ISA Mapper: A Compute and Hardware Agnostic Deep Learning Compiler

Matthew Sotoudeh*
University of California, Davis

Anand Venkat
Intel Corporation

Michael Anderson
Intel Corporation

Evangelos Georganas
Intel Corporation

Alexander Heinecke
Intel Corporation

Jason Knight[†]
Intel Corporation

ABSTRACT

Domain specific accelerators present new challenges for code generation onto novel instruction sets, communication fabrics, and memory architectures. We introduce a shared intermediate representation to describe both deep learning programs and hardware capabilities, then formulate and apply *instruction mapping* to determine how a computation can be performed on a hardware system. Our *scheduler* chooses a specific mapping and determines data movement and computation order.

With this system, we demonstrate automated extraction of matrix multiplication kernels from recent deep learning operations. We demonstrate 2-5X better performance on GEMM and GRU execution versus state-of-the-art on new hardware and up to 85% of state-of-the-art performance on existing hardware.

ACM Reference Format:

Matthew Sotoudeh, Anand Venkat, Michael Anderson, Evangelos Georganas, Alexander Heinecke, and Jason Knight. 2019. ISA Mapper: A Compute and Hardware Agnostic Deep Learning Compiler. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3310273.3321559>

1 INTRODUCTION

Modern computer programs are typically written in high-level programming languages that abstract away details of individual hardware architectures. To that end, a large body of work exists in the field of *compilation* techniques, the process of automatically translating high-level program descriptions into the low-level instruction set understood by the hardware. Crucially, there are usually many (infinite) mappings from high-level program to low-level executable, and the compiler is charged with finding a close to optimal (with respect to size, execution time, or energy use) low-level executable that preserves the computational semantics of the high-level program.

*Work completed during an internship at Intel.

[†]Corresponding author: jason.knight@intel.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CF '19, April 30-May 2, 2019, Alghero, Italy

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6685-4/19/05...\$15.00

<https://doi.org/10.1145/3310273.3321559>

Historically, existing work has focused on general-purpose compilers such as GCC and LLVM [11] that compile input programs written in high-level languages like C to a compute device following the traditional Harvard or Von Neumann architectures composed of caches in a memory hierarchy and single CPU operating on scalar or vector values.

Unfortunately, in domains such as dense linear algebra, despite decades of compiler work it is still widely accepted that hand written and optimized assembly surpasses the performance of code output by today's standard compilers. Additionally, the demand for performance in these domains is driving further hardware innovation which only exacerbates these existing problems in this domain.

Some of the issues exhibited by general-purpose compilers such as GCC or LLVM are:

- (1) They assume that the code is being compiled for a single, synchronous compute unit or multiple devices with particular forms of parallelism and shared memory capabilities.
- (2) They assume a particular form of memory hierarchy, with a large main memory accessible by the CPU and a cache hierarchy on the chip that is managed completely by hardware.
- (3) They assume a scalar or vector instruction set, and are unable to map programs onto broader types of instructions like matrix multiplication.

In response to these issues, a number of *domain-specific* deep learning compilers have been proposed.

In particular, TVM [3] builds on Halide [14] to allow users to express computational kernels in a high level description language (essentially a “Tensor IR”) and then expose a device specific set of scheduling primitives for users to describe loop blocking, memory prefetching, and other considerations to lower the computational description to efficient code in LLVM (or similar) intermediate representation (IR). The requirement that users manually schedule kernels was partially addressed in the AutoTVM [4] extension by leveraging learned cost models, but this still requires good initial schedules. Furthermore, current support for *pattern matching* larger blocks of compute such as a matrix multiplication from the finer grained IR of TVM is inflexible given the memory hierarchy and instruction set assumptions made in TVM.

PlaidML [17] has a similar “Tensor IR” (Tile), but uses parameterized cost models and careful memory hierarchy analysis to generate automatically-scheduled kernels. However, PlaidML also makes strong assumptions about the underlying memory hierarchy and instruction sets supported by the individual compute units, making it difficult to directly apply to new hardware architectures.

The Intel® nGraph™ [7] library and TensorFlow™¹'s XLA [12] compiler both take as input a coarser, graph based representation of a deep learning computation and then allow each hardware backend to choose how to lower the coarse grained deep learning computations to machine code. As an example, nGraph™'s CPU backend leverages the Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) and Eigen for much of the execution capabilities whereas the XLA CPU backend lowers operators to kernel library calls or LLVM IR. Both of these systems are capable of leveraging the compilation approach described in this paper with the appropriate lowering pass to the Tensor IR described in Section 2.

