Assisting High-Level Synthesis Improve SpMV Benchmark Through Dynamic Dependence Analysis

Rafael Garibotti, *Member, IEEE*, Brandon Reagen, *Student Member, IEEE*, Yakun Sophia Shao, *Member, IEEE*, Gu-Yeon Wei, *Member, IEEE*, and David Brooks, *Fellow, IEEE*

***Abstract*—Recent advances in high-level synthesis (HLS) have enabled an automatic means of generating register-transfer level from high-level specifications without compromising performance. HLS provides substantial improvements to productivity and is a promising solution to designing future heterogeneous chips consisting of dozens of unique IP blocks (i.e., hardware accelerators). Despite their impressive capabilities, HLS tools today are commonly used to target a small subset of workloads, i.e., ones with inordinately regular control flow and memory access patterns. The challenges of achieving high-quality hardware for irregular workloads stems from HLS relying on static analysis. Static analysis is overly conservative when dealing with non-uniform memory access and imbalanced workloads, and identifying the most appropriate parallelizing strategy. In this brief, we propose the use of dynamic analysis to generate higher quality designs using commercial HLS tools. Our evaluations show that with dynamic dependence analysis, HLS designs achieve 3.3× performance improvement for the sparse matrix-vector multiply benchmark.**

***Index Terms*—Hardware accelerators, high-level synthesis, dynamic dependence analysis, SpMV benchmark.**

I. Introduction

TO CONTINUE scaling performance despite the power wall, heterogeneous chips consisting of dozens of hardware accelerators emerge as an alternative for the semiconductor industry. While accelerators overcome the power wall and offer substantial performance gains, they introduce non-negligible engineering costs as each requires a one-off design. To realize the potential of accelerators, hardware design and build process need improvements. High-Level Synthesis (HLS) automatically generates Register-Transfer Level (RTL) from a high-level workload specification, often written in C/C++. The productivity gains offered by HLS combined with recent advances that improve the Quality of Results (QoR), make it a promising solution and key to the feasibility of future accelerator-centric architectures.

HLS claims to increase productivity by reducing hardware design efforts, making large companies including Google, NVIDIA, and Qualcomm have already used HLS in recent projects [1]. However, HLS requires designers to restructure programs, tweak source code, and learn intricate details of how HLS works to apply convoluted compiler directives to obtain good results. Looking at this trend, researchers proposed techniques to improve the quality of HLS-generated designs, regarding performance [2], area-saving designs [3], and shortening optimization time with Design Space Exploration (DSE) techniques [4]. In this regards, *non-intrusive*[1] approaches are an alternative to reduce the gap between the benefits revealed by research and the results presented by industry.

Despite the increased use and maturity of HLS tools, several limitations still exist in off-the-shelf HLS tools. One well-known difficulty involves workloads whose behavior depends on dynamic information. Specifically, HLS struggles when the number of loop iterations (i.e., the loop's *trip count*) is unknown at design time [5]. When considering these irregular workloads, HLS tools are unable to efficiently exploit inter-loop iteration parallelism (i.e., hardware loop unrolling). The use of static analysis is insufficient to produce high-performing, parallel designs without user guidance. As this form of parallelism is a major source of efficiency for accelerators, this fundamental flaw is a significant drawback when considering commercial HLS tools.

In this direction, the *main contribution* of this brief is to propose the use of dynamic dependence analysis to assist designers to use commercial HLS tools to generate efficient accelerators for irregular workloads. Our approach evaluates the Sparse Matrix-Vector Multiply (SpMV) benchmark. This benchmark depends on the input data to reveal its inner loop trip count and has relevance to scientific applications [6]. This brief demonstrates how dynamic analysis can be useful for exposing unknown cycle information and improving the performance of accelerators generated by conservative HLS tools. Furthermore, our technique supports two state-of-the-art, commercial HLS tools, showing that the benefits are not specific to a tool. Results show that with dynamic dependence analysis, the performance of SpMV improves by up to 3.3×.

