Challenges Designing for FPGAs Using High-Level Synthesis

Clayton J. Faber
Steven D. Harris
Zhili Xiao
Roger D. Chamberlain
Anthony M. Cabrera

Clayton J. Faber, Steven D. Harris, Zhili Xiao, Roger D. Chamberlain, and Anthony M. Cabrera, “Challenges Designing for FPGAs Using High-Level Synthesis,” in *Proc. of IEEE High-Performance Extreme Computing Conference* (HPEC), September 2022.

McKelvey School of Engineering
Washington University in St. Louis

Architectures and Performance Group
Oak Ridge National Laboratory

Challenges Designing for FPGAs Using High-Level Synthesis

Clayton J. Faber, Steven D. Harris, Zhili Xiao, Roger D. Chamberlain
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, MO, USA
{cfaber,sharris22,xiaozhili,roger}@wustl.edu

Anthony M. Cabrera
Architectures and Performance Group
Oak Ridge National Laboratory
Oak Ridge, TN, USA
cabreraam@ornl.gov

***Abstract*—High-Level Synthesis (HLS) tools are aimed at enabling performant FPGA designs that are authored in a high-level language. While commercial HLS tools are available today, there is still a substantial performance gap between most designs developed via HLS relative to traditional, labor intensive approaches. We report on several cases where an anticipated performance improvement was either not realized or resulted in decreased performance. These include: programming paradigm choices between data parallel vs. pipelined designs; dataflow implementations; configuration parameter choices; and handling odd data set sizes. The results point to a number of improvements that are needed for HLS tool flows, including a strong need for performance modeling that can reliably guide the compilation optimization process.**

***Index Terms*—HLS, FPGA, High-Level Synthesis, Field-Programmable Gate Array.**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are receiving considerable attention in recent years as computational accelerators [1], [2]. Multiple orders of magnitude performance improvements are possible relative to traditional software executing on processor cores [3]–[6]. A significant challenge to their adoption, however, is the fact that historically FPGAs have been programmed at an extremely low level of abstraction. Hardware Description Languages (HDLs) such as VHDL or Verilog were (and still are) used to specify both the computational data path as well as the detailed schedule (what happens on each clock cycle). While this can result in highly-performant designs, the process is quite inefficient in human terms [7] and is also fairly error prone [8].

High-Level Synthesis (HLS) is an approach to alleviating this challenge, by allowing application authors to describe their computation using high-level languages. While HLS research has been ongoing for some time [9], [10], the technology has now reached sufficient maturity that both major FPGA manufacturers, Intel and Xilinx, offer HLS tools for developers using their parts. The availability of the tools, however, has not totally alleviated the challenge of producing performant designs [11]–[13].

This work was supported by NSF awards CNS-1763503 and CNS-1814739, the DARPA MTO Domain-Specific System-on-Chip program, and the DoE ASCR program.

Here, we describe a number of circumstances where the HLS tool flow gives counter-intuitive results. A programming construct, parameter setting, design choice, pragma, or other feature of the tools is applied to an application; the application is built and deployed on an FPGA, and performance measurements are taken; and rather than the performance benefit that was anticipated, in many cases a substantial performance degradation is experienced.

We provide examples from sixteen distinct implementations of four different applications, some of which have been described previously in the literature and some of which are first being documented here. In each case, we give a description of the performance optimization attempted (including why we believed, at the time, that it was a good idea) and measured performance results both before and after the optimization is applied. In a number of cases, we can discern why the performance was disappointing, and when that is the case, we report it. In other cases, the resulting performance degradation is still a mystery.

II. Background and Related Work

Since the adoption of HLS tools by the major manufacturers, the majority of research in this field has been focused on closing the gap between programmability and performance. Most HLS systems target an algorithm written in C, C++, or extensions like OpenCL or SystemC to generate an HDL kernel for synthesis. This allows designers to utilize the power of custom hardware along with high-level abstractions that allow for either ease in programming or use of code blocks already written for general execution. However, it is often in this translation from higher-level algorithms into their hardware implementation that performance is lost.