Less recent research has been published in the field of mapping computations onto complex instruction set (CISC) architectures, primarily using directed acyclic graphs to describe the computation. [1, 10], for example, describe both the program and the supported instructions as graphs similar to SSA graphs used in LLVM, then perform a pattern matching step to find isomorphisms between the two. However, such approaches in general fall under the general-purpose language assumptions made by compilers like GCC and LLVM, limiting applicability to deep learning as they cannot effectively analyze and exploit loop nests and loop-nest-reordering invariances that are extremely common in deep learning programs. Finally, they generally do not address the problem of actually scheduling memory movement and splitting up large computations over heterogeneous, parallel architectures.

In practice, the workaround to these limitations has been the creation of hand written kernel libraries, but these libraries have several issues. First, reliance on hand-written kernels means that each new hardware architecture and instruction set requires significant investment from the hardware vendor to even begin executing programs. Furthermore, when novel kernels are introduced, even existing devices require additions to the kernel library and/or compiler systems for support. Finally, even when lowering rules exist, the fact that such kernels are written and called in isolation to the rest of the program often misses significant optimization opportunities such as operator fusion and memory re-use.

1.1 Our Approach

In this paper, we propose ISAM, a **fully-automated, optimizing compiler to map linear algebra computations onto complex hardware architectures**. We limit our domain of interest to machine learning programs such as those supported by TVM and PlaidML, and show how such a domain-specific compiler can automatically produce optimized executable programs for heterogeneous systems that, until now, have been unaddressed by all existing compilation approaches. We demonstrate performance advantages of our approach in a variety of cases.

We separate the compilation problem into two steps. First, *instruction mapping* attempts to enumerate the multitude of ways that one program can be executed on all of the compute devices in the system. We show how our intermediate representation allows ISAM to perform this mapping between arbitrary input programs and hardware instructions. The second step is *scheduling*, which consists of a number of choices, including: which instructions to use,

how to break up the computation, device allocation, and memory movement throughout complex memory hierarchies. We propose using a graph-based hardware-abstraction layer and “dry-run” static scheduling to effectively schedule large computations on complex architectures. The result of the scheduling phase is a compiled executable which can then be executed on the target hardware platform.

In contrast to TVM and similar libraries, ISAM can *automatically* translate high-level descriptions of linear algebra programs into executable assembly for complex machine architectures and instruction sets. To compile for the same architectures with TVM-like libraries would require the programmer to effectively specify exactly which instructions should be run in what order on particular compute devices, forcing the programmer to make a majority of the decisions that should be handled by the compiler.

In comparison to kernel library approaches, we do not claim to have or utilize any fundamentally superior heuristics. In fact, our benchmarks on individual operations (Figure 4) demonstrate that our heuristics can be 6X worse than hand-optimized code. Instead, we claim that, by optimizing an *entire program at once*, ISAM can more effectively utilize even the simplest heuristics to provide better overall performance by taking advantage of reuse and optimization opportunities *between* individual operations that are otherwise inaccessible to kernel libraries.

2 INSTRUCTION MAPPING

One of the major assumptions modern compilers make is that the instruction set targeted (the language recognized by the hardware architecture) consists primarily of scalar and vector operations. Modern programming languages and compilers have been written so that layers of IR can be *lowered* onto predominantly scalar instruction sets. For example, in the pseudo-code shown in Listing 1, recursive lowering “templates” could be applied by the compiler to translate each line into viable x86 instructions.

```
for i, j, k {
    C[i][j] += A[i][k] * B[k][j];
}
```

Listing 1: Pseudo-code for a matrix multiplication. For succinctness, we do not explicitly show the loop nest ordering. Traditional compilers are typically unable to analyze deep loop nests such as this one, but can apply lowering “templates” if the underlying instruction set is scalar.

However, as hardware supported instructions become more complex, lowering is not adequate because the computational granularity of the compiler IR can be finer than that of the instructions provided by the hardware. For example, processing units may expose matrix multiplication instructions that can execute thousands of multiply-accumulate operations in a single cycle. A traditional compiler, which assumes scalar and vector instructions and works via statement-by-statement rewriting or limited template matching, would not be able to determine that the entire program in Figure 1(Ia) can be broken up and executed with a series of matrix and element-wise multiplication instructions.