Other contributions include:

- demonstrate the importance of addressing unknown loop trip counts in HLS DSE to achieve efficient designs;

Manuscript received June 21, 2018; revised July 19, 2018; accepted July 21, 2018. Date of publication July 26, 2018; date of current version September 27, 2018. This work was supported in part by PUCRS, in part by CNPq, in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and in part by DARPA under Contract HR0011-13-C-0022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. This brief was recommended by Associate Editor J. M. de la Rosa. *(Corresponding author: Rafael Garibotti.)*

R. Garibotti was with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA. He is now with the School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre 90619-900, Brazil (e-mail: rafael.garibotti@pucrs.br).

B. Reagen, G.-Y. Wei, and D. Brooks are with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA.

Y. S. Shao was with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA. He is now with the NVIDIA Research, NVIDIA, Santa Clara, CA 95050 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSII.2018.2860122

[1] *Non-intrusive* means a companion tool that assists HLS tools to generate efficient designs without modifying the source code of HLS tools.

– understand the syntax of currently available commercial HLS tools to take full advantage of the dynamic analysis phase to generate meaningful directives for HLS tools;
– provide a Source-to-Source (S2S) transformation at the C level to automatically apply the optimization opportunities extracted from the dynamic analysis phase.

II. Related Works

HLS tools rely on static compilers to discover parallelism, pipeline structure and memory access patterns from high-level descriptions of incoming workloads. The frequent absence of explicit parallelism leads commercial HLS tools to allow users set optimization parameters (e.g., pipeline initiation intervals and unrolling factors) based either on detailed application knowledge [7] or handwritten-RTL reference designs [8] to generate efficient accelerators. However, such an approach that exposes the optimization parameters to the user does not solve the performance problem of irregular workloads; it offloads the responsibility of finding the workload parallelism from HLS tools to users.

To overcome this HLS limitation, researchers start developing frameworks and companion tools [4], [9]–[11] that perform the DSE of a selected benchmark as an earlier step before invoking HLS tools. These proposed infrastructures help to understand the architectural design trade-offs introduced by hardware accelerators. Aladdin [4] and Lin-analyzer [9] construct a dynamic dependence graph directly from the C code and estimate the latency, area, and power for a variety of accelerators. HLScope+ [10] provides a fast and accurate HLS-based cycle estimate of the FPGA execution. PARADE [11] integrates the accelerator models with a cycle-accurate simulator, thus encompassing a DSE of the entire system. All these works focus on providing analysis and approximations for efficient benchmark exploration. In contrast, our approach differs from previous ones in the sense that we use DSE as an initial phase to analyze the original C code and propose optimization parameters that can be directly fit into HLS tools.

Table I shows works with similar objectives. They propose techniques that can be used in conjunction with commercial HLS tools to improve the quality of generated accelerators. Alle *et al.* [2] paved the way to use S2S transformations in C code for assisting HLS. However, they rely on the user to apply their technique. For example, they require the user inform (i.e., through directives) the loop latency to allow loop pipelining in applications with data-dependent memory access. Liu *et al.* [12] also propose an automated S2S transformation framework that generates pipelines to select dynamically among multiple schedules during runtime. However, their flow requires an HLS pass to extract scheduling information. Lattuada and Ferrandi [13] present another work that heavily relies on the user, e.g., the benchmarks have to be annotated by hand. On the contrary, our approach extracts the application parallelism through a DSE phase, showing the user the best opportunities for optimizing the accelerator. Besides, instead of requiring a user-defined optimization parameter, our flow automatically generates the optimization parameters.

Tan *et al.* [5] describe an approach that generates a dataflow pipeline architecture where multiple pipeline instances of a dynamic-bound inner loop are scheduled to execute in parallel. Dai *et al.* [14] and Josipović *et al.* [15] propose the inclusion of a hardware module, i.e., a hardware dynamic hazard resolution mechanism and elastic circuits, respectively, to resolve runtime conflicts caused by pipelining irregular loops. Unlike the three, our approach focuses on finding opportunities through available HLS directives to improve the quality of the generated accelerator without inserting any new hardware modules. Besides, our approach causes resource-saving opportunities in different parts of the design, not just in the loop pipeline.

