An MLIR-based Compiler Flow for System-Level Design and Hardware Acceleration

Nicolas Bohm Agostini*
PNNL
Richland, WA, USA

Serena Curzel†
Politecnico di Milano
Milan, Italy

Vinay Amatya
PNNL
Richland, WA, USA

Cheng Tan
PNNL
Richland, WA, USA

Marco Minutoli
PNNL
Richland, WA, USA

Vito Giovanni Castellana
PNNL
Richland, WA, USA

Joseph Manzano
PNNL
Richland, WA, USA

David Kaeli
Northeastern University
Boston, MA, USA

Antonino Tumeo
PNNL
Richland, WA, USA

ABSTRACT

The generation of custom hardware accelerators for applications implemented within high-level productive programming frameworks requires considerable manual effort. To automate this process, we introduce SODA-OPT, a compiler tool that extends the MLIR infrastructure. SODA-OPT automatically searches, outlines, tiles, and pre-optimizes relevant code regions to generate high-quality accelerators through high-level synthesis. SODA-OPT can support any high-level programming framework and domain-specific language that interface with the MLIR infrastructure. By leveraging MLIR, SODA-OPT solves compiler optimization problems with specialized abstractions. Backend synthesis tools connect to SODA-OPT through progressive intermediate representation lowerings. SODA-OPT interfaces to a design space exploration engine to identify the combination of compiler optimization passes and options that provides high-performance generated designs for different backends and targets. We demonstrate the practical applicability of the compilation flow by exploring the automatic generation of accelerators for deep neural networks operators outlined at arbitrary granularity and by combining outlining with tiling on large convolution layers. Experimental results with kernels from the PolyBench benchmark show that our high-level optimizations improve execution delays of synthesized accelerators up to 60x. We also show that for the selected kernels, our solution outperforms the current of state-of-the art in more than 70% of the benchmarks and provides better average speedup in 55% of them. SODA-OPT is an open source project available at https://gitlab.pnnl.gov/sodalite/soda-opt.

CCS CONCEPTS

• Software and its engineering → Compilers; **• Hardware → Emerging languages and compilers**; *HW-SW codesign.*

KEYWORDS

MLIR, High-Level Optimizations, HLS, Compilers

1 INTRODUCTION

High-level programming frameworks for scientific computing, data science, and machine learning (among others) interface with high-level compilers to optimize and map computational graphs onto various heterogeneous hardware substrates, including graphic processing units and tensor processing units. When performance, power efficiency, and latency are critical, the same applications dramatically benefit from custom systems containing specialized hardware accelerators implemented as either Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). For example, modern edge devices such as sensor networks [5, 11], embedded processing systems in scientific instruments [12], and autonomous cyber-physical systems [16] typically operate under tight latency constraints: they need to identify valuable information in the acquired data, process it, react to changes in the environment, and perform autonomous control in real-time.

The design of hardware accelerators is typically a lengthy process, requiring to identify commonly used computational patterns and develop correspondent low-level modules in hardware description languages (HDLs) such as Verilog, VHDL, or Chisel. On the other hand, high-level algorithms and programming frameworks evolve at a quick pace, complicating the development of highly specialized systems. To alleviate these burdens, it is paramount to generate custom accelerators directly from the high-level frameworks, with limited or no manual intervention from low-level hardware designers. Several existing tools perform a source-to-source translation from the functional description of the algorithms in the high-level framework into an imperative general purpose programming language (usually C), specifically annotated with OpenCL or custom pragmas for high-level synthesis (HLS) tools [18, 21, 23, 25, 27]. However, translation to an annotated imperative language restricts the accelerator generation to a specific HLS backend tool, target device, and technology. Moreover, it may cause a loss of information that had been encoded in the input computational graph (e.g., structural hierarchy, order, or parallelism) and cannot be expressed with general purpose imperative code.

This paper presents SODA-OPT, a novel solution to address these challenges. SODA-OPT is a compiler framework that searches, outlines, optimizes, dispatches, and accelerates computational kernels from high-level programming frameworks, targeting systems containing FPGAs or ASICs, without resorting to imperative language

*Also with Northeastern University.
†Also with Pacific Northwest National Laboratory.

templates. SODA-OPT leverages and extends the Multi-Level Intermediate Representation (MLIR) framework [19], which enables to define IRs at multiple levels of abstraction, making the best use of different information exposed at each step of the compilation process. SODA-OPT performs hardware/software partitioning, automatically outlining kernels from applications developed in high-level programming frameworks that interface with MLIR. It applies software- and hardware-oriented transformations, preparing the outlined kernels for synthesis through progressive IR lowerings. SODA-OPT interfaces with HLS in the backend through a lower-level compiler IR (which is specific to an HLS tool) and, by leveraging runtime libraries, it can generate code to control the synthesized hardware accelerators from a host processor. SODA-OPT employs a design space exploration (DSE) engine to identify effective compiler passes and options to optimize the IR for hardware synthesis.

In summary, SODA-OPT makes the following contributions:

- it leverages the MLIR framework to enable automatic outlining of relevant kernels from applications written in high-level software frameworks. Backends supported by SODA-OPT can utilize this feature to automatically define the generated accelerator scope;
- it provides an automated flow that transforms the high-level outlined kernels into fully custom hardware implementations through HLS. SODA-OPT can target two different state of the art HLS backends enabling synthesis for FPGAs and ASICs;
- it offers a configurable pipeline of high-level optimization passes that improve the performance of the generated accelerators;
- it connects to a DSE engine to tune the optimization pipeline to different applications, tools, and design constraints.

The paper proceeds as follows. Section 2 compares and discusses the related work. Section 3 describes SODA-OPT, the outlining and tiling, the code transformations, and the DSE engine. Section 4 presents an experimental evaluation on typical linear algebra and neural network kernels. Finally, Section 5 concludes the paper.

2 RELATED WORK

Different works facilitate the generation of domain-specialized architectures starting from high-level programming frameworks. Some of them map input applications on generic accelerator templates such as data processing units, matrix multiplication units, and systolic arrays (e.g. [24, 27–29] for deep neural networks). Other tools, including ours, translate functional descriptions from high-level programming frameworks into specialized accelerators through a compiler IR, or HLS-specific C/C++ code. This second approach allows to explore compiler optimizations and code transformations to increase the quality of the synthesized accelerators. Table 1 compares the most relevant state of the art tools that translate generic high-level applications into hardware accelerators.

