

# Code Transpilation for Hardware Accelerators

Yuto Nishida, Sahil Bhatia, Shadaj Laddad, Hasan Genc, Yakun Sophia Shao, Alvin Cheung

UC Berkeley

{sahilbhatia, shadaj}@berkeley.edu

**Abstract**—DSLs and hardware accelerators have proven to be very effective in optimizing computationally expensive workloads. In this paper, we propose a solution to the challenge of manually rewriting legacy or unoptimized code in domain-specific languages and hardware accelerators. We introduce an approach that integrates two open-source tools: *Metalift*, a code translation framework, and *Gemmini*, a DNN accelerator generator. The integration of these two tools offers significant benefits, including simplified workflows for developers to run legacy code on Gemmini generated accelerators and a streamlined programming stack for Gemmini that reduces the effort required to add new instructions. This paper provides details on this integration and its potential to simplify and optimize computationally expensive workloads.

## I. INTRODUCTION

In recent years, the software industry has witnessed a trend where Domain-Specific Languages (DSLs) have increasingly become part of the existing workflows. These specialized programming languages offer high-level abstractions that are tailored to solving particular problems or expressing computations within specific domains. Some examples of DSLs are Numpy for matrix operations, Tensorflow for deep learning, and Halide for image processing, among others. On the other hand, in the hardware industry accelerators have become increasingly popular. These special-purpose execution engines are optimized to perform specific tasks, and by offloading certain parts of computations to them while performing the remainder on a general-purpose CPU, applications can achieve performance optimizations.

One common outcome of these trends in both industries is the development of frameworks that facilitate the adoption of these specialized tools. By offering high-level abstractions and automating many low-level implementation details, these frameworks have made it easier for users to take advantage of the benefits of DSLs and accelerators. One example of such a framework is **Metalift** [1], which enables users to build custom compilers for translating code written in general-purpose languages to DSLs. Leveraging program synthesis, Metalift frees developers from writing syntax-driven rules that can be error-prone and difficult to specify. On the hardware side, **Gemmini** [2] is an open-source framework for building custom DNN accelerators. It allows developers to generate accelerators and customize them end-to-end, from architectural templates (such as spatial arrays and scratchpads) to programming support (including ONNX format and low-level C++ APIs) to system support (such as microcontrollers and server-class CPUs).

At present, Gemmini offers two front-ends for running DNN workloads: high-level push-button support for executing workloads from ONNX files and low-level C/C++ APIs of

its instruction set for running workloads on the generated accelerators. However, running legacy or unoptimized code, such as Fortran-based scientific computing kernels or C++-based image processing kernels on Gemmini could pose a challenge for developers. To do so, the developer would either need to manually translate these kernels or write a syntax-driven compiler to perform the translation, both of which can be error-prone and time-consuming. Combining Metalift with Gemmini can be a powerful approach for automating this translation from general-purpose languages to Gemmini’s ISA. Additionally, Metalift’s search-based technique for translation can be guided by the performance of the translated code on the generated hardware. This allows for the search to potentially find hardware parameters that are optimal for running a particular kernel.

This paper presents our initial work on combining Metalift and Gemmini. By integrating these two frameworks, we aim to enable more efficient computations in various domains and significantly simplify the workflow for developers. Additionally, since both frameworks are open-source it allows for greater flexibility in integrating the two frameworks which could lead to improvements and optimizations that benefit both the communities.

## II. APPROACH

We apply Metalift to map array processing code written in standard Python or C++ (using standard lists and loops, rather than specialized library functions) to operations that can be accelerated using Gemmini. Instead of having to hand-write pattern matching logic to translate common patterns to their accelerated equivalents, which requires significant engineering and yields brittle results, we can focus on specifying the formal semantics of the Gemmini operators and let Metalift search for an appropriate mapping.

The verified lifting framework that Metalift enables involves three key steps: defining a grammar to search for the target language, specifying the semantics of the target language, and analyzing the behavior of the source program we want to match the behavior of. The specifications provided to Metalift are written using a custom intermediate representation that resembles the APIs of SMT solvers such as Z3. To analyze the source program, Metalift provides several front-ends that perform static analysis over Python or LLVM inputs and generates a symbolic expression over the inputs in the same Metalift IR.

