Vitis High-Level Synthesis User Guide (UG1399)

Design Principles

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 is 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. 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 with 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 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 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 responds 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 be strategically placed on the FPGA 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 using 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.

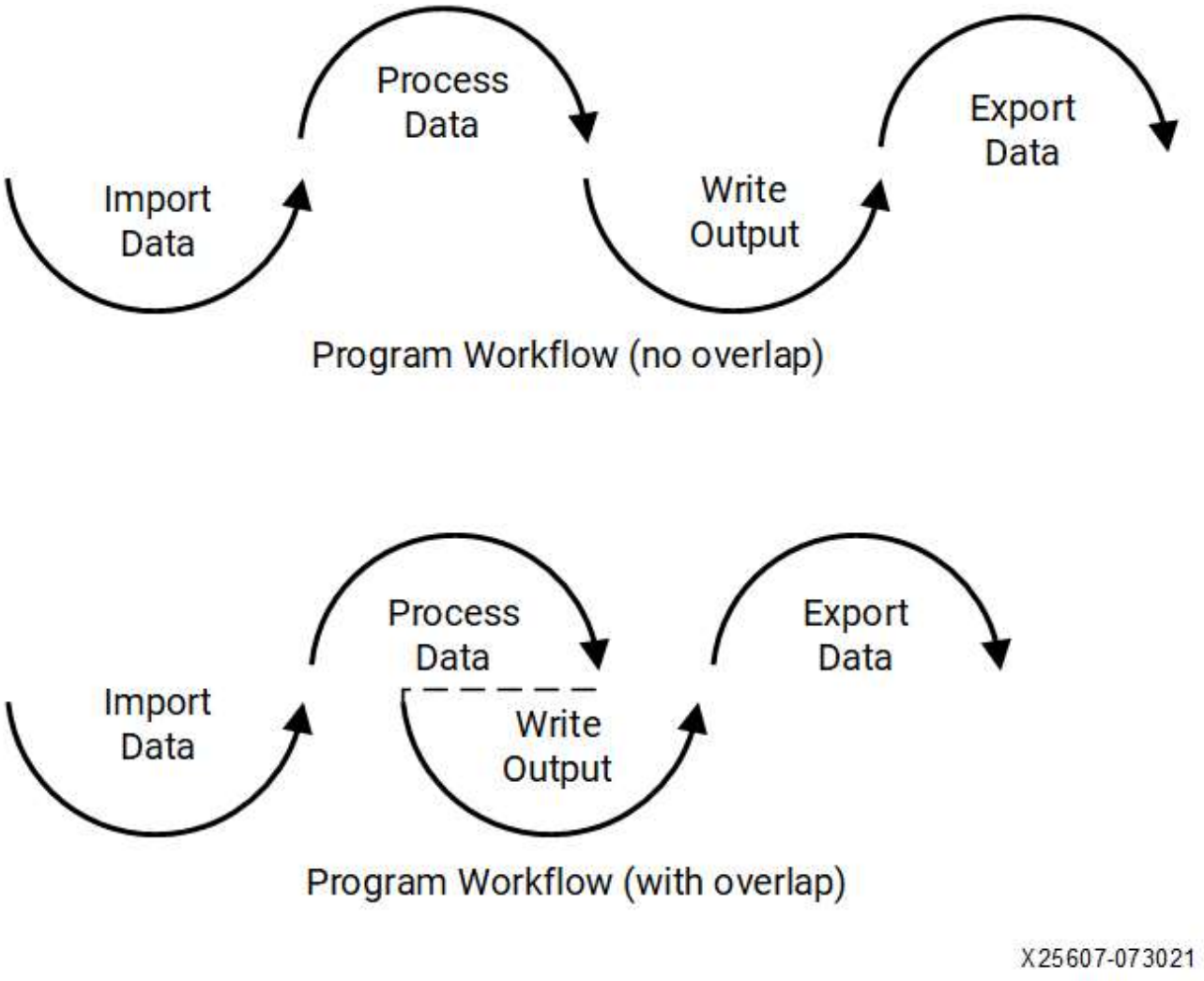
Three Paradigms for Programmable Logic

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 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 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), requires additional work such as rewriting the software to help the HLS tool achieve the desired performance goals. The next few sections discuss how you can first identify some macro-level architectural optimizations to structure your program and 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 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 can import items into a list. You can then process each item in the list. The processing of each item takes around 2 seconds. After processing, you write the result in another datasheet and this action takes an additional 1 second per item. So if you have a total of 100 items in the input Excel sheet then it takes 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. 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 allows for a maximal overlap of I/O with computation. Reading or writing to input/output ports in the middle of the computation step limits 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 can delve a little deeper into this *streaming* paradigm that was introduced in this section.