TABLE I
Related Works That Present Frameworks and Companion Tools That Enhance the Quality of HLS-Generated Designs

Works	Year	Irregular Workloads	Non-intrusive Technique	Target Device FPGA	Target Device ASIC	Human Interaction	Number of HLS tools
[2]	2013	✓	✓	✓		High	2
[5]	2015	✓	✓	✓		High	1
[12]	2017	✓	✓	✓		High	1
[13]	2017	✓	✓	✓		High	1
[14]	2017	✓	✓	✓		High	1
[15]	2018	✓		✓		Medium	1
[16]	2016		✓	✓		Low	2
Our work	2018	✓	✓	✓	✓	Low	**2**

Lastly, the commercial Merlin Compiler [16] is an example of a companion tool that plays with optimization parameters and demand low user interaction, similar to our approach. However, they provide an environment that covers the entire system, from design optimization to accelerator communication. Unlike, our approach targets irregular workloads and is the first to show improvements for both FPGA and ASIC.

The *original* contribution of this brief is the proposition of a design flow that uses commercial HLS as a back-end tool to improve the performance of the SpMV benchmark. The key difference from other works is the presence of a DSE phase that extracts the profile information from the irregular workload and uses it to generate optimization parameters that guide the HLS tools automatically. Additionally, our approach is *the first* that addresses both FPGA and ASIC flow in HLS companion tools, as shown in Table I.

III. Methodology

Commercial HLS tools hide many architectural details from designers [10]. As a result, designers need to rely on companion tools and analytical methods to help them optimize their accelerators efficiently. Following this trend, our *non-intrusive* approach assists HLS tools to generate efficient accelerators from the SpMV benchmark through dynamic analysis. This brief extends the former [17] in the sense that dynamic analysis is applied to improve the performance of the SpMV benchmark instead of saving resources from regular workloads. This distinction made different optimization parameters gain importance, such as those that expose the *loop trip count*.

A. SpMV Benchmark

SpMV is an important kernel present in a variety of applications, including image processing [18] and text classification [19]. However, parallelizing SpMV remains a challenging problem because it deals with non-uniform memory access and imbalanced workload. Recent works aiming to accelerate SpMV on modern multi- and many-core architectures [20] and GPUs [21]. In this regards, our proposal addresses this optimization problem through the use of accelerators. To demonstrate that HLS can efficiently generate accelerators

Algorithm 1 SpMV Code Supporting CSR Format

```
1: procedure SPMV_CSR(double *val, int *col, int *row, double *vec,
   double *out, int lenght)
2:     for i = 0; i < lenght; i++ do
3:         sum = 0;
4:         for j = row[i]; j < row[i + 1]; j++ do
5:             sum = sum + val[j] * vec[col[j]];
6:         out[i] = sum;
```

for this irregular workload, we choose 10 of the most well-known matrices (5 integer type and 5 double type) from the SuiteSparse Matrix Collection [22]. Because each input results in radically different behavior, each is treated as a single benchmark. These carefully selected inputs cover *91.1%* of the sparsity behavior from a total of 2757 available matrices, which rigorously evaluate our proposed technique.

B. Irregular Loop Trip Count

To demonstrate how static analysis limits HLS-generated designs, Algorithm 1 shows a SpMV example in Compressed Sparse Row (CSR) format. The inner loop processes the non-zero elements (nnz) of the matrix in each row. By default, HLS tools would not apply any unrolling factor to this loop as the loop trip count is unknown at the static time. Thus, such HLS absence directly degrades the performance achieved. Let's consider that a designer has a detailed application knowledge, knows the maximum number of nnz (e.g., 64 per row) and applies it as a user-defined optimization parameter in a state-of-the-art HLS flow, as shown at the top of Figure 1. It guides the HLS tool to generate the highest performance accelerator. However, this HLS-generated design may still be inefficient as many rows of the matrix is not as dense as 64. For example, if the input matrix has only one row with 64 nnz, and the remaining have 2 nnz per row, such aggressive design is over-designed as most of the allocated hardware is idling during the execution. In this case, either over- or under-provision of hardware resource would lead to inefficient designs.

C. Dynamic Dependence Analysis

Dynamic dependence analysis comes to eliminate user guidance to generate efficient HLS designs. In state-of-the-art HLS flow, the user needs to learn the HLS syntax, understand the available directives and how they are more effective, besides needing detailed application knowledge to produce a meaningful optimization parameter, as shown at the top of Figure 1. On the contrary, our approach generates an optimized RTL design with minimal human interaction.

