for HLS

In order to achieve high performance hardware, the HLS tool must infer parallelism from sequential code and exploit it to achieve greater performance. This is not an easy problem to solve. In addition, good software design often uses well-defined rules and practices such as run-time type information (RTTI), recursion, and dynamic memory allocation. Many of these techniques have no direct equivalence in hardware and present challenges for the HLS tool. This generally means that off-the-shelf software cannot be efficiently converted into hardware. At a bare minimum, such software needs to be examined for non-synthesizable constructs and the code needs to be refactored to make it synthesizable. Even if a software program can be automatically converted (or synthesized) into hardware, to assist the tool you need to understand the best practices for writing good software for execution on the FPGA device.
The Design Principles section introduced the three main paradigms that need to be understood for writing good software for FPGA platforms: producer-consumer, streaming data, and pipelining. The underlying parallel programming model that these paradigms work on is as follows:

- The design/program needs to be constructed as a collection of tasks that communicate by sending messages to each other through communication links (aka channels)
- Tasks can be structured as control-driven, waiting for some signal to start execution, or data-driven in which the presence of data on the channel drives the execution of the task
- A task consists of an executable unit that has some local storage/memory and a collection of input/output (I/O) ports.
- The local memory contains private data, i.e., the data to which the task has exclusive access
- Access to this private memory is called local data access - like data stored in BRAM/URAM. This type of access is fast. The only way that a task can send copies of its local data to other tasks is through its output ports, and conversely, it can only receive data through its input ports
- An I/O port is an abstraction; it corresponds to a channel that the task will use for sending or receiving data and it is connected by the caller of the module, or at the system integration level if it is a top port
- Data sent or received through a channel is called non-local data access. A channel is a data queue that connects one task's output port to another task's input port
- A channel is assumed to be reliable and has the following behaviors:
  - Data written at the output of the producer are read at the input port of the consumer in the same order for FIFOs. Data can be read/written in random order for PIPOs
  - No data values are lost
- Both blocking and non-blocking read and write semantics are supported for channels, as described in HLS Stream Library

Figure: Blocking/Non-Blocking Semantics


Abstract Parallel Programming Model
Blocking Semantics
Blocking Reads/Writes
Can result in
Deadlocks caused by bounded FIFOs or rate mismatches
Deterministic simulations
Task and/or Channel can never lose data
Nonblocking Semantics
Nonblocking Reads/Writes
Can result in
Deadlocks caused by bounded FIFOs
Non-deterministic simulations
Task can lose data

When blocking semantics are used in the model, a read to an empty channel results in the blocking of the reading process. Similarly, a write to a full channel results in the blocking of the writing process. The resulting process/channel network exhibits deterministic behavior that does not depend on the timing of computation nor on communication delays. These style of models have proven convenient for modeling embedded systems, high-performance computing systems, signal processing systems, stream processing systems, dataflow programming languages, and other computational tasks.
The blocking style of modeling can result in deadlocks due to insufficient sizing of the channel queue (when the channels are FIFOs) and/or due to differing rates of production between producers and consumers. If non-blocking semantics are used in the model, a read to an empty channel results in the reading of uninitialized data or in the re-reading of the last data item. Similarly, a write to a full queue can result in that data being lost. To avoid such loss of data, the design must first check the status of the queue before performing the read/write. But this will cause the simulation of such models to be non-deterministic since it relies on decisions made based on the run-time status of the channel. This will make verifying the results of this model much more challenging.
Both blocking and non-blocking semantics are supported by theVitis HLS abstract parallel programming model.

Control and Data Driven Tasks

Using this abstract model as the basis, two types of task-level parallelism (TLP) models can be used to structure and design your application. TLP can be data-driven or control-driven, or can mix control-driven and data-driven tasks in a single design. The main differences between these two models are:

- If your application is purely data-driven, does not require any interaction with external memory and the functions can execute in parallel with no data dependencies, then the data-driven TLP model is the best fit. You can design a purely data-driven model that is always running, requires no control, and reacts only to data. For additional details refer to Data-driven Task-level Parallelism.
- If your application requires some interaction with external memory, and there are data dependencies between the tasks that execute in parallel, then the control-driven TLP model is the best fit. Vitis HLS will infer the parallelism between tasks and create the right channels (as defined by you) such that these functions can be overlapped during execution. The control-driven TLP model is also known as the dataflow optimization in Vitis HLS as described in Control-driven Task-level Parallelism.

The next few sections describe these major modeling options that are available to use. You can use any of these models to write your source code using C++ in order to optimize the execution of the program on parallel hardware.

Data-driven Task-level Parallelism

Data-driven task-level parallelism uses a task-channel modeling style that requires you to statically instantiate and connect tasks and channels explicitly. Tasks in this modeling style only have stream type inputs and outputs. The tasks are not controlled by any function call/return semantics but rather are always running waiting for data on their input stream.
Data-driven TLP models are tasks that execute when there is data to be processed. In Vitis HLS C-simulation used to be limited to seeing only the sequential semantics and behavior. With the data-driven model it is possible during simulation to see the concurrent nature of parallel tasks and their interactions via the FIFO channels.
Implementing data-driven TLP in the Vitis HLS tool uses simple classes for modeling tasks (`hls::task`) and channels (`hls::stream`/`hls::stream_of_blocks`)

‼ **Important:** While Vitis HLS supports `hls::tasks` for a top-level function, you cannot use `hls::stream_of_blocks` for interfaces in top-level functions.