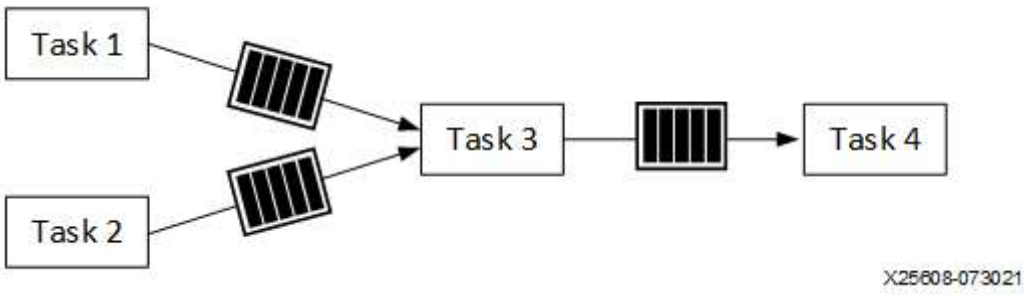
Streaming Data Paradigm

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 sees 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 alternates 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 are needed.
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 and 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 can 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.

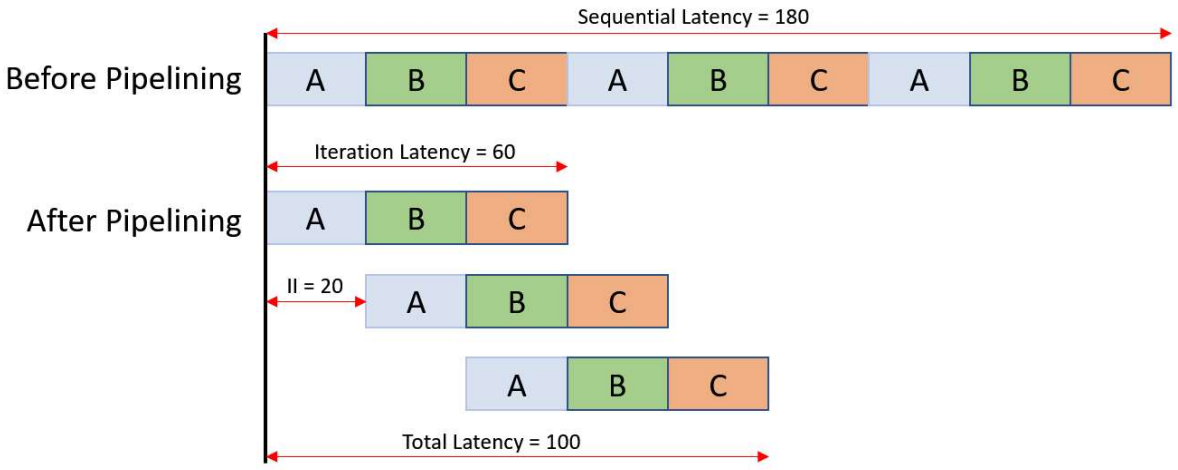
Suppose that assembling one car requires three tasks A, B, and C that takes 20, 10, and 30 minutes, respectively. Then, if all three tasks were performed by a single station, the factory would output one car every 60 minutes. By using a pipeline of three stations, the factory would output the first car 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.

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 can be to achieve an II of 1 - for example, after the initial setup delay, the output is 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 can 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 because 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 are 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 can in turn be composed of many smaller predefined components that typically implement basic functions such as add, sub, and multiply. Functions call other functions although recursion is not supported. Functions that are small and called less often can be also inlined into their callers, showing 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.

