Compiling Deep Learning Workloads for Distributed Computing Clusters

Candidate no. 1065328

Thesis word count: 13073

Submitted in partial completion of the
Master of Science in Advanced Computer Science

30th August 2022

Abstract

Compute required to train deep learning models has experienced exponential growth over the past decade. Hardware has not been able to keep up with these demands, and as a result, training models on tens to thousands of accelerators has become the norm.

Partitioning a workload for such a cluster is a challenging problem. Existing approaches rely on the composition of manually specified parallel algorithms for each supported operator, limiting generality.

In this work, we propose a system for automatic partitioning of generic array programs. Our approach partitions workloads based on work rather than data, enabling parallelism beyond tensor dimensions.

Given a partitioning of the workload iteration space, our system automatically extracts communication constraints, which are then solved using one of several communication solvers. This eliminates the need operator-specific annotations, and enables automatic discovery of parallel algorithms. This makes it, to the best of our knowledge, the first such system that is able to generalize to a large class of array programs.

Our system compiles Python-based (NumPy or PyTorch) array programs to hardware-optimized executables that communicate using MPI. We demonstrate correctness on several workloads, including execution of a program across a cluster of 128 machines on the Piz Daint supercomputer.

Compiling Deep Learning Workloads for Distributed Computing Clusters

Candidate no. 1065328

Word count estimate: 13073

Submitted in partial completion of the

Master of Science in Advanced Computer Science

☑ I hereby certify that this is entirely my own work unless otherwise stated.

30th August 2022

Abstract

Compute required to train deep learning models has experienced exponential growth over the past decade. Hardware has not been able to keep up with these demands, and as a result, training models on tens to thousands of accelerators has become the norm.

Partitioning a workload for such a cluster is a challenging problem. Existing approaches rely on the composition of manually specified parallel algorithms for each supported operator, limiting generality.

In this work, we propose a system for automatic partitioning of generic array programs. Our approach partitions workloads based on work rather than data, enabling parallelism beyond tensor dimensions.

Given a partitioning of the workload iteration space, our system automatically extracts communication constraints, which are then solved using one of several communication solvers. This eliminates the need operator-specific annotations, and enables automatic discovery of parallel algorithms. This makes it, to the best of our knowledge, the first such system that is able to generalize to a large class of array programs.

Our system compiles Python-based (NumPy or PyTorch) array programs to hardware-optimized executables that communicate using MPI. We demonstrate correctness on several workloads, including execution of a program across a cluster of 128 machines on the Piz Daint supercomputer.

Contents

List of Figures

List of Abbreviations

(D)NN . . . . (Deep) Neural Net

DL . . . . . Deep learning

ML . . . . . Machine learning

AI . . . . . Artificial intelligence

SDFG . . . . . Stateful dataFlow multiGraph

IR . . . . . Intermediate representation

FLOP . . . . . Floating point operation

GPU . . . . . Graphics processing unit

ONNX . . . . Open Neural Network eXchange

AST . . . . . Abstract syntax tree

JIT . . . . . Just in time

DSL . . . . . Domain specific language

HPC . . . . . High performance computing

SPMD . . . . . Single program, multiple Data

MPI . . . . . Message passing interface

1 Introduction

The rise of Deep learning (DL) is one of the most significant developments of computer science in the last decade [LeCun et al., 2015]. Unlike other computer programs, these Deep Neural Networks (DNNs) require immense compute resources to *train* during development.

One of the driving discoveries of this revolution has been the interaction between compute and model generalization ability. Models with more parameters perform better [Neyshabur et al., 2015; Belkin et al., 2019; Advani et al., 2020], and the relation between compute and model performance has continued to hold in the face of orders of magnitude of growth [Kaplan et al., 2020; Rae et al., 2021].

