Introduction to HLS

Simone Bologna
simone.bologna@bristol.ac.uk

University of Bristol

23 October 2019
Outline

- Introduction to FPGA, VHDL, and HLS
- Getting started with HLS
- Life of a toy project from conception to (almost) implementation
- Tips and tricks
- Using C++ constructs in Vivado HLS
Introduction
Field Programmable Gate Arrays (FPGA)

- FPGA are circuits that are programmable on the field
- FPGAs are powerful and flexible devices
- Components of FPGA
  - Flip-Flops (FF), small memory component able to store a bit
    - Typical used as a fast register to store data
  - Look-Up Tables (LUT), small memories used to store truth tables and perform logic functions
    - Typically used to perform operation such as “and”, “or”, sums or subtractions
  - Digital Signal Processor (DSP), small processor able to quickly perform mathematical operation on streaming digital signals
    - Typically used for multiplication and additions
  - Block RAM (BRAM), memory able to store data
    - Can store a fair amount of data, but slow and with a limited number of ports limiting memory throughput
VHDL and HLS

- What is VHDL?
  - VHSIC Hardware Design Language
    - Very High Speed Integrated Circuit Hardware Description Language
      - ... ergh...
    - Used to describe circuits that will be implemented on FPGA via code
    - Not covered here!
- High-Level Synthesis (HLS) enables user to transform (synthesize) C/C++/SystemC code into VHDL
  - Enables users to program FPGA in high-level languages!
  - Focusing on C++
- Analogies with assembly and high-level languages are stretched
  - Each language works better in specific situations
When to use each language?

- Collecting here opinions I have heard from various experts

When to use VHDL?
- When you want full control on how your design is going to be implemented
- When you need some clock-dependent applications
  - i.e. receive data and hold it for three clock cycle
- Receiving data and sorting in specific manners

When to use HLS?
- Rapid prototyping
  - I would suggest to use it in doubt, implementing stuff in HLS will generally take less time than using VHDL
- Designing some processing/analysis block
  - i.e. developing some particle identification algorithm
Getting started
HLS in Bristol

- excession.phy.bris.ac.uk is the FPGA development machine
- Two strategies to develop in HLS:
  - Write code in your favourite editor and use Vivado HLS’ command line interface (CLI)
  - Use Vivado HLS’s GUI to do both editing and synthesis
- Vivado HLS’ command line does not provide all the tools
  - Vivado HLS GUI is required when you need to investigate design performance in detail
- Using editor + Vivado HLS CLI here
- I recommend using VNC to log into excession if you want to use Vivado GUI
  - Feel free to ask me help to set it up :)
How to begin

- Get an account on excession
- Add Vivado to your environment
  - source /software/CAD/Xilinx/2018.2/Vivado/2018.2/settings64.sh
  - 2019.1 is available, I started on 2018.2 and I am keeping it for consistency
- Run `vivado_hls` in your terminal to open the `vivado_hls` GUI
  - Use if you have mounted /software locally or if you are working via VNC
- `vivado_hls -i` opens the interactive TCL shell
  - Development tools through command line
- `vivado_hls <.tcl file>` runs a .tcl script
  - Typically I use it to build my firmware and test
    - Back to tcl in a sec
- `vivado_hls -f <.tcl file>` runs a .tcl script and keeps the console open
  - Useful for .tcl scripts that sets up your project before running some interactive operation
Terminology

- **HLS file**, C/C++ code that will be synthesised and run on FPGA
- **Test bench (TB) file**, C/C++ code that is run to test the HLS code. It calls the HLS functions and can run tests on their output, e.g. C asserts.
- **Tcl scripts**, set of tcl instructions executed by the Vivado HLS shell
- **Synthesis**, C/C++ \(\rightarrow\) HDL lang (VHDL/Verilog)
- **Project**, collection of HLS and test bench (TB) files
  - Has a top-level function name that is the starting point for synthesis
- **Solution**, specific implementation of a project
  - Runs on a specific device at a specific clock frequency
- **C simulation**, HLS + TB files are compiled with gcc against HLS headers and lib and plainly run as any other executable
- **C/RTL cosimulation**, synthesised HLS code is run on a simulator and results tested on the C/C++ test bench
Setting up your first project

