An Analysis of Programming Paradigms in High-Level Synthesis Tools

Pieter Staal
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.j.staal@student.utwente.nl

ABSTRACT

The imperative, functional, and concurrent programming paradigm are compared to determine which paradigm best fits the task of synthesizing an algorithm to digital hardware. For this comparison three tools are investigated: the Intel HLS Compiler, Clash, and the Intel OpenCL SDK. The performance of the synthesized product is assessed by analyzing the maximum runnable frequency, the die area used on the FPGA, simulations, and verification on the FPGA. The throughput of the written code when synthesized for an FPGA and when compiled for a regular processor is compared. The tools are compared by some metrics related to wide-spread adoption of the tool. The experience of developing an algorithm in the languages and with the tools is discussed.

It was found that the Intel HLS compiler usually performs better in both area used and maximum frequency. However, it was also found running the Haskell code on an FPGA instead of on a CPU is in both tested cases more than one hundred times faster. It is concluded that despite the in general worse absolute performance the functional paradigm is the best fit, since the experience with the tools showed that Clash has the most straight-forward transformation from source to hardware. The worse performance is explained by the fact that the Intel HLS Compiler has had more resources committed to it.

Keywords

Embedded Systems, High-Level Synthesis, Programming Paradigms, OpenCL, Clash, Haskell, C++.

1. INTRODUCTION

Since the decline of Moore's law, devices other than a conventional processor are gaining popularity. One of these is the *Field Programmable Gate Array* (FPGA). An FPGA is an integrated circuit that can be reconfigured by reprogramming it. This makes it possible to rapidly prototype digital logic. Applications requiring processing of data streams or parallelizable algorithms can be greatly accelerated using the configurable data-paths of FPGAs [14].

FPGAs are configurable with a *hardware description language* (HDL). Popular languages are VHDL, Verilog, and SystemVerilog. These languages are low-level specifications of the hardware, at the so-called *Register Transfer Level* (RTL). An FPGA programmer, therefore, needs expertise on the subject of hardware design to implement their algorithms. These low-level languages do not offer powerful abstraction constructs available to software designers in higher level languages like C++ or Haskell. These abstractions make it easier to develop and maintain large complex systems. Being able to work with these abstractions could help digital designers to build larger systems and to get their product to the market sooner.

To bring these abstractions from software development to hardware development several so-called *high-level synthesis* (HLS) tools have been developed. Most of these tools accept a C-like language as their input [16] and infer the structure of the FPGA from the supplied input file. Since C is an imperative language it is very much sequential in nature, while the inferred FPGA structure should exploit the parallel nature of digital hardware as much as possible. This makes the synthesis process convoluted and inconsistent across tools.

Not all high-level synthesis tools use C however. Some tools are based on languages following different paradigms. Some popular paradigms are the imperative, functional, and concurrent paradigm. In the imperative paradigm computations are described as sequential instructions. The functional paradigm treats computation as the evaluation of mathematical functions. The concurrent paradigm focuses on programming for heterogeneous systems, where a host executes the main computations and is supported by accelerators. *In this research, the paradigm which best fits digital hardware design is investigated.*

2. RELATED WORK

Almost all high-level synthesis tools that are widely used are based on C. Nane et al. [16] presents a comparison of 33 HLS tools used at the point of writing. It presents an overview of the inputs and outputs of the languages, whether they are still maintained, licensing options, and much more. They ran benchmarks for four of the evaluated tools to evaluate their performance in certain algorithms. Here, they found that the performance of the same algorithm written in the same language could be very different when synthesized by different HLS tools. This shows that most HLS tools do not use a structured approach but instead optimize and infer what they can from the input file.

Meeus et al. [15] shows the difference between the traditional RTL design flow and the HLS design flow in a Gasjki-Kuhn Y-chart [11]. These diagrams show that in the RTL flow behaviour is directly described by the low-

32nd Twente Student Conference on IT Jan. 31st, 2019, Enschede, The Netherlands.

level HDL. Designers working with the RTL flow have to work at the logic level to describe the system, but with the HLS flow the system can be specified on the more abstract algorithmic level.

A practical example of using HLS tools is given in Cong et al. [10]. They show that current HLS tools are viable solutions and they show that using an HLS tool improves design productivity and even FPGA resource usage.

In Chapter 3 of FPGAs for software programmers [14] a comparison of high-level synthesis solutions is presented. They evaluate availability, target architectures, type of computation, parallelization and abstraction support, and their input file format and language.