As a result, since 2010, the amount of compute required to train the largest DL models has grown exponentially, doubling every 5.7 months [Sevilla et al., 2022] (see Figure 1.1). In this era of scaling, seminal breakthroughs in various domains—including Vision [Krizhevsky et al., 2012], Natural Language Processing [Vaswani et al., 2017; Brown et al., 2020], Reinforcement Learning [Silver et al., 2016]—have often been characterized by their ability to efficiently leverage large scale compute. Efficient utilization and orchestration of compute resources has revealed itself one of the fundamental enablers of modern AI research: the bitter lesson [Sutton, 2019] has taught us that often, FLOPs trump theory.

Figure 1.1: Training compute required to train milestone ML systems has grown exponentially over the past decade (Figure reproduced from Sevilla et al. [2022]).

This demand for compute was initially met by specialized hardware. Training workloads almost always run on *accelerators* (GPUs, TPUs, etc.). Characterized by their massively parallel architecture, these chips are able to execute tensor operations, such as matrix multiplication, orders of magnitude faster than a typical CPU. Although these chips have grown more powerful through aggressive specialization towards deep learning workloads, their performance growth has been far keeping up with the 5.7 doubling in compute demands. Further, the similarly exponential growth in model parameters has also far outpaced the memory capacity of these accelerators, with the largest models requiring thousands of chips to fit all parameters and activations. While hardware specialization using specialized, low-bit floating point representations [Burgess et al., 2019] helps, it is also not enough.

In the face of these constraints, research has scaled horizontally, and running training workloads on tens to hundreds of accelerators is now commonplace. However, mapping the workload onto an efficient execution schedule for these clusters of resources is not a simple task. While simple techniques like data-parallelism are popular, they quickly run into constraints with larger models, necessitating more complex partitioning schemes. In search of higher efficiency,

modern implementations of popular models [Shoeybi et al., 2019; Rasley et al., 2020] now combine and interleave a large variety of techniques, including tensor partitioning schemes (e.g., model, operator and data-parallelism), complex pipelining schedules [Narayanan et al., 2019; Fan et al., 2021; Li and Hoefler, 2021], and even reach into DL-specific constructs such as the gradient descent optimizer states [Rajbhandari et al., 2020].

While manual efforts to combine these techniques have been initially successful, developments in hardware towards more specialized systems present ever-increasing complexity in interactions between them. Achieving high throughput requires careful consideration of these effects [Narayanan et al., 2021], and these considerations are often highly specific to the exact model architecture or cluster configuration. As a result, performing research on novel massive architectures necessitates large, highly-specialized engineering teams to navigate the search space and to produce an efficient distributed implementation, implicitly constraining the field's search for novel architectures.

For these reasons, interest in improved scheduling of these training workloads has grown, and various approaches have been proposed. We discuss and review these in Chapter 3.

In this work, we investigate a novel approach to scheduling of deep learning workloads. We propose, to the best of our knowledge, the first distributed training framework that exploits a full-stack view of the workload, lowering the model from high-level, PyTorch [Paszke et al., 2019] code, all the way to hardware optimized GPU kernels that are distributed across nodes in a cluster. By considering the problem from the full-stack view and analyzing operator implementations with generic techniques, we remove the need for manual, operator-centric annotations. This enables the exploration of a new approach to scheduling DL workloads, namely work- rather than parameter- focussed partitioning. We perform our cost-modeling and analysis on a data-centric intermediate representation of the workload, where all data movement is made explicit at all granularities.

1.1 Contributions

We make the following contributions.

- We propose a new approach to partitioning deep learning workloads based on work rather than data (Chapter 3).
- We propose a technique for automated partitioning of generic workloads based on symbolic data movement analysis. The approach is the first of its kind that is able to function *without operator-specific partitioning annotations*, enabling generalization beyond current sets of operators supported by other frameworks (Chapters 4 & 5).
- We build a system for compilation of deep learning workloads for distributed execution on clusters by extending the DaCeML framework [Rausch et al., 2022].
- We evaluate the system by demonstrating automatic partitioning of NumPy code on the Piz Daint supercomputer, scaling up to 128 machines (Chapter 8).

