Vitis HLS: High-Performance Design Using Task-level Parallelism

WP554 (v1.0) August 23, 2023

Abstract

The increasing complexity of algorithms used in domains such as AI, wireless communication, and image processing poses challenges for efficient hardware implementation. AMD Vitis™ High-Level Synthesis (HLS) addresses these challenges by enabling accelerated IP creation through the synthesis of C/C++ code into RTL code for AMD's FPGA and adaptive System-on-Chip (SoC).

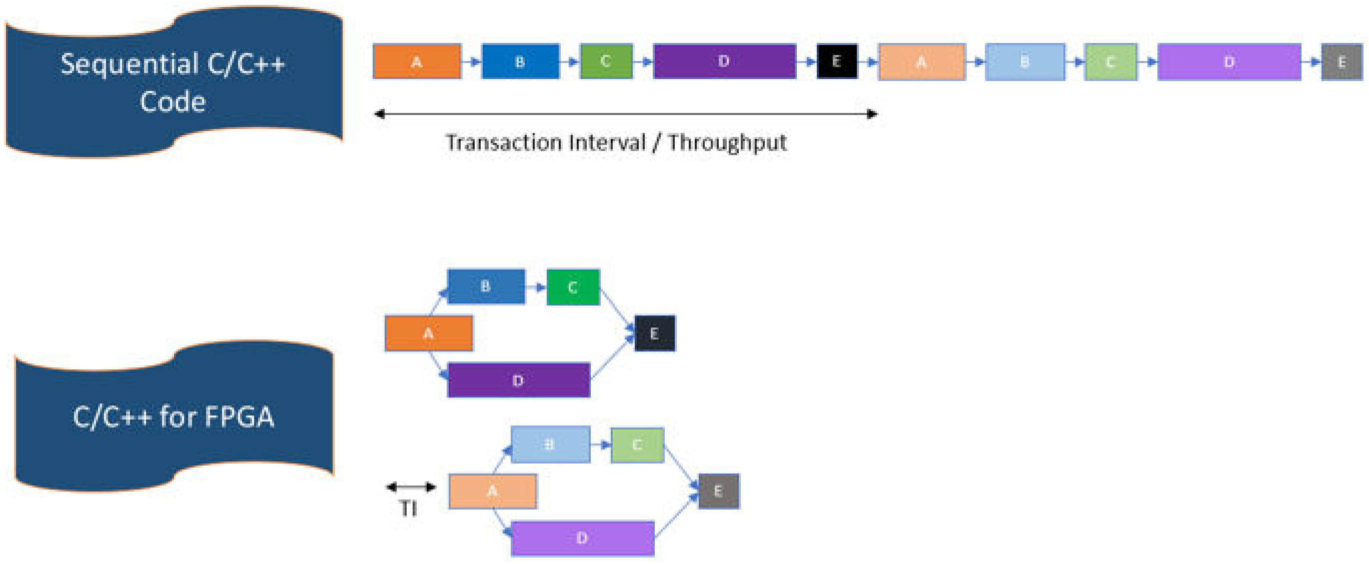
Introduction

Vitis HLS leverages the unique benefits and characteristics offered by AMD FPGAs to optimize the C/C++ code for high-performance designs. The tool supports various parallel programming constructs to model a desired implementation. Extracting task-level parallelism (TLP) is crucial for designing efficient C-based IP and kernels.

Why Use Task-level Parallelism?

Generic C/C++ code written for CPUs typically executes the tasks sequentially, which can be inefficient when dealing with tasks composed of multiple subtasks as shown in the following figure. To maximize the performance on parallel architectures such as FPGAs, the C/C++ code needs to incorporate parallelism. Task level parallelism is one such technique to achieve different forms of parallel execution, breaking down tasks into subtasks that can execute concurrently or in a pipeline manner.

Figure 1: **Sequential vs Task Parallelism**



C/C++ code targeted for Vitis HLS must be designed with TLP architecture to:

- Maximize performance and efficiency.
- Enhance synthesizability and simplify timing closure by breaking tasks into smaller, manageable units rather than large monolithic tasks.

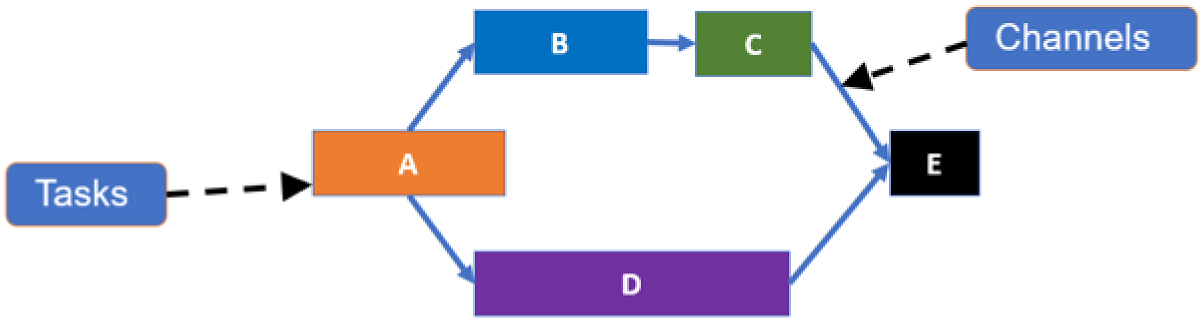
Tasks and Channels

The TLP regions in Vitis HLS (as shown in the following figure) consists of two building blocks:

- Computational units known as tasks.

