A Data-Centric Optimization Framework for Machine Learning

Oliver Rausch [*1] **Tal Ben-Nun** [*1] **Nikoli Dryden** [1] **Andrei Ivanov** [1] **Shigang Li** [1] **Torsten Hoefler** [1]

Abstract

Rapid progress in deep learning is leading to a diverse set of quickly changing models, with a dramatically growing demand for compute. However, as frameworks specialize optimization to patterns in popular networks, they implicitly constrain novel and diverse models that drive progress in research. We empower deep learning researchers by defining a flexible and user-customizable pipeline for optimizing training of arbitrary deep neural networks, based on data movement minimization. The pipeline begins with standard networks in PyTorch or ONNX and transforms computation through progressive lowering. We define four levels of general-purpose transformations, from local intra-operator optimizations to global data movement reduction. These operate on a data-centric graph intermediate representation that expresses computation and data movement at all levels of abstraction, including expanding basic operators such as convolutions to their underlying computations. Central to the design is the interactive and introspectable nature of the pipeline. Every part is extensible through a Python API, and can be tuned interactively using a GUI. We demonstrate competitive performance or speedups on ten different networks, with interactive optimizations discovering new opportunities in EfficientNet.

1 Introduction

The modern development of deep learning is spearheaded by the confluence of algorithms (Bottou et al., 2018), data (Halevy et al., 2009; Sun et al., 2017), hardware (Raina et al., 2009; Jouppi et al., 2017), and systems (Abadi et al., 2015; Paszke et al., 2019). Today, machine learning systems have become critical in light of the increasing complexity of models and the breadth of emerging hardware. Training performance in particular is key to both researcher productivity and reducing the environmental impact of machine learning (Strubell et al., 2019; Patterson et al., 2021). To this end, systems and compilers such as XLA (Google, 2021), ONNX Runtime (Microsoft, 2021), and TorchScript (PyTorch Team, 2020) are widely used to automatically optimize training.

Most such compilers tend to focus on *operator-centric* optimization, that is, defining specific rule-sets based on a set of predefined building blocks. Thus, these frameworks tend to perform very well on popular neural networks, but often have limited performance improvements on new, unseen models — which many researchers would like to explore, but may not be able to train given the high compute costs. Indeed, this effect has led to a situation where the models that perform well on existing hardware and systems are more likely to succeed than others (Hooker, 2020).

In this work, we propose to shift the paradigm from operators to their memory access patterns by performing data-centric DNN optimization. It is well-established that the performance of modern DNNs such as Transformers hinges on data movement minimization (Ivanov et al., 2021; Microsoft, 2020), and that a certain FLOP reduction does not guarantee matching speedup (Tan & Le, 2019). Up until now such optimizations have been performed by specialized engineering teams for specific architectures, remaining out of reach for most research groups (Barham & Isard, 2019).

We present **DaCeML**[1], the Data-Centric Machine Learning framework, which provides a simple, flexible, and customizable Python-based pipeline for optimizing training and empowering deep learning research. DaCeML seamlessly integrates with PyTorch (Paszke et al., 2019) and ONNX (ONNX, 2021) to enable accelerating and tuning existing models, both in evaluation and backpropagation. The input models are then optimized with a general-purpose transformation pipeline, which reduces data movement at fine and coarse grain, regardless of the internal computation. Lastly, DaCeML provides interfaces to programmatically and interactively guide the optimization process further.

Internally, the framework uses Stateful Dataflow Multigraphs (SDFGs) (Ben-Nun et al., 2019) as a data-centric, hierarchical graph-based intermediate representation, which enables it to work with operators and data movement at all levels, from registers to distributed memory. The DaCeML optimization pipeline begins with a standard PyTorch or

[*]Equal contribution [1]Department of Computer Science, ETH Zurich, Switzerland. Correspondence to: Oliver Rausch <firstname.lastname@inf.ethz.ch>.

Preprint.

[1] https://github.com/spcl/daceml

ONNX model and transforms computation through progressive lowering, using four levels of general-purpose transformations: coarse-grained; local data movement reduction; global data layout optimization; and hardware specialization. Data-centric optimizations manifest in different ways, especially when training is considered. One of the several unique controls DaCeML provides, for example, is choosing whether to recompute or retain data for backpropagation. With the included automatic optimizations, DaCeML often *already matches or outperforms modern frameworks*.

The central focus of DaCeML is the ability to then go further – it unpacks the DNN compiler-black box and puts the machine learning practitioner in the driver's seat. This starts with the visual, introspectable intermediate representation that makes finding and understanding performance issues, such as excessive data movement, intuitive. These are subsequently addressed by manipulating data movement or the data layouts of the parameters and intermediate storage. The SDFG IR allows automated or manual searches to be performed on multiple granularities *simultaneously*, rather than in the traditional compiler-based fixed set of passes. These can be performed either using a Python API or interactively using the Visual Studio Code (Microsoft, 2021) IDE.

We provide an overview of DaCeML's design, its approach to progressive lowering, and its transformations, including novel optimizations not supported by other frameworks. In particular, we demonstrate up to 3.43× speedups over prior best on non-mainstream activations; state-of-the-art performance with automatic optimization of a wide range of DNNs from various domains (Bochkovskiy et al., 2020; Devlin et al., 2019; He et al., 2015; Long et al., 2015; Naumov et al., 2019; van den Oord et al., 2016; Sandler et al., 2019; Tan & Le, 2019; Tolstikhin et al., 2021; Zagoruyko & Komodakis, 2017), compared with PyTorch, TensorFlow + XLA, and JAX; and two guided optimization case studies on BERT (Devlin et al., 2019) and EfficientNet (Tan & Le, 2019), the former highlighting the importance of data layout and the latter showing that nonlocal data movement minimization can yield 1.33× speedup over cuDNN.

2 Background and Related Work

Most closely related to our work is that of Ivanov et al. (2021), which also takes a data-centric view of optimizing deep learning training and finds that transformers (Vaswani et al., 2017; Devlin et al., 2019) are memory bound, illustrating the importance of data-centric analysis. However, their work focuses exclusively on BERT (Devlin et al., 2019), and is entirely manual. Similarly, DeepSpeed (Microsoft, 2020) also provides manually-optimized primitives for transformers. Numerous other works have focused on specific optimizations (Frostig et al., 2018; Jia et al., 2019b; Sivathanu et al., 2019; Baghdadi et al., 2019; Vasilache et al., 2018; Lethin, 2017; Wei et al., 2018; Truong et al., 2016; Venkat et al., 2019; Dong et al., 2019; Elango et al., 2018; Hu et al., 2020; Oyama et al., 2018; Li et al., 2016; Zheng et al., 2020; Li et al., 2020; Steiner et al., 2020; Yang et al., 2021) that can be implemented as DaCeML transformations.

Training vs. inference compilation Inference optimization elides several concepts required in training. Most importantly, no automatic differentiation is needed. Secondly, as opposed to training, intermediate activations need not be stored for backpropagation, which drastically changes the optimization search space. Thirdly, several operators (e.g., batch normalization, dropout, convolution) behave differently during training. For example, batch normalization stores running statistics throughout the process, and convolutions that use basis transformations (FFT, Winograd) can pre-transform the weights once, since they will not change during inference. Lastly, apart from backpropagation, gradient updates and stochastic optimizer rules must be applied.

State-of-the-art DNN compilers Researchers have often used custom kernels, linear algebra primitives (e.g., BLAS), and vendor-optimized libraries (e.g., cuDNN, Chetlur et al., 2014, oneDNN, Intel, 2021a). Recently, with the modularization and diversity of DNNs, the focus is shifting towards using general compiler infrastructure in DNN optimization, which uses Just-in-Time (JIT) compilation to both optimize individual operators and fuse them.

The methods by which DNN compilers optimize (transform) code can be classified into three categories: *graph rewriting rules*, *expansions*, and *global passes*. Graph rewriting represents DNNs as DAGs and pattern-matches certain subgraphs to replace them with others. Expansions convert a known operation (e.g., batch normalization) directly into an explicit, pre-optimized version. Finally, global passes operate on the entire code (e.g., memory scheduling). Ultimately, the first two categories, and sometimes the third, are implemented on an operator-centric basis. Below, we discuss state-of-the-art DNN compilers and their inner workings.

XLA Advanced compiler infrastructure (Google, 2021) used by TensorFlow (Abadi et al., 2015) and JAX (Bradbury et al., 2018), which contains multiple intermediate representations (e.g., HLO, LLO) and various domain-specific expansion transformations and global passes. Apart from those, general purpose optimizations (such as dead-code elimination) are performed by the LLVM (Lattner & Adve, 2004) and MLIR (Lattner et al., 2020) (experimental) infrastructure. Prior to XLA, graph rewriting rules in TensorFlow were provided by a component called Grappler.

TorchScript (`torch.jit`) The main static optimization effort in PyTorch (Rotem et al., 2018; PyTorch Team, 2020; Paszke et al., 2019). Through either tracing or Python introspection, TorchScript can convert PyTorch modules into

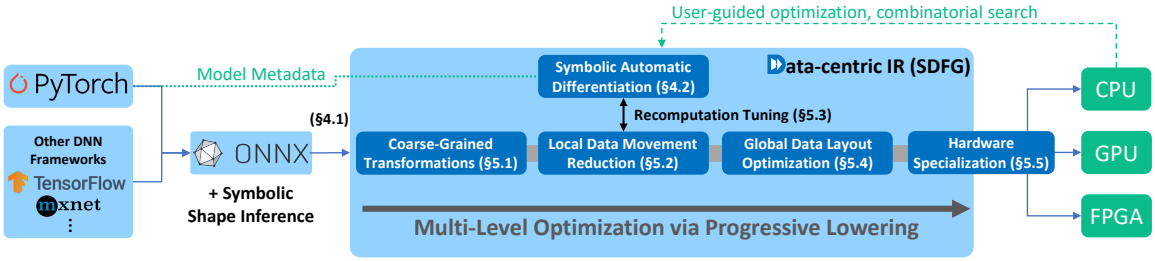


Figure 1. DaCeML system overview.

a DAG-based IR, and contains a pass manager to manage transformations. As opposed to XLA, optimizations are focused on the forward pass, leaving backpropagation to the `autograd` module. A recent addition to PyTorch is the experimental `torch.fx` module, which allows lifting optimized graphs back to Python code. The types of transformations available are global passes (called "direct graph manipulation"), pattern-matching, and expansion transformations (called "proxy/retracing").