Numerous researchers have demonstrated that the process of going from a high-level language to HDL is not a guaranteed speed up and in some cases can end up performing worse than single-threaded implementations executing on a traditional processor [9]. To assist in porting efforts, manufacturers have released a set of best practices and programming guides that help developers maximize the HLS tool's potential [14], [15] and researchers have demonstrated the benefit of re-writing code tuned to these guidelines [16]. However, somewhat counter-intuitively, there have been situations where applying

said advice causes no change or in some cases even worse performance [17].

In light of these problems, much of the research today falls into two categories: design space exploration (DSE) and development of tools that are used as additional steps in the HLS process. In the DSE space there are many examples of studying the effects of various parameters that change how hardware is generated for the HLS system. One example is the various `#pragma` commands that are available to change the parameters for code blocks. Sohrabizadeh et al. [13] describe a "simple" CNN HLS code (of 24 lines) that runs 80× slower than a single thread on an individual core. After inserting 28 `#pragmas`, it speeds up 7000×. This example illustrates the challenge before us, the number of `#pragmas` used to achieve the best performance exceeds the number of lines of code in the initial implementation. Other works doing design space exploration for HLS include [17], [18]. A recent approach, investigated by Sun et al. [19], uses machine learning approaches.

Beyond pure HLS DSE, other research tools focus on either on looking at domain specific changes [20] or general code changes at the source [21], [22] and/or the Intermediate Representation (IR) level [23], [24]. These types of tools have been extremely important in filling in some of the gaps that are missed by the HLS tools, however, they add another engineering burden on the developer.

III. Design Choices and Applications

The design choices available to an application developer using HLS come in a number of different forms. We report instances of unanticipated poor performance results in four distinct categories of design choices: (1) programming paradigm choices between data parallel vs. pipelined designs; (2) dataflow implementations; (3) configuration parameter choices; and (4) how to handle odd data set sizes. Each of these is described in turn next.

A. Data Parallel vs. Pipelined Implementations

For the most part, achieving improved performance on FPGAs is primarily an exercise in identifying and exploiting parallelism. In OpenCL (supported by both major FPGA manufacturers), this primarily happens in two forms. The first, which is the common approach for deployment on graphics engines, is data parallelism (often called Multiple Work Item, or MWI), in which multiple copies of the data path are driven via common control, and different data elements are deployed across the data paths. This is akin to the Single-Instruction, Multiple-Data (SIMD) terminology of Flynn [25]. The second is pipeline parallelism (often called Single Work Item, or SWI), in which a stream of data elements' computations are deeply pipelined in a single data path. For both manufacturers, the SWI approach is recommended for applications expressed in OpenCL [14], [15].

Our experience has not been consistent with this guidance. We have experience with a number of applications for which the MWI implementation performs better than the SWI implementation. Figure 1 shows the speedup of the MWI implementation relative to the SWI implementation for four applications (data from [26]–[28]). These four applications can be split into two categories: compute intensive applications in the form of a standard matrix-matrix multiplication and streaming computations drawn from the Data Integration Benchmark Suite (DIBS) [29], [30]. To factor out issues with I/O, the input and output data reside in main memory. For the matrix-matrix multiply applications, the MWI implementation outperforms the SWI implementation by more than two orders of magnitude, and the performance of MWI over SWI is more than one order of magnitude for the two applications chosen from DIBS. It should be noted that for the streaming applications from DIBS the computation is primarily memory-bound. The measured data throughput for each of the applications is shown in Table I.