3. HLS TOOLS

To analyze different programming paradigms three HLS tools were chosen that all use a different paradigm. The Intel HLS Compiler compiles regular C++ to a project structure containing the HDL files necessary to synthesize the component. The Intel OpenCL SDK provides support for the OpenCL language, which was originally developed for highly concurrent computing systems like GPUs. The final tool is Clash, which can compile the functional language Haskell to VHDL, Verilog, or SystemVerilog.

3.1 Intel HLS Compiler (i++)

The Intel HLS Compiler [3] or i++ uses untimed ANSI C++ as the design source. The C++ language is a combination of the imperative and object oriented paradigm. In this research, the focus is on the imperative aspects of C++.

3.2 Intel OpenCL SDK

OpenCL [17] is a language written for heterogeneous computing platforms. Nowadays, it is used most commonly in GPU programming. Like FPGAs, GPUs excel at performing the same computations in parallel. However, a GPU consists of hundreds of processors instead of the reconfigurable net of logic modules that make up an FPGA. Since some of the language constructs of OpenCL are relevant to both GPUs and FPGAs it makes sense to use OpenCL as an input language for an HLS tool.

The Intel OpenCL Software Development Kit [5] allows users to develop for systems using Intel CPUs and FPGAs with OpenCL. The SDK focuses on the hybrid use of both a CPU and an FPGA by offering a common development environment for both the CPU ('host'), and the FPGA ('accelerator').

3.3 Clash

The Clash compiler [1][8][9] is a modification of the *Glasgow Haskell Compiler* (GHC), which is the de facto standard Haskell compiler. Clash transforms high-level functional descriptions of hardware to low-level synthesizable VHDL, Verilog, or SystemVerilog. The high-level descriptions are written in the functional programming language *Haskell*. Since Haskell is a pure functional language it is possible to directly map functions to hardware components and to intuitively connect them to form a larger system through higher order functions, which are also representable in hardware.

Clash allows the written designs to be evaluated in a custom version of Haskell's GHCI called *clashi*. With this interactive interpreter it is easy to test designs and verify that the system requirements are correctly implemented.

4. BENCHMARKS

To evaluate the chosen programming paradigms the benchmarks are divided in two categories. First, the end-result of the synthesis process is evaluated. This is investigated by comparing the throughput of the same code on a CPU and on an FPGA after synthesis. Secondly, the tool is assessed on its ease of use and the general experience of using the tool.

The synthesized HDL code generated by each tool was run on a DE1-SoC [7]. This is a development board with an *Intel Cyclone V FPGA*. This FPGA is more than powerful enough to run the benchmarks. The source code from which the HDL code is synthesized is also benchmarked on a regular desktop computer. The CPU used in this benchmark is an *Intel Core i5-9400 CPU* running at 2.90GHz.

4.1 Performance

To test the performance, two of the algorithms implemented in the CHStone [12] benchmark suite are partially implemented in C++ and Haskell and synthesized by their respective tools. This benchmark suite is specifically designed for C-based HLS tools [13]. The performance of a synthesized program is evaluated by data gathered from the report generated by Intel Quartus, which is the tool used for compiling HDL files to programming files for the FPGA. Furthermore, the performance is evaluated by data gathered from simulations and benchmarks.

Quartus Report

- **Area used**: FPGAs have limited available space to use for designs. A more efficient HLS tool will generate designs that occupy a smaller area on the chip.
- **Maximum frequency**: The final step in synthesis to hardware is to put the design into a timing model and analyze its maximum runnable frequency. If the design is run at a frequency higher than that there can be no guarantee of correct results when reading from the output since the computation is not done yet. A better HLS tool will generate a design that can be run at a higher frequency, provided that this actually speeds up the calculations.

Benchmarks

- **Cycles / calculation**: HLS tools can choose to split a design in several steps. This means the overall design will be able to run at a higher frequency, but to perform the full calculation several cycles are necessary. This metric is determined by investigating simulations of the generated designs.
- **FPGA throughput**: The throughput of the design in operations per second on an FPGA in millions of calculations per second. This number is derived from the maximum frequency Quartus reports divided by the amount of cycles per calculation.
- **CPU throughput**: The throughput of the original source code in millions of calculations per second when compiled and run on a regular desktop computer.

The chosen test programs are a simplified implementation of the MD5 hashing algorithm and an implementation of positive floating point addition.