In some limited cases, a compiler might support a textual template for such programs, matching them to compiler-supported

¹Other names and brands may be claimed as the property of others.

intrinsic, or allow the programmer to manually call an architecture-specific library that exposes support for custom instruction sets and architectures. This approach, however, is not robust to syntactic (but semantic preserving) changes like loop nest reordering or buffer transpositions.

```
for b, i, j, k, di, dj, q, r {
  C[b][i][j][k] += A[b][s * i + di]
                  [s * j + dj][q]
                  * D[di][dj][q][r]
                  * P[c * q + r][k];
}
```

Listing 2: Pseudo-code describing a separable depthwise convolution

In other cases, the computation must be transformed at a coarse level of granularity before mapping to fixed-function accelerator compute blocks. For example, Listing 2 shows pseudo-code for a separable-depthwise convolution [15], a relatively recent kernel used in computer vision. This operation can be executed on matrix-multiplication and convolution accelerators, however when expressed in this type of IR, a number of transformations are required before the matrix-multiplication can be pattern matched directly. We developed our IR and compiler in order to address these mapping and transformation challenges in the deep learning domain.

2.1 Representation

We represent *both* the program to be executed and the instructions exposed by hardware in the same IR (“ISAMIR”) which then casts the overarching problem as one of finding isomorphisms between sub-computations in the “haystack” program (describing the desired computation) and the “needle” program (describing a hardware instruction). Representing both the capabilities of the hardware and the kernels in the same IR is a simple but powerful approach to automating compilation for coarser grained hardware.

We focus ISAMIR on a subset of programs, namely deep learning kernels like matrix multiplication shown in Listing 1. These usually consist of simple arithmetic operations on scalar elements in high-dimensional arrays, with indices of the arrays being determined by affine combinations on a set of loop variables (axis). Importantly, such kernels are usually loop-order invariant when ignoring floating point associativity, which is typical in the deep learning domain. This invariance makes such programs simpler to analyze and optimize than general-purpose programs, as all loop reordering operations are valid. This property has been leveraged by Halide, TVM, and PlaidML, which similarly restrict their input domain to such dependency-free programs (but because they assume low-level instruction sets, they do not need to perform a mapping analysis of their programs as we describe here).

```
forall i, j, k {
  tmp[i][j][k] := A[i][k];
  tmp[i][j][k] *= B[k][j];
  C[i][j] += tmp[i][j][k];
}
```

Listing 3: ISAMIR for a matrix multiplication

An example of ISAMIR is shown in Listing 3. Effectively, we implement a two-operand language where each statement performs exactly one algebraic operation, in-place with two operands. This retains the iteration order-invariance of TVM while requiring that each statement performs exactly one operation for effective analysis.

Notably, for analysis purposes, our system assumes that each statement (line) in the intermediate representation is executed in isolation over its entire iteration domain before executing the next statement. Thus, the “forall” loop surrounding the program is semantically different than a “for” loop in languages such as C, and only serves to explicitly enumerate the loop axes (not stating anything about iteration order). This again leads to simpler analysis to enable parallel statement execution when appropriate.

We now turn our attention to Figure 1, which provides a motivating example to demonstrate our instruction mapping process. In Column (a), we have made up a variant of a one-dimensional convolution which demonstrates the capabilities of our IR (the exact details are not relevant to this exposition). We then want to compile this computation to run on a hardware architecture supporting matrix multiplication (Listings 3 and 1) and element-wise multiplication instructions. The top row contains the internal representation used by ISAM, while the bottom row shows the corresponding C-style pseudo-code.

2.2 IR Transformations

The first thing we notice in Column (a) is that the computation itself cannot directly map onto a matrix multiplication instruction, because there is no “assign, multiply, add” sequence in the program (as in Listing 3). The corresponding issue in the pseudo-code, is a sum over two multiplications as opposed to the one in Listing 1.

Thus, ISAM first performs a series of *IR transformations* which can express the same computations using a different set of operations. In (b), ISAM has reversed the order of the final two statements without affecting the semantics (exposing the “assign, multiply, add” pattern of a matrix multiplication and “multiply” pattern of an in-place, element-wise product). This is an application of the *factorization* transformation pass, which describes the basic algebraic law of distributivity $((a \times b) + (a \times c) = a \times (b + c))$. ISAM also implements transformations that describe commutativity $(a + b = b + a)$ and “identities” that add otherwise-extraneous copies or 0-length dimensions which can be useful in some cases to cause the underlying operation orders to match. We have found that factorization, commutativity, and buffer-copy transformations are enough to map all tested programs (including separable depthwise convolutions) to matrix multiplication and element-wise operations in a fully automated manner.

We note that these transformations form an often-infinite search space, but in practice almost all of our tested mappings require fewer than three transformations. Furthermore, the transformation search can be strongly guided by the mapping process — e.g., in Figure 1(a), ISAM can determine that changing the third statement into a sum would allow it to map a matrix multiplication and only consider transformations that produce such a result. In practice, we have found that the time spent searching this transformation space is negligible (sub-second).

