SODA-OPT
System-Level Design in MLIR for HLS

Nicolas Bohm Agostini
nicolas.agostini@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

David Kaeli
kaeli@ece.neu.edu
Northeastern University
Boston, MA, USA

Antonino Tumeo
antonino.tumeo@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

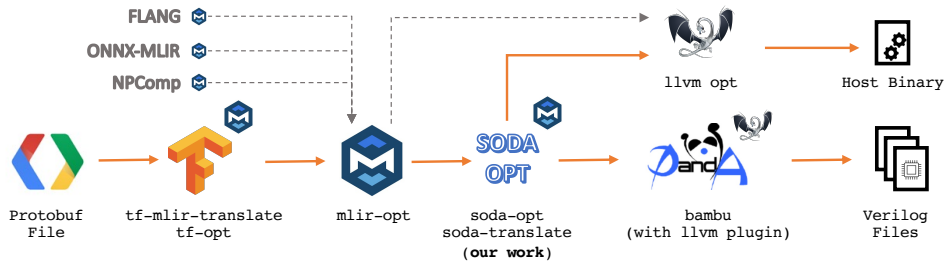

Figure 1: From High-Level applications to HLS with SODA-OPT.

1 INTRODUCTION

General purpose computing units are experiencing progressively smaller performance gains due to technology and power limitations. This directly affects High-Performance Computing clusters and Edge devices alike. Computer architecture innovations are key to keep performance steadily increasing, thus domain specific accelerators are receiving renewed interest and have shown to provide great benefits to different scientific and machine learning applications [1, 3]. High-Level-Synthesis (HLS) provides a way to quickly generate hardware descriptions for domain specific accelerators starting from applications implemented with high-level languages. However, state-of-the-art tools typically require the application to be manually translated to C/C++ and carefully annotated to improve final design performance. This cumbersome process prevents scientists and researchers from tapping into the power of HLS, as many of their applications are written in higher-level languages such as python, and require significant effort to be ported.

To overcome this challenge, we propose SODA-OPT, a front-end compiler tool that leverages the MLIR framework [4] to provide host binaries and pre-optimized accelerator code for a back-end tool of choice. SODA-OPT requires no manual code translations and presents a novel approach to automate the outlining, mapping, and generation of custom accelerators. Thanks to our selection of MLIR optimizations at the appropriate level of abstraction, our tool creates high-quality accelerated kernels for HLS. SODA-OPT provides the following contributions to the field of system-level design:

- Extensions to the MLIR framework, enabling the outlining of relevant kernels for custom accelerator generation from high-level python frameworks.
- A new frontend for HLS tools that ingest LLVM IR.
- Optimizations for HLS applied within the MLIR framework, rather than described through a C frontend.
- Generation of *optimized* LLVM IR for HLS that can be synthesized into Verilog code, achieving *superior* performance (1.36x to 1.64x speedup) with respect to LLVM IR code with no high-level optimizations.

2 SODA-OPT

SODA-OPT is a high-level compiler tool developed using MLIR [4]. MLIR is a framework that allows building reusable, extensible, and integratable compiler infrastructures. MLIR allows defining *dialects*, which are self-contained IRs that follow the rules of MLIR's meta-IR and model specific abstractions with their related operations. MLIR already provides a number of architecture dependent and independent dialects such as `linalg` (describes linear algebra operations and their optimizations), `affine` (with abstractions for polyhedral analysis and transformations), and `scf` (to describe and optimize structured control flow operations, including loops and branches). To leverage these dialects, new MLIR dialects need to define transformation and lowering passes. Multiple dialects can coexist at the same time empowering MLIR's representation power. The approach makes it easy to define dialects that represent the (domain) specific languages of high-level frameworks, thus facilitating integration with the MLIR compiler framework.

Figure 1 shows how SODA-OPT participates in an end-to-end generation flow, from a high-level framework to Hardware Description Language (HDL) code and driver binary. For the purpose of this paper, we specifically use the `tf-mlir-translate` and `tf-opt` in TensorFlow to convert a *protobuf* file to one of the SODA-OPT MLIR input dialects, but this input MLIR code could have been generated by any other high-level frameworks (e.g., NPCOMP and ONNX). After partitioning and optimizing the code, SODA-OPT generates two types of LLVM IR as output: one that LLVM `opt` tool can use to compile the host application (including the accelerator driver calls) and one, which includes HLS-specific optimizations, to be ingested by an HLS tool to generate the accelerators. In this paper we use the Bambu HLS tool [2], which explicitly decouples the compilers' frontends that parse and optimize the input specifications (typically GCC or CLANG for C/C++) from the actual HLS process. Hence, Bambu directly takes as input the frontend-optimized IRs and performs the hardware generation steps by converting LLVM IR (or GCC GIMPLE) to its own internal IR.

3 EVALUATION

In this section we evaluate our approach by performing HLS of MLIR kernels produced by SODA-OPT. All experiments are carried out with Bambu version 0.9.7, which supports the clang-10 frontend. We evaluate the kernels execution latency simulating the designs using Verilator. Finally, we synthesize the designs, targeting a Xilinx XC7Z020-2CLG484I device, using Vivado *v*2020.1 and specifying a desired frequency of 100MHz.