First, the dynamic data dependence graph (DDDG) is constructed using a dynamic trace, built through an LLVM instrumentation step [23]. In DDDG, the nodes represent operations (or instructions), and the edges represent dependencies between operations [24]. The scheduling heuristic consists of parsing registers and memory dependencies that yields an original DDDG that only contains the true read-after-write data dependencies. Then, we find places in the graph where control flow and data dependencies cannot be disambiguated statically, adding edges (i.e., true dependencies) at these locations, in addition to manipulate the original DDDG to consider different directives. Next, nodes are scheduled for execution when all of the nodes they depend on (i.e., their parent nodes) finish. Finally, the graph is re-balanced. Parent nodes on the critical path and nodes that access memory with true dependencies are left intact, but the remaining node operations can suffer

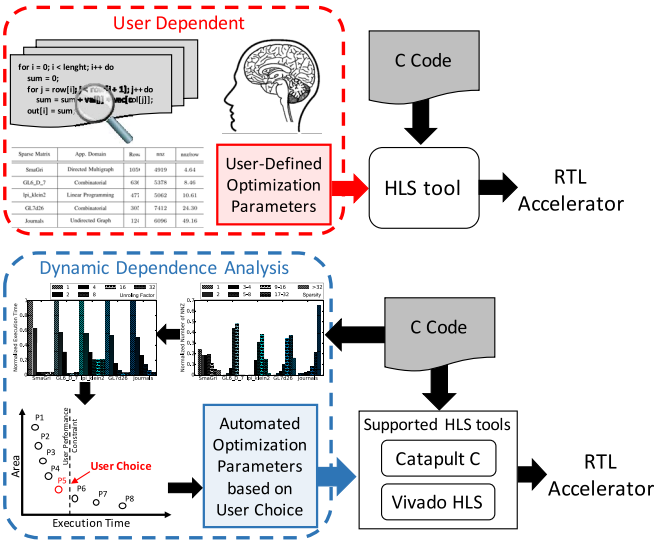

Fig. 1. Difference between state-of-the-art and proposed HLS flow, highlighting the included dynamic dependence analysis.

reordering. The output of this backward step is a scheduled DDDG that exposes the hidden parallelism of SpMV benchmark and is even more balanced to improve resource usage.

The bottom of Figure 1 shows our dynamic analysis flow. First, we use the original C code as input to perform a DSE based on a specific HLS tool, i.e., our approach automatically evaluates several DDDGs. This phase results in several design points where the best ones (e.g., from P1 to P8) are shown in a Pareto curve. At this point, the user needs to pick up one design, which is the only moment of human interaction. Here we assume that the user has a performance constraint, as shown in Figure 1. After his choice (e.g., P5), the extracted profile information is used to (*1*) apply a S2S transformation at the C level, and (*2*) to automatically generate an input configuration (i.e., optimization parameters or directives) for a specific HLS tool that reveals, for example, the unknown loop bounds that depend on the input data. This flow ensures design portability between HLS tools without human interaction, which encourages more software designers to adopt HLS tools.

D. Commercial Tools

The choice of commercial HLS tools was made based on their market coverage. We aim to improve designs for both FPGA and ASIC, and not be restricted to a single technology. In this regards, the market-leading HLS tools chosen were:

Vivado HLS [25]: This HLS tool from Xilinx is used as the solution when aiming FPGA platforms. Reported FPGA numbers are for a Virtex-7 FPGA (xq7v585t).

Catapult C [26]: To demonstrate our proposed optimization techniques' generality both across different back-ends and HLS tools, we leverage Catapult C as our ASIC HLS flow using a commercial 40nm CMOS technology library.

Both HLS tools generate RTL and test benches starting from C code, as shown in Figure 1. All synthesized RTL designs are simulated using QuestaSim 10.4c; simulation results provide performance numbers and validation for each design.[2]

[2]This brief intends to improve designs generated by commercial HLS tools and **not offer any comparison** between them. In this regards, experimental results fall into two categories, and each HLS tool evaluated a different type of sparse matrix input (i.e., integer for ASIC flow and double for FPGA flow).