ScaleHLS [26] performs high-level optimizations in MLIR and generates annotated C/C++ for Vivado HLS. Differently from SODA-OPT, it does not deal with hardware/software partitioning and system-level concerns. ScaleHLS provides several valuable features (e.g., a quality of results estimator and automated design space exploration); however, its optimizations strictly depend on the capabilities of the HLS backend tool, with features such array partitioning and loop pipelining that are only effective if the backend tool applies them. SODA-OPT, instead, performs all optimizations directly on the compiler IR, so that they are equally available to any supported backend tool.

Table 1: Qualitative comparison of tools translating high-level applications into hardware accelerators.

Tool	Input	Backend tools	F1	F2	F3	F4	F5
SODA-OPT	MLIR (Affine or Higher)	Bambu, Vitis HLS	✓	✓	✓	✓	✓
ScaleHLS [26]	MLIR (Affine or Higher)	Vivado HLS	✓	✗	✓	✓	✗
CIRCT HLS [7]	MLIR (Affine or Higher)	CIRCT	✓	✗	✗	✗	✓
FROST [8]	Halide, Tiramisu	Vivado HLS	✗	✗	✗	✓	✗
Hot & Spicy [23]	Annotated Python	SDSoC	✗	✗	✗	✓	✗
HeteroCL [18]	Annotated C, OpenCL	Intel or Vitis HLS	✗	✗	✗	✓	✗
HPVM2FPGA [10]	HeteroC++	Intel HLS	✓	✓	✓	✓	✗
Phism [30]	C/C++	Vitis HLS	✓	✗	✗	✓	✗

F1: Optimizations Without Manual Annotations; **F2**: Automatic Partitioning of Host and Kernel; **F3**: DSE; **F4**: FPGA Support; **F5**: ASIC Support.

The CIRCT [9] project is leveraging MLIR to provide a hardware compiler infrastructure. CIRCT-HLS [7] is implementing the initial HLS support that takes advantage of CIRCT hardware abstractions and transformations. Our outlining mechanisms, optimization passes, and overall system design methodology can easily integrate with CIRCT once conversion passes become available. Once this happens, SODA-OPT will support CIRCT-HLS as another backend target by exploiting the modularity of the MLIR framework.

FROST [8] accelerates applications in Halide, a domain-specific language (DSL) for image processing, through the Xilinx SDAccel framework. Hot & Spicy [23] translates annotated Python into an accelerator and its runtime Python wrappers. HeteroCL [18] defines a new DSL in Python that decouples algorithmic description from hardware optimizations, generating the accelerator and the host code driving it. All of these tools generate hardware by producing annotated C code for a specific HLS backend tool.

HPVM2FPGA [10] promises to synthesize efficient RTL descriptions for FPGAs by leveraging a "hardware-agnostic" subset of C++ (HeteroC++), a custom compiler intermediate representation (HPVM), and a DSE engine that identifies effective compiler optimizations. Phism [30] proposes to exploit MLIR to bridge the gap between polyhedral tools and HLS. The authors intend to use Polygeist [20] as a frontend for C/C++ programs, and to exploit MLIR `affine` abstraction to implement polyhedral optimizations that can benefit their target backend. Phism aims to support Vitis HLS through its LLVM frontend, thus eliminating the need to rewrite annotated C/C++ code for HLS. To the best of our knowledge, both HPVM2FPGA and Phism have not yet released fully functional implementations.

SODA-OPT leverages the MLIR framework, consequently it does not require to rewrite the input application in a different language: any high-level software framework that can be translated to an in-tree MLIR dialect can serve as input to our tool. Moreover, SODA-OPT does not generate C/C++ specifically annotated for one backend HLS tool, rather it applies architecture-independent high-level compiler transformations with the aim of restructuring and optimizing the IR itself (which is finally translated to LLVM IR before HLS).

Determining effective combinations of compiler passes and parameters that improve the generated code is a well known problem, also in the field of HLS [15]. The typical approaches look at implementing DSE with a variety of methods, including machine learning [2]. SODA-OPT also integrates a DSE engine to select the most appropriate combination of high-level optimizations.

3 SODA-OPT

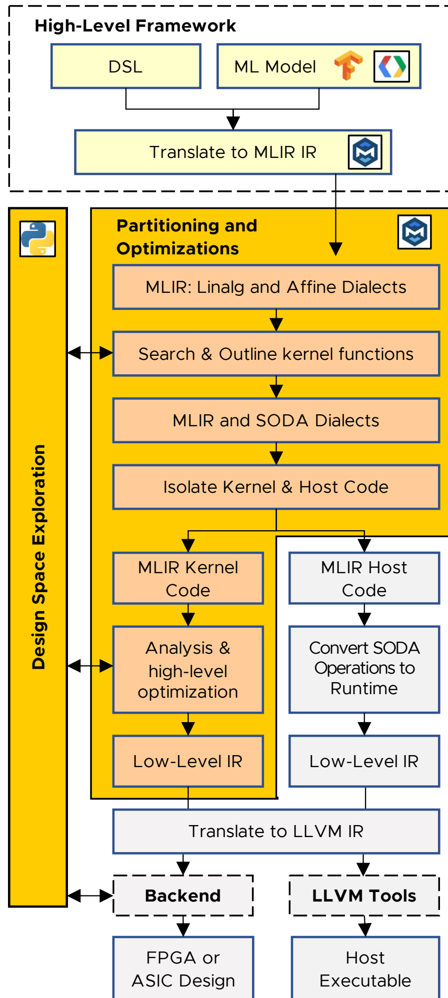


Figure 1: The compilation pipeline of SODA-OPT, from high-level applications to custom host code and accelerator design.