- In a base project you will typically have
  - At least a HLS .c/.cpp files
  - A header used to link HLS code to test bench code
  - At least a TB .c/.cpp file
  - A .tcl script to set up your Vivado HLS project and solution
General workflow

- Problem
- Define your inputs & output
  - They will translate as the parameters of your HLS top-level function
- Write up your code
- Test your C++ code
- Synthesis, i.e. convert to VHDL code
  - Optimise it to get the desired performance while staying in your HW limits
- Test synthesised design
- Export design, typically in Vivado IP (Intellectual Property) format
- Implement in Vivado on actual FPGA
Building and optimising a project
Our problem

- Problem definition:
  - We want to design a high-throughput vector adder and multiplier
    - *Throughput*: amount of data items passing through the process

  \[
  \text{Throughput} = \frac{\text{data in input} \times \text{clock frequency}}{\text{initiation interval}}
  \]

- Input & output definition
  - *We receive two 100-dimensional vector of 16-bit signed integer*
  - *We output a 100-dimensional vector of 16-bit signed integer as the sum and an additional 16-bit integer as the product*
Code time!

https://github.com/simonecid/VivadoTutorial
Testing

- Before optimising your design, you need a reliable system to check that it works as expected

- Testbench!
  - C++ which runs your HLS function with a defined sets of inputs, of which you already know the output
    - e.g. two vectors you know the sum and product of

- Having a test bench that runs through tests is extremely beneficial
  - You can use it to keep on checking that your code keeps on working fine after you have altered it
    - After going through synthesis you might want to redesign parts of it in order to better suit your needs or optimise it

- Typical test runs the function and checks its results via C asserts
  - More extensive and sophisticated test unit libraries, e.g. CPPunit, are available, but let’s keep it simple :)
Testing

- Add test bench files with
  - `add_files -tb "FILE"`
- Run your test bench with
  - `csim`
    - Abbreviation of `csim_design`

```c
// checking sum and product of non-null vectors
for (int a = 0; a < ARRAY_SIZE; a++)
{
    lArray1[a] = 0;
    lArray2[a] = 0;
}
lArray1[10] = 12;
lArray2[10] = 90;
lArray1[67] = 67;
lArray2[67] = -2;
lArray2[88] = -179;

hls_vectorOperations(lArray1, lArray2, lArray3, lProduct);
for (int a = 0; a < ARRAY_SIZE; a++)
{
    switch (a) {
    case 10: assert(lArray3[a] = 102); break;
    case 67: assert(lArray3[a] = 65); break;
    case 88: assert(lArray3[a] = -179); break;
    default: assert(lArray3[a] = 0); break;
    }
}
assert(lProduct == 946);
```
Synthesis

- If the design is working and has been tested, you can proceed with the synthesis
  - Run **csyn** (abbreviation of **csynth_design**)
- Vivado HLS synthetises VHDL and Verilog (another HDL language) from your C++ code
- Synthesis starts from a top-level function, declared in your .tcl file with **set_top**
- Parameters of the top-level functions are translated into ports, by default:
  - N-bit variables are translated into STD_LOGIC_VECTORS, i.e. array of 1-bit ports
  - Structs and classes are converted to ports by creating ports for each one of their attributes
  - Arrays are translated into ports able to read from an external memory
Post-synthesis analysis

- After synthesis, HLS produces a report describing the performance of your design under `<ProjectName>/<SolutionName>/syn/report/` in .rpt format, human readable, and .xml, useful for automated analysis.