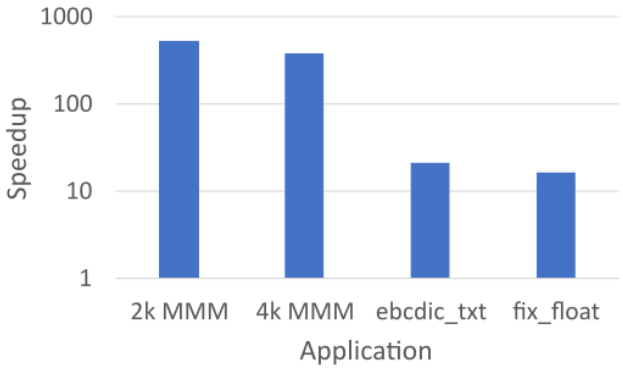

Fig. 1. Speedup of MWI over SWI implementations. Applications are 2k×2k and 4k×4k matrix-matrix multiply plus data integration applications `ebcdic_txt` and `fix_float` from DIBS [29].

TABLE I
Measured throughputs for MWI and SWI implementations.

Application	SWI (MiB/s)	MWI (GiB/s)
2k MMM	8	4.2
4k MMM	8	3.1
`ebcdic_txt`	260	5.5
`fix_float`	400	6.5

This first set of examples is a case where the advice provided by the device manufacturers did not result in a more performant implementation. Manufacturers in their programming guides recommend a SWI approach for a programming model [14], [15], whereas a MWI model appears to be more preferred when porting these codes into the FPGA space. We have previously investigated the use of memory access patterns as a predictor of which paradigm results in more performant designs [27], that study was limited to a small set of applications within a restricted class (data integration applications).

B. Dataflow

In both OpenCL and C variant HLS tools, manufacturers suggest the use of SWI work items instead of their MWI counterparts. This often involves deeply pipelining unrolled loops within the kernel code executing on the FPGA. When working with the SWI approach, it is important to have each loop be as independent as possible across iterations so as to not cause loop dependence, which can increase the loop iteration interval resulting in a longer running time. In this approach, it can sometimes be difficult to separate reads and writes in such a way that enables the pipeline parallelism to work well. The Xilinx Vitis toolchain has attempted to address this with a programming strategy available for both OpenCL and C HLS kernels known as *dataflow optimization*.

In the documentation, dataflow optimization is described as a way to implement "task-level pipelining," which allows for code blocks contained in functions to be scheduled in a way to achieve pipeline parallelism, similar to the approach that loop unrolling does within a loop. As shown in Figure 2, the idea is to create a collection of tasks or functions that would normally run in sequence and allow the compiler, with the help of `#pragma` directives, to create hardware that allows for execution of downstream tasks to start before their preceding task has completed. This is achieved through the use of first-in-first-out (FIFO) buffers to pass data elements from one code block to the next. These code blocks could be some type of read from global memory, general compute functionality, then a write back to global memory, which allows the hardware to take advantage of blocked reads and writes. Ideally, this type of coding style, as the name suggests, can work well for data streaming applications such as the previously mentioned data integration applications.

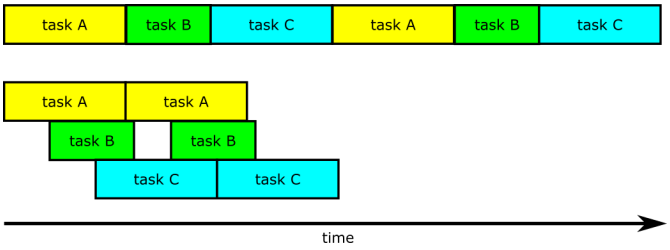

Fig. 2. Illustration of dataflow execution with 3 tasks. The top diagram illustrates a sequential execution timeline, while the bottom diagram illustrates a pipelined execution timeline.