SODA-OPT is a high-level compiler tool that can partition applications from high-level software frameworks into an orchestrating host program and custom hardware accelerators (Figure 1). In this work we describe in detail the system level design and how SODA-OPT selects and delivers the optimizations providing better inputs for the target backends. To provide these features, SODA-OPT extends the MLIR framework [19]. MLIR allows developers to build reusable, extensible, and modular compiler infrastructures by defining *dialects* (i.e., self-contained intermediate representations - IRs -, also called abstractions) that can model code at different levels of abstraction. MLIR tackles both translation and optimization concerns. Dialects can be architecture dependent or independent and can co-exist in the same program to achieve specific representation semantics. Examples of available architecture-independent dialects include: `linalg`- providing abstractions to describe linear algebra operations and their optimizations; `affine`- providing abstractions for polyhedral analysis and transformations; `scf`- providing abstractions to describe and optimize structured control flow operations such as for and while loops; `cf`- providing lower-level control flow operations such as branches and switches; and the `llvm` dialect - which represents LLVM IR operations in the MLIR IR.

MLIR developers can define new dialects and transformations to represent domain-specific languages and abstractions, enabling high-level software frameworks to use MLIR-based optimizations, and progressively lower high-level dialects into low-level dialects. TensorFlow, FLANG, ONNX-MLIR, Polygeist, and Torch-MLIR are examples of projects providing an entry point to the MLIR framework. In Section 4.1, for example, we use the `tf-mlir-translate` and `tf-opt` tools to convert a TensorFlow *protobuf* file containing a pre-trained neural network into an MLIR representation. However, SODA-OPT approach can be used for any other high-level description that can be translated to MLIR.

The SODA-OPT analysis and transformation passes ingest MLIR inputs, including operations from the `linalg`, `affine`, or `scf` dialects, identify key code regions, and outline them into separate MLIR `modules` (Section 3.1). Afterwards, the host program and the outlined kernels follow two parallel compilation flows. The host `module` is lowered into an LLVM IR file that includes runtime calls to control the custom accelerators synthesized from the kernel's `modules`. The kernel's `modules` are progressively lowered and optimized through several MLIR dialects (`linalg` → `affine` → `scf` → `cf` → `llvm`), and finally translated to LLVM IR.

The kernel LLVM IR is specifically optimized and restructured to facilitate hardware synthesis, as it will later be ingested by an HLS tool to generate an RTL accelerator design. While it is optimized for synthesis, it contains neither transformations nor annotations specific to any HLS tool, and thus can interface to synthesizers that accept standard LLVM IR as input. In this work, we use the Bambu HLS tool [14], which accepts LLVM IR as input and generates synthesizable Verilog for a variety of target devices (including Xilinx FPGAs, Intel FPGAs, and ASICs with the OpenROAD flow and FreePDK 45nm library). We also use Vitis HLS from Xilinx bypassing its frontend and feeding the LLVM IR to the synthesis backend; in this case, we also need to "downgrade" the IRs, as Xilinx tools work with an old LLVM version that is not compatible with MLIR.

Alongside SODA-OPT, we introduce a Design Space Exploration (DSE) engine that can explore the selection of compiler passes and their options in the front-end compilation pipeline and backend HLS engine (Section 3.2). We leverage the DSE engine to identify a high-level optimization pipeline that results in the generation of faster accelerators.

3.1 SODA Dialect and Code Transformations

SODA-OPT transforms an input application into host program and accelerator through four steps: i) Search, ii) Outline, iv) Dispatch,

Table 2: The soda dialect operations

Operation	Semantic
`soda.launch`	Marks code regions to be outlined and extracted into kernel modules and functions.
`soda.terminator`	Indicates the end of a region to be outlined.
`soda.launch_func`	Calls outlined functions and is replaced by an accelerator launch mechanism in the host code.
`soda.module`	Holds a list of outlined kernel functions to be optimized and later become a unique accelerator.
`soda.func`	Defines an outlined kernel function, including its interface.
`soda.return`	Indicates the end of an outlined function.

and v) Accelerate. To support this process, we introduced the `soda` dialect and a set of transformation passes that operate on `soda`. The `soda` dialect follows similar design principles as provided in the `gpu` dialect. Table 2 describes the `soda` dialect operations. Additionally, SODA-OPT includes Optimization passes that aim at improving the performance of the generated accelerators.

Search passes - The multiple levels of abstraction provided in MLIR allow us to maintain the semantics from the original specification (e.g., ML operators, linear algebra operators, etc.) throughout the compilation pipeline, simplifying the task of automatically identifying specific kernel regions that are well suited for acceleration. In fact, the SODA-OPT compiler can identify these regions by matching key patterns in the earliest stages of the compilation process. Search passes insert `soda.launch` operations around the code to be outlined. The block inside `soda.launch` is then completed by injecting a `soda.terminator` operation for termination detection.

Our search targets are operations belonging to the set of linear algebra functions or affine structures with arithmetic operations inside, with a focus on patterns that are common in neural networks models and scientific applications. However, more patterns and operations can be included as MLIR dialects and supported applications evolve.

Once a matching pattern is found, we follow the steps of Figure 2 to decide whether outlining the whole kernel region or selecting only a loop tile. In fact, the set of optimizations that will be applied by SODA-OPT after outlining trade off area for performance by increasing parallelism, so we need to ensure as early as possible that the accelerator generated from the optimized IR fits on the hardware resources of the target. If our estimates indicate that the number of operations is too high to outline the entire region, SODA-OPT offers two different tiling options: 1) *marking* the region to outline and *tiling* inside the marked region, or 2) *tiling* and *marking* the tiled region for outlining. The decision largely depends on the available device memory, as option 1) requires that the entire inputs and outputs reside in the accelerator memory. Option 2) handles a smaller portion of inputs and outputs, but it is only suitable to systems where the host and accelerator memory are tightly coupled, as it incurs in redundant host-accelerator memory transfers. Option 1) does not change the amount of computation inside the outlined region, but tiling introduces new loop nests which will drastically change the effect of downstream loop transformations.