Others Frameworks that maintain a graph structure also provide optimizing transformations. ONNX Runtime (Microsoft, 2021) provides graph rewrite rules and global passes, mostly focused on cleaning up artifacts resulting from the ONNX (ONNX, 2021) format (e.g., constant folding) and algebraic fusion involving chained ONNX operators (e.g., `MatMulAddFusion`). OpenVINO (Intel, 2021b) uses nGraph (Cyphers et al., 2018) with graph rewrite rules and single-node matching transformations to optimize inference. For interactivity, it allows users to deselect certain optimization passes on given node names through its command-line optimizer. TVM (Chen et al., 2018) provides passes on its two IRs (Relay, TIR) based on statement visitors/mutators, similar to AST manipulation. TASO (Jia et al., 2019a) also provides subgraph pattern rewrite rules, focused on linear algebra. TensorRT (NVIDIA, 2021) and cuDNN (Chetlur et al., 2014) provide functionality to fuse elementwise operations to DNN primitives, the former for the purpose of accelerating inference and the latter for training as well.

Limitations Each of the aforementioned DNN compilers is affected by at least one of three limiting factors. First, most of the transformations rely on specific operator types, which change from network to network, and over time. Second, the transformations are written in lower-level languages (typically C++) and sometimes require compiler analysis knowledge to develop. Lastly, the transformations are applied in passes, usually in a greedy fashion, inhibiting opportunities for further tuning and analysis.

3 System Overview

We propose to tackle the problem of DNN optimization through a holistic and user-centered approach. Data-Centric Machine Learning (**DaCeML**) is a semi-automated, user-extensible pipeline for compiling and optimizing deep learning workloads. It takes PyTorch models and ONNX files, optimizes the program through data-centric means, and generates code with state-of-the-art performance for multiple platforms. The goal of DaCeML is threefold: (1) **Usability**: keeping the simplicity and expressiveness of PyTorch and its powerful model definition; (2) **Generality:** using general-purpose transformations that generalize well to new models, from coarse-grained optimizations to low-level code generation; and (3) **Interactivity**: translating the simplicity of defining models to optimizing them by enabling easy extensibility and interactive reasoning.

To achieve this goal, DaCeML defines a *progressive lowering* pipeline, depicted in Figure 1. At the core of DaCeML is the data-centric Stateful DataFlow multiGraph (SDFG) representation (Ben-Nun et al., 2019), a parametric graph IR designed for high-performance computing. SDFGs promote separation of concerns between domain scientists and performance engineers by optimizing the data movement of a program separately from the algorithmic part of the input code. The DaCe framework is written in Python and has recently powered multiple applications in several domains, including quantum chemistry (Ziogas et al., 2019) and numerical weather prediction (de Fine Licht et al., 2020), achieving state-of-the-art performance on CPUs, GPUs, and FPGAs.

Once within that representation, DaCeML can automatically optimize inference, forward, or backward passes of a model to performance that is on par with (or faster than) state-of-the-art DNN compilers. Furthermore, DaCeML's API is geared towards extensibility — from defining new operator implementations, through local and global data movement planning, to low-level work partitioning in code generation — allowing users to assume direct control and guide the optimization process towards faster performance. The process can be done interactively, through APIs and IDE support, or through combinatorial searches over the optimization space.

4 Data-Centric Progressive Lowering

DaCeML supports loading models from the Open Neural Network eXchange (ONNX) format (ONNX, 2021), or by

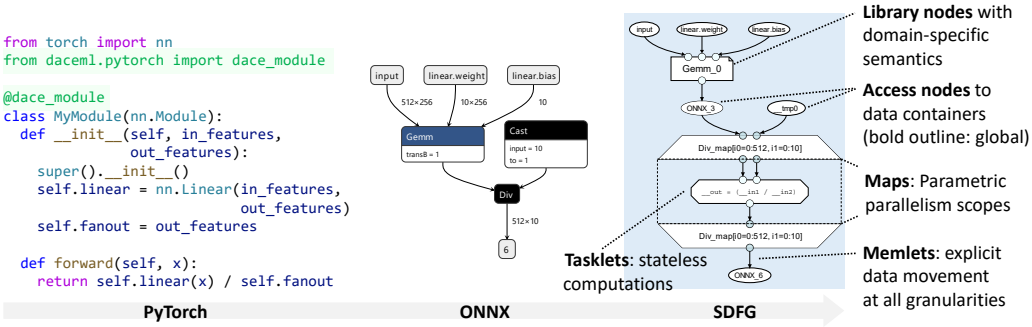


Figure 2. SDFG Intermediate Representation and conversion. ONNX graph rendered in Netron (Roeder, 2021).

wrapping PyTorch `nn.Modules` with a `@dace_module` decorator. The loaded models can be invoked directly, or used as optimized replacements of sub-networks in existing training and inference codes. Figure 2 provides a summary of the conversion process and SDFG representation.

4.1 From PyTorch and ONNX to Symbolic Stateful Dataflow

The ONNX format is a DAG representation of DNNs, which aims to standardize deep learning primitives across frameworks and networks. In the representation (Figure 2, middle), nodes represent operators and edges represent tensors that form data dependencies between them. As of version 1.9, there are 186 operator types in ONNX (ONNX, 2021), which cover the vast majority of deep learning models.

Most deep learning frameworks, including PyTorch, have converters to the ONNX representation (typically used for deployment). For this reason, we chose to build DaCeML around the ONNX representation. When a PyTorch module is wrapped, we use PyTorch's ONNX exporter to generate the computational graph. As this is not sufficient for training and parameter management, we augment the ONNX input with metadata such as positional arguments, module hierarchy, and parameter structure. From those elements, we generate a replacement `nn.Module` object that is controlled by DaCeML and seamlessly integrates with the PyTorch `autograd` module for training. Internally, the module keeps either one (inference) or two (forward, backward) SDFGs for execution and optimization.

SDFG We use the example in Figure 2 to explain the SDFG representation. Full details and operational semantics for SDFGs are provided by Ben-Nun et al. (2019). In particular, SDFGs are graphs of graphs, representing state machines of acyclic dataflow multigraphs. Each *state* in the state machine (light blue region) represents data access and computations (the equivalent of the full ONNX graph); the outer, state machine graph describes coarse-grained control flow (e.g., the training loop).

Two central concepts in SDFGs are the explicit separation of data movement from computation, and a multi-level view of data movement. The first concept is aided by the structure of the state graph itself — computations (*tasklets*, octagonal nodes) and *access* (oval nodes) to data containers are represented by nodes, and data movement is represented explicitly by the edges, called *memlets*. A tasklet cannot access non-local memory that is not connected to it via a memlet, and memlets are represented as symbolic ranges that connect access nodes and tasklets. Data container names are unique, and indicate disjoint memory regions, but access nodes that refer to them can appear multiple times. Furthermore, containers that are not local to the computation (i.e., cannot be removed in subsequent transformations) are called global memory and visually marked with a bold outline.

The concept of multi-level data movement is facilitated by three other types of nodes: (1) computations with predefined semantics (e.g., matrix multiplication generated from the `Linear` layer in the figure) are represented by *library nodes*; (2) parallel regions, such as the `Div` operator in the figure, are grouped together between *map* (trapezoidal) nodes, which contain iteration ranges and a schedule (OpenMP loop, GPU kernel, FPGA region replication, and others); and (3) in case control flow is necessary within a parallel region, SDFGs can be nested in each other with *Nested SDFG* nodes.

ONNX library nodes The predefined semantics of library nodes allows them to be expanded to lower-level implementations, either "native" SDFG elements (tasklets, memlets, access nodes, and maps) or fast library calls (e.g., cuBLAS and cuDNN). As we shall show, native expansions can sometimes be optimized further than such libraries. In DaCeML, each ONNX operator is represented by a library node. Native implementations for many ONNX operators use the NumPy DaCe frontend to generate SDFGs. For example:

```
@python_pure_op_implementation
def Softplus(X, Y):
  Y[:] = numpy.log(1 + numpy.exp(X))
```

As not all operators are natively implemented, and the ONNX standard is expected to grow, we also support a fall-

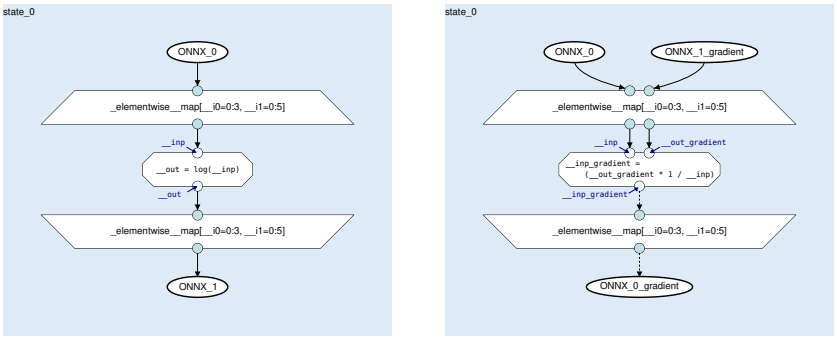


Figure 3. Elementwise logarithm SDFG (left) and symbolic auto-differentiated version (right).

back for any ONNX operator, using ONNX Runtime (Microsoft, 2021) as a backend. Specifically, the expansions generate code that eagerly calls the operators within the compiled module.