2 Background

Contents

In the following, we outline the necessary background, including an overview of the training workloads, contemporary parallelism schemes, the distributed scheduling problem statement and an outline of the existing compiler frameworks on which we base our work.

2.1 Neural Network Training

We now briefly provide background on the workload of training DNNs. A *Neural Network* (see Figure 2.1) is built up of basic components historically called *neurons*, based on analogy to biological neural networks [Rosenblatt, 1958]. These units aggregate contributions from their inputs, and produce an output to which an

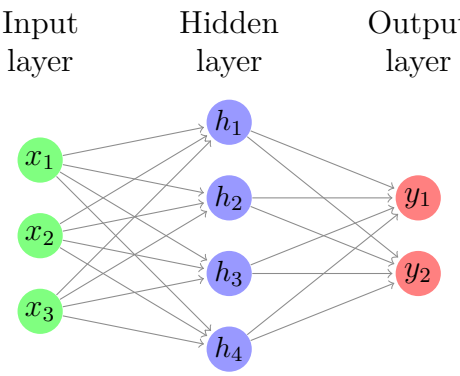


Figure 2.1: A simple, feedforward neural network with 3 inputs and 2 outputs. The single hidden layer contains 4 neurons.

activation function σ is applied. The neurons are parameterized by a weight vector w and a bias vector b, and given an input vector x, compute

$$\sigma(wx + b).$$

Critically, the use of a non-linear activation function, such as the sigmoid function, enables NNs to approximate universal functions given enough width [Cybenko, 1989] or depth [Yarotsky, 2017]. The weight and bias of a neuron are referred to as its *learnable parameters*. In an NN, multiple neurons are grouped into *layers*. The most basic layer is a fully-connected, or dense layer, where each input is connected to each output. However, this connectivity is often relaxed; for example, the commonly used convolutional layer only connects inputs to neurons according to spatial locality.

In modern DNNs, layers have grown diverse, and popular layers include resampling layers [LeCun et al., 1989; Ronneberger et al., 2015], multi-head attention [Vaswani et al., 2017] and various statistical normalization layers [Bridle, 1990; Ioffe and Szegedy, 2015; Ba et al., 2016].

The goal of the *training* process is to find suitable values for the parameters of the layers. To do this, one first selects a *loss*, or *objective* function, which, given the output of the NN will produce a score that we seek to minimize.

In the common case of supervised learning, we aim to learn some unknown function $f : X \to Y$. We are given a set of *training data*

$$\{(x_1, f(x_1)), \ldots, (x_n, f(x_n))\} \subseteq X \times Y.$$

Our objective function $\mathcal{L} : Y \times Y \to \mathbb{R}$ takes as input a candidate output label, and the true label, producting a scalar loss. We use this function by passing training samples x_i through the neural network to obtain outputs y_i, and then compute the loss $\mathcal{L}(y_i, f(x_i))$, where $f(x_i)$ is known from the training set. Popular objective functions include cross-entropy or mean squared error.

Parameter search is almost always done using variants of gradient descent, known as *optimizers*. These provide an *update rule* that describes how the parameters are updated given the gradients and the values from the previous iteration. Each update of the parameters is known as a *step*, and many such steps are performed in a given training run. By computing the gradients w.r.t. the loss, we can perform weight updates that over time, decrease the loss.

Gradients are typically computed by *reverse-mode automatic differentiation* [Baydin et al., 2018], where the computation can be decomposed into a *forward* pass, where loss for a given input is computed, and a backward pass, where the NN is traversed in reverse-order and partial derivatives are propagated backwards through the graph. The backward pass typically has data dependencies on the activation values from the forward pass.

As discussed in Chapter 1, the compute required for DNN training workloads has grown exponentially. In terms of the workload characteristics, these increased costs are primarily due to growth in the number of compute FLOPs and parameters in DNNs, as well as increases in the number of training steps taken. Training on accelerators, such as GPUs, has been a driving factor in enabling this growth [Raina et al., 2009; Krizhevsky et al., 2012].