In both cases SODA-OPT considers characteristics of the code region of interest and of the hardware target to select a tiling factor

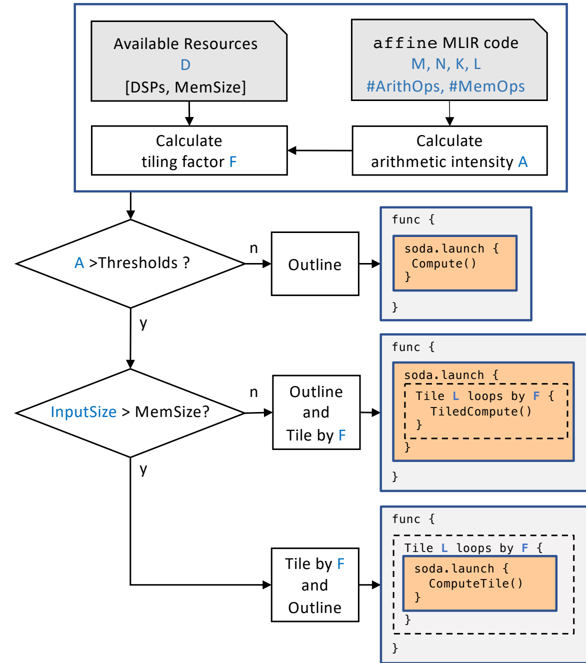


Figure 2: Outlining and tiling strategies under different target constraints.

F. Specifically, SODA-OPT approximates the arithmetic intensity of the kernel region (A) by counting the number of nested loops (L) and their iterations, and the number of arithmetic operations in the innermost loop. For example, a `linalg.matmul` operation contains 3 nested loops with M, N, K iterations, and 2 arithmetic operations inside the innermost loop (a `multiply` and an `add`), which results in an approximate arithmetic intensity of $A = 2 \cdot M \cdot N \cdot K$. The amount of available resources (D) for an ASIC target is estimated based on the allocated chip area and on the area consumed by multiply and add units (characterized post-synthesis); on FPGA we take into account the number of available Digital Signal Processing blocks (DSPs, used to implement fused multiply-adds) as D. With all these information, we calculate the tiling factor as follows:

$$Tiling\ Factor:\ F = \frac{A}{D^L}$$

Outline passes - This step extracts the code marked within `soda.launch` blocks into a separate module, including all the functions that are part of the call graph of the offloaded operator or block of code. The outlining process, described in Algorithm 1, analyzes use-def chains of values inside the `soda.launch` region to generate the interface of the top kernel functions. The process then transforms the marked code so that each `memref` descriptor allocated outside the `soda.launch` region, but referenced inside the region, becomes an argument of the kernel. Constant values, instead, are pulled inside the outlined region. The process then proceeds by moving each operation within `soda.launch` into the generated `soda.module`/`soda.func`, finally replacing the `soda.launch` block with `soda.launch_func` calling the outlined kernel.

Algorithm 1 Outlining soda.launch functions

```
1: to_dup ← ∅                                   ▷ Set of values to duplicate
2: to_arg ← ∅                                   ▷ Set of values to become argument
3: for ∀v ∈ soda.launch region do
4:     for ∀u ∈ use_def_chain of v & ∉ soda.launch region do
5:         if isConstantDeclaration(u) then
6:             to_dup ← u
7:         end if
8:         if isParentFuncArgument(u) then
9:             to_arg ← u
10:        end if
11:        if isMemrefDescriptor(u) then
12:            to_arg ← u
13:        end if
14:    end for
15: end for
16: dst_func ← createKrnlFuncWithArgs(to_arg)
17: for ∀v ∈ soda.launch region do
18:     cloneIntoFunction(v, dst_func)
19:     for ∀u ∈ to_dup do
20:         duplicateAndReplaceUse(u)
21:     end for
22: end for
23: replaceByFuncCall(soda.launch, dst_func)
```

Dispatch and Accelerate passes - These passes separate the host and kernel code. They generate a host MLIR file without the definitions of the outlined kernel code, and the kernel MLIR file without references to the host code. These files serve as input to their respective targets (host compiler or HLS tool).

Optimize passes - After Outlining the kernel, the compilation flow starts its Optimization step. In the current implementation, SODA-OPT exploits several dialect-specific optimization passes from the `affine` and `scf` dialects, aiming at restructuring the final low-level IR to make it more suitable for hardware synthesis. The optimization pipeline can easily include future contributions from new MLIR dialects or new optimization passes, and it can be fine-tuned to target any HLS tool that supports the LLVM IR as input. Figure 3 summarizes the transformations in the pipeline that are beneficial to the HLS process. SODA-OPT optimization pipeline enables efficient scheduling of operations in the compute-intensive region, resulting in significant performance gains.

Table 3: Effects of Clang -O2 vs SODA-OPT optimizations on the LLVM IR for a GEMM kernel (M,N,K=2) synthesized with Bambu.

Operations	Clang -O2	SODA-OPT
Loads	20	12
Stores	12	4
Multiplications	20	16
Additions	8	8
Alias analysis?	No	Yes
Scheduling Task?	Harder	Easier
Runtime (Cycles)	103	16
DSPs	8	16
LUTs	1929	2537

In fact, HLS tools can synthesize faster accelerators with the LLVM IR code generated by SODA-OPT than with a standard LLVM IR input optimized by standard C/C++ frontends such as Clang. As an example, in Table 3 we show the effect of the SODA-OPT optimization pipeline on a small (M,N,K=2) generic matrix-multiply kernel. The baseline version is derived from a C implementation and optimized with `clang -O2`; the second one is derived from SODA-OPT and undergoes the optimizations described above. Some code structures can improve the resource allocation and the instruction scheduling steps of the HLS process. SODA-OPT optimizations expose code structures that present simpler dependency chains, exhibit load-compute-store patterns, and have a low number of redundant instructions. When synthesized with the opensource Bambu HLS compiler, the SODA-OPT version in Table 3 runs 6.4x faster than the optimized C version. This trend is evident throughout our experiments in Section 4.2.

The individual passes and parameters included in the SODA-OPT pipeline for Bambu were tuned with a wide set of benchmarks using the DSE engine implemented in Python. Details on this step are discussed in Section 3.2.

3.2 Design Space Exploration

The design space exploration (DSE) engine is implemented in Python. Its main role is to identify the most effective combination of compiler passes, but it can also tune additional parameters, such as inputs to specific passes (unroll factor, tile sizes, buffer sizes to promote malloc to alloca operations, target design delay, target number of memory ports, etc.). The DSE engine interfaces with both SODA-OPT and the HLS backend tool to select passes through Python bindings and collect synthesis results, respectively.