4.1.1 Hashing algorithm

The implemented hashing algorithm is based on the MD5 hashing algorithm. MD5 can take an arbitrary length message and processes it to a 128-bit length hash. MD5 was chosen because there are many reference implementations available. The version of MD5 written for this research can receive a message of 32 bits, encoded as a single 32 bit unsigned integer. It then processes the message like MD5 would, but only returns the first 32 bits of the generated hash. The algorithm was downgraded to this because this still preserves the main processes of the algorithm while working with it in an HLS and FPGA context becomes easier. With only a 32-bit input and output to be considered, no RAM modules have to be used to work with the data. This also allows us to use the readily available datatype `unsigned int` in C++ and the `Unsigned 32` type in Clash. The removed features are not necessary to demonstrate the difference between high-level synthesis tools since both tools still compile the same algorithm, but simplified.

4.1.2 Floating point arithmetic

Another test class of the CHStone suite is a floating point division algorithm, which is why floating point arithmetic was added as one of the tests. A simplified version of floating point addition was implemented which ignores the sign bit. The algorithm is still a representative algorithm for testing since it performs a useful task, simple positive floating point addition.

4.2 Language Experience

This benchmark is far harder to quantify objectively than the performance. For each tool the focus is on certain features that would either hinder or aid wide-spread adoption of the tool. The experience of developing an algorithm in the language and the tool is also discussed.

The following specific features are taken into consideration.

- **Language adoption of the supported language**: The amount of developers actually familiar with the high-level language is considered. Adoption of an HLS tool will be faster if new developers do not need to learn a new language to use it.
- **Availability of support**: Active support forums or support plans can prevent developers reinventing the wheel and can make solving problems with the tool faster.
- **Operating system support**: Being available on major operating systems would aid in reaching a more wide-spread adoption of the tool.

5. RESULTS

In this section the results are presented. They are split in two sections, the performance of the tools and the experience of developing in them.

The Intel OpenCL SDK could not be used to generate the necessary results for the performance comparison. This tool focuses on the integration of an FPGA in a hybrid system containing both an FPGA and a processor, instead of being able to generate isolated components. This is noticeable in the fact that this tool needs access to a ”Board Support Package”, which defines not only the FPGA it needs to compile for but also other hardware available on the board. Clash and i++ take less than a minute for simple designs, and not much longer for larger designs. Furthermore, the compilation times are impractically long. A simple demonstration program in OpenCL already takes over 20 minutes on a powerful server and the ”getting started” guide states that compilation can take ”several hours” [4].

Because of these reasons and general time constraints the examining of the exact performance of the Intel OpenCL SDK was dropped. It is however discussed in the language experience section.

5.1 Performance Results

	F_{max} (MHz)	Area (%)
Clash	3.13	14.0
i++	113.77	1.77

Table 1. Quartus report for MD5

	F_{max}(MHz)	Area (%)
Clash	78.03	0.76
i++	268.67	0.61

Table 2. Quartus report for floating point addition

	Cycles/hash	FPGA (MH/s)	CPU (MH/s)
Clash	1	3.13	0.018
i++	78	1.46	2.8

Table 3. MD5 benchmarks

	Cycles/FLOP	FPGA (MFLOPS)	CPU (MFLOPS)
Clash	1	78.03	0.0939
i++	1	286.67	15.0

Table 4. Floating point addition benchmarks

In Table 1 and 2 the results of compiling the synthesized algorithms in Intel Quartus are presented. F_{max} is the maximum frequency at which the synthesized design is able to run according to Quartus. *Area* is the percentage of *Adaptive Logic Modules* (the basic building blocks of Intel FPGAs) used by the design in the FPGA.

Table 3 and 4 show the results of the CPU and FPGA benchmarks. *Cycles / hash* is the amount of cycles it takes before the complete calculation of a hash is finished. Likewise, *Cycles / FLOP* is the amount of cycles one *Floating Point Operation* takes. The FPGA and CPU columns depict the throughput of the algorithms on the FPGA and on the CPU in million hashes per second or million floating point operations per second.

5.1.1 Semantics Verification

Both the Haskell and C++ implementation of the algorithms in question were tested whether they produced correct results. For the algorithms written in Haskell this was done inside the interactive interpreter that comes with clash. The C++ programs were built as normal C++ programs, with the special HLS keywords and types omitted.

The hashing algorithm was tested against a reference implementation written in C. Both the Haskell and C++ implementation gave the same results as the reference implementation. The positive floating point addition was tested by adding the values 100.0, 20.0, 0.375, and +0 to each other in different combinations and checking whether the result was correct, this was indeed the case. These numbers were chosen because these cover most of the edge cases in floating point addition.

An important requirement of an HLS tool is that the result of the program is not different after synthesis. Therefore it was verified on an FPGA whether the programs produced the same output for the same inputs. This was the case for programs synthesized by both Clash and i++;

5.2 Language Experience Results

This section is based on the experience of developing the algorithms with Haskell, C++, and to some extent OpenCL. Further results are based on the experience with setting up and using the tools that were investigated.