### Clock estimate:
gives an initial estimate of whether your design meets the required clock period.

*Note: final clock can only be known after implementation on actual device, sometimes HLS really messes up.*

### Latency:
minimum and maximum number of clocks to finish processing, may change if you have variable length loops.

### Initiation Interval (II):
number of clocks before new data can be processed.

### Pipeline:
if the function has been pipelined (more on this soon).

### Loop breakdown:
label your loops to make sure you can see and study its performance.

### Trip count:
the number of iteration of the loop.

### Utilisation estimates:
breakdown of resource usage.

*Note: LUTs and FFs are typically overestimated, even by a factor 2.*
Post-synthesis analysis breakdown

- You can see how your resources are being used
- 1 DSP used by multiplication
- 75 for the sums
- 108 used for temporary memory
Optimising your design

- Base throughput: 1.2 Gb/s
- Let’s work on improving this
- Introducing three new concepts:
  - **Pipelining**: enables an iteration of a function or a loop to be executed before the previous one is over
    - Increases throughput with minimal resource usage increase
  - **Unrolling**: enables multiple iterations of a for loop to run in parallel, if independent
    - Greatly reduces latency and throughput
    - Can have an impact on resource usage based on loop size
  - **Memory partitioning**: splits array (implemented in BRAM1P/2P or memory port by default) into single registers or ports, enable fast parallel memory access

Base design throughput 1.2 Gb/s
Pipelining

• Let’s partition the memories and pipeline the main body of the loops
  – Partition in the body of the function where the variable or the parameter is declared; in main:
    #pragma HLS array_partition variable=inVector1/2/3
    • Breaks down the memory interface into single 16-bit ports
  – Put this pragma in loop body to pipeline it; in sumLoop and productLoop:
    #pragma HLS pipeline

• Following pipelining and partitioning
  – Latency: 802 → 206
  – II: 802 → 206
Pipelining

Base design throughput 1.2 Gb/s

Pipelined throughput 4.7 Gb/s
Unrolling

- Let’s unroll the loops
  - Instead of instantiate logic for a single loop and execute it 100 times, instantiate logic for each iteration and execute in parallel
    - Essentially you increase resource usage by a factor 100
      - DSP: 1 → 100
    - Put this pragma in loop body to unroll it; in sumLoop and productLoop:
      #pragma HLS unroll

- Latency: 206 → 8
- II: 206 → 8
Unrolling

Base design throughput 1.2 Gb/s

Pipelined throughput 4.7 Gb/s

Unrolled throughput 120 Gb/s
Pipelining the top-level function

- The pipeline pragma pipelines the function in which it is located and unroll and pipelines every underlying loop
  - If we place a pipeline pragma in the top-level function body, everything will be unrolled and pipelined, maximising performance

- Latency: 8 → 8
- II: 8 → 1, data can be input every clock cycle, **max. throughput**
Pipelining the top-level function

- Base design throughput: 1.2 Gb/s
- Pipelined throughput: 4.7 Gb/s
- Unrolled throughput: 120 Gb/s
- Fully pipelined throughput: 960 Gb/s
Finishing touches

- Whenever you create a function, HLS creates a separate logic block and connects it to the logic block of the main function
  - Increases latency
  - Prevents HLS from running optimisations that reduces resource usage
  - In the function body (not top-level): `#pragma HLS inline`
    - Inlines and integrates the sub-function in the calling one
- Latency: 8 → 7
Testing and exporting the synthesised design

- Synthesised design can be tested in HDL simulator in the C test bench
  - Run **cosim** (abbreviation of **cosim_design**)
    - First tests the C code, then the synthesised design
  - If everything looks good, you can export it for actual implementation
    - Using IP catalog now, but other formats are available
    - Final product of Vivado HLS
      - **export_design -format ip_catalog**
        - Exported design can be found in `<ProjectName>/<SolutionName>/impl/ip`
  - From here on is Vivado domain, not covered here, but you can load IP and implement it
Tips and tricks
Various tips and tricks

- You can use C++11 and higher constructs, e.g. auto or constexpr:
  `add_files -cflags "-std=c++11 "<HLS_FILE>"

- Run thorough tests on software, do not be lazy like me!
  - Debugging stuff at later stages is just way harder and confusing
    - If you do not trust me, ask Aaron!

