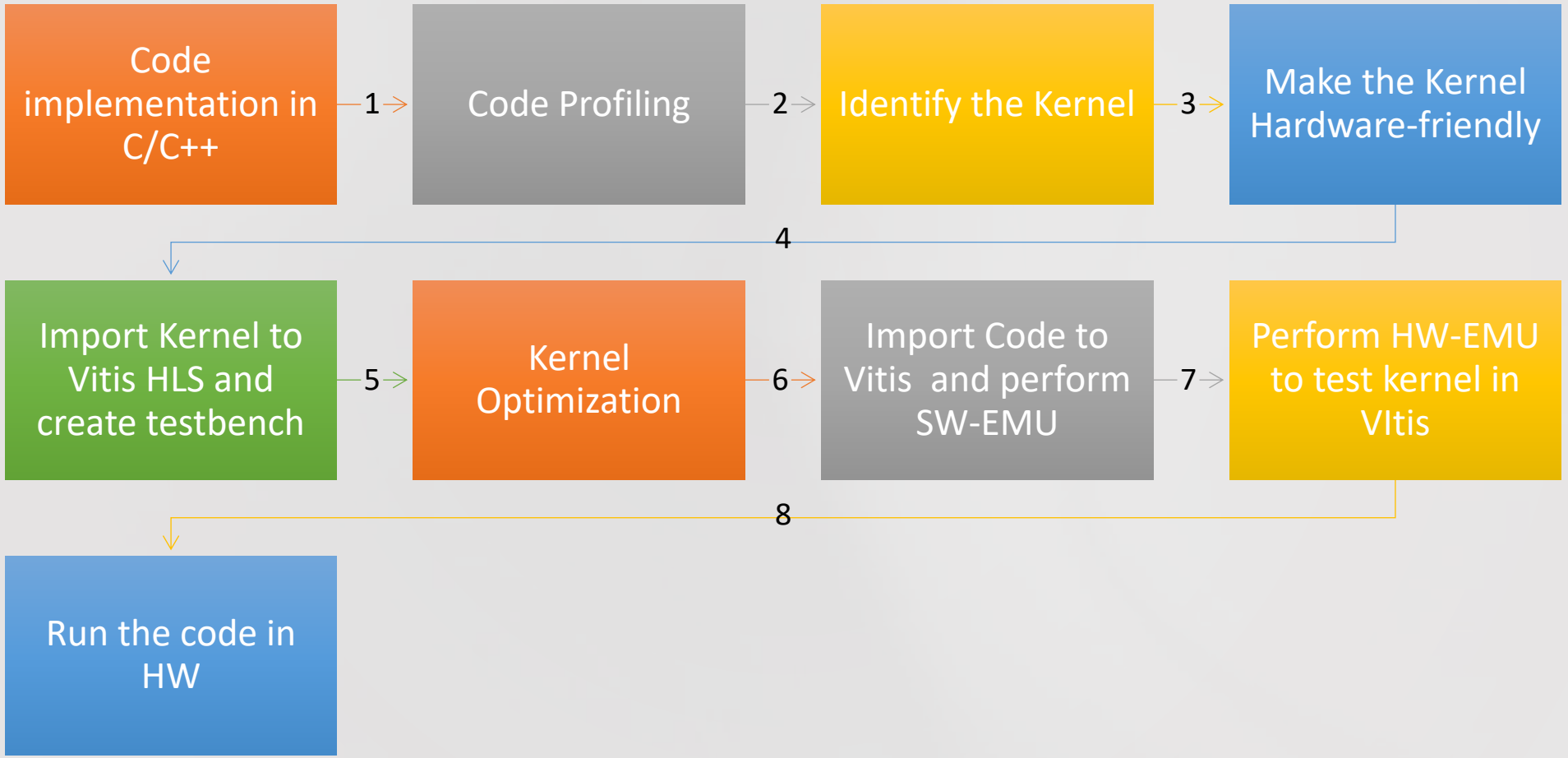
# Αρχιτεκτονική Προηγμένων Υπολογιστών και Επιταχυντών

Αθανασιάδης Άγγελος

# FPGA Vitis Acceleration Workflow



# Code implementation in C/C++

- Source code usually is in Python
- Implement Code in C/C++
- Test if it produces the same results with Python

# Code Profiling

Very important step!