Progressive lowering of library nodes Several library node expansions lower to other ONNX nodes. As an example, the `MatMul` node is lowered first to an Einstein sum (`Einsum`), which enables some tensor-contraction algebraic optimizations. Another is the `im2col` implementation of `Conv`, in which a `Gemm` node is included in the lowering output. This multi-level, progressive lowering reduces implementation redundancies and facilitates operator kernel authoring due to reuse.

4.2 Symbolic Automatic Differentiation

To compile and optimize for training, DaCeML uses source-transformation-based reverse-mode Automatic Differentiation (AD) to compute Vector-Jacobian Products (VJP) of the operators. Guided by the principle of generality, all lower level optimization on the backward pass is done using the same optimizing transformations as the forward pass. To enable this, the DaCeML AD engine transforms a forward pass SDFG into the corresponding SDFG that computes the required gradients. Critically, the differentiation is performed on the lowered, native SDFG, removing the need for manually specified VJPs.

Differentiating SDFGs is challenging, since nodes not only represent computation, but memory accesses and parametric replication. It is important to differentiate the computation symbolically, even if all sizes are known, since the access pattern (memlets) within each operator is still symbolic.

Given an SDFG, the engine first determines the subgraph to differentiate, based on the output access nodes and the access nodes of inputs that require gradients (part of the metadata obtained from PyTorch). The subgraph is then traversed in reverse topological order and reversed as follows. *Access nodes* are reversed by "inverting" the access of the node, and replacing the data that is written/read with the adjoint of that data. *Maps* can be reversed by simply converting map entry nodes to map exit nodes and vice versa. The reverse of a code node — whether *tasklet, library node, or nested SDFG* —- is a node that computes the VJP of the forward node. The inner tasklets of most machine learning operators work with scalar values. To automatically differentiate these scalar expressions, DaCeML uses the SymPy (Meurer et al., 2017) symbolic differentiation engine (see example in Figure 3). Nested SDFGs are reversed recursively, and in the case of library nodes, they are lowered to their native SDFG implementation for further differentiation, with the possibility to provide manual backward expansions.

The VJPs of code nodes often require values that were inputs to the corresponding forward node. These values are either *forwarded* by storing the value in the forward pass and reading it in the backward pass, or recomputed in the backward pass.

Manual backward expansions DaCeML's AD engine is able to automatically differentiate almost all operator implementations, producing backward passes comparable to the hand-written implementations. However, it is often beneficial to specify manual backward operators (e.g., `Einsum` can be reversed directly for performance or `Softmax`/`LogSoftmax` for numerical stability). DaCeML provides this capability by manual expansions.

5 Optimization Pipeline

DaCeML follows DaCe's white-box optimization approach, in which optimization can be performed automatically as a starting point, and then is interactive and guided by performance engineers. We further simplify the interface and extend its capabilities, in order to allow ML practitioners to productively write transformations of their own.

DaCeML transformations are written in Python, and can use the existing graph rewriting tools from DaCe. A transformation can perform local replacement (complete example in Figure 4), graph pattern-matching (see Appendix C), or global modifications. The replacement API is graph-based,

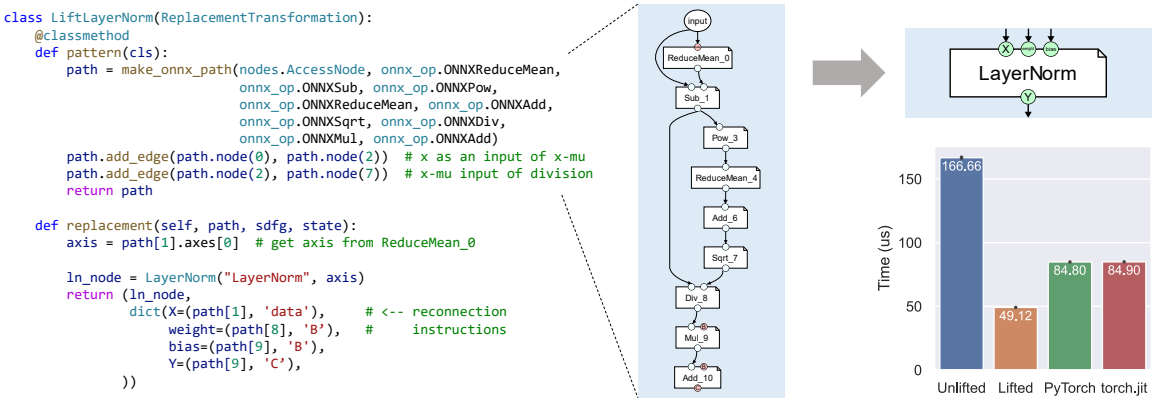


Figure 4. Coarse-grained transformation example (§ 5.1): lifting Layer Normalization from PyTorch.

where users provide the subgraph to match (`pattern`), and replacement instructions (`replacement`), which include returning a new node or nested SDFG and instructions on how to reconnect to the pattern graph or add new arrays (via the mapping in the return value). For more complex behavior, any method (exact matching condition, application, reconnection) can be overridden.

The transformations provided in DaCeML's standard set can be categorized into four distinct levels, ordered by their application as the graph is lowered: (1) Coarse-grained transformations on the ONNX-SDFG representation; (2) local data movement reduction, e.g., operator fusion and replication; (3) global (network-wide) data movement optimization; and (4) hardware specialization and workload partitioning. From those categories we sub-select transformations to apply globally on every network in our experiments, forming a "recipe" for DNN optimization, and show how guided application of the rest of them can yield even higher performance.

5.1 Coarse-Grained Transformations

The first step in transforming the SDFG is leveraging the semantics of ONNX operators. We can further categorize this into transformations that clean up the graph from framework- and ONNX-related clutter, and transformations that find subgraph combinations to improve the performance or numerical stability of the model.

Cleanup We provide a general ONNX transformation called `ConstantFolding`, which evaluates nodes whose inputs are not part of the weights or model inputs, and replaces the subsequent paths with the evaluated value. Following expansion to native SDFG, another transformation (`InputToConstant`) complements this by inlining constant inputs into their respective tasklets. This results in a chain of optimizations, for example, a 3.0 exponent in a `Pow` operator is first considered as a constant array; after cleanup, the code generator will subsequently replace the operation with much faster multiplications. Overall, this eliminates many unnecessary copy and computational operations (see Appendix A), and enables more complex transformations, such as algebraic fusion.

Algebraic fusion Due to the progressive lowering approach taken in DaCeML, many linear algebra operations (e.g., matrix-vector multiplication, transpose, batched tensor contraction) are lowered into `Einsums`. We provide a set of transformations, such as horizontal and vertical fusions, which can fuse together a transposition into a single Einsum (e.g., `ij->ji` and `ik,jk->ij` to `ki,jk->ij`, see Appendix C.1 for visualization). This information potentially feeds into combinatorial searches of optimal data layouts — the shape of a weight can be modified through DaCeML if the Einsum does not generate an optimized BLAS call due to layouts (for which we provide a check). The ability to modify the shape of intermediates and weights is necessary to maximize performance in today's DNN architectures (Ivanov et al., 2021) and, to the best of our knowledge, unique to DaCeML.

Lifting and omitting operations Other transformations relate to removing nodes that do not perform computations through symbolic size analysis, such as removing an Adaptive Pooling operator if the input/output sizes are the same, or fusing together padding and convolution operations. Known subgraphs generated by frameworks can be also be "lifted" to new nodes (e.g., with custom implementations or backward expansions), as is the case for Layer Normalization (Ba et al., 2016) in Figure 4.

5.2 Local Data Movement Reduction

While powerful, domain-specific transformations on the ONNX representation do not suffice, as they only capture operator-centric behavior and will not work with unseen operators or unexpected combinations thereof. We thus focus on transformations on the native SDFG (after ONNX library node expansion) that apply directly to ML workloads.

DaCe contains a library of standard transformations that use the structure of the graph to manipulate data movement. Of particular importance are `MapTiling` and `LocalStorage`, where the former partitions the workload of a map into tiles by introducing another nested map, and the latter adds an access node between two such maps in order to create storage only accessible by the current tile (e.g., placing weights in shared memory in Section 6.3). For DaCeML, we develop additional transformations to handle data movement bottlenecks in deep learning workloads.

Transformations Given an arbitrary subgraph, the `SubgraphFusion` transformation tries to find a way to combine multiple map scopes into one scope. This reduces write/read roundtrips to global memory and, on GPUs, kernel launch overhead. In elementwise operations where iteration spaces are equal, the operation is trivial. However, in many cases the spaces are permuted (when different data layouts are involved) or do not share the entire space (e.g., in the Softmax operator, see Appendix B.2). For this reason, we extend subgraph fusion to find permutations, offsets (e.g., `start:end` vs. `0:end-start`), parallel regions in reduction, and the greatest common subset of map iteration domains to fuse over. The transformation extracts them out first and then fuses the requested outer maps, greatly extending the realm of fusion possibilities.

Fusion space exploration To automate the process of optimizing data movement in graphs, we develop automatic optimization heuristics that enumerate the space of fusible subgraphs. Currently, DaCeML supports greedy enumeration with pruning (based on path constraints for fusion), and ranking according to scoring functions. For evaluation, we rank by largest neighboring regions to minimize data movement and GPU kernels.

5.3 Backpropagation and Data Movement

When optimizing the forward and backward passes at the same time, data-centric optimizations can control aspects of auto-differentiation. As AD requires intermediate value forwarding, the memory footprint can become infeasible and movement too demanding (Chen et al., 2016; Jain et al., 2020). However, fusing maps on the forward pass means omitting intermediate values, and replicating them means recomputation on the backward pass. This process can create a tunable "knob" to optimize backpropagation.

For this purpose, we develop two transformations: `TaskletFusion`, which fuses two computations into one symbolic tasklet; and `MapReplication`, a transformation that detects access nodes being read more than once, and replicates the immediate map leading to it. Example uses of the two on the Mish activation (Misra, 2020) and statistical normalization are shown in Appendix B.