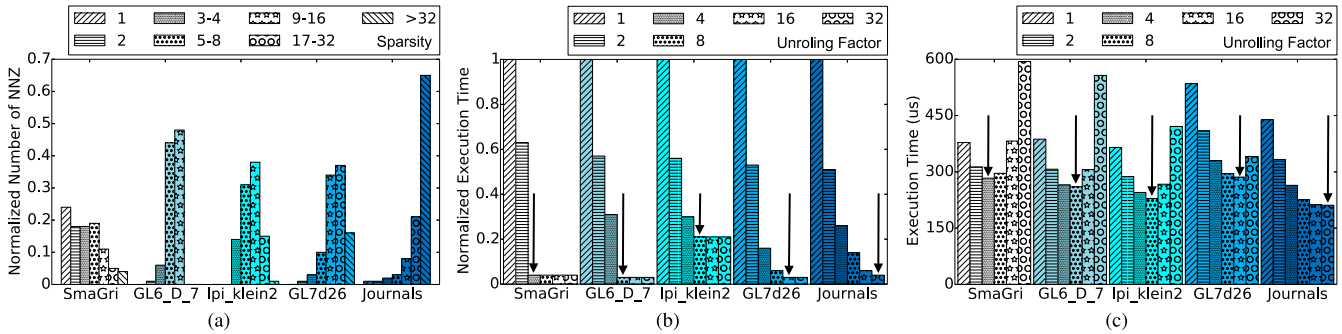

Fig. 2. Evaluation of integer input sparse matrices for SpMV: (a) input matrix distribution, (b) dynamic analysis and (c) design performance using Catapult C.

TABLE II
INTEGER INPUT SPARSE MATRICES SET

Sparse Matrix	App. Domain	Rows	nnz	nnz/row
SmaGri	Directed Multigraph	1059	4919	4.64
GL6_D_7	Combinatorial	636	5378	8.46
lpi_klein2	Linear Programming	477	5062	10.61
GL7d26	Combinatorial	305	7412	24.30
Journals	Undirected Graph	124	6096	49.16

TABLE III
DOUBLE INPUT SPARSE MATRICES SET

Sparse Matrix	App. Domain	Rows	nnz	nnz/row
bcsstm25	Structural	15439	15439	1.00
poli3	Economic	16955	37849	2.23
dw8192	Electromagnetics	8192	41746	5.10
bcsstk12	Structural	1473	17857	12.12
ex7	Fluid dynamics	1633	54543	33.40

IV. RESULTS

This section presents how dynamic dependence analysis can increase the quality of HLS-generated designs. At large, imbalanced workloads and non-uniform memory access account for the most significant source of inefficiency in HLS-generated RTL, due to the limitations of static analysis and required conservatism to implement correct hardware. An archetype of irregularity, the SpMV benchmark varies wildly across input sets. For example, Table II shows the extent of disparity among 5 integer input sets, with sparsity range from 4.64 to 49.16.

Algorithm 1 presented in Section III-B shows the source code for SpMV-CSR benchmark. Note the inner-loop in Algorithm 1, where unknown static trip counts limit the amount of parallelism that HLS can extract from the sequential specification. Unlike regular applications where increasing design area (e.g., duplicating resources) through HLS directives (e.g., loop unrolling) produce greater parallelism and better performance. The issue of optimizing SpMV-CSR using HLS alone is the irregularity increase complexity of static dependence analysis in HLS tool, which induces the generation of inefficient designs. On the other hand, dynamic analysis can address this problem by deciphering the optimal design rapidly.

Figure 2 shows the three phases performed to validate our proposed optimization technique. First, (a) profile information is extracted from the input matrix. This chart is optional and helps users to understand where are the points to be improved. Then, (b) a DSE is performed based on several scheduled DDDG. Finally, (c) a commercial HLS tool (Catapult C or Vivado HLS) automatically fed by our optimization parameters, generates the accelerator with the most effective solution.

Although phase (a) is optional, it produces insights to optimize the HLS-generated design. For example, sparsity shown in the last column in Table II corresponds to the average density of non-zero elements of the matrix. This is typical information used by designers to optimize the accelerator by unrolling the inner-loop shown in Algorithm 1 [20]. However, this information can be misleading, because depending on the distribution of the non-zero elements across the matrix, it will not match the normal curve average and this initial directive will undoubtedly generate an inefficient design.