Use a profiling tool (ex. Vtune) or perform profiling “by hand”

Profiling can happen in both Python and C/C++

We prefer to do in C/C++

# Identify the Kernel

Prerequisite: Profiling in C/C++

Locate the function/functions that take run in most of the runtime

This function/functions will be used as kernel/kernels

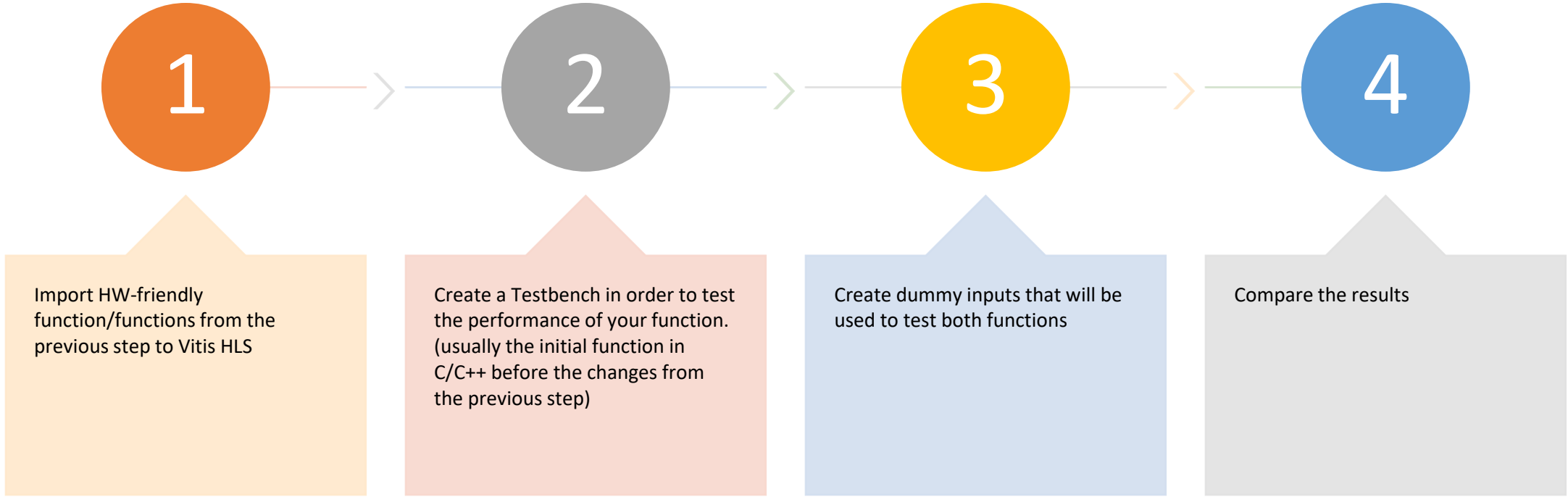
Not all codes can be accelerated with FPGAs!

# Make the Kernel Hardware-friendly

Important factors to consider when making a function FPGA hardware-friendly:

- Data Parallelism: Identify opportunities to parallelize data processing.
- Pipelining: Implement data pipelining to maximize throughput.
- Memory Access Patterns: Optimize memory access patterns for FPGA-specific memory hierarchies.
- Resource Utilization: Minimize resource usage and efficiently utilize FPGA resources.
- Latency vs. Throughput: Balance latency and throughput according to application requirements.

# Import Kernel to Vitis HLS



# Kernel Optimization (1)

- Improved Performance: Faster execution of the kernel on the FPGA.
- Resource Efficiency: Minimize resource utilization to accommodate more functionality.
- Reduced Latency: Decrease the time it takes to process data.
- Lower Power Consumption: Enhance energy efficiency for embedded applications.

# Kernel Optimization (2)

- Loop Unrolling: Expand loops to expose more parallelism.
- Loop Pipelining: Introduce pipeline stages for continuous data processing.
- Data Tiling: Divide data into tiles to enhance memory access patterns.
- Data Precision Reduction: Use smaller data types when possible, to reduce resource usage.
- Loop Interchange and Loop Fission: Reorder and split loops for improved parallelism.
- Memory Optimization: Minimize data movement between on-chip and off-chip memory.
- Resource Allocation: Efficiently utilize FPGA resources.

# Import Code to Vitis and perform SW-EMU

1. Create a Vitis project for your application
2. Import the whole application including the parts that will run in SW only
3. Import the kernel in the corresponding area
4. Verify that the code runs and produces correct results

# Perform HW-EMU

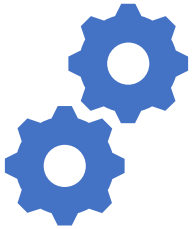
Change the build configuration to HW-EMU

Run the code once, with small size of parameters to test if the kernel runs correctly before we test it on hardware

We usually use it once, only to test our kernel

Takes lots of time for each kernel execution, which usually makes it unaffordable to wait for the whole application to run.