5.2.1 General remarks

During development of the benchmark algorithms it stood out how well the files Clash generated matched with the original implementation. Some parts of the files are even annotated with line numbers from the original Haskell source. As is shown in Section 4 in the thesis presenting Clash [9], the intermediate representation of the original source produced by the front-end of GHC is transformed such that it can be compiled to hardware.

Great contrast to this is the output of i++. Instead of a single HDL file, this tool generates a complete Quartus project. This also locks users of i++ into Intel's ecosystem instead of giving them the freedom to compile their component the way they want. i++ does generate a subdirectory with HDL files that contains the generated component. However, even a simple component is built up from over fifty files in three different languages (Verilog, SystemVerilog, and VHDL). Compiling this with any other tool than Quartus requires reverse engineering the complicated project structure and is therefore impractical.

The output files of i++ show that a large supporting structure is generated around the component. This structure supports features like stacks, RAM access, queues, and more. These files are always included when a component is compiled by i++, even with a simple 32-bit counter. Whether all this extra structure is necessary is unclear. This is different from Clash, which generates only one file per component.

5.2.2 Language adoption

According to the StackOverflow developer survey of 2017 [6] 19.3% of professional developers use C++, and about 15.4% use C. A developer proficient in these languages will certainly be able to find their way around the constructs used by the Intel HLS compiler since it supports regular C++, and the specific API calls are all in a fashion recognizable to C and C++ developers. The survey of 2017 is cited here because this is the most recent one where Haskell appears: 1.4% of professional respondents use it.

5.2.3 Support availability

The Intel HLS compiler and the Intel OpenCL SDK are both proprietary tools made by Intel. The Clash compiler is an open-source project licensed under the BSD2 license. Their different development methods gave rise to different support structures.

For the Intel tools documentation is made available through their website. A support forum also exists. The documentation is very extensive, but made available in such a way that search engines have trouble searching through it, making finding specific parts of the documentation hard. In the context where these tools are usually used, namely at a company, this works fine. There it is possible to walk to a colleague who has encountered a problem before and work the problem with them, when developing outside that environment solving problems becomes harder because of the generally low amount of online open discussion regarding the tools and the hard to search through documentation.

The documentation for Clash follows the standard format for Haskell documentation, so anyone who is familiar with reading Haskell documentation will have no problem with reading the documentation for Clash. Since Clash is a very direct modification of GHC, most problems encountered with Clash after getting it up and running are with Haskell. Unlike the Intel tools, Haskell is discussed very often and in-depth on online open forums. This means that with only the internet it is often possible to solve any encountered problems.

5.2.4 Operating system support

All tools support both Windows and Linux. The Operating system of a potential user of a high-level synthesis tool therefore has little effect on the choice of which tool to use, since this accounts for a large majority of users [2].

6. DISCUSSION

6.1 Performance Discussion

As shown in Table 1 and Table 2, the F_{max} of the generated design by Clash is lower than that of the design generated by i++. Furthermore, the area used by the Clash design is larger than that used by design generated by i++. The higher F_{max} can be explained by the fact that the i++ design takes more cycles per calculation in the case of the MD5 benchmark. By using more, briefer, steps the frequency can be increased. This technique also uses less area since components calculating the same thing can be re-used in the next cycle.

In the case of the floating point benchmarks both designs use the same amount of cycles per calculation. Here there are two more possible explanations for the better results of i++. Firstly, one could suspect that the functional paradigm is less suited for hardware synthesis and needs a lot of boilerplate surrounding it to make it work. As seen in Section 5.2.1 the reverse is the case, i++ is the tool that generates a large support structure around the functions to make them work. The other explanation is that i++ has more resources invested in it, and is therefore more sophisticated and more tuned to the rest of the Intel environment. This is more plausible since the entire tool-chain that generated this data is owned and maintained by Intel. Clash is, however, a far smaller open source project and presumably has not been able to implement many optimizations.

Looking at the ratio between the throughput on the FPGA and the throughput on the CPU results in the following table:

	Algorithm	ratio
Clash	MD5	173
	Float	778
i++	MD5	0.52
	Float	19.1

It is clear that the performance increase gained by running Haskell on an FPGA is far larger than that of running C++. In the case of the MD5 benchmark it is even slower to run the code on an FPGA instead of a CPU. Comparing how data flows through a program in an imperative language and in a functional language explains this difference. In an imperative language every line of code is an instruction, and the program steps through those instructions. Data is stored and accessed in some data bank. This is nearly exactly how modern computers are engineered, their processors step through small instructions and all the data is stored and accessed in cache and memory. In a functional language the data flows through functions, starting with some input data at the top level function and then branching out, flowing deeper into the program, and finally flowing back to the start to return the end result. A modern computer does not correspond well with this model, but this does come much closer to how digital hardware works.