Metalift performs the software synthesis procedure with an iterative algorithm which enumerates candidate programs from the provided grammar, and then verifies the correctness

```

1 vector<int> program(vector<int> data){
2   vector<int> result;
3   for (int i = 0; i < data.size() - 1; i++)
4     result.push_back(data[i] + data[i + 1]);
5   return result;
6 }

```

(a) User-Provided Sequential C++ Code

```

1 def gemmini_conv(data, kernel, stride):
2   if length(data) < len(kernel):
3     return []
4
5   return prepend(
6     dot_product(data, kernel),
7     gemmini_conv(data[stride:], kernel, stride))

```

(c) Semantics of the convolution operator provided to Metalift

Initial Condition	$Inv(i = 0, result = \{\}, data)$
Preservation	$Inv(i, result, data) \wedge (i < size(data)) \rightarrow Inv(i + 1, result.push\_back(data[i] + data[i + 1]), data)$
Termination	$Inv(i, result, data) \wedge \neg (i < size(data)) \rightarrow PS(result, data)$

(b) Verification conditions for the source code.

```

1 Synthesized:
2 program = gemmini_conv(data, kernel=[1,1], stride=1)
3
4 Invariant:
5 (i ≥ 0) && (i < (length(data))) &&
   out = gemmini_conv(data[:i+1], kernel=[1,1], stride=1)

```

(d) Program lifted to use Gemmini operators

Fig. 1: The four steps of the Metalift pipeline, adapted for the domain of translating C++ kernels to Gemmini.

of the candidate by querying an SMT solver. To perform this verification step for code involving loops with dynamic bounds, Metalift synthesizes additional *loop invariants*, which make it possible to reason about the result of the program for any number of iterations.

Consider the input code in Figure 1a, which computes the sums of values within a sliding window. Readers familiar with primitives for tensor accelerators may recognize this as a convolution, but this is not explicit in the code and the developer may not have this same insight. Our goal is to use verified lifting to automatically synthesize an provably equivalent function that uses an accelerated convolution operator.

Our first step using Metalift is to specify the grammar to search for the target program. We model each accelerated operator as a function that takes input tensors and configuration parameters as inputs and produces a new tensor as an output. Then, we can build up the grammar by starting with the inputs to the programs as leaves and layering compositions of the available Gemmini operators.

The other half of specifying the target language is to provide formal models of each operator, so that we can verify synthesized candidates against the source program using an SMT solver. In Metalift, the target language is specified by providing implementations in terms of the Metalift IR, which is later passed to the solver. In our implementation, we include specifications for core Gemmini operators such as matrix multiplication and convolutions (Figure 1c). But beyond these core specifications, Metalift is given no additional information about when these operators should be used—it has to discover appropriate mappings by searching the grammar.

### III. RESULTS AND FUTURE WORK

Metalift can translate the code in fig. 1a to generate the equivalent code in Gemmini’s ISA (Figure 1d) in <1min and this compiler was implemented in <100 LOC. Our initial prototype encodes the semantics of matrix multiplication and

convolution operator from Gemmini’s ISA. Using our initial compiler, we will translate the cloverleaf benchmarks in [3] and the image processing kernels introduced in [4].

At present, Metalift can perform code translation for a fixed Gemmini generated accelerator. Our plan is to integrate the search for hardware parameters into the synthesis procedure of Metalift’s. The integrated workflow would involve the user providing the source code to be translated and potential Gemmini’s generator parameters, such as the scratchpad size or the systolic array dimensions. In addition to searching for equivalent code in Gemmini’s ISA, Metalift can also search for the most optimal accelerator parameters for executing the given source program. The search procedure would be guided by the performance of the translated code on the accelerator. However, running every possible candidate on the accelerator may be too expensive and slow down the translation process. Therefore, we need to develop a reliable proxy cost model that can be evaluated quickly and guide Metalift’s search procedure.

A unified framework with Metalift and Gemmini could open up new research directions in improving the hardware software co-design. Overall, this combination offers an exciting opportunity for automatically translating legacy code and achieving high-performance execution on custom hardware.

### REFERENCES

- [1] “Metalift,” <https://github.com/metalift/metalift>, 2023.
- [2] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, “Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration,” in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.
- [3] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama, “Verified lifting of stencil computations,” *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 711–726, 2016.
- [4] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil, “Automatically translating image processing libraries to halide,” *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–13, 2019.