2.2 Compiler Frameworks

Our work builds on DaCe [Ben-Nun et al., 2019], a data-centric compiler for high performance computing. We build directly on the DaCeML deep-learning framework [Rausch et al., 2022] and formulate our proposed system as a set of extensions to DaCeML. We introduce these frameworks in the following sections.

```python
from torch import nn
import daceml.torch

@daceml.torch.dace_module
class MyModule(nn.Module):

def __init__(self, n_in, n_out):
    super().__init__()
    self.linear = \
      nn.Linear(n_in, n_out)
    self.fanout = n_out

def forward(self, x):
  x = self.linear(x)
  return x / self.fanout
```

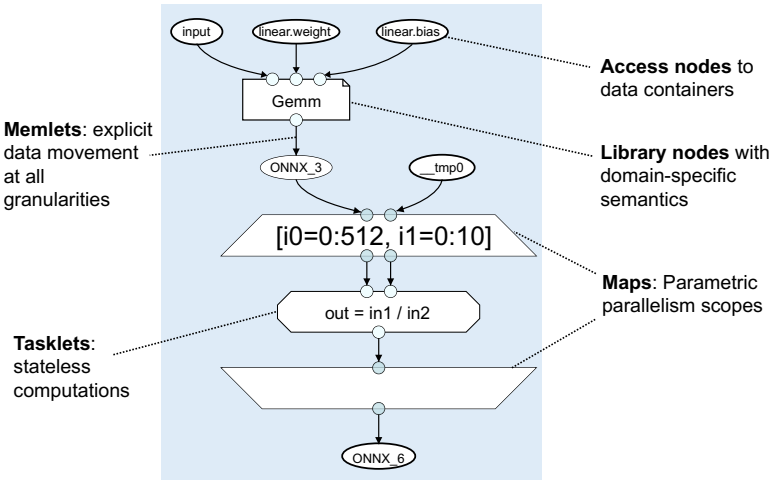


Figure 2.2: Translation of PyTorch code (left) to the SDFG IR (right) using DaCeML.

2.2.1 DaCe: Data-Centric Parallel Programming

Due to recent trends in hardware development, data movement has steadily gained relevance as a central bottleneck in high performance numerical computing [Unat et al., 2017]. Optimization of data movement throughout the memory hierarchy to exploit data locality, e.g., through loop tiling or fusion, has grown in relevance, and these data-movement-optimizing transformations typically cover the bulk of optimization efforts. In deep learning workloads, where it is typical to process large volumes of data on many compute nodes, often with specialized accelerators, some of the most popular networks have been shown to be data-movement bound [Ivanov et al., 2021].

DaCe is a compiler framework with a focus on explicitly modeling, analyzing and optimizing data-movement in high performance computing programs. It has been applied in scientific domains, such as Quantum Transport Simulations [Ziogas et al., 2019] and Weather Modeling [Ben-Nun et al., 2022], and has been used as a compiler for HPC workloads [Ziogas et al., 2021; Calotoiu et al., 2022; Ziogas et al., 2022].

In this work, the *Stateful Dataflow Multigraph* (SDFG) intermediate representation is used as the main and only IR. SDFGs are directed graphs representing state machines, where the nodes in the state machine (named *SDFG states*) are acyclic

data-flow graphs. In our work, the SDFGs will typically consist of few (usually only one), large states. An example of such an SDFG is shown in Figure 2.2 (right). The blue region is the single state in the SDFG. In an SDFG state, all data-movement is explicit using *memlets*, the black edges between the state nodes. Data is read or written from *access nodes*, and is moved into *code nodes* to perform computation. The simplest code node is a *tasklet*; in the shown example, the element-wise division by `self.fanout` is performed in a tasklet. In this case, the tasklet has two scalar inputs, and a single scalar output. The tasklet is contained within a *map scope*. The sub-graph within a map scope is parametrically replicated, with the symbols denoted in the map entry (in this case `i0` and `i1`) iterating over the given iteration ranges. The map schedule defines how the sub-graph is executed, (e.g., in parallel on CPU multicore systems, or GPUs, or sequentially).