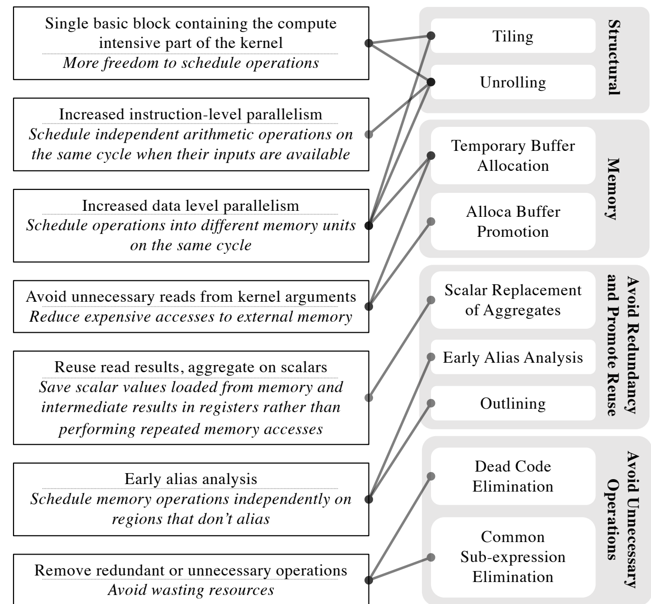
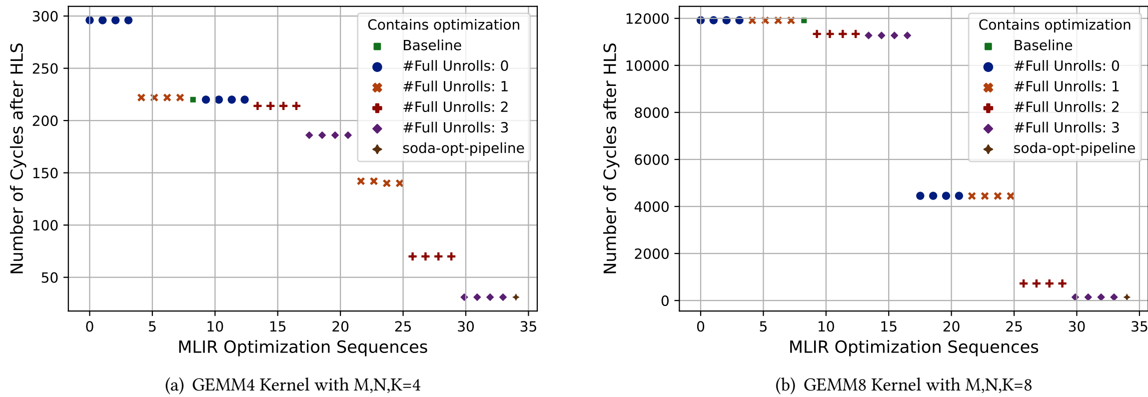


Figure 3: High-level optimizations, *benefits to HLS*, and implemented passes in SODA-OPT.



(a) GEMM4 Kernel with M,N,K=4

(b) GEMM8 Kernel with M,N,K=8

Figure 4: Performance results for different MLIR optimization sequences including no high-level optimization (Baseline) and the derived optimization pipeline for Bambu.

To determine a generic and reusable high-level optimization pipeline that improves performance of synthesized kernels with the different HLS backends, we performed DSE while considering sequences of high-level transformation passes including mandatory passes to lower MLIR code to LLVM IR, and optional passes that can improve the HLS process. The optional passes considered here (Figure 3) transform MLIR at the `affine` and `scf` abstractions. We tested all generated sequences with different linear algebra kernels and observed that they follow similar trends. Below, we provide further insights by analysing the results of one of these kernels while tuning the optimization pipeline for the Bambu tool.

Figure 4 shows the effects of different sequences of optimization passes on two Generalized Matrix Multiplication (GEMM) kernels of different dimensions. Figure 4(a) shows an implementation with input matrices of size 4x4 (M,N,K=4) and Figure 4(b) shows an implementation with matrices of size 8x8 (M,N,K=8). We present the execution delay (in clock cycles) of the synthesized designs for each of the 34 unique sequences of optimization passes, generated combining the passes in Figure 3.

In most instances, the sequence of passes that exhibit the worst performance are the ones without loop unrolling. This is expected, as loop unrolling is a key optimization for HLS, since it allows increasing instruction-level parallelism. However, in GEMM8 shown in Figure 4(b), some sequences with full unroll optimization perform as bad as sequences without any unrolling. In these cases, the HLS tool cannot exploit the increased instruction-level parallelism because the kernel reads and stores all intermediate results from/to external memory, and parallel loads and stores are limited by the number of available memory ports.

Figure 4(b) shows a considerable increase in performance after sequence number 17, where unrolling becomes more effective. The speedup is correlated to the selection of the *temporary buffer allocation+alloca buffer promotion* optimization passes, which enable pre-fetching of the kernel inputs allocated in external memory into a pre-allocated register file. GEMM4, in Figure 4(a), also exhibits this effect after sequence 17, but observes a smaller speedup since the smaller kernel has fewer opportunities for reuse.

Some sequences of optimization passes exhibit the same performance. For example, *CSE+DCE* and *SRoA* are optimization passes also executed by the backend, and if the backend is successful in transforming the code, these passes can be removed from the high-level optimization sequence. However, including these passes among the high-level transformations can provide improvements for other kernels. For this reason, we kept these optimizations in the final sequence of optimization passes for Bambu and Vitis, referred as *SODA-OPT Optimization Pipeline*, in the following experiments.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate SODA-OPT by generating accelerators for different applications and algorithms. Our experiments show the effectiveness of SODA-OPT search/outline algorithm, and the benefits of the optimization pipeline for quality-of-results (QoR) on both FPGAs and ASICs. For HLS backends, we use Vitis HLS 2021.1 with default optimizations enabled, and Bambu 0.9.7 with default options and `-O2` optimization level. We generate custom FPGA designs targeting a Xilinx Virtex 7 xc7vx690t-ffg1930-3 device at 100MHz through both Vitis HLS and Bambu, and custom ASIC designs at 500MHz through Bambu using the OpenROAD flow [17] with the FreePDK 45nm cell library. Accelerators derived from our framework undergo all the steps through place-and-route in a completely automated process.