	(a) Program Description	(b) IR Transformations	(c) Loop Ordering	(d) Mapping
(I) ISAMIR	<pre>forall i, x, d, ki, ko { tmp[i][x][d][ki][ko] := A[i][x+d][ki]; tmp[i][x][d][ki][ko] *= B[d][ki][ko]; tmp[i][x][d][ki][ko] += C[x]; D[i][x][ko] += tmp[i][x][d][ki][ko]; }</pre>	<pre>forall i, x, d, ki, ko { tmp[i][x][d][ki][ko] := A[i][x+d][ki]; tmp[i][x][d][ki][ko] *= B[d][ki][ko]; D[i][x][ko] += tmp[i][x][d][ki][ko]; D[i][x][ko] *= C[x]; }</pre>	<p>ISAMIR loop ordering is unspecified. Here, ISAM determines corresponding axis with target instruction:</p> <pre>i <-> i j <-> ko k <-> ki</pre>	<pre>forall x, d { A_t := transpose(A, [1, 0, 2]); D_t := transpose(D, [1, 0, 2]); matmul(A_t[x+d], B[d], D_t[x]); D_tt := transpose(D_t, [1, 2, 0]); ewmul(D_tt[i][ko], C[x], D_tt[i][ko]); D := transpose(D_tt, [0, 2, 1]); }</pre>
(II) Pseudo	<pre>for i, x, d, ki, ko { D[i][x][ko] += A[i][x+d][ki] * B[d][ki][ko] * C[x]; }</pre>	<pre>for i, x, d, ki, ko { D[i][x][ko] += A[i][x+d][ki] * B[d][ki][ko]; } for i, x, ko { D[i][x][ko] *= C[x]; }</pre>	<pre>for x, d { for i, ko, ki { D[i][x][ko] += A[i][x+d][ki] * B[d][ki][ko]; } } for i, ko { for x { D[i][x][ko] *= C[x] } }</pre>	<pre>A_t = transpose(A, [1, 0, 2]); D_t = transpose(D, [1, 0, 2]); // (i, x, ko) for x, d { D_t[x] += matmul(A_t[x+d], B[d]); } D_tt = transpose(D_t, [1, 2, 0]); // (i, ko, x) for i, ko { for x { D_tt[i][ko] = ewmul(D_tt[i][ko], C[x]); } } D = transpose(D_tt, [0, 2, 1]); // (i, x, ko)</pre>

Figure 1: A simplified example of the transformation and mapping process performed by ISAM on a variant of a one-dimensional convolution modified to demonstrate the capabilities of our IR. The top row has the internal ISAMIR representation of the computation, while the bottom row has the corresponding C-style pseudo-code. Here, we ask ISAM to map the example computation onto a device that exposes matrix multiplication operations (described in ISAMIR in Listing 3). In (b), ISAM transforms the computation such that the operations performed match between the program and target computation. In (c), ISAM determines which loop axis in the program correspond to which axis in the instruction description in order to reorder the loops such that the inner-most loop set performs the desired operation. In (d), the buffers are reorganized such that the relevant computation can be performed over the most-minor dimensions.

2.3 Deterministic Mapping

Once ISAM identifies a match between desired and available computational descriptions, ISAM attempts to actually identify the inner matrix and element-wise multiplication computations (Column (c)).

In the pseudo-code representation, this corresponds to reordering the loops such that the inner-most loop set “looks like” the target computation. In (IIc), for example, we see that the inner “for i, ko, ki” loop effectively computes a matrix multiplication (Listing 1), where the i axis corresponds to the i axis; ko to j ; the D buffer to C (dimensions 0, 2 of D corresponding to 0, 1 of C); B to B (dimensions 1, 2), etc.

In the ISAMIR representation loop ordering is not explicit, so instead ISAM must identify and store these mappings at an index level. Effectively, we are looking for a bijection between buffers, axis, and buffer dimensions in the target instruction’s ISAMIR representations and some subset of those in the transformed program’s ISAMIR (there may be multiple valid mappings, so we maintain the set of all *possible* target dimensions). We note that identifying corresponding buffers is trivial, as the buffers of each statement must correspond between program and instruction ISAMIR. Thus, we focus on determining a bijection between loop axis and buffer dimensions, assuming we already know the bijection between buffers themselves.

To do this efficiently, we utilize the common approach of *backtracking*. We begin with an empty map between program and instruction axis, along with a map from instruction buffer dimension to *possible* program buffer dimensions. At each step, we add a new axis mapping and update the potential buffer dimension map. We recursively repeat this until all axis are mapped, and use the potential-dimension maps to determine the bijection between buffer

dimensions. Importantly, each axis-mapping step always *constrains* the number of potential dimension maps, so if at any step we determine that one instruction dimension has no corresponding program dimension, we can immediately discard the last-mapped axis and backtrack to the previous step. In this way, we can efficiently and *deterministically* find valid axis and dimension maps between a given program and instruction described in ISAMIR (assuming that the relevant transformations have been performed such that the underlying operations and buffers match up).