So this shows that when using C++, porting it to run on an FPGA will not always yield an increased performance. Haskell, however, strongly under-performs on a CPU, therefore when using Haskell greatly increased performance can be attained by compiling it for an FPGA.

6.2 Language Experience Discussion

Working with OpenCL and imperative C++ is both quite similar to C. So with the numbers from the StackOverflow survey it is clear that an HLS tool based on a C-like language would be far easier to adopt by most professional developers since they are already familiar with the language. This is probably why most HLS tools use a C-like language. Using a language unfamiliar to developers sounds unwise: they might as well learn VHDL or Verilog, although this would not give them access to the strong abstractions higher level languages like Haskell have to offer.

The different support structures say more about how and by whom the tools were developed than the programming paradigms their languages are based on. There was no connection found between paradigm and support structure. While interesting to anyone who needs to choose an HLS tool, it does not help with determining which paradigm is better suited for an HLS tool. The same is true for the operating system support.

Languages like OpenCL following the concurrent paradigm can be fitting high-level language for an HLS tool because of their origins in hybrid systems. However, as was found the current set-up of the development environment is highly specific to the target board. This limits the HLS tool in that it can only be used for less general purposes. Where in other tools it would be easier to create libraries with reusable functions, this would be harder with OpenCL.

7. CONCLUSION

In this research, the paradigm that best fits hardware design was investigated. Of the tools whose performance was benchmarked, i++ performed better in both the F_{max} and the occupied area. However, it was noted that the relative improvement of the algorithms written in Haskell was much greater than those written in C++. It is concluded that this is because the program flow in a functional program is very unlike the operation of a modern computer and much more like digital hardware. Furthermore, it was shown that the synthesis process to an HDL from a functional language is much more straight-forward and less convoluted than synthesis from an imperative language. *Therefore it is concluded that the functional paradigm is the better fit for the design of digital hardware.*

However, it must also be stated that functional languages are far less popular than imperative languages. This makes it hard to justify investing many resources in an HLS tool based on a functional language, since adopting the tool will be very hard for most professional developers.

Limitations of this study are that the large differences in optimization and how well the tools fit in the ecosystem for both compared tools made it hard to isolate the comparison to the programming paradigm. Furthermore, the concurrent paradigm was not benchmarked for performance. Future work could for example select tools that have a roughly equal amount of resources invested in them, or try to build prototypes for tools for all paradigms from the ground up. Future studies could also look into whether the FPGA model matters, since the algorithms were only synthesized for Intel FPGAs. Furthermore, it was not investigated to which extent the programmer implementing the algorithms matters. Different approaches in either paradigm could change the results.

8. REFERENCES

[1] Clash. `https://clash-lang.org/`. Accessed: 2019-12-01.

[2] Desktop operating system market share worldwide. `https://gs.statcounter.com/os-market-share/desktop/worldwide/2019`. Accessed: 2020-01-14.

[3] High-level synthesis compiler - intel®hls compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. Accessed: 2019-11-30.

[4] Intel fpga sdk for opencl pro edition. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf`. Accessed: 2020-01-13.

[5] Intel®sdk for opencl™applications. `https://software.intel.com/en-us/opencl-sdk`. Accessed: 2019-11-30.

[6] Stack overflow developer survey 2017. `https://insights.stackoverflow.com/survey/2017#technology`. Accessed: 2020-01-10.

[7] Terasic - soc platform - cyclone - de1-soc board. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&No=836`. Accessed: 2019-11-28.

[8] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. Clash: Structural descriptions of synchronous hardware using haskell. In *Proceedings - 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2010*, pages 714–721, 2010. Cited By :39.

[9] C. P. Baaij. *Digital Circuits in ClaSH – Functional Specificiations and Type-Directed Synthesis.* PhD thesis, University of Twente, PO Box 217, 7500AE Enschede, The Netherlands, jan 2015.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. Cited By :379.

[11] D. Gajski and R. Kuhn. New vlsi tools. *Computer*, 16(12):11–14, dec 1983.

[12] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009. Cited By :181.

[13] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009. Cited By :183.

[14] D. Koch, F. Hannig, and D. Ziener. *FPGAs for software programmers*, pages 1–327. FPGAs for Software Programmers. 2016. Cited By :5.

[15] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. Cited By :79.

[16] R. Nane, V. . Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. Cited By :152.

[17] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–72, 2010. Cited By :723.