Using this technique, we have seen success deploying a data integration application as a dataflow kernel, with performance better than the MWI and even the initial SWI implementation using an AWS F1 system utilizing a UltraScale+ Virtex VU9P card [31]. However, in other data integration applications we see that this style of kernel does not perform quite as well as expected. In an experiment implementing a handwriting database from an ASCII picture representation to a bit array representation (essentially the `optidigits` application from DIBS [29]) on an UltraScale+ Alveo U250, we find that the best performance is with the original SWI programming model with a throughput of 221 MiB/s. Believing that the dataflow approach would result in better performance, we made a pair of attempts using it. With our first attempt at the dataflow model we observed a slight reduction in throughput to 218 MiB/s, and on a second version, removing a write back to global memory at the end of the computation, we observed a more significant slowdown to 29 MiB/s. While we have a general understanding of why the second revision of the dataflow kernel is dramatically slower (the replacement of a bulk global memory write and replacing it with an iterative write back), it is still unclear to us why the first approach using the dataflow model is slower than the initial SWI version given our previous success with this model, algorithm type, and porting methods.

Furthermore, the dataflow model allows programmers to isolate and focus in on the computation step of a streaming application, isolating the data movement (reading and writing to both data streams and global memory) to separate tasks that are then combined in a larger kernel. This benefit of abstraction can sometimes be a frustration to a newcomer to HLS tools as it can become harder to determine where exactly the origin of the slowdown is located as it requires intricate knowledge of what is being created and why one method works over the other. It is our hope that developing effective performance models that inform the costs and benefits of deployments will cut down on development time.

The instances of unexpected poor performance described in the previous two sections are the result of programming paradigm choices, in effect, how the application developer expresses the available parallelism in the code. We next explore an example relating to the setting of configuration parameters.

C. Setting Configuration Parameters

Whenever an FPGA kernel is specified, there are a number of configuration parameters that impact the low-level details of the kernel's execution. For many of these configuration parameters, they impact fundamental tradeoffs in the design space, often between execution speed and FPGA resources consumed. Examples of this include loop unrolling depth, data path replication, etc.

Figure 3 illustrates the performance, shown in terms of execution time, for six separate implementations of the `fix_float` DIBS application implemented on the Intel HARPv2 platform using the Intel FPGA SDK for OpenCL. They are:

1) *CPU Seq* – a sequential implementation on a traditional processor core.
2) *CPU OpenMP* – a parallel implementation on processor cores of the HARPv2 using data parallelism expressed using OpenMP compiled using `-O3`.
3) *FPGA Naive* – A naive SWI kernel deployed on an FPGA with no pipeline flags installed.
4) *FPGA SIMD* – An MWI kernel deployed on an FPGA with the largest work-group size supported. Example in Listing 1.