For example, we may initially attempt to map instruction axis j to program axis ko , finding that the only possible program-dimension for (template buffer) C ’s 1-dimension is (program buffer) D ’s 3-dimension (since both have a coefficient of 1 on the $j \leftrightarrow ko$ axis in all relevant statements). If we next attempted to map $k \leftrightarrow x$, we would find that there is no satisfactory program-dimension corresponding to (template buffer) B ’s 1-dimension, as no dimension in the right-hand-side access of the multiplication (in the program) has a coefficient of 1 on x . Thus, we know that all maps including $j \leftrightarrow ko, k \leftrightarrow x$ are invalid, allowing the algorithm to “rule out” entire branches of possibilities at a time.

For completeness, we note that this problem is equivalent to that of bipartite sub-graph isomorphism, and as such has complexity no greater than the general sub-graph isomorphism problem $O(n^{0.729w})$ where n and w are the number of vertices of the graph representation of the source program and target hardware instruction respectively [13]. The algorithm described above was heavily inspired by existing backtracking sub-graph isomorphism solvers [6].

Thus, the matching system can be seen as a loop, Figure 2, where the non-deterministic mapper is constantly sampling points from

Figure 2: ISAM combines a fast but limited “deterministic mapper” with a set of “IR transformations” to effectively map complex computations onto tensor-level instruction sets.

the space of IR transformations (Column (b)), and the deterministic mapper is analyzing each of those points for potential mappings (Column (c)). The result is a set of transformations, each with an associated set of axis and dimension mappings (for succinctness, only one such transformation-mapping pair is considered in Figure 1).

Finally (Column (d)), given a desired mapping between buffers, loop axis, and buffer dimensions, ISAM inserts calls to transpose the buffers such that the most-minor dimensions are used by the instruction call, and replaces the relevant statements with calls to the instruction.

2.4 Instruction Selection

This system often produces multiple potential mappings for a single input program or set of statements. For example, anything that can be mapped to a matrix multiplication instruction could also have been mapped to a dot product instruction. Similarly, some architectures may expose “fused” instructions, for which the system can choose whether to call two independent instructions serially or the single fused instruction. We use the often-reasonable heuristic of picking the non-overlapping instructions that lead to the minimum number of final instructions used (so, for example, a single fused instruction will be preferred to two individual instructions).

3 SCHEDULING

Now that we know which instruction we wish to use to compute each portion of the source program, the system must actually produce executable instructions that can be executed on the target devices.

3.1 Compile-Time Scheduling Approach

Many existing deep learning compilers [3, 12, 16, 17] make use of *static scheduling*, where the computation is broken up into *scheduling* (or compilation) and *execution* phases. During the scheduling phase, the system emits a series of instruction calls that can later be executed to produce the desired output. This scheduling approach is unable to adapt to changing hardware or program conditions, but produces zero runtime overhead.

ISAM also performs static scheduling, as we have found the overhead of runtime scheduling too great for the limited benefits. ISAM’s static scheduler operates through a “dry-run” approach where ISAM attempts a simulated execution of the program, while recording the instructions and associated system state needed to perform the computation. This instruction record from the simulated “dry-run” execution is then stored and can be “replayed” on real hardware for the final execution.

3.2 System Description Graph

In order to flexibly schedule across a wide variety of future systems, we utilize a *system description graph* as an abstraction layer for the underlying hardware. This graph is provided by the user to describe the machine they wish to execute their program on. The system description graph contains three types of nodes: *compute nodes* that expose support for computational instructions, *memory nodes* that contain information about available size and allocation instructions, and *data movement nodes* that describe instructions used for moving data between memory nodes (essentially edges between the memory nodes).

Notably, these nodes are critical objects which interact during the scheduling process by retaining the system state during scheduling. For example, each compute node contains the list of the instructions that it will execute at runtime, memory nodes contain a compile-time list of memory buffers that the system will allocate on them, and edges encode the device or devices that can emit instructions to control memory movement across them. In this way, the graph nodes themselves operate similarly to a hardware abstraction layer (HAL) in a traditional compiler.

3.3 Unrolling

In order to analyze data dependencies more accurately, the first step of the scheduling process is to *unroll* the computation, imposing an explicit order on the sub-computations (instruction calls) to be made. We refer to these sub-computations as *compute tiles*, and each will be associated with a single call to the underlying instruction. In parallel computations, this determines the *dependency order* — if two different devices need to update the same memory, the one earlier in the unrolling will have priority and the second will be dependent on the first’s completion. We use the reasonable heuristic to place computations using the same memory close together in the unrolling.