5.4 Global Data Layout Optimizations

After tuning local data movement, one can look beyond the traditional "peephole optimizations" and schedule data assignment and movement on the entire graph with data-centric transformations. Briefly, such transformations do not have a pattern to match, but view the entire graph and generate multiple options. One such optimization is `TransientReuse`, which detects arrays that have the same volume but are used in non-overlapping segments of time (using DAG level analysis). References to these arrays are replaced by a reference to a single memory region, and unused arrays are removed. Another potential example is finding an optimal set of data layouts for weights and intermediate values. Since DaCeML generates native SDFGs, it can generate optimized code for each layout, constrain the search space by ensuring layouts match, and prune it based on domain-specific constraints, e.g., by only considering BLAS-optimized layouts for Einsums.

Allocation and deallocation of memory can create significant overheads when happening within critical code. To avoid them, the SDFG provides fine-grained control over their lifetimes. For example, when storage is annotated as Persistent, it will outlive multiple SDFG invocations.

5.5 Hardware Specialization

Lastly, we need to consider the underlying system we are compiling for. This may mean the accelerator architecture(s), or whether we are running on one or more nodes.

GPU specialization DaCe transformations such as `GPUTransformSDFG` and `FPGATransformSDFG` can offload code to different platforms by introducing copies and kernel code (Ben-Nun et al., 2019). In DaCeML, we extend the transformation capabilities to partition workload efficiently on GPUs. `WarpTiling` is a variant of `MapTiling` that takes a GPU kernel map and divides its work across a warp. It detects reductions inside the map and inserts efficient warp-collective reductions as necessary. `Vectorization` allows using target-specific vector instructions. We extend the system to support reduced-precision vector types, provide fast implementations of vector-to-scalar reductions, and "fill in gaps" in math library functions by calling sequences of scalar operations.

Distributed computing DaCeML supports data-parallelism for distributed training. `DistDataParallel` adds a distributed `allreduce` library node after each weight gradient access node in the backward-pass SDFG. Since the code generator traverses SDFGs in topological order, communication is performed as soon as the data are ready, promoting pipelining.

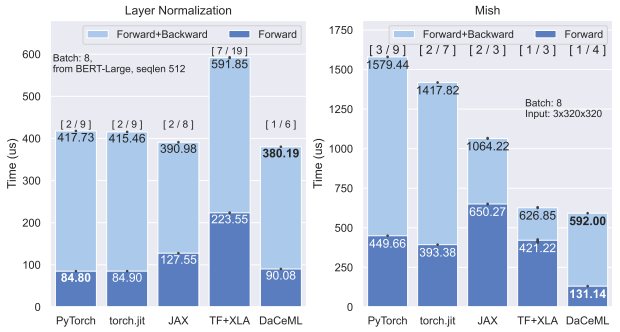


Figure 5. Runtime results for the NN building blocks. Brackets [·] indicate number of kernels launched in forward / fwd+bwd.

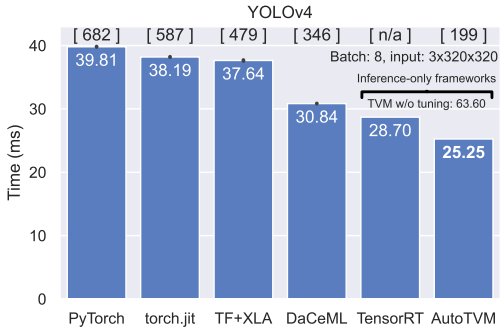


Figure 6. Runtime results for inference with YOLOv4. Brackets [·] indicate number of kernels launched. Due to profiler issues, we could not measure kernel counts for TensorRT.

5.6 Automatic Transformations

Now that we have established a set of transformations, we can create an automated sequence of their application for arbitrary DNNs. Our transformation recipe used in the results is as follows: (1) Clean up the ONNX-based SDFG with constant folding and algebraic fusion; (2) lower ONNX nodes to implementations, choosing from ONNXRuntime, cuDNN or PyTorch implementations depending on the node type; (3) in the resulting SDFG, inline nested SDFGs, remove all redundant copies, and greedily fuse subgraphs; (4) specialize the program to the hardware: in fused maps, hoist out initialization to kernel start, eliminate trivial (0, 1 element) maps, specialize memory types of transients and call `WarpTiling`; (5) flatten map dimensions to coalesce memory accesses, and vectorize maps to 128-bit accesses if applicable. When manually applying guided transformations, this recipe can be used as a starting point.

6 Evaluation

Experimental setup We run each experiment at least 100 times and report the median value with a 95% nonparametric confidence interval. For measurement, we use a server with an Intel Xeon Gold 6140 CPU (2.30GHz), 768 GiB RAM, and an NVIDIA Tesla V100 GPU (16 GiB RAM). We use Python 3.8.8, DaCe 0.10.8, CUDA 11.4, CUDNN 8.2.0.54. PyTorch 1.8.1, ONNXRuntime 1.7.0, TensorFlow 2.5.0, TensorRT 8.0.1.6, TVM 0.8.0dev0 (commit dbfbebe), and JAX 0.2.13. The torch-based frameworks were all run from the same model source code, unmodified except for the addition of a decorator for `torch.jit` and DaCeML.

6.1 Neural Network Operators

We demonstrate DaCeML's ability to optimize single operators, and then examine how these perform when used as part of a larger model.

Layer Normalization We first examine layer normalization (Ba et al., 2016), a primitive widely used in transformer models; see Figure 5 (left). The generated code outperforms all frameworks on the forward+backward pass, where JAX closely matches in backpropagation but misses several fusion opportunities. On the forward part, PyTorch is slightly faster due to DaCeML storing more information for backpropagation, which ends up being faster overall. Without this storage, DaCeML takes 49 μs (Figure 4). DaCeML's native SDFG is also fusible with neighboring operators, which delivers improved overall performance when used in larger models. In Appendix B.2, we show that the same transformations, which partition the work across GPU warps, apply to other classes of statistical normalization, e.g., softmax.

Mish activation YOLOv4 (Bochkovskiy et al., 2020) is a widely-used network for object detection. Among other features, it relies on the Mish (Misra, 2020) activation, $\mathrm{mish}(x) = x\tanh(\mathrm{softplus}(x))$, which is not natively supported by PyTorch. We evaluate Mish in isolation, and overall inference in YOLOv4 with Mish optimized.

Figure 5 (right) shows our Mish results. For forward evaluation, PyTorch produces three kernels: $\tanh$, $\mathrm{softplus}$, and multiplication. `torch.jit` and JAX are able to fuse two out of the three operations. DaCeML, however, is able to fuse the entire activation function, resulting in $3.43\times$ improvements over PyTorch and $3\times$ over `torch.jit`. In backpropagation, similar results hold, and we achieve $2.67\times$ improvements over PyTorch and $1.06\times$ over TF+XLA.

YOLOv4 Figure 6 lists inference results for YOLOv4 with the automated recipe (§ 5.6), which optimizes Mish in the context of the full network. The figure shows reduction in kernel counts, as well as 1.22–1.3$\times$ speedup over training frameworks. This nearly reaches the optimization levels of inference-only frameworks, such as TensorRT and TVM. Upon deeper inspection, the two frameworks make use of inference-specific optimizations, such as custom implementations for convolutions and pre-transforming weights. As DaCeML is designed to optimize training workloads, such techniques are outside the scope of this work.

		PyTorch		torch.jit		JAX		TF+XLA		DaCeML	
		→	⇄	→	⇄	→	⇄	→	⇄	→	⇄
Guided	EfficientNet (👁)	2.05	6.90	2.04	6.94	2.39	7.40	1.54	6.37	**1.40**	**5.97**
	$\text{BERT}_{\text{LARGE}}$ (mixed) (🤖)	2.94	8.18	2.92	8.20	3.19	8.11	3.80	10.76	**2.74**	**7.62**
Automatic	ResNet-50 (👁)	14.55	32.04	**9.98**	**31.94**	14.17	33.93	12.33	35.57	10.03	32.45
	Wide ResNet-50-2 (👁)	22.50	70.94	22.45	70.83	40.49	98.13	32.79	99.06	**20.62**	**67.99**
	MobileNet V2 (👁)	9.98	18.45	6.22	15.53	—	—	7.42	20.29	**4.74**	**14.77**
	EfficientNet (👁)	2.05	6.90	2.04	6.94	2.39	7.40	**1.54**	**6.37**	1.57	15.00
	MLP Mixer (🙈)	1.63	**3.65**	**1.36**	3.66	1.77	4.01	—	—	1.48	4.25
	FCN8s (🧩)	46.85	158.42	46.82	**158.40**	—	—	—	—	**45.97**	166.30
	WaveNet (🔊)	23.21	46.39	**18.67**	41.49	—	—	—	—	26.16	**41.07**
	$\text{BERT}_{\text{LARGE}}$ (single) (🤖)	11.05	31.76	11.05	31.82	**10.93**	**29.94**	11.14	38.73	11.44	32.98
	$\text{BERT}_{\text{LARGE}}$ (mixed) (🤖)	2.94	8.18	**2.92**	8.20	3.19	**8.11**	3.80	10.76	3.34	9.25
	DLRM (📈)	118.07	126.55	**117.38**	126.83	—	—	—	—	117.69	**126.42**

Table 1. Median runtime for the forward (→) and forward + backward (⇄) passes for convolutional vision (👁), non-convolutional vision (🙈), audio (🔊), image segmentation (🧩), transformer (🤖) and recommendation system (📈) models. $\text{BERT}_{\text{LARGE}}$ reports for a single encoder layer; EfficientNet for the MBConv 1 layer. A — indicates we could not find an implementation for the model.