- Channels that handle data transfer between tasks.

Figure 2: **Tasks and Channels**



Tasks: Types of Task-Level Parallelism

There are two types of task-level parallelism (TLP).

- Control driven
- Data driven

Control-Driven TLP

- Tasks are identified based on the control structures in the code when the *dataflow* pragma is added.
- Execution of tasks is determined by the control signals defined in the code.
- Control signals preserve sequential semantics, allowing tasks to synchronize and start/stop accordingly.
- Control signals are necessary for external memory interface access.

The following code example shows the concept of control-driven parallelism. Without the pragmas, the sequential control flow dictates that task A executes first followed by the other tasks B, C, and D. The next iteration of the task begins after all the subtasks have completed. There is an opportunity to use TLP by adding the *dataflow* pragma. With the presence of the dataflow pragma, the tool will infer TLP, and any tasks in that region are separated into subtasks. In this way, the region is pipelined, and the subtasks can operate on subsequent iterations of data simultaneously.

```
void diamond(date_t vecIn[N], data_t vecOut[N]) {
 data_t c1[N], c2[N], c3[N], c4[4];
#pragma HLS dataflow
 funcA(vecIn, c1, c2);
 funcB(c1, c3);
 funcC(c2, c4);
 funcD(c3, c4, vecOut);
}

void funcA(data_t *in, data_t *out1, data_t *out2) {
 for (int i=0; i<N; i++) {
   data_t t = in[i] *3;
   out1[i] = t;
   out2[i] = t;

  }
}
```

Control driven TLP is a powerful optimization, particularly suitable for designs involving additional control for memory-mapped I/O operations like DDR memory and high-bandwidth memory (HBM).

Data-Driven TLP

- Users explicitly define task level parallelism using `hls::tasks`.
- Each task executes independently based on the availability of streaming input data.

```
void diamond(hls::stream<data_t> &vecIn,
hls::stream<data_t> &vecOut) {

hls::stream<data_t> c1,c2,c3,c4;

 hls_thread_local hls::task taskA(funcA,vecIn, c1, c2);

 hls_thread_local hls::task taskB(funcB,c1,c3);

 hls_thread_local hls::task taskC(funcC,c2,c4);

 hls_thread_local hls::task taskD(funcD,c3,c4,vecOut);

}

 void funcA(hls::stream<data_t> &in,

       hls::stream<data_t> &out1

       hls::stream<data_t> &out2)

  data_t t = in.read();

  out1.write(t);

  out2.write(t);

}
```

The code example illustrates the concept of data-driven parallelism. As you can see, a key difference between control-driven and data-driven TLP is that in the latter case, tasks are explicitly dictated by the user. With the absence of control signals, each task can be thought of as an infinite loop. Because there are no control signals, the data-driven tasks require streaming interfaces to indicate when the channel contains data, and thus, when the task can run.

Data-driven TLP is useful for purely streaming designs like video processing or network packet processing, where tasks can exhibit parallelism and require user control for designs involving feedback.

It is important to see that control driven and data driven TLP are not mutually exclusive concepts. They will commonly be used in different parts of the same project.

Channels

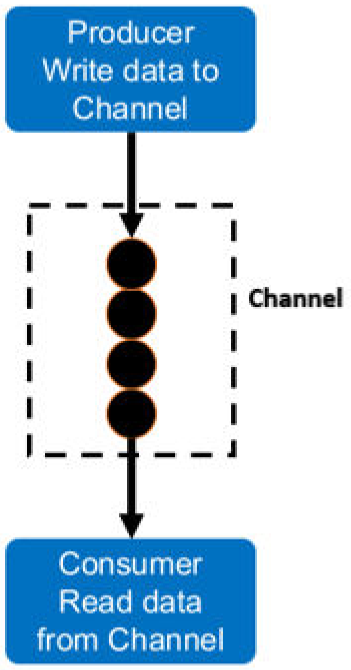
Vitis HLS provides two libraries to define the channels in the TLP region:

- hls::stream – Enables sequential data transfer.
- hls::stream of blocks – Supports non-sequential data transfer.