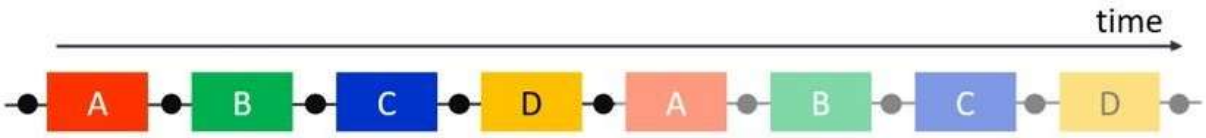
Loops are one of the most important constructs in your program. Because the body of a loop is iterated over a number of times, this property can be easily exploited to achieve better parallelism. There are several transformations (such as pipelining and unrolling) that can be made to loops and loop nests to achieve efficient parallel execution. These transformations enable both memory-system optimizations, mapping to multi-core and SIMD execution resources. Many programs in science and engineering applications are expressed as operations over large data structures. These can be simple element-wise operations on arrays or matrices or more complex loop nests with loop-carried dependencies - for example, 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 (for example, 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. 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);
}
```

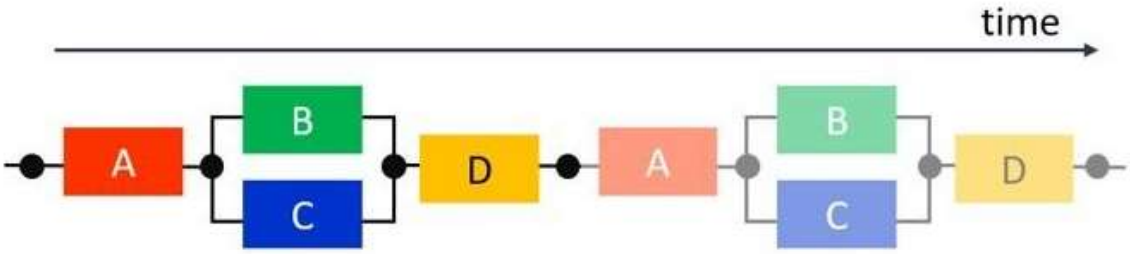
The code example above shows the C/C++ source snippet for how these functions are invoked. The tasks B and C have no mutual data dependencies. A fully-sequential execution corresponds to the following figure where the black circles represent some form of synchronization used to implement the serialization.

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 following figure, 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.

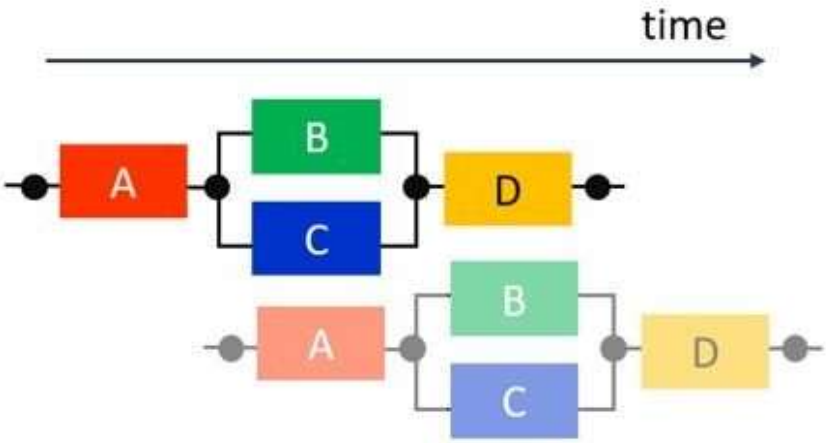
Figure: Task Parallelism within a Run



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 know that this is exactly 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 (for example, A, B, C, or D) reuses the same computation hardware as for

its previous run, you might 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

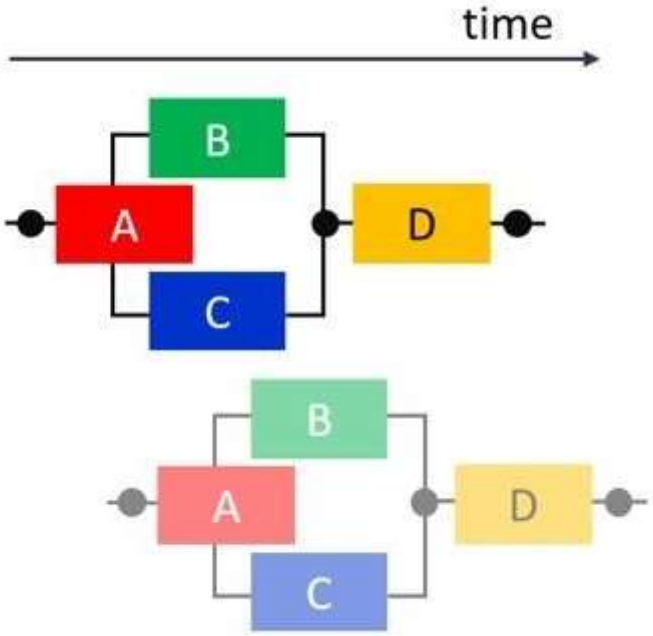


Now, however, when the first run of B reads from the memory where A placed its first result, the second run of A is possibly already writing in the same memory. To avoid overwriting the data before it is consumed, you can rely on a form of memory expansion, namely double buffering or PIPOs to allow for 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 runtime 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 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 gives 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 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 gives 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 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 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 might 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 are required during each phase of your design is important. Knowing this influences how you write your application for FPGAs.
    - Think about the critical path (for example, 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 might 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.