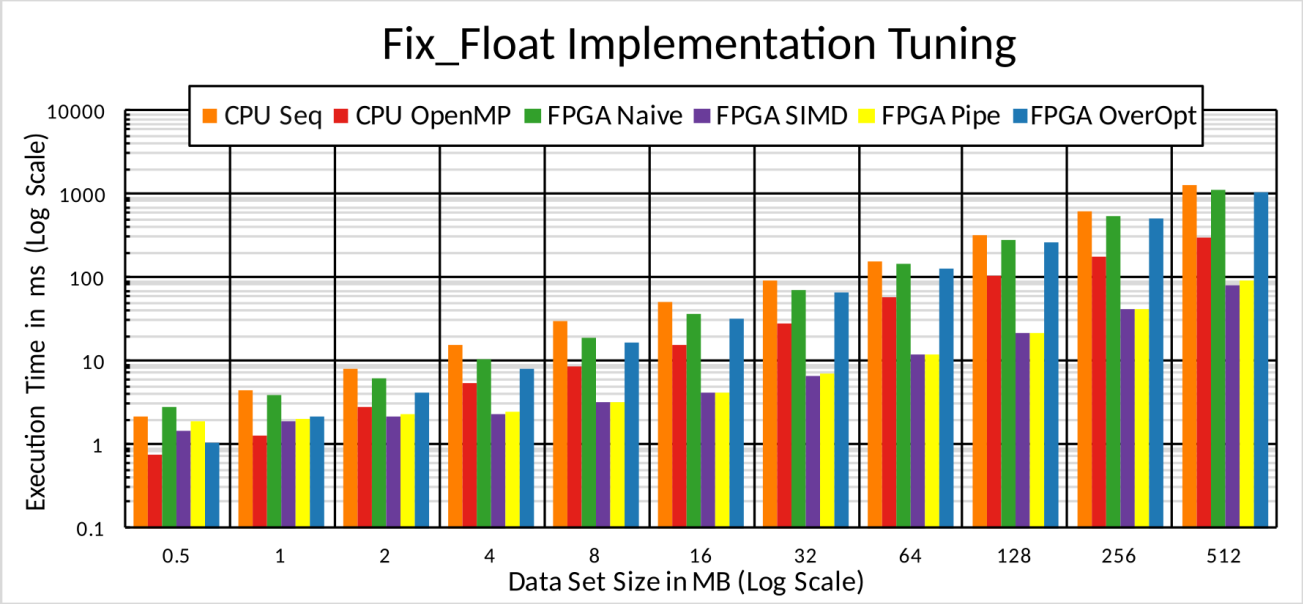

Fig. 3. Performance of several implementations of the `fix_float` application. Time and data set size are presented on a log/log scale.

5) *FPGA Pipe* – An SWI kernel that has been aggressively pipelined. Example in Listing 2.
6) *FPGA OverOpt* – An MWI kernel that has been maximally optimized for parallelism. Example in Listing 3.

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void computation(...){
    size_t gid = get_global_id(0);
    //Perform kernel computation
}
```

Listing 1. OpenCL kernel using a MWI programming style. Here, a global work ID (`gid`) determines what data piece a kernel works on. This kernel is queued with a range value equal to the number of elements to be computed.

```
__attribute__((reqd_work_group_size(1,1,1)))
__kernel void computation(...){
    #pragma unroll
    for(unsigned int i = 0; i < n_items; ++i)
        //Perform kernel computation
    }
}
```

Listing 2. An OpenCL kernel using an SWI programming style. In this style, a for loop is used to iterate through elements with a `#pragma unroll` to let the compiler know to agressively pipeline the computation.

The two CPU implementations are included in the figure to illustrate the performance of the FPGA implementations relative to traditional processor cores. The best-performing FPGA implementations are substantially faster than the fastest CPU implementation. The naive kernel has comparable performance to the sequential CPU implementation stressing the fact that the tools will not give you free performance on the FPGA and a number of configuration tweaks are needed. The performance

```
__attribute__((num_compute_units(2)))
__attribute__((num_simd_work_items(16)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void computation(...)
{
    size_t gid = get_global_id(0);
    //Perform kernel computation
}
```

Listing 3. Listing for the OverOpt kernel, similar in operation to the MWI kernel in Listing 1 however the attributes are added in an attempt to improve kernel performance.

of the naive kernel is not surprising, as we would expect such a simple kernel to not necessarily perform well on an FPGA given the current state of the design tools.

Our primary interest, however, is in the performance of the last kernel. Here, an MWI kernel has the largest work-group size supported (64 work-items), two compute units, and a SIMD work-item count of 16. Essentially, at every opportunity, the area-performance tradeoff was set to use area in an attempt to improve performance (primarily by increasing parallelism).

While this configuration performs well for small data sets, as the data set size grows its performance lags significantly behind the best performing implementations and even runs comparable to our naive FPGA implementation. For 512 MiB data sets, it runs a full order of magnitude slower than the regular MWI and SWI kernels. A developer could look at small data set size runs and make an incorrect choice in implementation strategy, strengthening the need for ways to evaluate the effects of configuration parameters before full synthesis runs. In general, how to set these configuration parameters is the research goal of a number of groups [13], [17]–[19].

D. Non-integral Data Size

There are a number of seemingly simple things within designs that can ultimately result in substantial performance implications (both to the good and to the bad). Here, we illustrate one of them, based on the size of the input data.

When designing an MWI kernel in OpenCL (in this case targeting the Intel FPGA Arria 10), the work-group size needs to evenly divide into the global number of work-items (so that work-groups each have the same number of local work-items). If the total amount of work to be done (the global number of work-items) doesn't divide evenly, it is common to pad the input appropriately so that it does. This necessitates the inclusion of a conditional bounds checking test within the code, as illustrated by Listing 4.