Benefits of high-level optimizations - High-level optimizations performed at the MLIR level can have a direct impact on the final performance of the generated Verilog code, by enabling further optimization opportunities within the HLS flow. For example, task level parallelism information allows exposing instruction and data level parallelism, which HLS can efficiently exploit for better scheduling decisions.

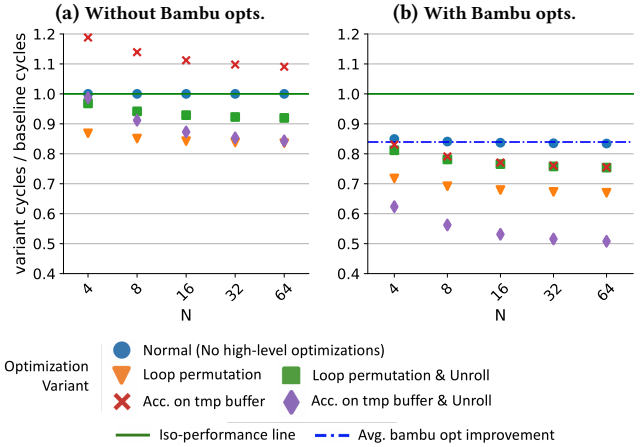

Figure 2: Relative difference between execution times of Matmul kernels with different MLIR and Bambu optimizations.

To support our claim, we synthesize Matmul kernels with different input sizes while allowing SODA-OPT to apply different high-level optimizations: *permuting loop iterations*, *unrolling innermost loops*, and *allocating a temporary buffer on which to perform accumulations*. Figure 2a compares the execution latency of the optimized variants against the non-optimized variant (Normal). The solid line in bold represents the iso-performance line. Points under this line represent faster execution than the non-optimized baseline. In Figure 2a, no additional Bambu optimizations have been applied.

In order to further showcase the contributions of SODA-OPT, we performed the same experiment using speculative SDC scheduling (SSDCS [5]), one of the low-level optimizations offered by Bambu (`-s` command-line option). Results are presented in Figure 2b. Using Bambu's SSDCS in isolation is beneficial, as it results in faster execution times; this is shown by the non-optimized (Normal) variants of Figure 2b with circle markers. The average contribution of SSDCS without other high-level optimizations (18% clock cycles reduction) is marked by the dashed line. Points under the dashed line represent designs with faster execution time than without any high-level optimization. In these designs, SODA-OPT exposes code structures that Bambu is able to better schedule, providing additional speedup that was not achievable otherwise.

The results in Figure 2 lead to several conclusions. First, high-level optimizations enabled by SODA-OPT have a direct impact on the final execution latency of the simulated designs. Second, carefully selecting individual or a combination of these optimizations is paramount for improving runtime performance. Finally, without high-level optimizations applied, Bambu with SSDCS provides a small speedup, but SODA-OPT can enable up to 1.64*x* better performance when the best combination of high-level optimizations is used along with SSDCS.

Neural Network case study - To demonstrate SODA-OPT capabilities on full dataflow applications, we automatically translated a LeNet model trained in TensorFlow to the `linalg` dialect and employed SODA-OPT to search, outline, and optimize different regions of the network, generating different specialized accelerators. Table 1 lists LeNet's first layers and shows the number of cycles to execute them as SODA-OPT is used to fuse these layers in unique accelerators. Accelerator runtime, represented by merged cells in the table, start from individual operators and move to progressively coarser granularities, including the operator fusion strategy performed in TensorFlow. We observe, in this case, that outlining these three layers in a single accelerator provides best results.

Table 1: Runtime results in cycles for the Lenet model (first 3 layers) with different outlining strategies.

		Outlining Strategy		
Layer type	**Layer params**	**Individual Ops**	**Fused (TF kernels)**	**Fused (Coarser)**
Conv2D	5x5, 6, padded	2,423,374	2,388,122	2,443,073
Activation	Relu	39,230		
AvgPool2D	2x2, 2x2 stride	88,244	88,224	
Total		2,551,848	2,476,366	2,443,073

4 CONCLUSIONS

This paper presents SODA-OPT, a hardware/software outliner and optimizer tool for users of high-level programming frameworks to generate systems composed of automatically synthesized hardware accelerators. SODA-OPT leverages the MLIR compiler framework to automatically perform kernel outlining and HLS-specific optimizations. We extended MLIR with the `soda` dialect, and assembled pass pipelines that have shown increase in performance of key linear algebra kernels predominant in current dataflow applications. Differently from other approaches, SODA-OPT does not need to generate C/C++ code with specific HLS annotations to guide the underlining HLS tool, and it is not limited by a library of hand-optimized parametrized RTL modules to compose the accelerators.

Experiments on key linear algebra kernels show that our high-level optimizations expose code structures that result in up to 1.6x faster runtime performance when paired with advanced HLS tool scheduling optimizations. Finally, we demonstrate the practical applicability of the framework by showing automatic generation of accelerators for Deep-Neural-Network operations fused at different granularities.

REFERENCES

[1] Jeremy Appleyard and Scott Yokim. 2017. Programming Tensor Cores in CUDA 9. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/
[2] Fabrizio Ferrandi. 2021. Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. https://panda.dei.polimi.it/?page_id=31
[3] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *ISCA*. 1–12.
[4] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. 2–14.
[5] M. Lattuada and F. Ferrandi. 2015. Code transformations based on speculative SDC scheduling. In *ICCAD*. 71–77.