To model what data needs to be moved, memlets are annotated with a (typically symbolic) *subset* that they transfer. For instance, the left memlet leading to `in1` in the tasklet above has subset `ONNX_3[i0, i1]`. The subset of the outer memlet will be automatically propagated using symbolic interval analysis; in this case, its subset will be `ONNX_3[0:512, 0:10]`. Since all data movement is explicit and annotated with memlets, SDFG transformations have constant time access to symbolic data-movement information, including source, destination and volume at all granularities. As we shall explore in Chapter 4 this makes it particularly suitable for modeling and solving communication constraints in a distributed workload.

The final type of node we introduce is another code node, the *library node*. Library nodes represent domain specific computation that can be *expanded* to one of several implementations. For example, the `Gemm` node here represents the ONNX [Bai et al., 2019] `Gemm` operator. It can be lowered to coarse-grained external library implementations such as cuBLAS or MKL. Further, most library nodes are equipped with *pure* implementations that only use SDFG elements. These pure implementations which enables fine-grained analysis transformation and analysis.

This concludes the brief description of the SDFG IR we will use in the remainder of this work. For a more detailed introduction, including operational semantics, we refer to Ben-Nun et al. [2019].

Beyond the IR, compiler and optimizing transformations, the DaCe framework also provides a Python frontend that is capable of parsing Python/NumPy code into SDFGs. We will work with, and introduce further details on this frontend in Chapter 7.

2.2.2 DaCeML

DaCeML [Rausch et al., 2022] is an optimization framework for deep learning models that uses the DaCe compiler to generate high performance implementations of models. In this work, we make use of parts of DaCeML, and we now provide a short description of the lowering process for given model.

In DaCeML, PyTorch modules can be annotated with the `@dace_module` decorator. Upon the first call to the model, the execution is traced and converted to an SDFG. The SDFG initially contains coarse-grained operator nodes for the ONNX Operators. DaCeML includes a set of pure implementations for these nodes, which are naive implementations parsed from DaCe's Python frontend. Given a fully-pure SDFG, DaCeML's symbolic automatic differentiation can produce a backward pass for this graph. Notably, and in contrast to engines in other frameworks—since the engine can operate on SDFGs—the framework can perform optimizations *before* automatic differentiation. This engine will be extended in Chapter 7 to computation beyond the model's forward and backward passes.

2.3 Prior work on Distributed Computation with DaCe

To clearly delineate our contributions from existing work, we briefly discuss existing facilities for distributed computing in DaCe. As mentioned above, DaCe has been used in the past to generate MPI programs. Most of the work has been done by manual use of MPI communication primitives.

Recently Ziogas et al. [2021] have worked on detecting certain elementwise computation patterns in SDFGs, and distributing these with MPI. Our approach is far more general, and enables partitioning of patterns far more complex that these (see Chapter 8 for examples).

Ziogas et al. [2022] have proposed a system of compiling tensor contractions to near-IO optimal distributed programs by generating Python code that is later parsed by the DaCe Python frontend. Again, our system more general, and is able to parse programs far beyond tensor contractions. Our system is first approach to partitioning and compiling generic SDFGs to distributed MPI programs.

Over the course of prior works, several abstractions for distributed computing have been introduced in the DaCe framework, notably an abstraction for process grids and distributed communication nodes that lower to MPI calls. While our system used these initially, we soon developed our own communication abstractions to fit our more complex needs (see Chapter 5).

While the DaCeML framework includes a `DistDataParallel` transformation for data parallelism, this is based on naive partitioning of the first axis of every tensor in the graph. Our proposed approach generalizes this, and many other advanced forms of parallelism that will be introduced in the next section.

2.4 Distributed Training

Distributed training of DNNs refers to the execution of DNN training workloads on a set more than one machine—a *cluster*—connected by an interconnection network (or *interconnect*). It is motivated by demands to reduce the latency in training a DNN (commonly on the order of days to months), as well as to satisfy memory constraints of accelerators (single accelerators cards can currently store on the order of tens of GB in local memory) in the face of ever-growing model sizes.