Consider the simple task-channel example shown below:

```
#include "test.h"

void splitter(hls::stream<int> &in, hls::stream<int> &odds_buf, hls::stream<int> &evens_buf)
{
    int data = in.read();
    if (data % 2 == 0)
        evens_buf.write(data);
    else
        odds_buf.write(data);
}

void odds(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 1);
}

void evens(hls::stream<int> &in, hls::stream<int> &out) {
    out.write(in.read() + 2);
}

void odds_and_evens(hls::stream<int> &in, hls::stream<int> &out1, hls::stream<int> &out2) {
    hls_thread_local hls::stream<int> s1; // channel connecting t1 and t2
    hls_thread_local hls::stream<int> s2; // channel connecting t1 and t3

    // t1 infinitely runs function splitter, with input in and outputs s1 and s2
    hls_thread_local hls::task t1(splitter, in, s1, s2);
    // t2 infinitely runs function odds, with input s1 and output out1
    hls_thread_local hls::task t2(odds, s1, out1);
    // t3 infinitely runs function evens, with input s2 and output out2
    hls_thread_local hls::task t3(evens, s2, out2);
}
```

The special `hls::task` C++ class is:

- A new object declaration in your source code that requires a special qualifier. The `hls_thread_local` qualifier is required in order to keep the object (and the underlying thread) alive across multiple calls of the instantiating function (`odds_and_evens` in the example).
  The `hls_thread_local` qualifier is only required to ensure that the C simulation of the data-driven TLP model exhibits the same behavior as the RTL simulation. In the RTL, these functions are already in always running mode once started. In order to ensure the same behavior during C Simulation, the `hls_thread_local` qualifier is required to ensure that each task is started only once and keeps the same state even when called multiple times. Without the `hls_thread_local` qualifier, each new invocation of the function would result in a new state.
- Task objects implicitly manage a thread that runs a function infinitely, passing to it a set of arguments that must be either `hls::stream` or `hls::stream_of_blocks`

  ★ **Tip:** No other types of arguments are supported. In particular, even constant values cannot be passed as function arguments. If constants need to be passed to the task's body, define the function as a templated function and pass the constant as a template argument to this templated function.
- The supplied function (`splitter/odds/evens` in the example above) is called the task body, and it has an implicit infinite loop wrapped around it to ensure that the task keeps running and waiting on input.
- The supplied function can contain pipelined loops but they need to be flushable pipelines (FLP) in order to prevent deadlock. The tool will automatically select the right pipeline style to use for a given pipelined loop or function.

‼ **Important:** An `hls:task` should not be treated as a function call - instead a `hls::task` needs to be thought of as a persistent instance statically bound to channels. Due to this, it will be your responsibility to ensure that multiple invocations to any function that contains `hls::tasks` be uniquified or these calls will use the same `hls::tasks` and channels.

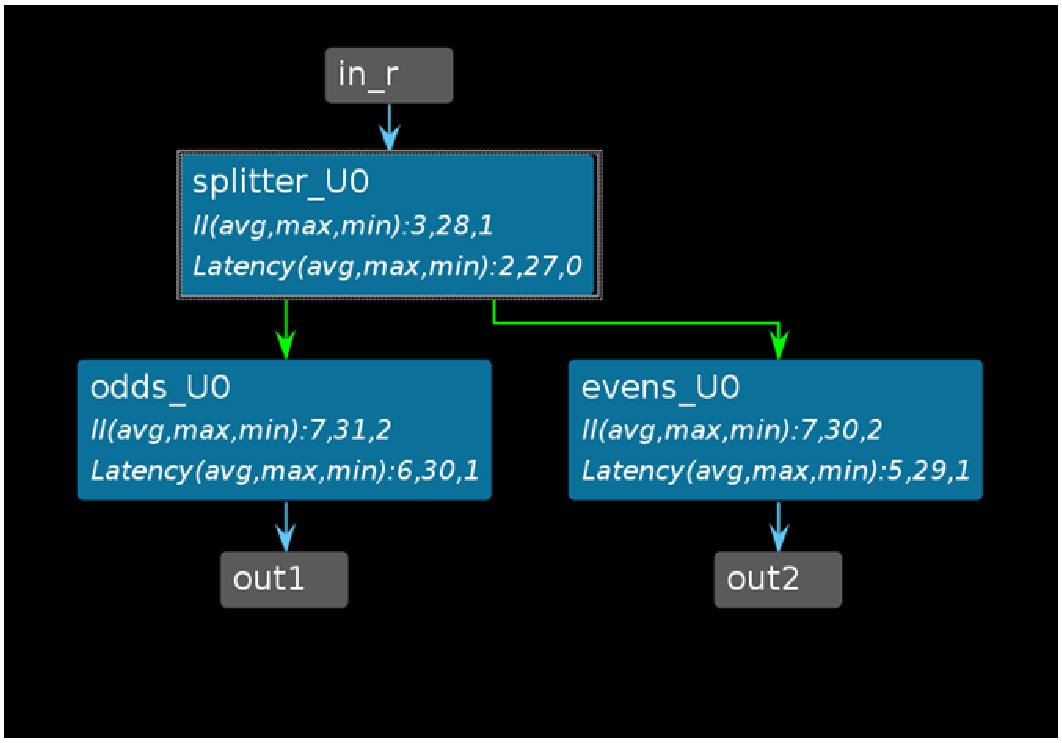
Channels are modeled by the special templatized `hls::stream` (or `hls::stream_of_blocks`) C++ class. Such channels have the following attributes:

- In the data-driven TLP model, an `hls::stream<type,depth>` object behaves like a FIFO with a specified depth. Such streams have a default depth of 2 which can be overridden by the user.
- The streams are read from and written to sequentially. That implies that once a data item is read from an `hls::stream<>` that same data item cannot be read again.

