AMD

Section I

Introduction

This section contains the following chapters:

- Navigating Content by Design Process
- Supported Operating Systems for Vitis HLS
- Benefits of High-Level Synthesis
- Introduction to Vitis HLS
- Introduction to the new Vitis Unified IDE
- Tutorials and Examples

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All AMD Versal™ adaptive SoC design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the AMD Vivado™ timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Launching Vitis HLS
  - Verifying Code with C Simulation
  - Synthesizing the Code
  - Analyzing the Results of Synthesis
  - Optimizing the HLS Project

Supported Operating Systems for Vitis HLS

AMD supports the following operating systems on x86 and x86-64 processor architectures.

- Microsoft Windows Professional/Enterprise 10.0 20H2 Update; 10.0 21H1 Update; 10.0 21H2 Update; 10.0 22H2 Update
- Microsoft Windows 11.0 21H2 Update; 11.0 22H2 Update
- Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6, 7.7, 7.9, 8.3, 8.4, 8.5, 8.6, 8.7, 9.0 and 9.1 (64-bit), English/Japanese
- CentOS 7.4, 7.5, 7.6, 7.7, and 7.9 (64-bit), English/Japanese
- SUSE Linux Enterprise 12 SP4 and 15 SP2 (64-bit), English/Japanese
- Amazon Linux 2 AL2 LTS (64-bit)
- Ubuntu Linux 18.04.1 LTS; 18.04.2 LTS, 18.04.3 LTS; 18.04.4 LTS; 18.04.5 LTS; 18.04.6 LTS; and 20.04 LTS, 20.04.1 LTS, 20.04.2 LTS, 20.04.3 LTS, 20.04.4 LTS; 20.04.5 LTS; 22.04 LTS and 22.04.1 LTS (64-bit), English/Japanese

TIP: *Ubuntu 20.04 can require the* `libtinfo.so.5` *as described in AR# 76616.*

Benefits of High-Level Synthesis

High-Level Synthesis (HLS) is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer level structure that realizes the given behavior.

A typical flow using HLS has the following steps:

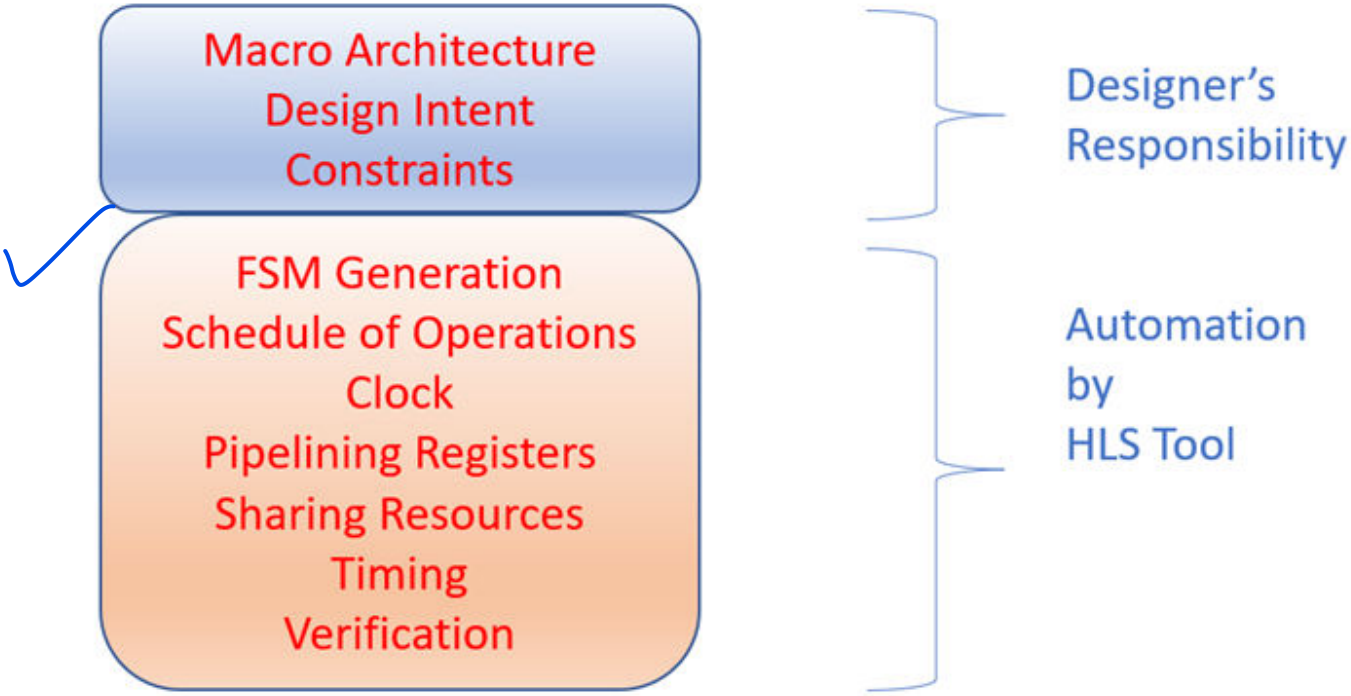
1. Write the algorithm at a high abstraction level using C/C++/SystemC with a given architecture in mind
2. Verify the functionality at the behavioral level
3. Use the HLS tool to generate the RTL for a given clock speed, input constraints
4. Verify the functionality of the generated RTL
5. Explore different architectures using the same input source code