3.4 Device Allocation

In the general case, there may be multiple physical devices which can execute a given instruction. For example, in our test architecture (see Section 5), each of the individual compute units can execute a matrix multiplication operation. The system must decide which compute unit will run which portion of the computation (compute tile). We use the reasonable heuristic to place computations which use the same memory on the same compute device.

3.5 Scheduling Memory Movement

At this point in the scheduling process, we have assigned compute statements to devices and their order. However, we have not expressed how the compute devices will access the data they need to

Figure 3: In this small example, we attempt to execute two matrix multiplication instructions then sum the products together into the C matrix. First, we need to determine an unrolling order and device allocation (a) for each individual instruction. In this example, we assume that our architecture has two processing units with separate registers. After allocation and unrolling (b), ISAM determines how to move the appropriate memory onto the appropriate computation devices using graph traversals, where the system records the necessary data movement commands and which device needs to execute them by labels attached to the graph hardware representation. For subsequent compute instructions (c), the latest version of data layout is considered which enables efficient data reuse amongst computing devices and registers. When data is updated in place, static cache invalidation is performed to invalidate other copies of that data across the graph.

compute on. For this, we keep track of where any given piece of data is in the system at any given stage in the (simulated) execution process.

For example, before any computation is run, we assume that all relevant memory buffers are stored in the system memory. Now, imagine we wish to execute a matrix multiplication instruction on a particular compute unit in our test architecture described in Section 5, reading data buffers “A”, “B”, and “C”, writing to buffer “C”. We first ask the compute unit’s representative node in our system graph which memory units each operand for the desired instruction may be in before execution. In this step, the compute unit’s node itself can expose limitations such as a requirement that all operands be on different register files.

In order to place the data in one of these executable locations, the compiler must first decide which currently-stored location it wants to copy the data from (for example, if a copy of a network’s weights are stored both on the host memory and on the on-chip HBM), then which location it wants to copy the data to (for example, if there are multiple register files), and finally which path of intermediate nodes and copy instructions it wants to use to move the data (for example, if data must be copied to an HBM unit before it can be copied to the actual register file). All of these problems are difficult and, in

many cases, architecture-specific. We use the heuristic of finding the shortest-path through the system description graph, which can be improved by adding edge weights describing latency between memory units. However, in general there are more concerns than simply the raw latency between memory units. For example, evicting resident data from registers or HBM to make space for the desired data may have ripple effects and can multiply the required memory bandwidth. ISAM can deal with these complications using the modular heuristics support described in Section 4, but we find a simple, shortest-path heuristic sufficient in our evaluations.

Furthermore, as many memory movement paths involve multiple memory units, we keep track of intermediate copies of buffers as they are copied throughout the system, allowing the scheduler to use these intermediate copies later as essentially cached copies of the data. In this way, ISAM can utilize explicitly-allocated memory units as cache devices. However, when new data is written to a copy of a buffer in one location, our scheduler must perform a “virtual” *cache invalidation* to note that all previous copies of the data are now out-of-date.

3.6 Scheduling Recurrent Models

As we have seen, the limited, domain-specific nature of ISAMIR allows ISAM to effectively map and schedule complex linear algebra computations onto novel memory architectures in an effective and general-purpose way. However, although the limited ISAMIR syntax described above can represent and schedule practically all commonly-used deep learning kernels (such as matrix multiplication and convolution), it presents no clear way to compile more complex programs using recurrent and control-flow-guided computation.

Recurrent neural networks (RNNs), for example, execute groups of computation repetitively, re-using results from the previous iteration. This poses additional challenges and opportunities for scheduling. For example, a GRU [5] “cell” (the computation repeated on each iteration of the GRU RNN algorithm) may be executed hundreds of times on a single input sequence. Some compilers unroll these RNN loops to reduce the problem back to standard scheduling. However, this approach is expensive at compile-time, places further challenges on memory optimization passes, and limits flexibility for dynamic RNN length control.

Ideally, we would like to schedule a finite number of steps at compile-time and invoke these sub programs dynamically at execution-time, but naïvely scheduling the computation once is not sufficient since the buffers may not be in the same location after the computation as they were in the beginning. Also, scheduling with awareness of repeated execution offers additional optimization opportunities such as persistent weights. To address this, ISAM explicitly exposes the concept of a recurrent loop and schedules these loop bodies specially three separate times. First is the *priming iteration*, which performs one instance of the computation, then leaves the data buffers as close to the compute devices as possible. Next is the *recursive iteration*, which executes on the data buffers from the priming iteration and ensures all outputs overwrite the appropriate inputs. Finally, the scheduler emits a *finish iteration*, which performs the computation a final time and places the data buffers where they will be needed by the next instruction in the program. At execution time, a driver first executes the priming iteration, then the recursive iteration as many times as necessary, and finally executes the finish iteration.