  ★ **Tip:** Accesses to different streams are not ordered (e.g. the order of a write to a stream and a read from a different stream can be changed by the scheduler).
- Streams may be defined either locally or globally. Streams defined in the global scope follow the same rules as any other global variables.
- The `hls_thread_local` qualifier is also required for streams (`s1` and `s2` in the example below) in order to keep the same streams alive across multiple calls of the instantiating function (`odds_and_evens` in the code example below).

The following diagram shows the graphical representation in Vitis HLS of the code example above. In this diagram, the green colored arrows are FIFO channels while the blue arrows indicate the inputs and outputs of the instantiating function (`odds_and_evens`). Tasks are shown as blue rectangular boxes.

Figure: Dataflow Diagram of `hls::task` Example



Due to the fact that a read of an empty stream is a blocking read, deadlocks can occur due to:

- The design itself, where the production and consumption rates by processes are unbalanced.
  - During C simulation, deadlocks can occur only due to a cycle of processes, or a chain of processes starting from a top-level input, that are attempting to read from empty channels.
  - Deadlocks can occur during both C/RTL Co-simulation and when running in hardware (HW) due to cycles of processes trying to write to full channels and/or reading from empty channels.
- The test bench, which is providing less data than those that are needed to produce all the outputs that the test bench is expecting when checking the computation results.

Due to this, a deadlock detector is automatically instantiated when the design contains an `hls::task`. The deadlock detector detects deadlocks and stops the C simulation. Further debugging is performed using a C debugger such as gdb and looking at where the simulated `hls::tasks` are all blocked trying to read from an empty channel. Note that this is easy to do using the Vitis HLS GUI as shown in the handling_deadlock example for debugging deadlocks.
In summary, the `hls::task` model is recommended if your design requires a completely data-driven, pure streaming type of behavior, with no sort of control. This type of model is also useful in modeling feedback and dynamic multi-rate designs. Feedback in the design occurs when there is a cyclical dependency between tasks. Dynamic multi-rate models, where the producer writes data or consumer reads data at a rate that is data dependent, can only be handled by the data-driven TLP. The simple_data_driven design on GitHub is an example of this.

Note: Static multi-rate designs, in which the producer writes data or consumer reads data at rates that are data-independent, can be managed by both data-driven and control-driven TLP. For example, the producer writes two values in a stream for each call, the consumer reads one value per call.

Control-driven Task-level Parallelism

Control-driven TLP is useful to model parallelism while relying on the sequential semantics of C++, rather than on continuously running threads. Examples include functions that can be executed in a concurrent pipelined fashion, possibly within loops, or with arguments that are not channels but C++ scalar and array variables, both referring to on-chip and to off-chip memories. For this kind of model, Vitis HLS introduces parallelism where possible while preserving the behavior obtained from the original C++ sequential execution. The control-driven TLP (or dataflow) model provides:

- A subsequent function can start before the previous finishes
- A function can be restarted before it finishes
- Two or more sequential functions can be started simultaneously

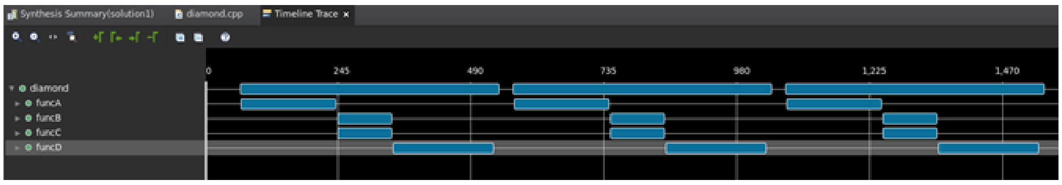
While using the dataflow model, Vitis HLS implements the sequential semantics of the C++ code by automatically inserting synchronization and communication mechanisms between the tasks.
The dataflow model takes this series of sequential functions and creates a task-level pipeline architecture of concurrent processes. The tool does this by inferring the parallel tasks and channels. The designer specifies the region to model in the dataflow style (i.e., a function body or a loop body) by specifying the DATAFLOW pragma or directive as shown below. The tool scans the loop/function body, extracts the parallel tasks as parallel processes, and establishes communication channels between these processes. The designer can additionally guide the tool to select the type of channels - i.e., FIFO (`hls::stream` or `#pragma HLS STREAM`) or PIPO or `hls::stream_of_blocks`. The dataflow model is a powerful method for improving design throughput and latency.
In order to understand how Vitis HLS transforms your C++ code into the dataflow model, refer to the simple_fifos example shown below. The example applies the dataflow model to the top-level `diamond` function using the DATAFLOW pragma as shown.

```
#include "diamond.h"

void diamond(data_t vecIn[N], data_t vecOut[N])
{
  data_t c1[N], c2[N], c3[N], c4[N];
#pragma HLS dataflow
  funcA(vecIn, c1, c2);
  funcB(c1, c3);
  funcC(c2, c4);
  funcD(c3, c4, vecOut);
}
```