HLS can enable the path of creating high-quality RTL, rather quickly than manually writing error-free RTL.

The designer needs to create the macro-architecture of the algorithm in C/C++ at a high level, meaning that the design intent and how this design interacts with the outside world should be carefully thought through. HLS tool also requires input constraints like clock period, performance constraints, etc.

Micro-architecture decisions like creating the state machine, datapath, register pipelines, etc are not needed at a high level. These details can be left to the HLS tool and optimized RTL can be generated by providing input constraints like clock speed, performance pragmas, target device, etc.

Figure 1: **Design Processes**



Using the defined macro-architecture of the C/C++ algorithm, designers can also vary constraints to generate multiple RTL solutions to explore trade-offs between performance and area. So a single algorithm can lead to multiple implementations, allowing designers to choose an implementation that best meets the needs of the overall application.

Improve Productivity

With HLS, the designer is working on a high abstraction level, meaning fewer lines of code will need to be written as input to HLS. Due to less time spent on writing the C++ code and quicker turnaround, less error-prone thus increasing overall design productivity. The designers can focus more time on creating efficient designs at a higher level than worrying about mechanical RTL implementation tasks.

HLS not only enables high design productivity but also verification productivity. With HLS, the testbench is also generated or created at a high level, meaning the original design intent can be verified very quickly. The designer can explore quick turnarounds of verified algorithms as the flow is still within the C/C++ domain. Once the algorithm is verified in C/C++, the same testbench can be used for generated RTL by the HLS tool. Nevertheless, the generated RTL can be integrated with the existing RTL verification flow for more comprehensive verification coverage.

The design and verification benefits of using HLS are summarized here:

- Developing and validating algorithms at the C-level for the purpose of designing at an abstract level from the hardware implementation details.
- Using C-simulation to validate the design, and iterate more quickly than with traditional RTL design.
- Creating multiple design solutions from the C source code and pragmas to explore the design space, and find an optimal solution.

Enable Re-Use

The designs created for High-level synthesis are generic and unaware of implementation. These sources are not tied to any technology node or any given clock period like a given RTL. With few updates of input constraints and without any source code changes, multiple architectures can be explored. A similar practice with the RTL is not pragmatic. The designers create the RTL for a given clock period and any change for a derivative product, however small it is leads to a new complex project. Working at a higher level with HLS, designers don't need to worry about the micro-architecture and can rely on the HLS tool to regenerate new RTL automatically.

Introduction to Vitis HLS

The Vitis HLS tool synthesizes a C or C++ function into RTL code for implementation in the programmable logic (PL) region of a Versal Adaptive SoC , Zynq MPSoC, or AMD FPGA device. Vitis HLS is tightly integrated with both the Vivado Design Suite for synthesis, place, and route, and the Vitis core development kit for heterogeneous system-level design and application acceleration.

