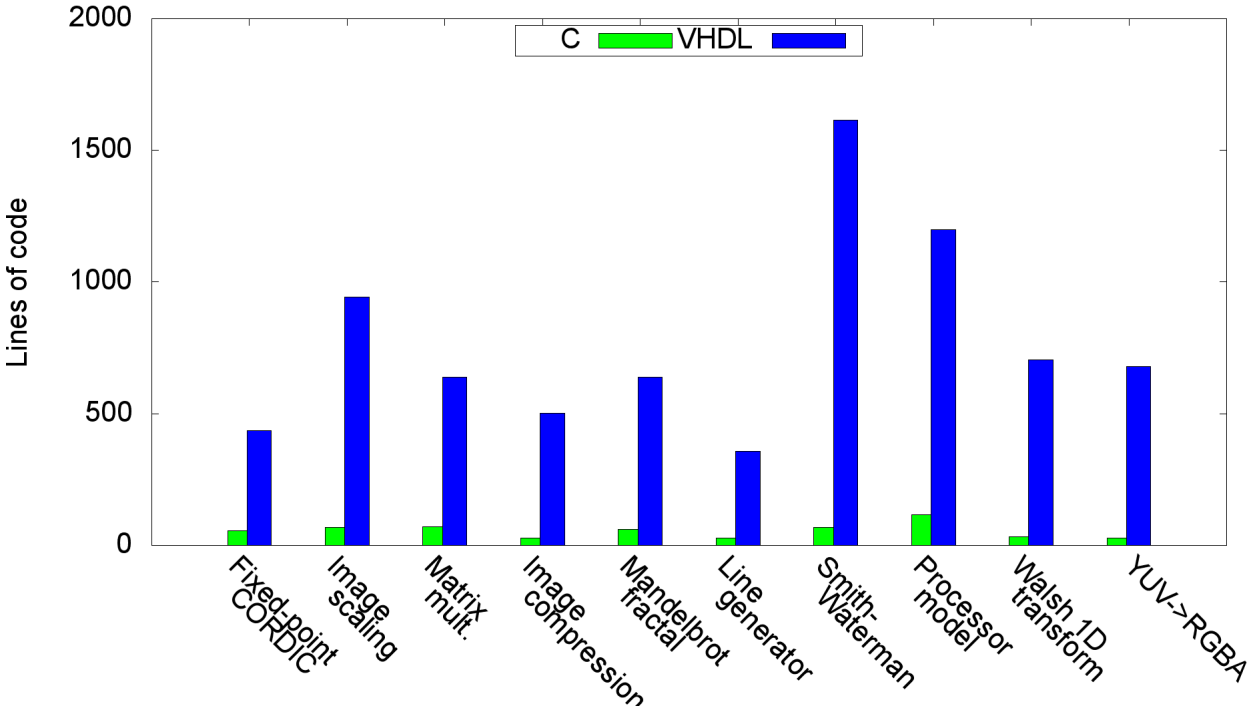
10x productivity boost with HLS: Myth, legend or fact?

There exists a saying about high-level synthesis (and any new disruptive technology, I must add). Maybe you have heard it before reading these lines; as an ingredient to a pitch, on colleagues' coffee break, at a press conference or maybe you have found yourself thinking more-less of this: "When I become proficient with this XYZ HLS tool, I will be 10x more productive! I will be able to fast-forward this task, put a smile on my supervisor's face; earn a good deal of quality time too!"

Well this reads as a dream-come-true, but is it reasonable to believe in it? Is it already possible, or will it become possible (as in "how soon is now?") Being an order of magnitude more productive; is this myth, legend or a factual thing?

HLS attempts to solve the tedious iterations at the RTL (primarily). This is done by having the developer describe the intent or behavior of the system using increased abstraction. No need to reason for each FSM state, exact timing, and taking every stepwise error-prone decision. Ideally, by giving the algorithm and a set of requirements to the HLS tool, you can effectively get a correct design with the desired QoR. Or get a no-go.

To add some data into the soup, let's examine the following figure.



For these ten algorithms, ANSI C code is passed to HercuLeS HLS (http://www.nkavvadias.com/hercules/). At the end of the flow, other things apart, we get the synthesizable RTL VHDL for the given C code. By measuring the VHDL-to-C ratio of useful lines of code (no comments etc), a 7.8x to 25.15x variation with an average of about 15 times (15.4x) is obtained.

This is by no means conclusive but it is indicative. HercuLeS takes your code through a series of transformations; from C to N-Address Code (intermediate language) to graph-based representation (in Graphviz) to VHDL finally. Every step is from a higher abstraction level to a lower one and every step results in a more detailed representation as it moves from software to hardware (or HW/SW).

This approach sounds obviously the correct one. But bare in mind the tarpit over there, this is no low-hanging fruit. 10x less code to write does not necessarily yield 10x less time to develop it. What must also be accounted for are:

1) Iterations in developing the C code itself.

2) Multiplicity in the number of generated RTL designs, as points in the architectural design space. These iterations are sloooow, so you have to keep them a) few, b) short. For this reason, they should not happen at the RTL, but rather at the level of an IL (intermediate language) estimator/profiler.

3) RTL design is only a fraction of concept-to-implementation. As cliche as it may read, this is Amdahl alright.

4) Coding for parallelism, whether implicit or explicit, is challenging in the sense of near-optimal scaling for irregular applications and adequacy of the underlying hardware paradigm.

And the verdict: tenfold increase in productivity is a FACT, and you should be able to experience it, if you do it the RIGHT way!