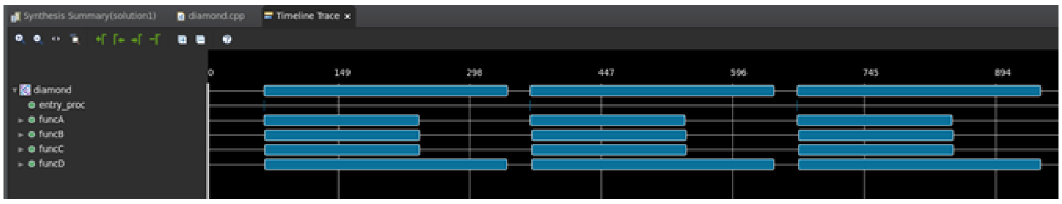
In the above example, there are four functions: `funcA`,`funcB`,`funcC` and `funcD`. `funcB` and `funcC` do not have any data dependencies between them and therefore can be executed in parallel. `funcA` reads from the non-local memory (`vecIn`) and needs to be executed first. Similarly, `funcD` writes to the non-local memory (`vecOut`) and therefore has to be executed last.
The following waveform shows the execution profile of this design without the dataflow model. There are three calls to the function diamond from the test bench. `funcA`,(`funcB`, `funcC`) and `funcD` are executed in sequential order. Each call to diamond, therefore, takes 475 cycles in total as shown in the figure below.

Figure: Diamond Example without Dataflow



In the following figure, when the dataflow model is applied and the designer selected to use FIFOs for channels, all the functions are started immediately by the controller and are stalled waiting on input. As soon as the input arrives, it is processed and sent out. Due to this type of overlap, each call to diamond now only takes 275 cycles in total as shown below. Refer to Combining the Three Paradigms for a more detailed discussion of the types of parallelism that can be achieved for this example.

Figure: Diamond Example with Dataflow



This type of parallelism cannot be achieved without incurring an increase in hardware utilization. When a particular region, such as a function body or a loop body, is identified as a region to apply the dataflow model, Vitis HLS analyzes the function or loop body and creates individual channels from C++ variables (such as scalars, arrays, or user-defined channels such as `hls::streams` or `hls::stream_of_blocks`) that model the flow of data in the dataflow region. These channels can be simple FIFOs for scalar variables, or ping-pong (PIPO) buffers for non-scalar variables like arrays (or stream of blocks when you need a combination of FIFO and PIPO behavior with explicit locking of the blocks).
Each of these channels can contain additional signals to indicate when the channel is full or empty. By having individual FIFOs and/or PIPO buffers, Vitis HLS frees each task to execute independently and the throughput is only limited by the availability of the input and output buffers. This allows for better overlapping of task execution than a normal pipelined implementation, but does so at the cost of additional FIFO or block RAM registers for the ping-pong buffer.

★ **Tip:** This overlapped execution optimization is only visible after you run the cosimulation of the design - it is not observable statically (though can be easily imagined in the Dataflow Viewer after C Synthesis).

The dataflow model is not limited to a chain of processes but can be used on any directed acyclic graph (DAG) structure, or cyclic structure when using `hls::streams`. It can produce two different forms of overlapping: within an iteration if processes are connected with FIFOs, and across different iterations when connected with PIPOs and FIFOs. This potentially improves performance over a statically pipelined solution. It replaces the strict, centrally-controlled pipeline stall philosophy with a distributed handshaking architecture using FIFOs and/or PIPOs. The replacement of the centralized control structure with a distributed one also benefits the fanout of control signals, for example register enables, which is distributed among the control structures of individual processes. Refer to the Task Level Parallelism/Control-Driven examples on Github for more examples of these concepts.

Canonical Forms

Vitis HLS transforms the region to apply the DATAFLOW optimization. For more predictibility of the resulting dataflow network, AMD recommends writing the code inside this region (referred to as the canonical region) using canonical forms. There are two main canonical forms for the dataflow optimization:

1. The canonical form for a function where sub-functions are not inlined. Note that these subfunctions can themselves be dataflow in function regions or dataflow inside loop regions. Note also that variable initialization (including those performed automatically by constructors) or passing expressions by value to processes are not part of canonical form. Vitis HLS does its best to implement the resulting dataflow but, if the code is not in canonical form, you should always check the GUI dataflow viewer and the cosimulation timeline trace to ensure that the dataflow happens as expected and the achieved performance is as expected.

```
void dataflow(Input0, Input1, Output0, Output1)
{
 #pragma HLS dataflow
 UserDataType C0, C1, C2;       // UserDataType can be scalars or arrays
 func1(Input0, Input1, C0, C1); // read Input0, read Input1, write C0, write C1
 func2(C0, C1, C2);             // read C0, read C1, write C2
 func3(C2, Output0, Output1);   // read C2, write Output0, write Output1
}
```

2. Dataflow inside a loop body enclosed in a function without any other code but the loop. For the `for` loop (where no function inside is inlined), the integral loop variable should have:
   a. The initial value is declared in the loop header and set to 0.
   b. The loop bound is a non-negative numerical constant or scalar argument of the function that encloses the loop.
   c. Increment by 1.
   d. Dataflow pragma needs to be inside the loop as shown below.