6.2 Automatic Optimization Recipe

While DaCeML's major strength lies in its manual tuning capabilities, the automated transformation recipe from Section 5.6 already yields performance improvements over state-of-the-art frameworks. We demonstrate networks from different domains, with varying data movement patterns, as well as utilizing different hardware units (e.g., tensor cores). We run the following models with the recipe alone, and list the results in Table 1 (bottom): ResNet-50 (He et al., 2015), Wide ResNet-50-2 (Zagoruyko & Komodakis, 2017), MobileNet V2 (Sandler et al., 2019), EfficientNet-B0's MBConv block (Tan & Le, 2019), MLP Mixer (Tolstikhin et al., 2021), Fully Convolutional Network (Long et al., 2015), WaveNet (van den Oord et al., 2016), BERT (Devlin et al., 2019) encoder block in single (32-bit) and mixed (16-bit) precision, and the DLRM (Naumov et al., 2019) recommendation system. An interested ML practitioner could then use DaCeML to further optimize performance as necessary.

The table shows that no single framework operates best across all DNNs. Additionally, with the automatic optimizations alone, we see that DaCeML roughly matches and in multiple cases outperforms the other compiler frameworks. This is especially interesting in variants of popular networks, such as Wide ResNets. DaCeML is roughly 2× slower on Wide ResNet-50-2 than on ResNet-50, as expected for performing twice the computations; yet other frameworks are up to 2.89× slower. This potentially indicates specialization for certain operators and sizes, which does not occur with the data-centric transformations in DaCeML.

We now proceed with two case studies that highlight the possibilities enabled by DaCeML's user-guided optimization workflow, where the results are summarized in Table 1 (top).

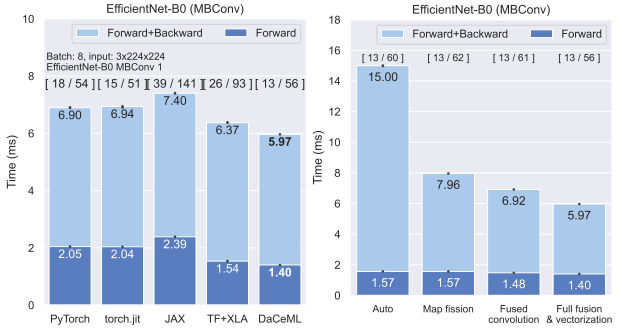


Figure 7. Runtime results (left) and transformation breakdown (right) of the EfficientNet-B0 guided optimization case study.

6.3 Guided Optimization Case Study: EfficientNet

In this case study, we consider EfficientNet-B0 (Tan & Le, 2019). Since EfficientNets use repeated blocks, we can optimize one such block and reuse the techniques for the rest of the network. We focus on the first MBConv block, where performance results are reported in Figure 7, as well as the progressive improvements gained by various optimizations, using the automatic recipe as a base that was further improved. For this case study, we perform the guided optimizations using the Visual Studio Code plugin (see Figure 8), and report the number of clicks performed in the UI.

Map fission (6 clicks) We use two fission transformations to split a map in the backward pass. This allows us to avoid atomic operations by swapping iteration order and accumulating into thread-local memory.

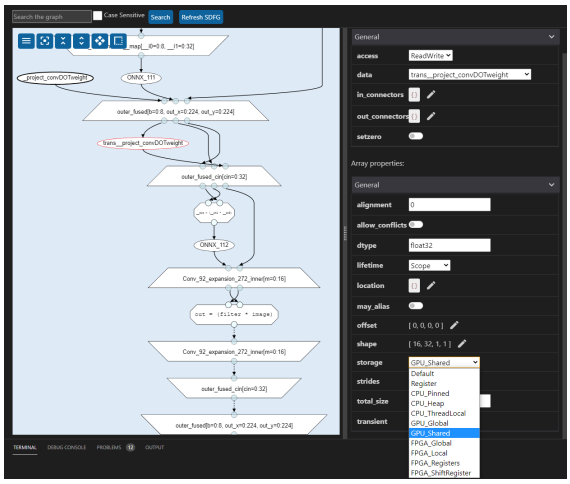


Figure 8. Guided EfficientNet optimization in Visual Studio Code.

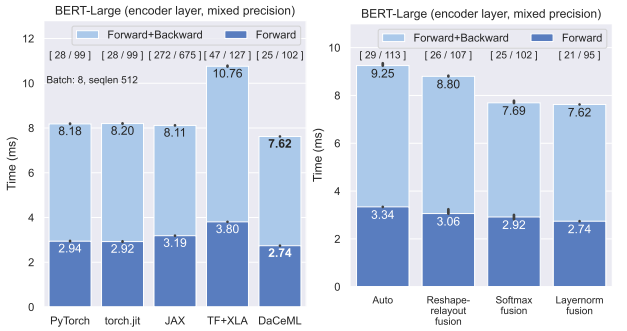


Figure 9. Runtime results (left) and transformation breakdown (right) of the $\text{BERT}_{\text{LARGE}}$ guided optimization case study.

Fused convolution (10 clicks) We discover a new fusion opportunity by expanding the convolutional operators to native SDFG, rather than cuDNN. Finding this automatically is non-trivial, requiring an exchange of the loop iteration variables in two operators to enable fusion. The data-centric GUI aids in this process—viewing data movement volumes in the graph highlights a potential bottleneck even without running the program. This fused convolution nets a $\sim$1.33$\times$ speedup over cuDNN and PyTorch's hand-optimized versions in the forward pass.

Full fusion and vectorization (2 clicks) The map fission performed enables further fusion in the backward pass. We also tune the backpropagation to recompute intermediate values rather than loading and storing them. This is done by applying `TaskletFusion` before the AD engine runs (in 27 lines of code). We fully fuse, flatten and vectorize the maps where possible.

6.4 Guided Optimization Case Study: BERT

Our second case study is the widely-used BERT (Devlin et al., 2019) transformer. We optimize the $\text{BERT}_{\text{LARGE}}$ architecture with a batch size of 8 and sequence length 512. We run in mixed precision: here the advantages of the data-centric optimization are more pronounced as data movement becomes important. For example, we already see in Figure 9 that TF+XLA exhibits slower performance due to suboptimal data layouts that yield slower tensor contractions.

All transformations are performed using the Python API starting from the automatic recipe; we report lines of code.

Reshape-relayout Fusion (25 lines) After applying the algebraic fusion heuristic and forced BLAS operation generation, we observed extra relayout (transposition) calls with data reshapes. We write a transformation that detects this pattern and fuses the output write into the prior (or subsequent) computation, if elementwise. This applies six times (thrice forward, thrice backward) in the graph.

Softmax Fusion (39 lines) Instead of calling a pre-optimized library for softmax, we decide to "break out of the library jail" and expand it to the native SDFG, followed by applying the recipe in Appendix B.2.1. This results in fusing scaling into softmax, at a 27 μs overhead instead of 223 μs in the forward pass, and 1.1 ms gain in total.

Layer-normalization Fusion (47 lines) Here we use the lifted layer normalization scheme to nest the preceding linear layer bias into normalization. We extend and adapt the lifting transformation to include the prior/subsequent operations. For the backpropagation, since bias is reduced into 1,024 elements, we use DaCe's warp-based reduction schedule and combine it with the layer normalization weight gradient kernel, which has the same iteration space.

In both case studies, the resulting code outperforms all compiler infrastructures. This demonstrates the strength of guided data-centric optimization — inspecting complex DNN models from a bird's eye view for data movement bottlenecks, and mitigating them via transformations.

7 Conclusion

We explore the concept of data-centric optimization for deep learning with DaCeML. The framework enables general-purpose data layout and movement transformations to be applied on arbitrary networks written in PyTorch, matching and outperforming state-of-the-art compilers. The two key insights of the data-centric view are focusing on data movement minimization based on memory access patterns rather than operator types, and allowing practitioners to further tune global and local movement. The former can perform a superset of the optimizations applied by DNN compilers; and the latter can turn massive engineering effort into a click of a button in the analysis and transformation UI. Either automatic or human-in-the-loop, DaCeML helps practitioners speed up training without sacrificing productivity.

ACKNOWLEDGEMENTS

This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreements DAPP, No. 678880; EPiGRAM-HS, No. 801039; MAELSTROM, No. 955513; and DEEP-SEA, No. 955606). T.B.N. is supported by the Swiss National Science Foundation (Ambizione Project #185778). N.D. is supported by the ETH Postdoctoral Fellowship. The authors wish to acknowledge the support from the PASC program (Platform for Advanced Scientific Computing), as well as the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Baghdadi, R., Debbagh, A. N., Abdous, K., Zohra, B. F., Renda, A., Frankle, J. E., Carbin, M., and Amarasinghe, S. TIRAMISU: A polyhedral compiler for dense and sparse deep learning. In *Workshop on Systems for ML at NeurIPS*, 2019.

Barham, P. and Isard, M. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019.

Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T., and Hoefler, T. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. YOLOv4: Optimal speed and accuracy of object detection, 2020.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *Siam Review*, 60(2), 2018.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An end-to-end optimization stack for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., et al. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

de Fine Licht, J., Kuster, A., De Matteis, T., Ben-Nun, T., Hofer, D., and Hoefler, T. StencilFlow: Mapping large stencil programs to distributed spatial computing systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

Dong, X., Liu, L., Zhao, P., Li, G., Li, J., Wang, X., and Feng, X. Acorns: A framework for accelerating deep neural networks with input sparsity. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

Elango, V., Rubin, N., Ravishankar, M., Sandanagobalane, H., and Grover, V. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.

Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

Google. XLA: Optimizing compiler for machine learning, 2021. URL https://www.tensorflow.org/xla.

Halevy, A., Norvig, P., and Pereira, F. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2), 2009.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

Hooker, S. The hardware lottery. *arXiv preprint arXiv:2009.06489*, 2020.

Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., and Wang, Y. FeatGraph: A flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

Intel. oneAPI deep neural network library, 2021a. URL https://01.org/oneDNN.

Intel. Open visual inference and neural network optimization toolkit, 2021b. URL https://01.org/openvinotoolkit.