- Read the list of pragmas and experiment a lot with them
  - Array_partition, pipeline, and unroll accept options, study them!
  - Pragmas try to bridge the gap between C++ and HLS, master them

- HLS likes ternary operators, if possible use them instead of if statements!

\[
a = (b > 0) ? c : d;
\]

Ternary operator

\[
if (b > 0)
|   a = c;
else
|   a = d;
\]

Equivalent if statement
Various tips and tricks

Splitting designs

- Big designs take long to synthesise
- Split your problem in smaller projects
- Each project can be exported in IP format and then linked in a chain
- Saves lots of synthesis time
- Increases flexibility
  - Blocks can be run at different clock speeds
- Example: the jet trigger algorithm I work on is made of three blocks
  - Histogrammer
  - Data buffer
  - Jet finder
- Divide et impera reigns!
Various tips and tricks

Scaling designs

- Your time is precious! Do not waste it implementing large broken designs.
- Start small and write code that can be easily scaled up!
- For instance, let’s say you need to do some processing on a large number of inputs
  - Make the number of inputs a parameter of your code with a
    
    ```
    #define NUMBER_OF_INPUTS XX
    ```
    and make your code depend on it
  - Do your initial testing on a scaled-down version of your code, i.e. with few inputs, then increase it
    - Takes way less time to implement a smaller design
Various tips and tricks

Getting more accurate estimates

- Final timing and resource usage results are only obtainable after implementation
- Vivado HLS provides tools to implement design without using Vivado
  - Not sure how it works, I presume it makes some basic assumptions on how you are going to place your design in a FPGA and implements it
- By running it you can get a more accurate estimates of timing and resource usage, although not final they tend to be much closer
- Run `export_design -format ip_catalog -evaluate vhdl`
  - This implements the VHDL design on FPGA
  - 10 minutes to run for the small test design, against XX for synthesis
  - Results in `<ProjectName>/<SolutionName>/impl/report/vhdl/`
Various tips and tricks

Getting more accurate estimates

```plaintext
# Post-Implementation Resource usage ===
CLB: 1007
LUT: 2897  # to be added to SRL 3697 vs 4737
FF: 8551  # was 5608
DSP: 100  # was 100
BRAM: 0
SRL: 800  # to be added to LUT 3697 vs 4737
# Final timing ===
CP required: 3.333
CP achieved post-synthesis: 2.302
CP achieved post-implementation: 3.228 # was 2.693+-0.42
Timing met
```
Various tips and tricks

HLS libraries

• **FOR THE LOVE OF GOD DO NOT USE THE C/C++ STANDARD LIBRARY!**
  
  – I have heard it gives horrible results
    
    • I do not even know how they managed to get HLS to synthesise
  
• **Do not reinvent the wheel!**
  
  – Vivado HLS has libraries doing many interesting things
    
    • It is all in the manual
  
  – For instance, `#include <hls_math.h>` for HLS math libraries
Using C++ constructs
Using C++ constructs

Code time!

https://github.com/simonecid/VivadoTutorial/tree/cpp_version
Using C++ constructs

- Rewritten the vector add and multiply by developing a generic Vector class via template
  - Generic, flexible, easy to use
  - N-dimensional
  - Uses any type
- Same resource usage
- Clever usage of C++ constructs provides great flexibility without usage penalties
- Note:
  - Partitioning of class attributes must be invoked in constructors
  - Inline every class method!
Using C++ constructs

Did you make it a template?  Yes...

What did it cost you?  Nothing.
Summary

- HLS enables users to write FPGA firmware in high-level languages
  - More flexible and easier to use
- HLS pragmas can be used to produce high-throughput designs
  - Pipeline functions, unroll loops and partition memory
  - Used it on a vector adder and multiplier
- The machine excession is available in Bristol per FPGA development
- Went through a number of tips and tricks
- Using C++ classes and template does not affect resource usage while improving code flexibility and ease of use
- Collection of my FPGA bookmarks in next slide
- Contacts:
  - simone.bologna@bristol.ac.uk
  - Skype: simonecid
  - Office: 4.57
Useful links

- HLS tips, https://fling.seas.upenn.edu/~giesen/dynamic/wordpress/vivado-hls-learnings/