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

Canonical Body

Inside the canonical region, the canonical body should follow these guidelines:

1. Use a local, non-static scalar or an array variable. A local variable is declared inside the function body (for dataflow in a function) or loop body (for dataflow inside a loop). Refer to Limitations of Control-Driven Task-Level Parallelism for additional limitations on arrays.
2. A sequence of function calls that pass data forward (with no feedback unless using `hls::stream/hls::stream_of_blocks`), from a function to one that is lexically later, under the following conditions:
   a. Variables (except scalar) can have only one reading process and one writing process.
   b. Use write before read (producer before consumer) if you are using local non-scalar variables, which then become channels. For scalar variables, both write before read and read before write are allowed.
   c. Use read before write (consumer before producer) if you are using function arguments. Any intra-body anti-dependencies must be preserved by the design.
   d. Function return type must be void.
   e. No loop-carried dependencies are allowed among different processes via variables except when FIFOs are used. Forward loop-carried dependencies are supported for arrays transformed to streams and both forward and backward dependencies are supported for `hls::streams`.
      - Except when these dependencies exist across successive calls to the top function (i.e., inout argument written by one iteration and read by the following iteration).
   f. No control whatsoever is supported inside a dataflow region, except inside function calls (that define processes).

  - For canonical dataflow, there should be no conditionals, no loops, no return or goto statements, and no C++ exceptions such as throw.

Dataflow Checking

Vitis HLS has a dataflow checker which, when enabled, checks the code to see if it is in the recommended canonical form. Otherwise, it will emit an error/warning message to the user. By default, this checker is set to `warning`. You can set the checker to `error` or disable it by selecting `off` in the strict mode of the `config_dataflow` TCL command:

```
config_dataflow -strict_mode  (off | error | warning)
```

Configuring Dataflow Memory Channels

Vitis HLS implements channels between the tasks as either PIPO or FIFO buffers, depending on the user's choice:

- For scalars, Vitis HLS will automatically infer FIFOs as the channel type.
- If the parameter (of a producer or consumer) is an array, the user has a choice of implementing the channel as a PIPO or a FIFO based on the following considerations:
  - If the data is always accessed in sequential order, the user can choose to implement this memory channel as PIPO/FIFO. Choosing PIPOs comes with the advantage that PIPOs can never deadlock but they require more memory to use. Choosing FIFOs offers the advantage of lesser memory requirements but this comes with the risk of deadlock if the FIFO sizes are not correct.
  - If the data is accessed in an arbitrary manner, the memory channel must be implemented as a PIPO (with a default size that is twice the size of the original array).

  ★ **Tip:** A PIPO ensures that the channel always has the capacity to hold all samples produced in one iteration, without a loss.

Specifying the size of the FIFO channels overrides the default value that is computed by the tool to attempt to optimize the throughput. If any function in the design can produce or consume samples at a greater rate than the specified size of the FIFO, the FIFOs might become empty (or full). In this case, the design halts operation, because it is unable to read (or write). This might lead to a stalled, deadlock state.

★ **Tip:** If a deadlocked situation is created, you will only see this when executing C/RTL co-simulation or when the block is used in a complete system.

When setting the depth of the FIFOs, AMD recommends initially setting the depth as the maximum number of data values transferred (for example, the size of the array passed between tasks), confirming the design passes C/RTL co-simulation, and then reducing the size of the FIFOs and confirming C/RTL co-simulation still completes without issues. If RTL co-simulation fails, the size of the FIFO is likely too small to prevent stalling or a deadlock situation. The Vitis HLS GUI now supports an automatic way of determining the right FIFO size to use. Additionally, the Vitis HLS IDE can display a histogram of the size of each FIFO/PIPO buffer over time, after RTL co-simulation has been run. This can be useful to help determine the best depth for each buffer.

Specifying Arrays as PIPOs or FIFOs

All arrays are implemented by default as ping-pong to enable random access. These buffers can also be re-sized if needed. For example, in some circumstances, such as when a task is being bypassed, performance degradation is possible. To mitigate this effect on performance, you can give more slack to the producer and consumer by increasing the size of these buffers by using the HLS STREAM pragma or directive as shown below.

```
void top ( ... ) {
#pragma HLS dataflow
  int A[1024];
#pragma HLS stream type=pipo variable=A depth=3

  producer(A, B, …);  // producer writes A and B
  middle(B, C, ...);  // middle reads B and writes C
  consumer(A, C, …);  // consumer reads A and C
```

Arrays on the interface are defined as `m_axi` or `ap_memory` by default. However, arrays can be specified as streaming if the INTERFACE pragma or directive defines the array as `ap_fifo` or `axis`.
Inside the design, an array must be specified as streaming using the STREAM pragma or directive if a FIFO is desired for the implementation.

★ **Tip:** When the STREAM directive is applied to an array, the resulting FIFO implemented in the hardware contains as many elements as the array. The `-depth` option can be used to specify the size of the FIFO.