4.1 Search and Outline Strategies

We use a classical machine learning application (LeNet convolutional neural network) to illustrate the Search and Outline capabilities of SODA-OPT. We automatically translated a LeNet model trained in TensorFlow to the `linalg` MLIR dialect, and employed SODA-OPT to search, outline, and accelerate different regions of the network. Table 4 lists the layers that compose the model, and displays the number of cycles obtained with different outlining

Table 4: Execution times of LeNet with different outlining strategies. Merged cells correspond to a single accelerator.

<table>
<tr><th colspan="5">FPGA Target: Xilinx xc7vx690t-ffg1930-3 @ 100MHz</th></tr>
<tr><th>Layer type</th><th>Individual Operations</th><th>Fused as TF kernels</th><th>Fused Coarser</th><th>Full Network</th></tr>
<tr><td>Conv2D - 6x5x5 filters</td><td>2,423,374</td><td rowspan="2">2,462,602</td><td rowspan="3">2,526,225</td><td rowspan="12">6,806,682</td></tr>
<tr><td>ReLU</td><td>39,230</td></tr>
<tr><td>AvgPooling2D - 2x2</td><td>88,244</td><td>88,244</td></tr>
<tr><td>Conv2D - 16x5x5 filters</td><td>4,917,022</td><td rowspan="2">4,917,022</td><td rowspan="3">5,175,970</td></tr>
<tr><td>ReLu</td><td>12,912</td></tr>
<tr><td>AvgPooling2D - 2x2</td><td>30,092</td><td>30,092</td></tr>
<tr><td>Dense - 120 units</td><td>1,010,522</td><td rowspan="2">1,011,602</td><td rowspan="2">1,011,602</td></tr>
<tr><td>ReLU</td><td>962</td></tr>
<tr><td>Dense - 84 units</td><td>213,446</td><td rowspan="2">214,202</td><td rowspan="2">214,202</td></tr>
<tr><td>ReLU</td><td>674</td></tr>
<tr><td>Dense - 10 units</td><td>17,841</td><td>17,841</td><td rowspan="2">19,084</td></tr>
<tr><td>Softmax</td><td>454</td><td>454</td></tr>
<tr><td>Total</td><td>8,754,773</td><td>8,742,059</td><td>8,947,083</td><td>6,806,682</td></tr>
</table>

strategies: starting from the generation of one accelerator for each individual operator, we moved to progressively coarser granularities (fusing operators as usually performed in TensorFlow, and outlining the entire network to generate a single accelerator).

SODA-OPT allows the exploration of custom accelerators for fused operators at any level of granularity. As the results show, outlining the entire network provides the greatest opportunity for additional optimizations, with 1.26x speedup compared to the outlining of individual operations. Just combining parametrized accelerators for each operator would not allow to perform such exploration and obtaining the same improvements in QoR of the generated designs.

The decision to tile loops and outline the inner tile to create a reusable accelerator can also significantly improve performance, as it allows to adapt the size of the design to computational intensity and availability of resources. We show this effect by tiling convolution kernels from common convolutional neural networks (Table 5). The generated accelerator is invoked multiple times to run a convolutional layer, and our selected sizes ensure the accelerators can also be reused across different layers in the bigger models (35, 14, and 46 convolutional layers in MobileNetV2, ResNet-18, and ResNet-50, respectively). Table 5 shows results of the tile and outline strategy for an ASIC target with and without applying the SODA-OPT optimization pipeline established in Section 3.2.

4.2 Effects of SODA-OPT on QoR

We run experiments with linear algebra kernels from PolyBench [22] to assess the effect of the SODA-OPT high-level optimization pipeline on the performance of automatically generated accelerators. PolyBench offers a collection of microkernels containing static control parts and floating point arithmetic operations on multi-dimensional tensors, representative of computation patterns implemented in high-level programming frameworks for scientific computing and machine learning [1, 3, 4, 13]. Several prior studies have employed these benchmarks to quantify the effectiveness of MLIR transformations for CPUs, GPUs, and FPGAs [6, 20, 26].

We measured the performance (execution delay in clock cycles) obtained by synthesizing PolyBench kernels with different HLS tools and different high-level optimizers. In particular, we compare SODA-OPT against ScaleHLS [26], a recently published MLIR-based tool. However, our analysis needs to take into account that ScaleHLS can only optimize annotated C++ code for Vivado HLS, and that different HLS tools generate accelerators with different performance even when the high-level optimizations applied are the same. Figure 5 reports the average speedup observed with different HLS tools (Vivado HLS and Vitis HLS) without manual optimizations, with ScaleHLS [26], and with SODA-OPT preparing the code for Bambu or Vitis HLS synthesis. Standalone HLS tools and ScaleHLS start from plain C specifications. We normalized FPGA (F) and ASIC (A) results with respect to standalone Bambu synthesis targeting FPGA and ASIC devices, respectively. This approach allows visually comparing results from each toolchain (represented by a different bar) within the same benchmark. Higher is better. Each bar in Figure 5 is the average speedup of four different kernel sizes (2, 4, 8, 16), where sizes refer to all dimensions of input and output tensors.

From these results we observe that SODA-OPT outperforms ScaleHLS, in 6 of the 11 benchmarks. More importantly, SODA-OPT provides a speedup for all examples when compared to Bambu targeting FPGA or ASIC synthesis. SODA-OPT also provides, on average, faster execution in 10 of the 11 benchmarks when optimizing the high-level code for the Vitis HLS backend. Additional analysis of individual accelerators shows that our solution outperforms ScaleHLS in 70% of the 44 synthesized designs. Further investigation of the individual kernel's performance indicates that standalone Vitis HLS, without manual optimizations, outperforms ScaleHLS and Vivado HLS in 11 and 40 out of the 44 generated accelerators, respectively, showing that Vitis HLS provides a superior baseline among the Xilinx HLS tools.