In theory, this unrolling process can be applied recursively to nested loops, first determining an explicit schedule for the innermost loop, then scheduling loops at higher levels (using the now-scheduled inner-loops). However, in practice, modern RNN architectures only require the scheduling of a single loop (iterating over the input sequence).

Finally, although we here consider only recurrent models, the above idea (compiling the dense, conditional computations with ISAM then using a lightweight driver to orchestrate during execution-time) can be generalized to support more complex control-flow (however, modern deep-learning algorithms generally do not use any control-flow beyond the simple loops already supported in ISAM).

4 MODULAR HEURISTIC SUPPORT

Throughout the scheduling process, ISAM must use a number of heuristics (for example, when scheduling memory movement). In

practice, we have found that a small number of “common-sense” heuristics (most of which are mentioned in Section 3) are sufficient to produce competitive executables. These general-purpose heuristics are particularly powerful in ISAM because the system-description graph can accurately capture much of the important system details at a high level of abstraction.

However, many systems may have architecture-specific heuristics that can improve executable quality, compile times, or both. ISAM supports such architecture-specific heuristics through a plugable system, where heuristics are exposed via a class instance (named an “Approach”) that is queried whenever the ISAM scheduler needs to make a heuristic choice. A default Approach is provided by ISAM, using the general-purpose heuristics described above.

This fully-optional, fully-extensible system allows for ISAM to become gradually better at scheduling for a particular architecture, falling back on the general-purpose heuristics when the architecture is new while slowly supporting architecture-specific heuristics as they are developed. It also allows engineers to quickly test and compare proposed heuristics, providing a better understanding of new hardware architectures and benefiting even traditional kernel library development.

As our test architecture is relatively new, and the general-purpose heuristics work well, all experiments use the “common sense” heuristics discussed in prior sections (with some small compiler performance improvements, such as a filtering mechanism to avoid considering clearly-non-optimal memory movement paths during the shortest-path computation).

5 CASE STUDY ARCHITECTURE

While the ISAM architecture is hardware-agnostic, we evaluated the principles against both an existing CPU architecture and a novel deep learning architecture which exhibits many of the challenges discussed previously. This latter processor is made up of many compute units that can execute native matrix instructions (such as matrix multiplication), element-wise operations (useful for activation functions), and matrix-wide reductions (such as sum or max), in addition to other special purpose units for common operations and control flow. Furthermore, on-chip “clusters” of these compute units are programmed with the same instruction stream and share a set of large register files, while there are several high-bandwidth memory modules to enable rapid access to memory too large for the register files. There are a number of host and device-side instructions to move memory between register files, processing units, and high-bandwidth units. Generally speaking, the processing units can only execute instructions on data in their respective register files, and further restrictions on how many times a single register file can be used in an operation exist as well. All of the memory units in this system are explicitly managed, so there is no cache hierarchy (although, as mentioned, ISAM can use explicitly-managed memory as an effective cache by re-using intermediate copied buffers).

This architecture (with significant use of tensor-level instructions, large amounts of parallelism, and a complex, explicitly-managed memory hierarchy) presents a challenging problem for compilers. Furthermore, this test architecture has a hand-optimized kernel library developed for it, providing our performance baseline. For

Figure 4: A selection of results comparing cycles per operation for ISAM and the kernel library (“KL”) on matrix multiplication. Smaller is faster. When the KL is optimized for a given size, it can perform over 5X faster than ISAM kernels. However, for many sizes that are less common or significantly different than the library’s intended focus, ISAM can produce competitive or significantly faster kernels.

common operations and tensor sizes, the kernels in this library are able to achieve nearly-theoretical maximum utilization of the architecture.

6 RESULTS

6.1 Mapper

We first test the effectiveness of our mapping system, as described in Section 2. We compiled a set of recently-proposed convolution kernels and RNN cells (including depthwise and separable depthwise convolution and GRU cells), then used ISAM to map these kernels to a system with matrix multiplication (and manipulation) and element-wise instructions. Our mapper was able to automatically determine the expected mappings in all cases. By comparison, all previous work requires this mapping to be explicitly specified by the programmer.

Additionally, by exposing BLAS methods as “instructions” to ISAM, we can map the convolutions onto BLAS calls for x86 devices as well. In Section 7, we also show how to effectively target x86 by using ISAM mappings with TVM and LLVM. These results demonstrate how the system can be effectively utilized even on existing devices supporting scalar and vector instructions.