Vitis HLS can be used to develop and export:

- Vivado IP to be integrated into hardware designs using the Vivado Design Suite
- Vitis kernels for use in the Vitis application acceleration development flow

> **TIP:** *The Vitis kernel (`.xo`) is a Vivado IP with specific requirements and limitations as described in Interfaces for Vitis Kernel Flow; while Vivado IP have few restrictions and offer greater design flexibility as described in Interfaces for Vivado IP Flow.*

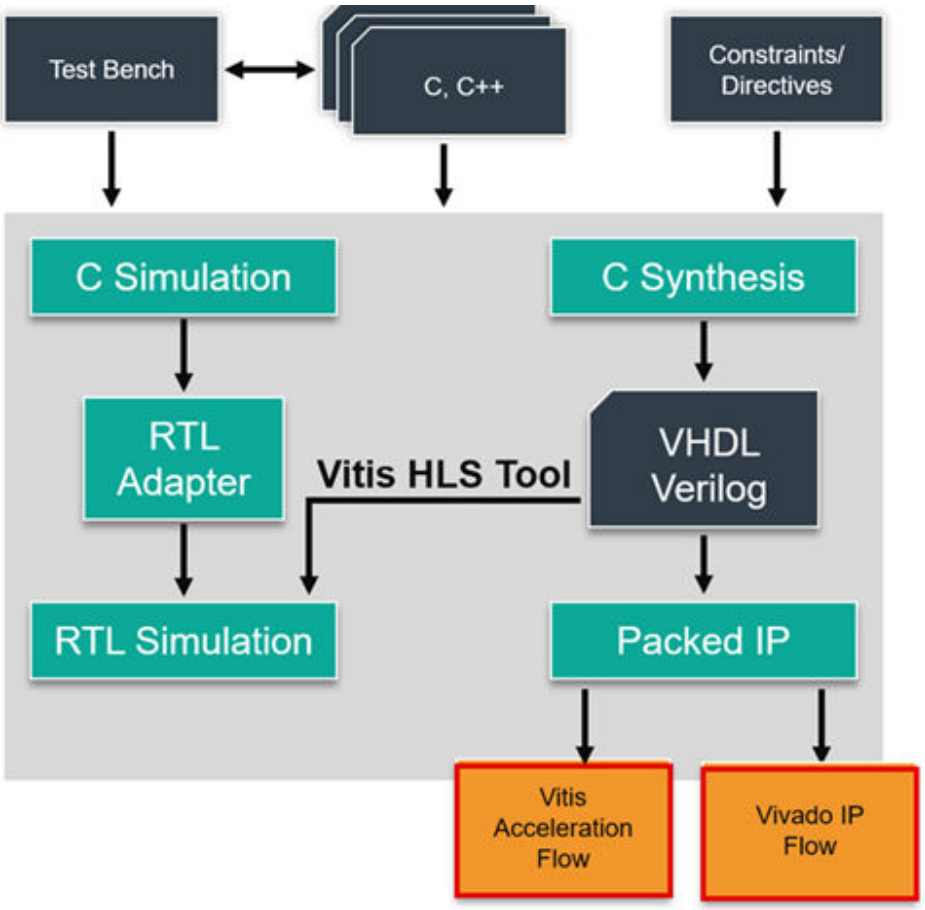
In the Vitis application acceleration flow, the Vitis HLS tool automates much of the code modifications required to implement and optimize the C/C++ code in programmable logic and to achieve low latency and high throughput. The inference of required pragmas to produce the right interface for your function arguments and to pipeline loops and functions within your code is the foundation of Vitis HLS.

In the Vivado IP flow, Vitis HLS also supports customization of your code to implement broader interface standards to achieve your design objectives. The RTL generated can be used as an IP directly within the Vivado tool or Model composer.

Here are the steps for the development of the C++ function.

1. Architect the algorithm based on the Design Principles
2. (C-Simulation) Verify the C/C++ Code with the C/C++ test bench
3. (C-Synthesis) Generate the RTL using HLS
4. (Co-Simulation) Verify the kernel generated with C++ outputs
5. (Analyze) Review the HLS synthesis reports and co-simulation reports, analyze
6. Re-run previous steps until performance goals are met.

Figure 2: **Vitis HLS Development Flow**



Vitis HLS implements the solution based on the target flow, default tool configuration, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Vitis HLS generates Vivado IP or Vitis kernel based on the target flow, default tool configuration, design constraints, and any optimization pragmas or directives you specify. You can use optimization directives to modify and control the implementation of the internal logic and I/O ports, overriding the default behaviors of the tool.