FIFO and hls::stream

First-in-first-out (FIFO) channels allow the producer to write data to the channel, which can then be read by the consumer. FIFO's only support sequential data transfer, while non-sequential data requires a different channel type.

Figure 3: **First-In-First-Out Channel**



In C/C++ code, the `hls::stream` behaves like an infinite-depth FIFO. The class offers accessor functions to write to and read from the buffer. They are templated to provide support for a wide variety of data types from simple data types like integers and floating-point types, to complex data types such as arbitrary precision or even user-defined structures. In addition, `hls::stream` provides for the optional capability to define side-channel signals. The following code example uses `hls::stream`. During synthesis, Vitis HLS infers a FIFO buffer for the channel.

```
void funcA(hls::stream<data_t> &in,
       hls::stream<data_t> &out1
       hls::stream<data_t> &out2 {
   data_t t = in.read();
   out1.write(t);
   out2.write(t);
}
```

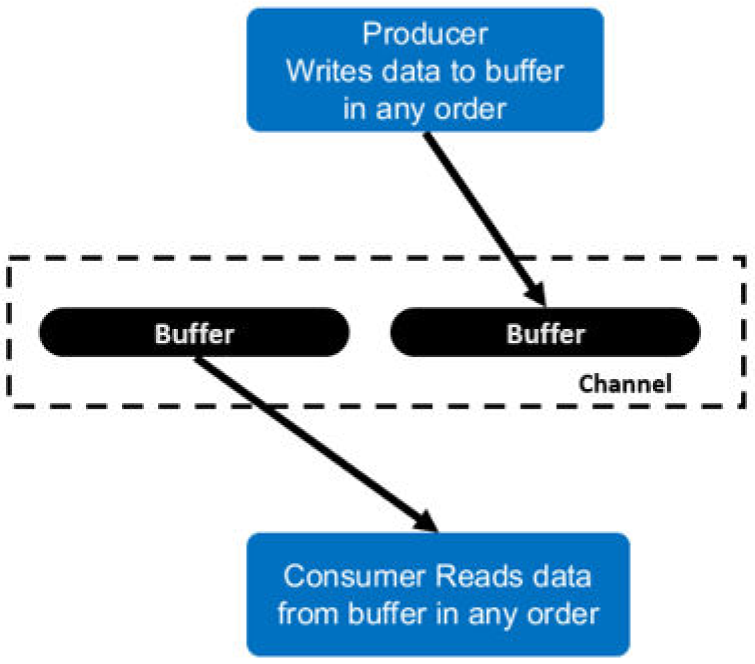
Benefits of hls::stream

- Enables designers to work at a higher-level abstraction.
- Focuses on algorithm aspects rather than hardware implementation.
- Shortens design times with minimal errors.

PIPO and hls::Stream_of_Blocks

In the situations where the data is being transferred is not purely sequential, Parallel-in-parallel-out (PIPOs) allow data to be passed between a producer and consumer in any order. Two buffers large enough to store the data are used for the data transfer. The producer obtains access to one of the buffers, writes to it, then cedes control of the buffer. The consumer then can use the same process to read the data from the buffer. By swapping control of the two buffers back and forth, the computation can continue without stalling.

Figure 4: **Parallel In Parallel Out Channel**



The `hls::Stream_of_blocks` library can be used instead of the `hls::stream` library for processing non-sequential data. As seen in the following code example, the stream of blocks channel is divided into fixed size blocks, which can be accessed through read and write.

```
void funcB(hls::stream_of_blocks<block_data_t> &in,
      hls::stream_of_blocks<block_data_t> &out) {
for(int i = 0; i <N/NUM_BLOCKS; i++) {
  hls::read_lock<block_data_t> inL(in);
  hls::write_lock<block_data_t> outL(out);
  for (unsigned int j = 0; j <NUM_BLOCKS; j++)
   outL[inL[j]] = j + 25;
  }
}
```

Benefits of hls::stream_of_blocks

- Offers explicit control and flexibility to customize the channel behavior.
- Provides higher performance Vitis HLS designs.

Conclusion

In this white paper, we introduced the concept of task level parallelism and the two ways of implementing it in the C/C++ code. Presenting several types of channels and how Vitis HLS allows users to control the behavior of the channel. By incorporating task level parallelism in the designs, users can achieve key performance benefits on real world HLS designs. These concepts allow user of HLS to get even more benefit from the tool, by enabling them to address a larger subset of their design needs using Vitis HLS and achieving higher performance from that subset.

References

These documents provide supplemental material useful with this white paper:

1. *Vitis High-Level Synthesis User Guide* (UG1399)
2. C/C++ Kernels in the *Vitis Unified Software Platform Documentation* (UG1416)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
08/23/2023 Version 1.0	
Version 1.0	Initial Release.

Please Read: Important Legal Notices

Copyright