Let's use `GL7d26` as an example. Table II indicates to use an unrolling factor of 24 for `GL7d26` based on sparsity. However, Figure 2a shows that `GL7d26` has almost half rows containing up to 16 nnz. This information reveals that a design using an unrolling factor of 16 can have *similar performance and 33% less area* compared to the same design generated with an unrolling factor of 24. `GL7d26` achieves this efficiency because the faster execution time on low-density parts compensates the large density parts.

Figure 2b illustrates the dynamic analysis phase. Each input matrix performs a DSE by sweeping the inner-loop unrolling factor from 1 to 32. Loop pipelining is set to minimize the initiation interval, and arrays are partitioned by 32, causing memory bandwidth does not limit inner-loop unrolling. Furthermore, Figure 2b highlights the implementation that saturates the normalized performance for each input matrix. The arrows are the efficient design that must be achievable (through directives) by HLS tools.

Looking at the implementation phase (Figure 2c), results show that all efficient unrolling factors exposed by our dynamic analysis (Figure 2b) match with the best performance design produced by Catapult C (Figure 2c).

Comparing the sparsity measures shown in Table II and the designs exposed by the dynamic analysis (Figure 2b), our *non-intrusive* approach improves the performance of HLS-generated accelerators. Otherwise, simplistic analysis using sparsity or other static information can lead to inefficient designs when dealing with irregular workloads. Results show that our approach achieves 2.07× better performance (`Journals`) for integer sparse matrices than our *baseline*, i.e., compared to the use of HLS tools without optimization parameters.

To demonstrate the approach's effectiveness across multiple platforms, we chose five large and complex double input sets (i.e., number of nnz larger than 15000) shown in Table III and evaluated over an FPGA device. Figure 3 shows the performance results generated by Vivado HLS, where are highlighted the hints produced by the dynamic analysis phase. Results show a performance improvement of up to 3.3× for *ex7* compared to our baseline design. Furthermore, dynamic analysis outcomes are precisely the set where are obtained

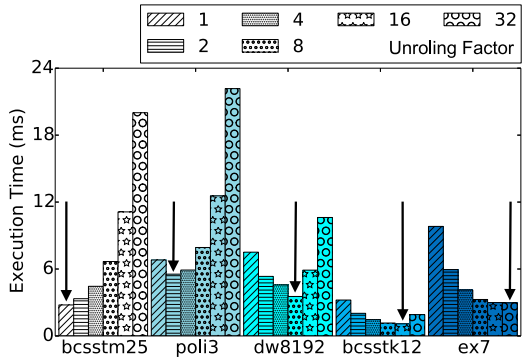

Fig. 3. SpMV design performance using Vivado HLS. These experiments help demonstrate the approach's effectiveness across multiple platforms.

the best results regarding performance, same behavior as shown in the ASIC flow. After this set, performance begins to deteriorate caused by the inefficiency to generate sizable Finite State Machines (FSM). FSM increases according to the number of Functional Units (FUs) assigned in each loop iteration, causing an unnecessary time control overhead if those FUs are underused. As our dynamic analysis does not model an entirely FSM, the point where performance saturates is the optimal design set, as previously illustrated in Figure 2b.

Although our approach shows good results for both ASIC and FPGA, their optimization parameters are entirely different. For example, FPGA presents scarcer DSP slices than LUTs and FFs. To save DSPs, the produced directives must prioritize share multipliers than other FUs. As shared FUs increase, the HLS tool introduces more MUXes which raises the number of LUTs. In ASIC design flow, on the contrary, the cost of additional MUXes are dwarfed by the cost of FUs. Therefore, the generated directives give less priority to shared FUs and more priority to increasing performance.

Our dynamic analysis approach has two caveats. First, irregular workloads have few area saving opportunities because standalone HLS tools cannot suitable parallelize the design as discussed previously. Further, relatively large FUs driven by unrolling factor dominates the area. This means that to achieve performance improvement for the SpMV benchmark, based on our approach or by handwritten modification, we have to deal with an area overhead. Second, the intrinsic delay in the overall C-to-RTL synthesis time brought by our *non-intrusive* approach. Fortunately, our dynamic analysis time accounts for less than 2% of the total synthesis time.

In short, ASIC and FPGA results show the same behavior, indicating that our *non-intrusive* approach improves the performance of the SpMV benchmark by up to 3.3× compared to standalone commercial HLS tools. However, this performance improvement shows a small increase in the synthesis time.