Here are some key areas related to coding and synthesizing the C++ functions in your HLS design with details covered in forthcoming sections:

- **Hardware Interfaces:** The arguments of the top-level function in a Vitis HLS design are synthesized into interfaces and ports that group multiple signals to define the communication protocol between the HLS design and components external to the design. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol used. The default interface protocols differ based on whether the HLS design is targeting for Vivado IP generation or the Vitis kernel. The default assignments of the interfaces can be overridden by using the INTERFACE pragma or directive.
- **Controlling the Execution of HLS Design:** The execution mode of an HLS design is specified by the block-level control protocol. The HLS design can have control signals to start/stop the execution or it can be only driven when the data is available. As a designer, you do need to be aware of how your HLS design will be executed, as described in Execution Modes of HLS Designs.
- **Task-Level Parallelism:**
  - To achieve high performance on the generated hardware, the HLS tool must infer parallelism from sequential code and exploit it to achieve greater performance. The Design Principles section introduces the three main paradigms that need to be understood for writing good software for FPGA platforms. Vitis HLS tool offers multiple types of task-level parallelism (TLP), either by specifying the DATAFLOW pragma or explicitly creating parallelism using `hls::task` object as described in Abstract Parallel Programming Model for HLS.
- **Memory Architecture:**
  - The memory architecture is fixed in the CPU but the developer can create their own architecture to optimize the memory accesses for running applications on FPGA
  - In C++ program, the arrays are fundamental data structures used to save or move the data around. In hardware, these arrays are implemented as memory or registers after synthesis. The memory can be implemented as local storage or global memory which is often DDR or HBM memory banks. Access to global memory has higher latency costs and can take many cycles while access to local memory is often quick and only takes one or more cycles.
  - Often the memory is allocated/deallocated dynamically in a C++ program but this can't be synthesized in hardware. So the designer needs to be aware of the exact amount of memory required for the algorithm.
  - The memory accesses should be optimized to reduce the overhead of global memory accesses. The redundant accesses, which means maximizing the use of consecutive accesses so that bursting can be inferred. The burst access hides the memory access latency and improves the memory bandwidth.
- **Micro Level Optimization:**
  - In C++ programs, there is a frequent need to implement repetitive algorithms that process blocks of data — for example, signal or image processing. Typically, the C/C++ source code tends to include several loops or several nested loops. Vitis HLS can unroll, or pipeline a loop or nested loops by inserting pragmas at appropriate levels in the source code. For more information, refer to the Loops Primer.

- Once the algorithm is architected based on the design principles, inferring parallelism, you still need the right combination of micro-level HLS pragmas like PIPELINE, UNROLL, ARRAY_PARTITION, etc. These pragmas may not be intuitive to new users. Vitis HLS offers the PERFORMANCE pragma as a way to specify top-level performance goals for a given body of loop or nested loops. The tool will automatically infer the necessary lower-level pragmas to meet the goal. With PERFORMANCE pragma, fewer pragmas are needed to achieve good QoR and is an intuitive way to drive the tool.

Re-Architecting the Design Code

The following is a simple program that includes a `compute()` function written in C++ for execution on the CPU. The program is similar to any other C++ program where there the main function sets up the data to be sent to compute function, calls the compute function, and checks the results against expected results. The execution of this program is sequential on the CPU. This example will need to be re-architected to achieve significant performance improvement when running on programmable logic.

```
#include <vector>
#include <iostream>
#include <ap_int.h>
#include "hls_vector.h"

#define  totalNumWords 512
unsigned char data_t;

int main(int, char**) {
    // initialize input vector arrays on CPU
    for (int i = 0; i < totalNumWords; i++) {
      in[i] = i;
    }
    compute(data_t in[totalNumWords], data_t Out[totalNumWords]);
    check_results();
}

void compute (data_t in[totalNumWords ], data_t Out[totalNumWords ]) {
  data_t tmp1[totalNumWords], tmp2[totalNumWords];
  A: for (int i = 0; i < totalNumWords ; ++i) {
    tmp1[i] = in[i] * 3;
    tmp2[i] = in[i] * 3;
  }
  B: for (int i = 0; i < totalNumWords ; ++i) {
    tmp1[i] = tmp1[i] + 25;
  }
  C: for (int i = 0; i < totalNumWords ; ++i) {
    tmp2[i] = tmp2[i] * 2;
 }
  D: for (int i = 0; i <  totalNumWords ; ++i) {
     out[i] = tmp1[i] + tmp2[i] * 2;
   }
}
```