```
...
  unsigned int i = get_global_id(0);
  if (i < total_work_items) {
    // perform work for item i
  }
...
```

Listing 4. Bounds checking for proper termination. The `if` statement guards against doing unnecessary work that arises due to inflating the input buffer so that it is a multiple of the work-group size.

Figure 4 shows what happens in the synthesized hardware, via the control data flow graph (CDFG), when this conditional is added to the design (the example is from DIBS's `ebcdic_txt` [29] and has 4 copies of the data path). The figure shows the cycle-by-cycle schedule (moving top to bottom), and while the CDFG on the left (without the conditional) has coalesced the accesses to memory, they have turned into sequential accesses in the CDFG on the right. What was intended to be a parallel operation has been transformed into a sequential operation, simply by the inclusion of a conditional statement.

We have previously described a workaround for this issue, in which the full data set is not processed by the FPGA, but rather, most of the data set is processed by the FPGA (an amount that is evenly divisible into work-groups) and the rest is the responsibility of a traditional processor core [32]. A cleaner solution, which could be implemented by the compiler rather than requiring action by the application developer, would be for the conditional to be moved (via loop unswitching) outside of the primary execution path automatically, even if still executed on the FPGA.

IV. Discussion

We believe that all of the above issues can, ultimately, be addressed by better tools. In particular, tool flows that have a more robust understanding of the performance implications of design choices that are made during the compilation process can make better decisions as to what choices to make. The needed tool improvements come, however, in two forms, which we discuss in turn below.

A. Immediately Fixable Issues

A number of the needed tool improvements are things that are known (i.e., reasonably well described in the current literature), but are not yet incorporated into the current tool flows. An example of this type of issue is illustrated in Section III-D, in which a performance degradation (in this case, serialization of memory accesses) is clearly known to the compiler, there is a well understood transformation that can mitigate it (loop unswitching [33]), and yet the current instantiation of the compiler didn't do so. There are any number of compiler optimizations that are readily available in standard software compilers but are seeming absent from HLS compilers. However, including them in the HLS tool flow can take a considerable amount of time and effort, which needs to be spent making sure that optimizations and their interactions are realizable in hardware.

While there might be other instances of this type of issue existing in the current tools, the majority of the issues we report on here are of the second type.

B. Issues That Require Additional Research

In virtually all of the tool improvements we believe are needed to address the issues we expose above, the compiler (and associated tools) is aware of the possible implementations that can be chosen to be deployed. Analysis of the problem and listing potential solutions (within the compiler's analysis steps) isn't the real issue. The majority of the time, the compiler can reasonably understand the options. What it can't do, today, is make good choices among those options, because it poorly predicts the performance implications of each option.