All mappings were performed in sub-second time on commercial hardware. Most mappings were found within two transformations, all within ten.

6.2 Scheduler

6.2.1 GEMM. Next, to test the effectiveness of ISAM’s scheduling, we compile and measure individual matrix multiplications using sizes from DeepBench [2] and internal tests representing real-world usage patterns for the architecture. We report a selection of the results in Figure 4.

For confidentiality reasons, Figures 4 through 5 have been normalized by the minimum value displayed on the plot to conserve relative differences (thus, the ratio of any two data points is accurate, but the absolute value or difference is not provided). Test results are averaged over ten trials each (run and re-compiled twice,

Figure 5: A selection of results comparing cycles per operation for ISAM and the kernel library when executing a GRU cell. Smaller is faster. Although the KL outperforms ISAM for many of the individual operations in the GRU cell (Figure 4), ISAM utilizes inter-operation fusion and memory reuse opportunities to outperform the equivalent KL computations.

with the lowest average taken, as the ISAM scheduler has some non-deterministic behavior that can affect the performance of the resulting executable), and we report on-device cycles (smaller is faster).

First, in configuration (a), the kernel library (KL) is well-optimized for the particular size and significantly out-performs ISAM-generated kernels. This is due to the significant prior knowledge, experimentation, and engineering effort that the library authors put into optimizing this operation for this device. By contrast, ISAM has only a few heuristics to use when scheduling a kernel. This demonstrates the continuing value in hand-optimizing hot spots (and hand-specifying heuristics — see Section 4), which kernel libraries are still well-suited to address. However, configurations (b) - (c) demonstrate the existence of (less-common) shapes for which the existing library has not yet been optimized and ISAM can produce comparable or slightly better-performing kernels. Finally, for shapes such as configuration (d) which do not currently fit the assumptions used in the kernel library well, ISAM can produce significantly faster kernels.

We note, again, that this benchmark is quite challenging to ISAM, since it is already well-optimized by the kernel library in many cases, and there is no opportunity for ISAM to perform inter-operation fusion. Nevertheless, we have found that there *are* situations (e.g., configuration (d)) in which ISAM can produce performant kernels (which can temporarily be used in place of the kernel library and as a starting point for hand-optimization).

6.2.2 GRU. To test ISAM’s benefit to recurrent computations, such as leaving re-used data on register files between kernels or pipelining kernels, we compiled and executed a GRU recurrent neural network over 128 steps, using matrix shapes adapted from the DeepBench [2] standardized benchmark. Full compilation times were no more than a few minutes per kernel. A selection of our results are presented in Figure 5.

As the selected results show, ISAM out-performs the composition of kernel-library operations in all tested GRU configurations. This is primarily due to the fact that ISAM has fine-grained control over

REFERENCES

- [1] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. 1989. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 4 (1989), 491–516.
- [2] Baidu. 2017. DeepBench: Benchmarking Deep Learning operations on different hardware. <https://github.com/baidu-research/DeepBench>
- [3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *arXiv preprint arXiv:1802.04799* (2018).
- [4] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. *arXiv preprint arXiv:1805.08166* (2018).
- [5] Kyunghyun Cho et al. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation* (2014). <https://doi.org/10.3115/v1/w14-4012>
- [6] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.
- [7] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv preprint arXiv:1801.08058* (2018).
- [8] James Demmel and Grace Dinh. 2018. Communication-Optimal Convolutional Neural Nets. *CoRR abs/1802.06905* (2018). arXiv:1802.06905 <http://arxiv.org/abs/1802.06905>
- [9] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke. 2018. Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures. *ArXiv e-prints* (Aug. 2018). arXiv:cs.DC/1808.05567
- [10] K. Keutzer. 1987. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC '87)*. ACM, New York, NY, USA, 341–347. <https://doi.org/10.1145/37888.37940>
- [11] Chris Lattner and Vikram Sadanand Adiv. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization*. 75–86.
- [12] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled! TensorFlow Dev Summit.
- [13] Jaroslav Nešetřil and P. Ossona de Mendez. 2012. Sparsity: Graphs, Structures, and Algorithms, volume 28 of Algorithms and Combinatorics.
- [14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédéric Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.
- [15] Laurent Sifre and Stéphane Mallat. 2014. *Rigid-motion scattering for image classification*. Ph.D. Dissertation. Citeseer.
- [16] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. [n. d.]. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* ([n. d.]).
- [17] Vertex.ai. 2017. Announcing PlaidML: Open Source Deep Learning for Every Platform. <http://vertex.ai/blog/announcing-plaidml>