This program can also run sequentially on an FPGA, producing correct results without any performance gain compared to the CPU. For the application to execute with higher performance on an FPGA, the program needs to be re-architected to enable parallelism at various levels. Examples of parallelism can include:

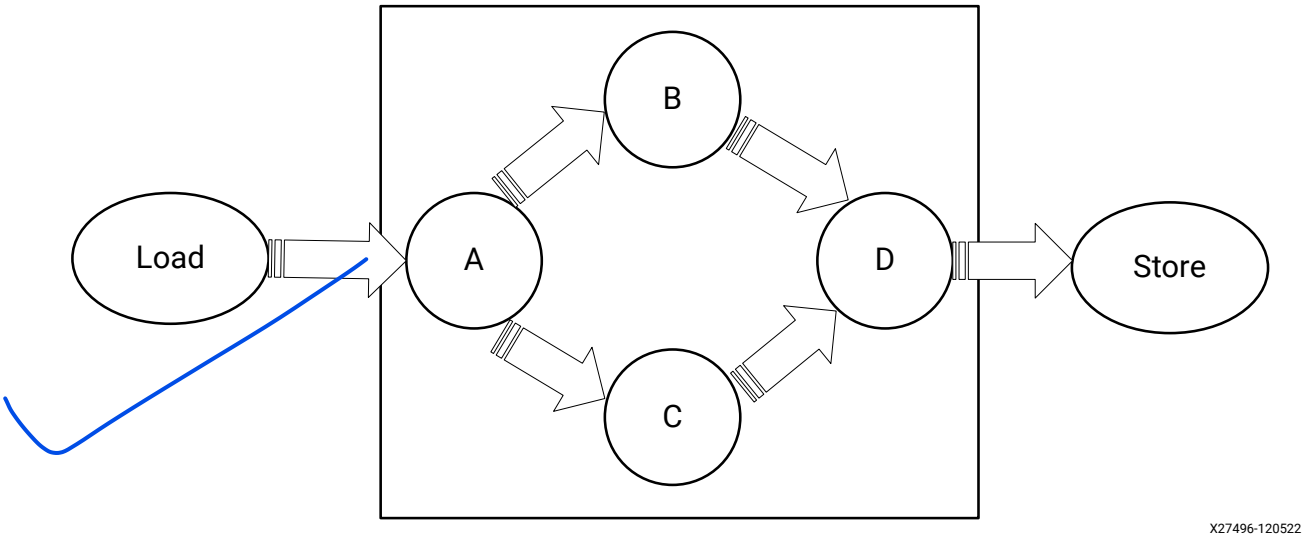
- The compute function can start before all the data is transferred to it
- Multiple compute functions can run in an overlapping fashion, for example a "for" loop can start the next iteration before the previous iteration has completed
- The operations within a "for" loop can run concurrently on multiple words and doesn't need to be executed on a per-word basis

Re-Architecting Kernel Code

From the prior example it is the `compute()` function that needs to be re-architected for FPGA-based acceleration.

The `compute()` function Loop A multiplies an input value with 3 and creates two separate paths, B and C. Loop B and C perform operations and feed the data to D. This is a simple representation of a realistic case where you have several tasks to be performed one after another and these tasks are connected to each other as a network like the one shown below.

Figure 3: **Kernel Architecture**



The key takeaways for re-architecting the kernel code are:

- Task-level parallelism is implemented at the function level. To implement task-level parallelism loops are pushed into separate functions. The original `compute()` function is split into multiple sub-functions. As a rule of thumb, sequential functions can be made to execute concurrently, and sequential loops can be pipelined.