In particular, what is lacking are good performance models that the compiler can use to assess the performance implications prior to instantiation and empirical measurement. Clearly, for the issues identified in Section III-C, there is ongoing research that is attempting to address this issue (e.g., see [13], [17]–[19]). For the SWI vs. MWI choice, previous work has hinted that memory access patterns might be informative [27], and for the dataflow analysis, queueing theory has been used in the past with some success [31], [34]. All of the above work, however, has various limitations, e.g., it has only been validated empirically, its breadth of utility is currently not known, etc. This is clearly an area that warrants future research. It gets even more complicated when one considers that the models we desire will also depend on the particulars of the FPGA board that is being targeted. For example, High-Bandwidth Memory (HBM) is now available with FPGAs, and the use of HBM has clear performance implications for deployed applications [35]–[37].

V. Conclusions and Future Work

While there are many ways that HLS tools have improved and come a long way since their humble beginnings, it can sometimes feel like there is still a lot to be desired when using HLS to program FPGAs in a heterogeneous computing environment. Here we have expressed a few of the pain points and issues from working in this space, but from those

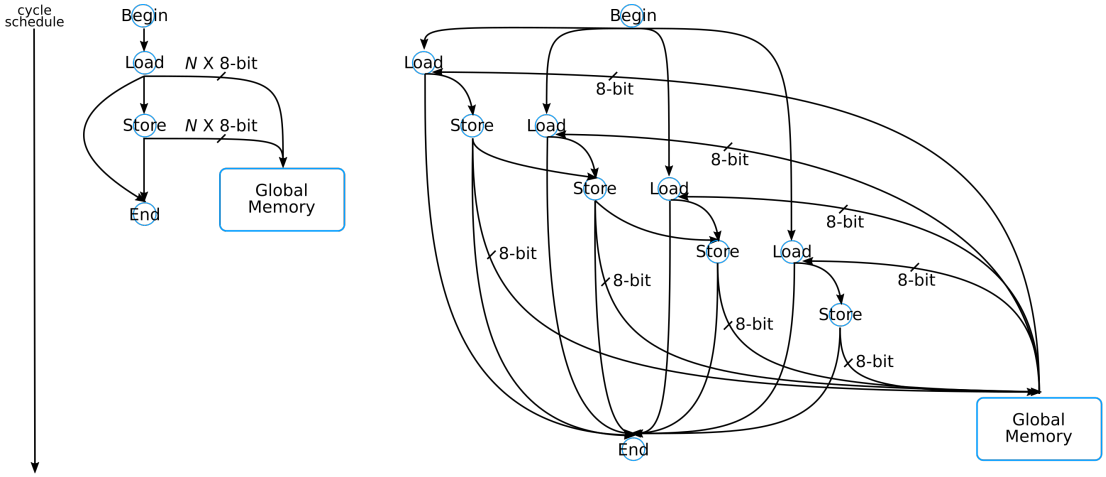

Fig. 4. Schedule of the control data flow graph when bounds checking is absent (left) and present (right) [32].

challenges arise opportunities for improvement. A good first step has been the open-sourcing of portions of the HLS toolchains by both manufacturers, allowing researchers the opportunity to tweak and improve upon an already rich tool set.

In our opinion, the way that the research community can be most helpful is in the development and validation of models that are truly predictive of performance using the information that is available at compile time. This would enable the compilers to make well informed decisions with the information that they have at their disposal.

Acknowledgements

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (available at http://energy.gov/downloads/doe-public-access-plan).

References

[1] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018.

[2] R. D. Chamberlain, "Architecturally truly diverse systems: A review," *Future Generation Computer Systems*, vol. 110, pp. 33–44, 2020.

[3] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69–76, 2008.

[4] N. Kapre and A. DeHon, "Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, Cell, and multi-core processors," in *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2009, pp. 65–72.

[5] P. Meng, M. Jacobsen, M. Kimura, V. Dergachev, T. Anantharaman, M. Requa, and R. Kastner, "Hardware accelerated alignment algorithm for optical labeled genomes," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 9, no. 3, pp. 18:1–18:21, 2016.

[6] X. Tian and K. Benkrid, "High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, pp. 26:1–26:22, 2010.

[7] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, "Design productivity of a high level synthesis compiler versus HDL," in *Proc. of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 140–147.

[8] A. Finder, J.-P. Witte, and G. Fey, "Debugging HDL designs based on functional equivalences with high-level specifications," in *Proc. of 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2013, pp. 60–65.

[9] 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*, vol. 30, no. 4, pp. 473–491, 2011.

[10] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *Proc. of Symposium on Field-programmable Custom Computing Machines (FCCM)*. IEEE, 2000, pp. 49–56.

[11] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2018.

[12] P. F. Silva, J. Bispo, and N. Paulino, "FPGAs as general-purpose accelerators for non-experts via HLS: The graph analysis example," in *Proc. of International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2021.

[13] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling software programmers to design efficient FPGA accelerators," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 4, pp. 32:1–32:27, 2022.

[14] "Intel® FPGA SDK for OpenCL™ Pro Edition: Best Practices Guide," https://www.intel.com/content/us/en/docs/programmable/683521, Intel, Apr. 2020.

[15] "Vitis High-Level Synthesis User Guide (UG1399)," https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS, 2021.