# Run the code in HW



Change the build configuration to Hardware

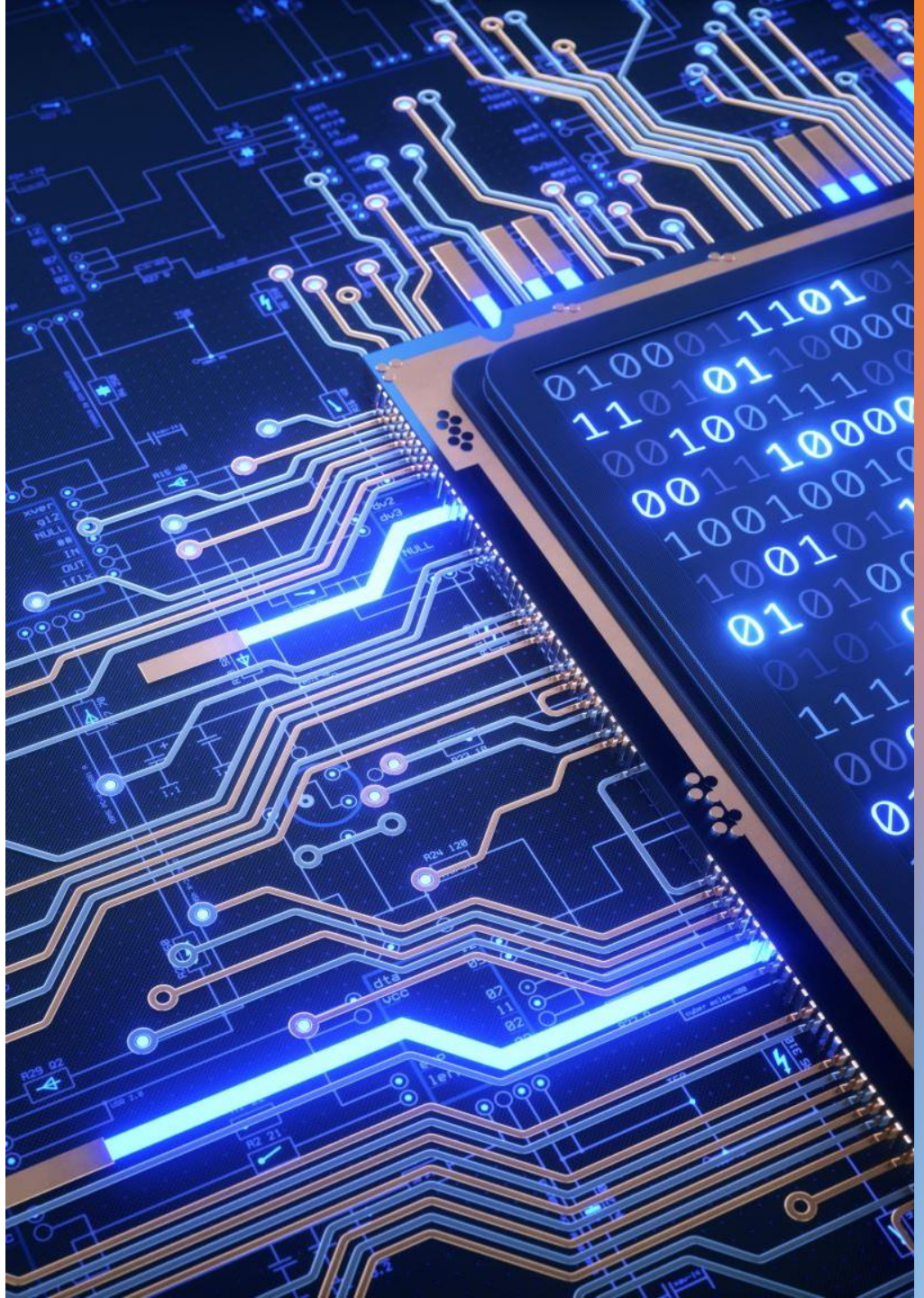
After build, check the utilization of your board/acceleration card

Finally, test your application on hardware

# Further optimization

If further optimizations are needed in order to achieve better performance, we usually do the following:

1. Go back to Vitis HLS Kernel Optimization Step
2. Optimize the kernel further
3. Test that it still produces the same results
4. Use the new kernel for Vitis Hardware build
5. Check again new utilization and run in on Hardware



# Further info on FPGAs

- https://fpgainsights.com/fpga/fpga-basics/
- https://learn-fpga-easily.com/the-essential-role-of-clbs-in-fpga/
- https://www.theeeview.com/what-are-luts-ffs-dsps-and-brams-in-an-fpga%EF%BF%BC/
- https://nandland.com/lesson-1-what-is-an-fpga/
- https://www.vemeko.com/blog/fpga-knowledge-overview.html

Thanks for your attention!
any questions?