We observe a wide range of speedups across the different kernels. Thanks to SODA-OPT, some individual accelerators reach speedups of 60x with respect to their baseline. On average the introduction of our high-level optimization pipeline results in designs 25x faster than Bambu HLS for FPGA, 18x faster than Bambu HLS for ASIC and 2x faster than Vitis HLS. The current pipeline of optimization passes is well suited for kernels that present dot product or matrix multiplication structures (e.g., *doitgen*, *gemm*, *two_mm*, *three_mm*). Other kernels (*syr2k*, *syrk*, and *symm*) exhibit small or no performance improvements: the reason is that these kernels include inner loop bounds dependent on the induction variables of the outer loops. This sometimes prevents loop unrolling and, in general, limits the effectiveness of the current SODA-OPT optimization pipeline.

Table 6 reports resource utilization of the accelerators synthesized on FPGAs with Bambu. Together with the number of Look-Up Tables (LUTs) consumed by baseline designs, we report the LUT overhead introduced by the SODA-OPT optimization pipeline, and the ratio between performance speedups and LUTs utilization increases. Values of the latter equal to or above *one* (1.00) indicate that the increase in performance is higher than the increase in resource utilization, providing an intuitive method to evaluate the increase in QoR. On average, we see that for a 2x increase in LUTs utilization SODA-OPT can provide a speedup of 10x.

5 CONCLUSION

This paper presents SODA-OPT, a compiler-based tool that generates systems composed of automatically synthesized hardware

Table 5: Synthesis results for Conv2D tiles @500MHz.

Operation and Kernel Information			Runtime Information				Synthesis Results for FreePDK 45nm		
Target Model	Tile size	MLIR Opt.	Cycles	Runtime (s)	GFLOPS	Speedup	Area(um^2)	Power(W)	GFLOPS/W
LeNet	1,1,14,8,1,1,1	No Opt.	1,809	3.62E-06	0.061	Baseline	38,375	0.01760	3.52
LeNet	1,1,14,8,1,1,1	SODA-OPT	125	250.00E-09	0.896	15.25	673,558	0.27400	3.27
MobileNetV2	1,7,7,4,1,1,1	No Opt.	3,194	6.39E-06	0.061	Baseline	26,811	0.01050	5.84
MobileNetV2	1,7,7,4,1,1,1	SODA-OPT	225	450.00E-09	0.871	14.20	752,356	0.38200	2.28
ResNet18,50	1,1,1,64,1,1,1	No Opt.	963	1.93E-06	0.066	Baseline	15,994	0.00533	12.47
ResNet18,50	1,1,1,64,1,1,1	SODA-OPT	99	198.00E-09	0.646	9.73	413,867	0.20200	3.20

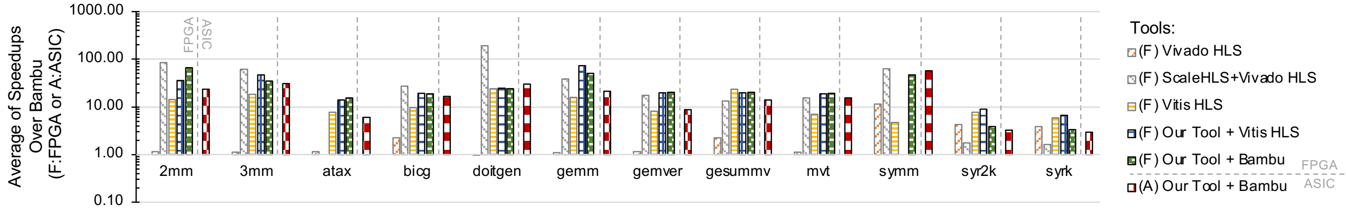


Figure 5: Average of speedups provided by different HLS tools targeting FPGA (F) or ASIC (A) synthesis. FPGA results are normalized to standalone Bambu targeting FPGA synthesis. ASIC results are normalized to standalone Bambu targeting ASIC synthesis.

Table 6: Performance gains and area overhead on the PolyBench kernels synthesized with Bambu. *: Tiling required. Resource utilization only represents the tile.

	FPGA Target: Xilinx xc7vx690t-ffg1930-3 @ 100MHz							
Opt. Strategy	LUTs Ratio				Speedup / LUTs Ratio			
Kernel Size	2	4	8	16	2	4	8	16
doitgen	1.81	3.14	22.75	26.98*	6.09	4.76	1.48	1.38
three_mm	1.68	4.07	30.05*	114.84*	5.95	10.71	1.46	0.38
two_mm	1.09	3.40	20.95	67.35*	6.45	9.41	5.46	1.66
gemm	1.32	3.51	22.22	59.55*	4.89	8.09	4.14	1.26
bicg	1.68	3.55	5.92	40.52*	3.34	2.96	4.36	0.81
mvt	1.57	1.47	5.23	7.78	3.63	9.39	4.91	4.20
gesummv	1.56	3.38	5.27	35.45	3.29	3.85	5.05	1.00
gemver	1.26	2.20	5.48	4.24	3.40	5.63	5.79	7.57
atax	1.04	2.51	5.61	8.80	3.49	3.50	3.48	3.36
syr2k	1.59	1.26	1.98	23.19	3.28	2.07	1.72	0.17
syrk	1.48	0.97	1.52	14.08	3.33	2.53	1.99	0.20
symm	1.51	0.75	0.97	0.98	3.21	1.85	1.72	1.93

$LUTs\ Ratio = \frac{\#LUT_{SODA-OPT}}{\#LUT_{NoHighLevelOpts.}}$ | $Speedup = \frac{\#CYCLES_{NoHighLevelOpts.}}{\#CYCLES_{SODA-OPT}}$

accelerators starting from applications written in high-level programming frameworks. SODA-OPT extends the MLIR compiler framework to automatically perform kernel outlining, introducing the soda dialect. It executes HLS-specific optimizations during progressive lowerings and restructures the IRs by exploiting MLIR specialized dialects. SODA-OPT interfaces with HLS tools to generate the accelerators by providing pre-optimized intermediate representations. SODA-OPT provides an extensible high-level compiler flow that can accept inputs from any high-level programming framework whose language is translated into an MLIR dialect, and can easily integrate new domain- and hardware- specific transformations through existing and new MLIR dialects. SODA-OPT adopts a DSE engine to identify the combinations of compiler optimization passes and parameters that provide the best QoR for the generated accelerators. These features enable SODA-OPT to outperform the state-of-the-art in many examples. We performed outlining and optimization for various kernels, targeting both FPGA and ASIC devices. We demonstrate the effectiveness of SODA-OPT automated outlining and tiling strategy by generating accelerators for deep neural networks operators fused at arbitrary granularity. Our pipeline of optimization passes, evaluated on the PolyBench kernels, provides a significant increase in QoR for the generated accelerators (performance and resource utilization).