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. Data movement is all you need: A case study on optimizing transformers. In *Proceedings of the Fourth Conference on Machine Learning and Systems (MLSys)*, 2021.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Keutzer, K., Stoica, I., and Gonzalez, J. E. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of the Third Conference on Machine Learning and Systems (MLSys)*, 2020.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019a.

Jia, Z., Thomas, J., Warszawski, T., Gao, M., Zaharia, M., and Aiken, A. Optimizing DNN computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning (SysML)*, 2019b.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

Lethin, R. Polyhedral optimization of tensorflow computation graphs. In *Sixth Workshop on Extreme-scale Programming Tools (ESPT)*, 2017.

Li, C., Yang, Y., Feng, M., Chakradhar, S., and Zhou, H. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 633–644. IEEE, 2016.

Li, S., Ben-Nun, T., Girolamo, S. D., Alistarh, D., and Hoefler, T. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pp. 45–61, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368186. doi: 10.1145/3332466.3374528. URL https://doi.org/10.1145/3332466.3374528.

Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. SymPy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

Microsoft. DeepSpeed, 2020. URL deepspeed.ai.

Microsoft. ONNX Runtime, 2021. URL https://www.onnxruntime.ai.

Microsoft. Visual Studio Code - Code editing. Refined. https://code.visualstudio.com/, 2021.

Misra, D. Mish: A self regularized non-monotonic activation function. In *Proceedings of the 31st British Machine Vision Conference (BMVC)*, 2020.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems, 2019.

NVIDIA. NVIDIA TensorRT, 2021. URL https://developer.nvidia.com/tensorrt.

ONNX. ONNX: Open neural network exchange, 2021. URL https://onnx.ai/.

Oyama, Y., Ben-Nun, T., Hoefler, T., and Matsuoka, S. Accelerating deep learning frameworks with micro-batches. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., and Dean, J. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021.

PyTorch Team. TorchScript, 2020. URL https://pytorch.org/docs/stable/jit.html.

Raina, R., Madhavan, A., and Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning (ICLR)*, 2009.

Roeder, L. Netron, 2021. URL https://netron.app/.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

Sivathanu, M., Chugh, T., Singapuram, S. S., and Zhou, L. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

Steiner, B., Cummins, C., He, H., and Leather, H. Value function based performance optimization of deep learning workloads, 2020.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.

Sun, C., Shrivastava, A., Singh, S., and Gupta, A. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2017.

Tan, M. and Le, Q. EfficientNet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pp. 6105–6114. PMLR, 2019.

Tolstikhin, I., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., and Dosovitskiy, A. Mlp-mixer: An all-mlp architecture for vision, 2021.

Truong, L., Barik, R., Totoni, E., Liu, H., Markley, C., Fox, A., and Shpeisman, T. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. Wavenet: A generative model for raw audio, 2016.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Venkat, A., Rusira, T., Barik, R., Hall, M., and Truong, L. SWIRL: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications*, 33(6), 2019.

Wei, R., Schwartz, L., and Adve, V. DLVM: A modern compiler infrastructure for deep learning systems. In *Proceedings of the Sixth International Conference on Learning Representations - Workshop Track (ICLR)*, 2018.

Yang, Y., Phothilimtha, P. M., Wang, Y. R., Willsey, M., Roy, S., and Pienaar, J. Equality saturation for tensor graph superoptimization, 2021.

Zagoruyko, S. and Komodakis, N. Wide residual networks, 2017.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

Ziogas, A. N., Ben-Nun, T., Fernández, G. I., Schneider, T., Luisier, M., and Hoefler, T. Optimizing the data movement in quantum transport simulations via data-centric parallel programming. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

A SYMBOLIC GRAPH ANALYSIS

In Figure 10, we can see an `MBConv` block of EfficientNet-B0, before and after cleanup transformations. As Figure 10 shows, the PyTorch ONNX exporter generates many extraneous computations, casting, and `Unsqueezes` that precede certain shape computations. While this is necessary to compute the block sizes, it generates many calls that can be hoisted out of the computation. Only after applying this transformation, a new transformation is exposed, `PadConvFusion`, which can fuse the convolution padding dimensions into the operator itself.

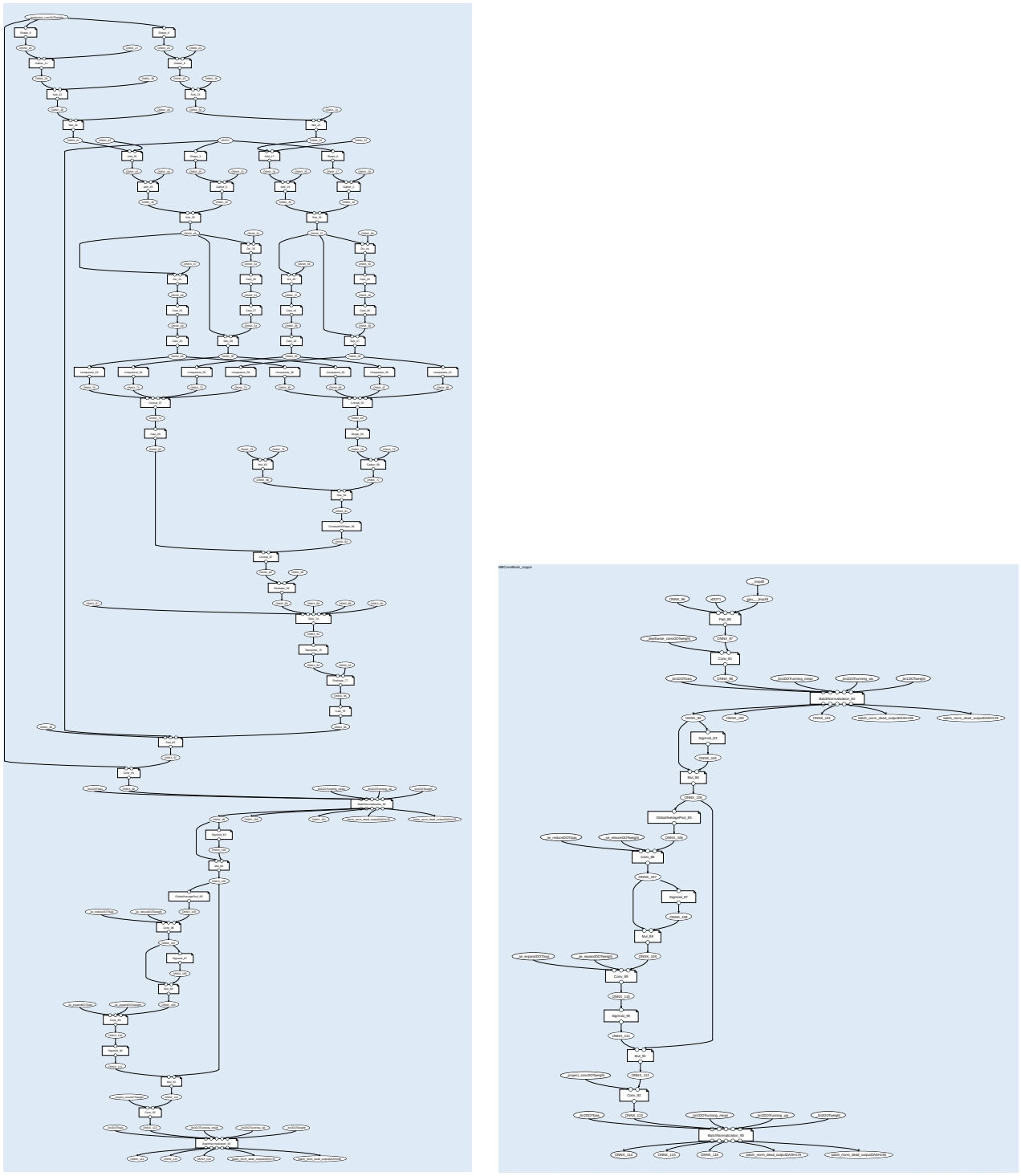


Figure 10. EfficientNet-B0 `MBConv` block before (left) and after (right) constant folding.

B DaCeML Transformation Recipe

B.1 Mish Activation

We optimize the Mish operator (Misra, 2020), a novel activation function that, among other uses, has been applied successfully in object detection (Bochkovskiy et al., 2020). Due to it being recent, it has not yet been implemented as a built-in activation in PyTorch, ONNX, or ONNXRuntime. We demonstrate how DaCeML can be used to optimize this operator.

We begin with code for the PyTorch Module, and import it into DaCeML by annotating it with the `@dace_module` decorator.

```
import torch
from torch import nn
from torch.nn import functional as F
from daceml.pytorch import dace_module

@dace_module(backward=True, auto_optimize=False)
class DaCeMish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x * (torch.tanh(F.softplus(x)))
        return x
```

The module works immediately with DaCeML for the forward and backward pass, and the unoptimized graphs (due to the `auto_optimize=False` flag) can be seen in Figure 11.

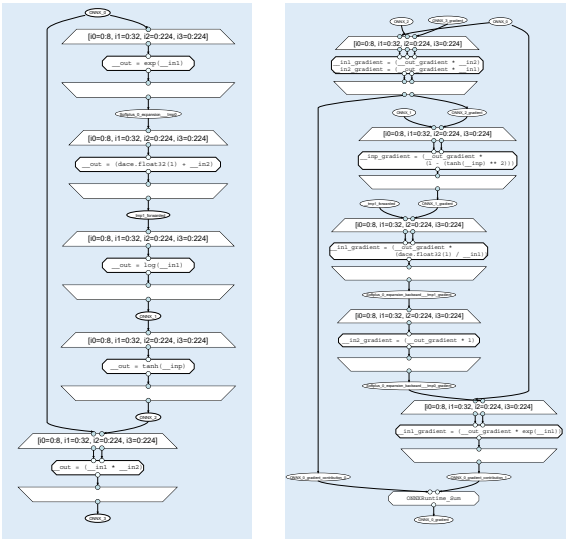


Figure 11. Unoptimized Mish activation forward-pass (left) and backward-pass (right) SDFGs.