- Instruction-level parallelism is implemented by reading 16 32-bit words from memory (or 512-bits of data). Computations can be performed on all these words in parallel. The `hls::vector` class is a C++ template class for executing vector operations on multiple samples concurrently.
- The `compute()` function needs to be re-architected into load-compute-store sub-functions, as shown in the example below. The load and store functions encapsulate the data accesses and isolate the computations performed by the various compute functions.
- Additionally, there are compiler directives starting with `#pragma` that can transform the sequential code into parallel execution.

TIP: *This is the using_fifos example found in the Vitis-HLS Introductory Examples on GitHub.*

```
#include "diamond.h"
#define NUM_WORDS 16
extern "C" {

void diamond(vecOf16Words* vecIn, vecOf16Words* vecOut, int size)
{
  hls::stream<vecOf16Words> c0, c1, c2, c3, c4, c5;
  assert(size % 16 == 0);

  #pragma HLS dataflow
  load(vecIn, c0, size);
  compute_A(c0, c1, c2, size);
  compute_B(c1, c3, size);
  compute_C(c2, c4, size);
  compute_D(c3, c4,c5, size);
  store(c5, vecOut, size);
}
}

void load(vecOf16Words *in, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in[i]);
  }
}

void compute_A(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out1, hls::stream<vecOf16Words >& out2, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    vecOf16Words t = in.read();
    out1.write(t * 3);
    out2.write(t * 3);
  }
}
void compute_B(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
```

```
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in.read() + 25);
  }
}


void compute_C(hls::stream<vecOf16Words >& in, hls::stream<vecOf16Words >&
out, int size)
{
Loop0:
  for (data_t i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in.read() * 2);
  }
}
void compute_D(hls::stream<vecOf16Words >& in1, hls::stream<vecOf16Words >&
in2, hls::stream<vecOf16Words >& out, int size)
{
Loop0:
  for (data_t i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out.write(in1.read() + in2.read());
  }
}

void store(hls::stream<vecOf16Words >& in, vecOf16Words *out, int size)
{
Loop0:
  for (int i = 0; i < size; i++)
  {
    #pragma HLS PERFORMANCE target_ti=32
    #pragma HLS LOOP_TRIPCOUNT max=32
    out[i] = in.read();
  }
}
```

Introduction to the new Vitis Unified IDE

The next generation AMD Vitis™ Unified Integrated Design Environment (IDE) is currently in preview mode for data center acceleration and embedded system design, AI Engine and High-Level Synthesis (HLS) component creation, platform creation, and embedded software design. Refer to the *Vitis Unified IDE and Common Command-Line Reference Guide* (UG1553) for more information.

The new Vitis tools embrace a bottom-up design flow that lets you develop components of a system and then integrate the components into a top-level system application. The next generation of tools include both a new Vitis Unified IDE, and new `v++` command-line flows for developing AI Engine components and HLS components.

The single integrated development environment provides all the features you need to compile, run, debug, and analyze the different elements of an FPGA-accelerated Data Center application, or heterogeneous embedded system design. The Vitis Unified IDE lets you create AI Engine components using the very-long instruction word (VLIW) processor arrays of AMD Versal™ devices; synthesize C/C++ code into RTL designs using HLS components, run C-simulation and C/RTL Co-simulation; review and analyze build and run summaries in the newly integrated Vitis analyzer tool.

The new Vitis Unified IDE works with the common command-line features of the `v++` and `vitis-run` commands. Whether working from the command-line or from the Vitis Unified IDE, the single environment provides you a tightly integrated design environment where you can accomplish most of your design objectives.

Tutorials and Examples

To help you quickly get started with the Vitis HLS, you can find tutorials and example applications at the following locations:

- **Vitis HLS Introductory Examples (https://github.com/Xilinx/Vitis-HLS-Introductory-Examples):** Hosts many small code examples to demonstrate good design practices, coding guidelines, design pattern for common applications, and most importantly, optimization techniques to maximize application performance. All examples include a `README` file, and a `run_hls.tcl` script to help you use the example code.
- **Vitis Accel Examples Repository (https://github.com/Xilinx/Vitis_Accel_Examples):** Contains examples to showcase various features of the Vitis tools and platforms. This repository illustrates specific scenarios related to host code and kernel programming for the Vitis application acceleration development flow, by providing small working examples. The kernel code in these examples can be directly compiled in Vitis HLS.
- **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.