The HLS STREAM pragma or directive can also be used to override the default implementation specified by the `config_dataflow` command for any arrays in a DATAFLOW region. In general, the use of the STREAM pragma or directive for a given array is preferable to the global option.

- If the `config_dataflow -default_channel` is set as pingpong, any array can still be implemented as a FIFO by applying the STREAM directive to the array.

  ★ **Tip:** To use a FIFO implementation, the array must be accessed in sequential order (and not in random access order).
- If the `config_dataflow -default_channel` is set to FIFO, any array can still be implemented as a ping-pong implementation by applying the `STREAM type=pipo` pragma or directive to the array.

‼ **Important:** To preserve sequential accesses, and thus the correctness of streaming, it might be necessary to prevent compiler optimizations (dead code elimination particularly) by using the `volatile` qualifier.

When an array in a DATAFLOW region is specified as streaming and implemented as a FIFO, the FIFO is typically not required to hold the same number of elements as the original array. The tasks in a DATAFLOW region consume each data sample as soon as it becomes available. The depth of the FIFO can be specified using the `config_dataflow -fifo_depth` option or the STREAM pragma or directive with the `-depth` option. This can be used to set the size of the FIFO to ensure the flow of data never stalls.
If the `-type=pipo` option is selected, the `-depth` option sets the depth (number of blocks) of the PIPO.

Specifying Arrays as Stream-of-Blocks

The `hls::stream_of_blocks` type provides a user-synchronized stream that supports streaming blocks of data for process-level interfaces in a dataflow context, where each block is an array or multidimensional array. The intended use of stream-of-blocks is to replace array-based communication between a pair of processes within a dataflow region. Refer to the using_stream_of_blocks example on Github.
Currently, Vitis HLS implements arrays written by a producer process and read by a consumer process in a dataflow region by mapping them to ping pong buffers (PIPOs). The buffer exchange for a PIPO buffer occurs at the return of the producer function and the calling of the consumer function in C++.

Stream-of-Blocks Modeling Style

On the other hand, for a stream-of-blocks the communication between the producer and the consumer is modeled as a stream of array-like objects, providing several advantages over array transfer through PIPO.
The use of stream-of-blocks in your code requires the following include file:

```
#include "hls_streamofblocks.h"
```

The stream-of-blocks object template is:

```
hls::stream_of_blocks<block_type, depth> v
```

Where:

- `<block_type>` specifies the datatype of the array or multidimensional array held by the stream-of-blocks
- `<depth>` is an optional argument that provides depth control just like `hls::stream` or PIPOs, and specifies the total number of blocks, including the one acquired by the producer and the one acquired by the consumer at any given time. The default value is 2
- v specifies the variable name for the stream-of-blocks object

Use the following steps to access a block in a stream of blocks:

1. The producer or consumer process that wants to access the stream first needs to acquire access to it, using a `hls::write_lock` or `hls::read_lock` object.
2. After the producer has acquired the lock it can start writing (or reading) the acquired block. Once the block has been fully initialized, it can be released by the producer, when the `write_lock` object goes out of scope.
   Note: The producer process with a `write_lock` can also read the block as long as it only reads from already written locations, because the newly acquired buffer must be assumed to contain uninitialized data. The ability to write and read the block is unique to the producer process, and is not supported for the consumer.
3. Then the block is queued in the stream-of-blocks in a FIFO fashion, and when the consumer acquires a `read_lock` object, the block can be read by the consumer process.

The main difference between `hls::stream_of_blocks` and the PIPO mechanism seen in the prior examples is that the block becomes available to the consumer as soon as the `write_lock` goes out of scope, rather than only at the return of the producer process. Therefore the amount of storage is much less with stream-of-blocks than with just PIPOs: namely 2N instead of 2xMxN.
The producer acquires the block by constructing an `hls::write_lock` object called `b`, and passing it the reference to the stream-of-blocks object, called `s`. The `write_lock` object provides an overloaded array access operator, letting it be accessed as an array to access underlying storage in random order as shown in the example below.
The acquisition of the lock is performed by constructing the `write_lock/read_lock` object, and the release occurs automatically when that object is destructed as it goes out of scope. This approach uses the common *Resource Acquisition Is Initialization* (RAII) style of locking and unlocking.

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer (hls::stream_of_blocks<buf> &s, ...) {
  for (int i = 0; i < M; i++) {
    // Allocation of hls::write_lock acquires the block for the producer
```

```
      hls::write_lock<buf> b(s);
      for (int j = 0; j < N; j++)
        b[f(j)] = ...;
      // Deallocation of hls::write_lock releases the block for the consumer
    }
  }

  void consumer(hls::stream_of_blocks<buf> &s, ...) {
    for (int i = 0; i < M; i++) {
      // Allocation of hls::read_lock acquires the block for the consumer
      hls::read_lock<buf> b(s);
      for (int j = 0; j < N; j++)
         ... = b[g(j)] ...;
      // Deallocation of hls::write_lock releases the block to be reused by the producer
    }
  }

  void top(...) {
  #pragma HLS dataflow
    hls::stream_of_blocks<buf> s;