Notice the bold outline on the intermediate arrays. This means that they are global, and are thus all retained for the backward pass. This is standard behavior for DNN compilers that cannot control both the forward and backward passes. This is an optimization opportunity for DaCeML: the AD engine is "forwarding" intermediate values to perform the differentiation, and we can avoid that by fusion.

B.1.1 Optimization

We can now run the full recipe, as described in Section 5. In particular, to demonstrate the API we will show how to programmatically call only a part of that recipe to optimize the operator:

```
from daceml.transformation import TaskletFusion
from dace.transformation.dataflow import (Vectorization,
    TrivialMapRangeElimination)
from dace.transformation.subgraph import SubgraphFusion
from daceml import onnx as donnx

dace_mish = DaCeMish()

# Expand the ONNX nodes and inline nested SDFGs
def expand_and_strict_transforms(module):
    with change_default(donnx, "pure"):
        utils.auto_optimize(module.sdfg,
                            apply_strict=True)
dace_mish.append_post_onnx_hook(
    "expand", expand_and_strict_transforms)

# Apply subgraph fusion to all nodes
def fuse_sg(module):
    module.sdfg.apply_transformations_repeated(
        TrivialMapRangeElimination)
    SubgraphFusion.apply_to(
        module.sdfg, *module.sdfg.node(0).nodes())
dace_mish.append_post_onnx_hook("subgraph_fusion",
                                fuse_sg)

# Apply tasklet fusion to recompute intermediate values
dace_mish.append_post_onnx_hook("fuse_tasklets",
    lambda mod:\
        mod.sdfg.apply_transformations_repeated(
            TaskletFusion))

# Apply vectorization
def vectorize(fwd, bwd):
    fwd.apply_transformations(Vectorization)
    bwd.apply_transformations(Vectorization)
dace_mish.append_post_autodiff_hook("vectorize",
                                    vectorize)
```

The resulting graphs generated are shown in Figure 12. All expressions in the backward tasklet were automatically generated by SymPy. Also note the change in map range indicating vectorization. The graphs in Section 6 are identical to those in the figure.

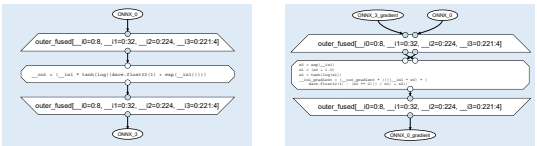


Figure 12. Mish SDFGs after optimization (forward: left, backward: right).

B.2 Statistical Normalization

Statistical normalization operations (e.g., Batch Normalization, Layer Normalization, Group Normalization) are

commonplace in modern DNNs. For example, BERT uses layer normalization twice in each encoder/decoder block. Despite the fact that all normalization operations are similar, their performance varies. All the aforementioned operations perform the general computation $\gamma \cdot \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}(x) + \epsilon}} + \beta$, but on different dimensions or subsets thereof.

We find, for example, that PyTorch has two separate implementations for layer and batch normalization. The implementation of batch normalization can use cuDNN, while layer normalization uses a manually-optimized function. Thus, despite batch normalization being more expensive than layer normalization (due to saving running mean/variance statistics to memory), the former is faster than the latter. Moreover, when exporting models from PyTorch to ONNX, since layer normalization is not an ONNX-native operator, it is split at arbitrary positions and a certain order of operations (mean computation followed by variance computation from the result of the mean) is enforced.

Here we show how our transformation recipe from Section 5 operates on layer normalization, and also applies to a more general form of statistical normalization — the Softmax operator.

B.2.1 Softmax

We start with a detailed account of automatically applying the recipe on the Softmax operator. First, DaCeML provides a numpy implementation of the operator itself (see Figure 13). This naive implementation, which corresponds to the more numerically-stable version of softmax, lowers the ONNX library node to four separate operations: two reductions (computing the maximum value to subtract; and summarizing the contributions for the denominator), and two elementwise operations (exponentiation, division). In the current mode, three extraneous tensors sized as the input would be generated. We can thus apply the second step of our recipe and locally reduce this data movement.

Fusion of those operators is not trivial, as the dimensions of the maps and reductions differ. The `greedy_fuse` method in DaCe automatically expands reductions (via `ReduceExpansion`) and finds common dimensions to extract out of the maps in the subgraph, which in turn allows to fuse them all to one map. This stores all memory locally, within the map. The end result is shown on Figure 14, on the top-left SDFG.

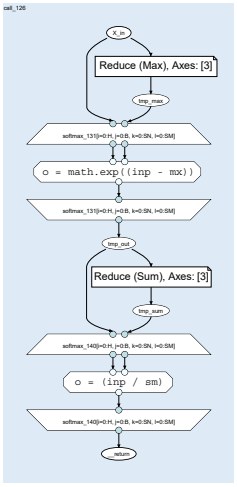
As the global data movement optimization step does not apply in this case, the recipe proceeds to specialize the code to the GPU. The code gradually lowers the representation (expanding the two reductions to nested SDFGs), transforms the map to be executed on a GPU (via `GPUTransformSDFG`), and calls `WarpTiling` to partition the work in the internal maps across warps. Note that

```python
@op_implementation(op="Softmax", name="pure")
class PureSoftmax(ONNXForward):
    @staticmethod
    def forward(node, state, sdfg):
        # Get properties and shapes
        axis = node.axis
        reduced_shape = list(copy.deepcopy(
          in_desc_with_name(node, state, sdfg,
                            "input").shape))
        reduced_shape[axis] = 1

        # Operator implementation in numpy
        def prog(input, output):
            max = np.max(input, axis=axis)
            max_keepdims = np.reshape(max, reduced_shape)
            exp_arr = np.exp(input - max_keepdims)
            sum = np.sum(exp_arr, axis=axis)
            sum_keepdims = np.reshape(sum, reduced_shape)
            output[:] = exp_arr / sum_keepdims

        return program_for_node(prog, sdfg, state, node)
```

(a) ONNX Library Node Implementation



(b) Resulting SDFG

Figure 13. Softmax implementation in DaCeML.

this automatically generates warp-level reductions, and by default *replicates* all computations that have multiple results dependent on reductions (the latter is configurable). This is shown in the top-right SDFG in Figure 14. Lastly, the SDFG is cleaned up via another pass of transformations — `HoistState` to move initialization up, another batch of map fusion, and vectorization (shown in the bottom graph of the figure).

The resulting performance matches the performance of state-of-the-art hand-written implementations, such as the PyTorch built-in softmax implementation.

B.2.2 Layer Normalization

The same exact recipe that was applied in Softmax can be applied to the Layer Normalization operator. In Figure 15 we can see the naive expansion of the PyTorch/ONNX op-

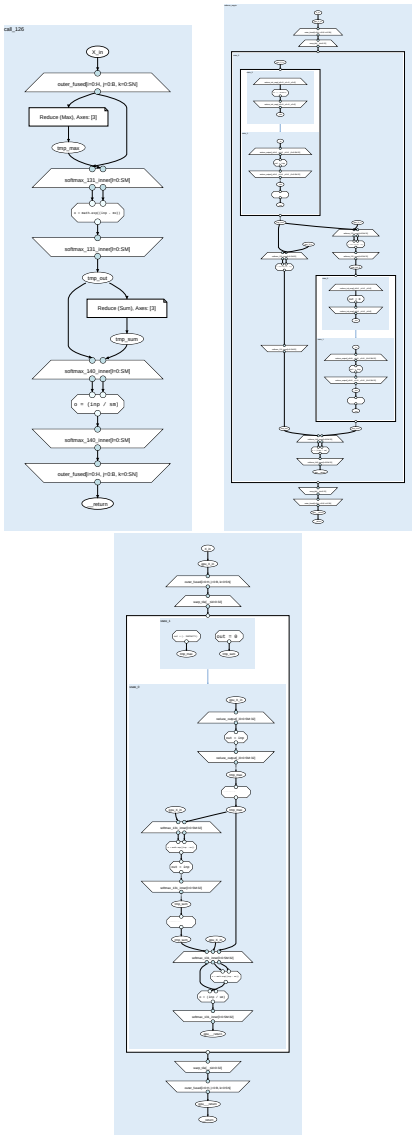


Figure 14. Softmax SDFG transformations: after subgraph fusion (top left), warp tiling (top right), cleanup and vectorization (bottom).

erator on the left-hand side, and an optimized graph on the right, which acts as the default expansion of the lifted `LayerNormalization` operator (from Figure 4).

In the figure, the transformed version is similar to the softmax code, with one important *algorithmic* distinction: the formula used for variance computation is $\mathbb{E}[x^2] - \mathbb{E}[x]^2$, which can be computed in parallel, as the graph shows. This is a direct result of domain knowledge in lowering. Also important with such an expansion is mixed-precision operation: since the input/output data types are known at lowering, we can control the intermediate data types, e.g., using 32-bit accumulators when 16-bit precision is involved.

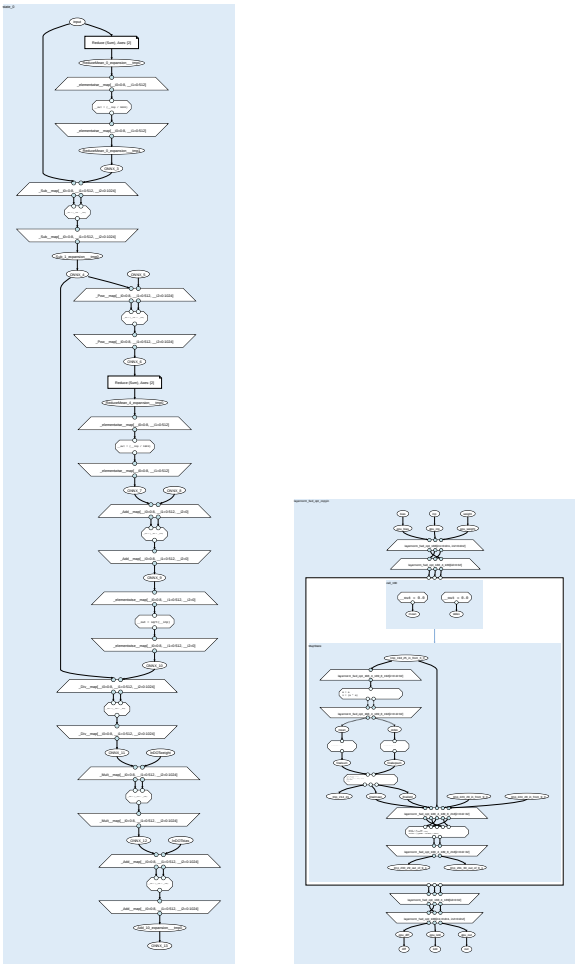


Figure 15. Layer Normalization SDFG transformations: naive (left) and optimized (right).