V. Conclusion

Performance improvement for irregular workloads challenges modern commercial HLS tools, mainly caused by static compilers that unpredict the loop's trip count. In this brief, we proposed a *non-intrusive* dynamic dependence analysis that works in conjunction with commercial HLS tools to increase the quality of accelerator designs.

Evaluation comprises a real benchmark (SpMV) through two commercial HLS tools. Results show that the proposed dynamic analysis can be used to accurately extract the parallelism profile of the SpMV benchmark, automatically producing optimization parameters that reached 3.3× more performance than unassisted HLS-generated accelerator designs.

References

[1] *HLS and RTL Low-Power: Customer Stories*. Accessed: Feb. 27, 2018. [Online]. Available: https://www.mentor.com/hls-lp/success/

[2] M. Alle *et al.*, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proc. DAC*, Austin, TX, USA, 2013, pp. 1–10.

[3] B. C. Schafer, "Allocation of FPGA DSP-macros in multi-process high-level synthesis systems," in *Proc. ASP-DAC*, Singapore, 2014, pp. 616–621.

[4] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. ISCA*, Minneapolis, MN, USA, 2014, pp. 97–108.

[5] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *Proc. ICCAD*, Austin, TX, USA, 2015, pp. 78–85.

[6] X. Liu *et al.*, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. ICS*, Eugene, OR, USA, 2013, pp. 273–282.

[7] P. Li *et al.*, "Resource-aware throughput optimization for high-level synthesis," in *Proc. FPGA*, Monterey, CA, USA, 2015, pp. 200–209.

[8] *The AutoESL AutoPilot High-Level Synthesis Tool*. Accessed: Mar. 13, 2018. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

[9] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proc. DAC*, Austin, TX, USA, 2016, pp. 136–141.

[10] Y.-K. Choi, P. Zhang, P. Li, and J. Cong, "HLScope$^+$: Fast and accurate performance estimation for FPGA HLS," in *Proc. ICCAD*, Irvine, CA, USA, 2017, pp. 691–698.

[11] J. Cong *et al.*, "PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *Proc. ICCAD*, Austin, TX, USA, 2016, pp. 380–387.

[12] J. Liu *et al.*, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published. [Online]. Available: https://ieeexplore.ieee.org/document/8207646/

[13] M. Lattuada and F. Ferrandi, "Exploiting vectorization in high level synthesis of nested irregular loops," *J. Syst. Archit.*, vol. 75, pp. 1–14, Apr. 2017.

[14] S. Dai *et al.*, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proc. FPGA*, Monterey, CA, USA, 2017, pp. 189–194.

[15] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proc. FPGA*, Monterey, CA, USA, 2018, pp. 189–194.

[16] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for HLS," in *FPGAs for Software Programmer*. Cham, Switzerland: Springer, 2016, pp. 137–163.

[17] R. Garibotti, B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Using dynamic dependence analysis to improve the quality of high-level synthesis designs," in *Proc. ISCAS*, Baltimore, MD, USA, 2017, pp. 1–4.

[18] Y. Wang, H. Yan, C. Pan, and S. Xiang, "Image editing based on sparse matrix-vector multiplication," in *Proc. ICASSP*, 2011, pp. 1317–1320.

[19] K. R. Townsend, P. Jones, and J. Zambreno, "A high performance systolic architecture for k-NN classification," in *Proc. MEMOCODE*, Lausanne, Switzerland, 2014, pp. 201–204.

[20] A. Elafrou, G. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on modern multi-and many-core processors," in *Proc. ICPP*, 2017, pp. 292–301.

[21] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. SC*, New Orleans, LA, USA, 2014, pp. 769–780.

[22] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011. [Online]. Available: https://sparse.tamu.edu/

[23] *The LLVM Compiler Infrastructure*. Accessed: Nov. 7, 2017. [Online]. Available: http://llvm.org

[24] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proc. ISCA*, 1992, pp. 342–351.

[25] *Vivado High-Level Synthesis*. Accessed: Sep: 29, 2017. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[26] *Catapult High-Level Synthesis*. Accessed: Oct. 21, 2017. [Online]. Available: http://calypto.com/en/products/catapult/overview/