    producer(b, ...);
    consumer(b, ...);
  }
```

The key features of this approach include:

- The expected performance of the outer loop in the producer above is to achieve an overall Initiation Interval (II) of 1
- A locked block can be used as though it were private to the producer or the consumer process until it is released.
- The initial state of the array object for the producer is undefined, whereas it contains the values written by the producer for the consumer.
- The principal advantage of stream-of-blocks is to provide overlapped execution of multiple iterations of the consumer and the producer to increase throughput.

Resource Usage

The resource cost when increasing the depth beyond the default value of 2 is similar to the resource cost of PIPOs. Namely, each increment of 1 will require enough memory for a block, e.g., in the example above N * 32-bit words.
The stream of blocks object can be bound to a specific RAM type, by placing the `BIND_STORAGE` pragma where the stream-of-blocks is declared, for example in the top-level function. The stream of blocks uses 2-port BRAM (`type=RAM_2P`) by default.

Specifying Compiler-Created FIFO Depth

Start Propagation FIFOs

The compiler might automatically create a start FIFO to propagate the `ap_start/ap_ready` handshake to an internal process. Such FIFOs can sometimes be a bottleneck for performance, in which case you can increase the default size (which can be incorrectly estimated by the tool) with the following command:

```
config_dataflow -start_fifo_depth <value>
```

If an unbounded slack between producer and consumer is needed, and internal processes can run forever, fully and safely driven by their inputs or outputs (FIFOs or PIPOs), these start FIFOs can be removed, at user's risk, locally for a given dataflow region with the pragma:

```
#pragma HLS DATAFLOW disable_start_propagation
```

Scalar Propagation FIFOs

The compiler automatically propagates some scalars from C/C++ code through scalar FIFOs between processes. Such FIFOs can sometimes be a bottleneck for performance or cause deadlocks, in which case you can set the size (the default value is set to `-fifo_depth`) with the following command:

```
config_dataflow -scalar_fifo_depth <value>
```

Stable Arrays

The `stable` pragma can be used to mark input or output variables of a dataflow region. Its effect is to remove their corresponding task-level synchronizations, assuming that the user guarantees this removal is indeed correct.

```
void dataflow_region(int A[...], ...
#pragma HLS stable variable=A
#pragma HLS dataflow
    proc1(...);
    proc2(A, ...);
```

Without the `stable` pragma, and assuming that `A` is read by `proc2`, then `proc2` would be part of the initial synchronization for the dataflow region where it is located. This means that `proc1` would not restart until `proc2` is also ready to start again, which would prevent dataflow iterations to be overlapped and induce a possible loss of performance. The `stable` pragma indicates that this synchronization is not necessary to preserve correctness.
With the `stable` pragma, the compiler assumes that:

- If `A` is read by `proc2`, then the memory locations that are read are still accessible and will not be overwritten by any other process or calling context, while the `dataflow_region` is being executed.
- If `A` is written by `proc2`, then the memory locations written will not be read, before their definition, by any other process or calling context, while `dataflow_region` is being executed.

A typical scenario is when the caller updates or reads these variables only when the dataflow region has not started yet or has completed execution.
In summary, the Dataflow optimization is a powerful optimization that can significantly improve the throughput of your design. As there is reliance on the HLS tool to do the inference of the available parallelism in your design, it requires the designer's help to ensure that the code is written in such a way that the inference is straightforward for the HLS tool. Finally, there will be situations where the designer might see the need to deploy both the Dataflow model and the Task-Channel model in the same design. The next section describes this hybrid combination model that can lead to some interesting designs.

Mixing Data-Driven and Control-Driven Models

The following table highlights factors of the HLS design that can help you determine when to apply control-driven task-level parallelism (TLP) or data-driven TLP.

Control-Driven TLP	Data-Driven TLP
• HLS Design requires control signals to start/stop the process • Design requires non-local memory access • Design requires interaction with an external software application • Designs with multiple processes running the same number of executions • Requires RTL simulation to model the effect of the parallelism	• HLS Design uses a completely data-driven approach that does not require control signals to start/stop the process • Design uses pure streaming data transfers • Designs with data-dependent multi-rate behavior ◦ Producer writes data or consumer reads data at a rate that is data dependent ◦ Easier to model for designs that require feedback between processes • Task-level parallelism is observable in C simulation as well as RTL simulation.

As the above table indicates, the two forms of task-level parallelism presented have different use cases and advantages. However, sometimes it is not possible to design an entire application that is purely data-driven TLP, while some portion of the design can still be constructed as a purely streaming design. In this case a mixed control-driven/data-driven model can be useful to create the application. Consider the following mixed_control_and_data_driven example from GitHub.