B.3 EfficientNet MBConv Block

To optimize the building block of the EfficientNet architecture, we use the popular PyTorch implementation from the `efficientnet_pytorch` PyPI package[2]. To import the block, we make no code modifications, and simply wrap the module using the DaceModule class (equivalent to the decorator).

```
dace_model = MBConvBlock(block_params[0],
                         global_params).cuda()
# ...
dace_model = DaceModule(dace_model, cuda=True)
```

To optimize this model, we perform the same recipe as in the Mish case (map fusion, tasklet fusion and vectorization). In particular, DaCeML fuses and vectorizes the Sigmoid and Mul operators, which form the Swish activation, since the Swish operator does not yet exist in ONNX.

[2] https://github.com/lukemelas/EfficientNet-PyTorch

The exported PyTorch implementation contains a large subgraph consisting of shape-based computation of the padding size for a Pad operator (see Figure 10). Using the information from the symbolic shape inference and the constant folding transformation, we are able to eliminate this subgraph to statically know the padding shape. Following constant folding, we are able to apply a high-level Pad-Conv fusion transformation that fuses the Pad operator into Conv by using the statically known padding sizes, and updating the `pads` attribute on the convolution operator (transformation implementation listed in Figure 17).

Following the automated transformation procedure, we perform interactive optimization using Visual Studio Code.

C TRANSFORMATION EXAMPLES

C.1 Algebraic Fusion through Einsums

The transformation `VerticalEinsumFusion`, described in Section 5.1, is shown in Figure 16. Its detection pattern is a chain of two Einsum nodes, and if the dimensions match, the replacement is a single, fused Einsum node. This pattern appears in several workloads where tensors with over two dimensions are used, transformers for example. It may result from code that uses `.permute()` or `.transpose()` operators, or may be generated automatically in some cases of the dot product operator.

In the transformation, since the pattern subgraph is connected, the tensor dimensions must match, but the index letters do not. Thus, we first parse the two expressions and make the test, keeping a mapping dictionary between one and the other. The replacement simply takes the output of the first Einsum and maps the characters to the one of the second Einsum, removing the access node and the original Einsum. Currently the only possible transformations are ones with one input (i.e., no contraction), and that do not change the dimensionality of the tensor, i.e., that the number of characters before and after the `->` matches. However, it is trivially possible to extend the matching condition to expansions, e.g., `ij->ijj`.

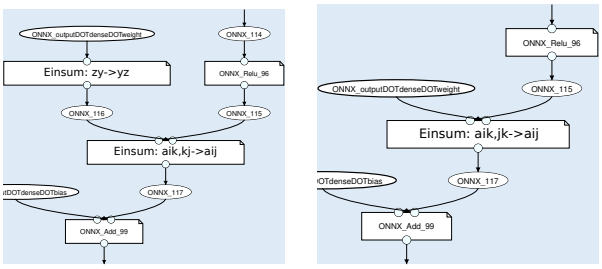


Figure 16. Vertical (dependent) Einsum fusion in action. On the left is the SDFG before transformation and on the right is the graph after transformation.

C.2 Replacement Transformations

In Figures 17–18, we see two additional examples of pattern-matching replacement transformations, along with their full source code.

As mentioned in Appendix B.3, `PadConvFusion` takes two consecutive Pad and Conv nodes, and modifies the Conv operator to include padding. In the implementation (Figure 17), we can see that the subgraph pattern definition (line 5) is straightforward — a path graph between the two nodes. The matching condition is extended beyond subgraph matching (lines 8–25), checking first that the subgraph matches using the superclass (lines 10–11), and then that the two values are foldable, i.e., not computed by the DNN itself (lines 14–21). Then, it checks the validity of the padding values themselves: whether the fill value is zero (in lines 22–23), and whether the padding is performed on the spatial dimensions of the sample, which is the only case supported by fast libraries (lines 24–32). The replacement part (lines 34–46) is straightforward as well: the `pads` values of the Conv node are modified according to the Pad node's values (lines 35–42). Lastly, to reconnect the graph (line 46), only the Conv node is returned, and the input (`X`) is reconnected to it directly, using the memlet that was connected to the Pad node's `data` input.

```
1  class PadConvFusion(ReplacementTransformation):
2    """ Fuse a constant pad into a convolution. """
3    @classmethod
4    def pattern(cls):
5      return make_onnx_path(onnx_op.ONNXPad,
6                            onnx_op.ONNXConv)
7
8    def can_be_applied(self, graph, candidate,
9                       expr_index, sdfg, strict):
10     if not super().can_be_applied(
11         graph, candidate, expr_index, sdfg, strict):
12       return False
13     # Check that the two pad inputs can be folded
14     cval = onnx_constant_or_none(sdfg,
15                                  self._pnode0(sdfg),
16                                  "constant_value")
17     pads = onnx_constant_or_none(sdfg,
18                                  self._pnode0(sdfg),
19                                  "pads")
20     if cval is None or pads is None:
21       return False
22     if cval != 0:
23       return False
24     if len(pads.shape) != 1 or pads.shape[0] != 8:
25       return False
26
27     # We can only eliminate the pad if it is along
28     # the spatial axes
29     if (not iterables_equal(pads[0::4], [0, 0])
30         and iterables_equal(pads[1::4], [0, 0])):
31       return False
32     return True
33
34   def replacement(self, path, sdfg, state):
35     pad: onnx_op.ONNXPad = path[0]
36     conv: onnx_op.ONNXConv = path[1]
37
38     pads = onnx_constant_or_none(sdfg, pad, "pads")
39     conv.pads[0] += int(pads[2::4][0])
40     conv.pads[2] += int(pads[2::4][1])
41     conv.pads[1] += int(pads[3::4][0])
42     conv.pads[3] += int(pads[3::4][1])
43
44     # Return the original convolution node and
45     # reconnect input
46     return (conv, dict(X=(pad, "data")))
```

Figure 17. Pad-Convolution Fusion, as used in the EfficientNet case study.

Figure 18 lists the code for a transformation that fuses matrix multiplications with a scalar multiplication that follows. In optimized BLAS libraries, this capability is directly supported in the GEneral Matrix-Matrix (GEMM) multiplication operator. Since this pattern occurs in multi-head attention in transformers, we can simply add a general-purpose transformation that would apply there as well. The pattern

in this case is a MatMul operator followed by a Div (or Mul) operator (line 4). If the scalar can be folded (line 19), the matrix multiplication node is replaced with a different Gemm ONNX operator node, which contains those scaling values. We set the value of α to the new scale (lines 23–30) and replace the node, reconnecting the inputs of the MatMul node into `A,B` and the outputs of the scaling to `Y` (lines 33–36). The resulting code is short, and more such use cases can be added with ease.

```
1  class MatmulScal(ReplacementTransformation):
2    @classmethod
3    def pattern(cls):
4      return make_onnx_path(onnx_op.ONNXMatMul,
5                            onnx_op.ONNXDiv)
6
7    def can_be_applied(self, graph, candidate,
8                       expr_index, sdfg, strict):
9      if not super().can_be_applied(
10         graph, candidate, expr_index, sdfg, strict):
11       return False
12
13     # Find other scaling input
14     other_node: nodes.AccessNode = next(
15         n for n in graph.predecessors(
16             self._pnode1(sdfg))
17         if n is not self._pnode2(sdfg))
18     # Check if the scalar can be folded
19     return onnx_constant_or_none(
20         sdfg, other_node) is not None
21
22   def replacement(self, path, sdfg, state):
23     other_node: nodes.AccessNode = next(
24         n for n in state.predecessors(path[1])
25         if n is not path[2])
26     scale = 1.0 / onnx_constant_or_none(other_node)
27
28     node = onnx_op.ONNXGemm('fused_gemm_div',
29                             alpha=scale,
30                             beta=0)
31     node.schedule = path[0].schedule
32
33     return (node,
34             dict(A=(path[0], 'A'),
35                  B=(path[0], 'B'),
36                  Y=(path[1], 'C')))
```

Figure 18. Matrix Multiplication Scaling transformation, as used in transformers.

C.3 Distributed data-parallelism transformation

Figure 19 presents an example of the `DistDataParallel` transformation, discussed in Section 5.5. The transformation detects a pattern of access nodes for weight gradients, which do not have output edges. Then, it adds a distributed `allreduce` library node after each weight gradient access node. During training, the gradients are aggregated across the workers by `allreduce`, implementing data-parallelism for distributed training. As mentioned, since the reduction operations are part of the generated code, they are inherently interleaved with computations, automatically creating overlap.

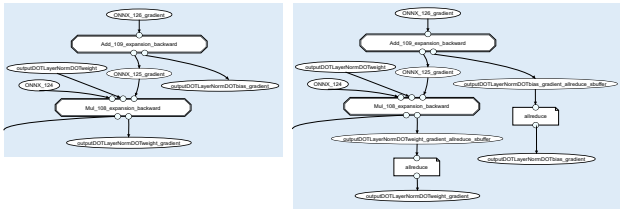


Figure 19. The effect of the `DistDataParallel` transformation for a part of a BERT encoder layer. On the left is the SDFG before transformation and on the right is the SDFG after transformation.