[16] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," in *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2016, pp. 409–420.

[17] A. Sanaullah, R. Patel, and M. Herbordt, "An empirically guided optimization framework for FPGA OpenCL," in *Proc. of International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 46–53.

[18] L. Ferretti, G. Ansaloni, and L. Pozzi, "Lattice-traversing design space exploration for high level synthesis," in *Proc. of 36th International Conference on Computer Design*. IEEE, 2018, pp. 210–217.

[19] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, "Correlated multi-objective multi-fidelity optimization for HLS directives design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 4, pp. 31:1–31:27, 2022.

[20] M. Shahshahani, B. Khabbazan, M. Sabri, and D. Bhatia, "A framework for modeling, optimizing, and implementing DNNs on FPGA using HLS," in *Proc. of 14th Dallas Circuits and Systems Conference (DCAS)*. IEEE, 2020.

[21] A. C. Ferreira and J. M. Cardoso, "Unfolding and folding: a new approach for code restructuring targeting HLS for FPGAs," in *Proc. of 5th International Workshop on FPGAs for Software Programmers (FSP)*. VDE Verlag, Aug. 2018, pp. 28–37.

[22] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proc. of 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 393–407.

[23] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, "EASY: Efficient Arbiter SYnthesis from multi-threaded code," in *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019, p. 142–151.

[24] A. Lotfi and R. K. Gupta, "ReHLS: Resource-aware program transformation workflow for high-level synthesis," in *Proc. of International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 533–536.

[25] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[26] A. M. Cabrera, "Domain specific computing in tightly-coupled heterogeneous systems," Ph.D. dissertation, Washington University in St. Louis, Aug. 2020.

[27] A. M. Cabrera and R. D. Chamberlain, "Designing domain specific computing systems," in *Proc. of 28th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, May 2020.

[28] S. Harris, R. D. Chamberlain, and C. Gill, "OpenCL performance on the Intel Heterogeneous Architecture Research Platform," in *Proc. of High-Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2020.

[29] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain, "DIBS: A data integration benchmark suite," in *Proc. of ACM/SPIE Int'l Conf. on Performance Engineering Companion*. ACM, Apr. 2018, pp. 25–28.

[30] A. M. Cabrera, C. Faber, K. Cepeda, R. Deber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain, "Data Integration Benchmark Suite v1," DOI:10.7936/K7NZ8715, Apr. 2018.

[31] C. J. Faber, T. Plano, S. Kodali, Z. Xiao, A. Dwaraki, J. D. Buhler, R. D. Chamberlain, and A. M. Cabrera, "Platform agnostic streaming data application performance models," in *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures (RSDHA)*. IEEE, Nov. 2021.

[32] A. M. Cabrera and R. D. Chamberlain, "Design and performance evaluation of optimizations for OpenCL FPGA kernels," in *Proc. of High-Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2020.

[33] K. D. Cooper and L. Torczon, *Engineering A Compiler*, 2nd ed. Morgan Kaufmann, 2011.

[34] J. C. Beard and R. D. Chamberlain, "Analysis of a simple approach to modeling performance for streaming data applications," in *Proc. of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, Aug. 2013, pp. 345–349.

[35] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "HBM connect: High-performance HLS interconnect for FPGA HBM," in *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2021, pp. 116–126.

[36] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *Proc. of IEEE/ACM International Conference On Computer Aided Design*. IEEE, 2021.

[37] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on FPGAs: A data analytics perspective," in *Proc. of 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.