ACKNOWLEDGMENTS

This research was partially supported by the Software Defined Accelerators for Data Analytics (SO(DA)^2) project in the Data Model Convergence Initiative under the PNNL's Laboratory Directed Research and Development (LDRD) program and the Defense Advanced Research Projects Agency's (DARPA) Real-Time Machine Learning (RTML) program.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI'16)*. USENIX, Savannah, GA, USA, 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (2018), 42 pages. https://doi.org/10.1145/3197978

[3] Junjie Bai, Fang Lu, Ke Zhang, and ONNX Community. 2019. ONNX: Open Neural Network Exchange. https://onnx.ai/

[4] Gary Bradski, Adrian Kaehler, and Others. 2018. OpenCV. https://opencv.org/opencv-4-0

[5] Yuan Cao, Hongkang Lu, and Tao Wen. 2019. A safety computer system based on multi-sensor data processing. *Sensors* 19, 4 (2019), 818–834. https://doi.org/10.3390/s19040818

[6] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive Raising in Multi-level IR. In *CGO*. IEEE, Seoul, South Korea, 15–26. https://doi.org/10.1109/CGO51591.2021.9370332

[7] CIRCT-HLS Developers. 2022. CIRCT-HLS. https://github.com/circt-hls/circt-hls

[8] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. 2018. A Unified Backend for Targeting FPGAs from DSLs. In *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP'18)*. IEEE, Milan, Italy, 1–8. https://doi.org/10.1109/ASAP.2018.8445108

[9] CIRCT Developers. 2020. "CIRCT" / Circuit IR Compilers and Tools. https://github.com/llvm/circt

[10] Adel Ejjeh, Aaron Councilman, Akash Kothari, Maria Kotsifakou, Leon Medvinsky, Abdul Rafae Noor, Hashim Sharif, Yifan Zhao, Sarita Adve, Sasa Misailovic, and Vikram Adve. 2022. HPVM: Hardware-Agnostic Programming for Heterogeneous Parallel Systems. *IEEE Micro* Early Access (2022), 1–12. https://doi.org/10.1109/MM.2022.3186547

[11] Iker Elorza, Iker Arrizabalaga, Aritz Zubizarreta, Héctor Martín-Aguilar, Aron Pujana-Arrese, and Carlos Calleja. 2021. A Sensor Data Processing Algorithm for Wind Turbine Hydraulic Pitch System Diagnosis. *Energies* 15, 1 (2021), 33. https://doi.org/10.3390/en15010033

[12] T Goji Etoh, Kazuhiro Shimonomura, Anh Quang Nguyen, Kosei Takehara, Yoshinari Kamakura, Paul Goetschalckx, Luc Haspeslagh, Piet De Moor, Vu Truong Son Dao, Hoang Dung Nguyen, et al. 2018. A 100 Mfps image sensor for biological applications. In *High-Speed Biomedical Imaging and Spectroscopy III: Toward Big Data Instrumentation and Management (BiOS'18)*. International Society for Optics and Photonics, SPIE, San Francisco, California, United States, 9–18. https://doi.org/10.1117/12.2288861

[13] Facebook. 2017. PyTorch: tensors and dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/

[14] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *58th ACM/IEEE Design Automation Conference (DAC'21)*. IEEE, San Francisco, CA, USA, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[15] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Dean Brown, and Jason Helge Anderson. 2013. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'13)*. IEEE Computer Society, Seattle, WA, USA, 89–96. https://doi.org/10.1109/FCCM.2013.50

[16] Rasheed Hussain and Sherali Zeadally. 2018. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1275–1313. https://doi.org/10.1109/COMST.2018.2869360

[17] Andrew B Kahng and Tom Spyrou. 2021. The OpenROAD Project: Unleashing Hardware Innovation. In *Government Microcircuit Applications and Critical Technology Conference (GOMAC'21)*. GOMACTech, Virtual, 1–6. https://vlsicad.ucsd.edu/Publications/Conferences/383/c383.pdf

[18] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. ACM, Seaside, CA, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[19] 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*. IEEE, Seoul, South Korea, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[20] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *30th International Conference on Parallel Architectures and Compilation Techniques) (PACT'21)*. IEEE, Atlanta, GA, USA, 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[21] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, et al. 2020. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *ML: Science and Technology* 2, 1 (2020), 1–14. https://doi.org/10.1088/2632-2153/aba042

[22] Pouchet, Louis-Noël and others. 2021. PolyBench/C 4.2.1. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[23] S. Skalicky, J. Monson, A. Schmidt, and M. French. 2018. Hot & Spicy: Improving Productivity with Python and HLS for FPGAs. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCMM'18)*. IEEE, Boulder, CO, USA, 85–92. https://doi.org/10.1109/FCCM.2018.00022

[24] TVM Developers. 2020. VTA: Deep learning accelerator stack. docs.tvm.ai/vta

[25] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized NN Inference. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, Monterey, California, USA, 65–74. https://doi.org/10.1145/3020078.3021744

[26] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. IEEE, Seoul, South Korea, 741–755. https://doi.org/10.1109/HPCA53966.2022.00060

[27] Hanchen Ye, Xiaofan Zhang, Zhize Huang, Gengsheng Chen, and Deming Chen. 2020. HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation. In *57th ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, San Francisco, CA, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218684

[28] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, San Diego, CA, USA, 1–8. https://doi.org/10.1145/3240765.3240801

[29] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-Mei Hwu, and Deming Chen. 2020. DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'20)*. IEEE, San Diego, CA, USA, 1–9. https://doi.org/10.1145/3240765.3240801

[30] Ruizhe Zhao and Jianyi Cheng. 2021. Phism: Polyhedral High-Level Synthesis in MLIR. In *LATTE virtual workshop (LATTE'21)*. arXiv, Virtual, 1–3. https://doi.org/10.48550/arXiv.2103.15103