```
void dut(int in[N], int out[N], int n) {
#pragma HLS dataflow
  hls_thread_local hls::split::round_robin<int, NP> split1;
  hls_thread_local hls::merge::round_robin<int, NP> merge1;

  read_in(in, n, split1.in);

  // Task-Channels
  hls_thread_local hls::task t[NP];
  for (int i=0; i<NP; i++) {
#pragma HLS unroll
    t[i](worker, split1.out[i], merge1.in[i]);
  }

  write_out(merge1.out, out, n);
}
```

In the above example, there are two distinct regions - a dataflow region that has the functions read_in/write_out in which the sequential semantics is preserved - i.e. `read_in` will be executed before `write_out` and a task-channel region that contains the dynamic instantiation of 4 tasks (since NP = 4 in this example) along with some special type of channels called a `split` or a `merge` channel. A split channel is one that has a single input but has multiple outputs - in this case, the split channel has 4 outputs as described in HLS Split/Merge Library. Similarly, a merge channel has multiple inputs but only one output.

Vitis Application Acceleration Development Flow Tutorials (https://github.com/Xilinx/Vitis-Tutorials)
Provides a number of tutorials that can be worked through to teach specific concepts regarding the tool flow and application development, including the use of Vitis HLS as a standalone application, and in the Vitis bottom up design flow.

HLS Programmers Guide

Introduction

This Programmers Guide is intended to provide real world design techniques, and details of hardware design which will help you get the most out of the AMD Vitis™ HLS tool. This guide provides details on programming techniques you should apply when writing C/C++ code for high-level synthesis into RTL code, and a checklist of best practices to follow when creating IP that uses the AXI4 interfaces. Finally, it details various optimization techniques that can improve the performance of your code, improving both the fit and function of the resulting hardware design.
This section contains the following chapters:

Design Principles

Introduction

You might be working with the HLS tool to take advantage of productivity gains from writing C/C++ code to generate RTL for hardware; or you might be looking to accelerate portions of a C/C++ algorithm to run on custom hardware implemented in programmable logic. This chapter is intended to help you understand the process of synthesizing hardware from a software algorithm written in C/C++. This document introduces the fundamental concepts used to design and create good synthesizable software in such a way that it can be successfully converted to hardware using high-level synthesis (HLS) tools. The discussion in this document will be tool-agnostic and the concepts introduced are common to most HLS tools. For experienced designers, reviewing this material can provide a useful reinforcement of the importance of these concepts; help you understand how to approach HLS, and in particular how to structure HLS code to achieve high-performance designs.

Throughput and Performance

C/C++ functions implemented as custom hardware in programmable logic can run at a significantly faster rate than what is achievable on traditional CPU/GPU architectures, and achieve higher processing rates and/or performance. Let us first establish what these terms mean in the context of hardware acceleration. *Throughput* is defined as the number of specific actions executed per unit of time or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. For example, the term "memory bandwidth" is sometimes used to specify the throughput of the memory systems. Similarly, *performance* is defined as not just higher throughput but higher throughput with low power consumption. Lower power consumption is as important as higher throughput in today's world.

Architecture Matters

In order to better understand how custom hardware can accelerate portions of your program, you will first need to understand how your program runs on a traditional computer. The von Neumann architecture is the basis of almost all computing done today even though it was designed more than 7 decades ago. This architecture was deemed optimal for a large class of applications and has tended to be very flexible and programmable. However, as application demands started to stress the system, CPUs began supporting the execution of multiple processes. Multithreading and/or Multiprocessing can include multiple *system processes* (For example: executing two or more programs at the same time), or it can consist of one process that has multiple *threads* within it. Multi-threaded programming using a shared memory system became very popular as it allowed the software developer to design applications with parallelism in mind but with a fixed CPU architecture. But when multi-threading and the ever-increasing CPU speeds could no longer handle the data processing rates, multiple CPU cores and hyperthreading were used to improve throughput as shown in the figure on the right.
This general purpose flexibility comes at a cost in terms of power and peak throughput. In today's world of ubiquitous smart phones, gaming, and online video conferencing, the nature of the data being processed has changed. To achieve higher throughput, you must move the workload closer to memory, and/or into specialized functional units. So the new challenge is to design a new programmable architecture in such a way that you can maintain just enough programmability while achieving higher performance and lower power costs.
A field-programmable gate array (FPGA) provides for this kind of programmability and offers enough memory bandwidth to make this a high-performance and lower power cost solution. Unlike a CPU that executes a program, an FPGA can be configured into a custom hardware circuit that will respond to inputs in the same way that a dedicated piece of hardware would behave. Reconfigurable devices such as FPGAs contain computing elements of extremely flexible granularities, ranging from elementary logic gates to complete arithmetic-logic units such as digital signal processing (DSP) blocks. At higher granularities, user-specified composable units of logic called kernels can then be strategically placed on the FPGA device to perform various roles. This characteristic of reconfigurable FPGA devices allows the creation of custom macro-architectures and gives FPGAs a big advantage over traditional CPUs/GPUs in utilizing application-specific parallelism. Computation can be spatially mapped to the device, enabling much higher operational throughput than processor-centric platforms. Today's latest FPGA devices can also contain processor cores (Arm-based) and other hardened IP blocks that can be used without having to program them into the programmable fabric.

Three Paradigms for Programming FPGAs

While FPGAs can be programmed using lower-level Hardware Description Languages (HDLs) such as Verilog or VHDL, there are now several High-Level Synthesis (HLS) tools that can take an algorithmic description written in a higher-level language like C/C++ and convert it into lower-level hardware description languages such as Verilog or VHDL. This can then be processed by downstream tools to program the FPGA device.
The main benefit of this type of flow is that you can retain the advantages of the programming language like C/C++ to write efficient code that can then be translated into hardware. Additionally, writing good code is the software designer's forte and is easier than learning a new hardware description language. However, achieving acceptable quality of results (QoR), will require additional work such as rewriting the software to help the HLS tool achieve the desired performance goals. The next few sections will discuss how you can first identify some