We note that this work focuses on the more popular regime of synchronous distributed training, where all nodes share a consistent view of the model parameters. While work on asynchronous distributed training exists, it is used significantly less popular than the synchronous approach, even in the largest training runs.

2.4.1 Communication

Communication plays a central role in distributed training. In these workloads, communication occurs both in between accelerator cards on a single node, and between nodes in the cluster.

Characteristics of these channels can vary heavily depending on the cluster hardware configuration. Some accelerators, such as NVIDIA GPUs, include high-bandwidth interconnects between nodes. In this work, we will abstract hardware details and assume the configuration to be constant, characterizing the different channels by metrics such as latency and bandwidth. We will make use of the MPI library [Walker and Dongarra, 1996] to implement certain communication patterns.

2.4.2 Parallelism Schemes

In the past decade, different forms of paralleling these workloads have been applied. There are many terms in use in different work, and it is not rarely applied consistently beyond the simpler forms. There is a distinction between intra-operator and inter-operator parallelism. Inter-operator parallelism are forms of parallelism where a model graph is split into separate sub-graphs, while intra-operator parallelism parallelizes individual layers and operators. In this work, we will not be concerned with specific classes of parallelism, but rather attempt to treat the problem of choosing a distribution scheme at a higher level of generality. For orientation and background, we nevertheless describe some of the more popular schemes here.

Data Parallelism Data parallelism is the simplest form of parallelism to implement, and it involves parallelizing along the batch dimension of the input data. As a result, each work stores a full replica of all model parameters, and gradient updates are communicated (typically using all-reduce) after each training step. In many networks (batch normalization can inhibit this), the different partitions of work are fully independent, and thus no communication is required until the end of the training step. However, the cost of replicating the model parameters quickly becomes prohibitive with larger models. Further,

the batch dimension only exposes limited parallelism as too large of a batch size can negatively impact convergence.

Model Parallelism Model parallelism is a less specific class of parallelism. In literature, model parallelism has been used to describe both intra-operator and inter-operator partitioning schemes. We will use tensor parallelism to refer to the former, and model parallelism for the latter. With this clarification, model parallelism is the scheduling of separate subgraphs of the model onto different devices, e.g., as in AlexNet [Krizhevsky et al., 2012] or Expert Parallelism in mixture of experts models [Shazeer et al., 2017].

Tensor Parallelism Tensor parallelism is a recently popularized term used to describe partitioning individual tensors, typically parameters of the model. Doing so implies a partitioning on the work performed by the operators, such that multiple devices compute the output of a single operator. This is particularly popular for tensor contractions, e.g., matrix multiplications due to their high arithmetic intensity. Different forms of tensor parallelism are often named by the underlying dimension they partition along: e.g. Row-, Column-, Sequence-, Channel- or Filter- parallelism.

Pipeline Parallelism Pipeline parallelism is a form of inter-operator parallelism that partitions the model graph into a sequence of sub-graphs, and then schedules the sub-graphs onto different devices in a pipelined fashion. It is typically combined with a form of data-parallelism to split the batches into *micro-batches* that enable deeper pipelining. The schedule used to execute the micro-batches is a complex and well studied problem [Huang et al., 2019; Narayanan et al., 2019; Fan et al., 2021; Li and Hoefler, 2021]. The pipeline bubbles introduced by these schemes are source of overhead: as a result, pipelining is not universally used in large model training.

For large language models, several recently trained milestone models such as PaLM [Chowdhery et al., 2022], LaMDA [Thoppilan et al., 2022] and [Du et al., 2022] have been trained without pipelining, while others, such as

Gopher [Rae et al., 2021] and Megatron-Turing NLG [Smith et al., 2022] did use pipelining.

Optimizer Partitioning Popular optimizers such as Adam [Kingma and Ba, 2014] maintain optimizer state, such as momentum and variances, that are 4× the size of the model parameters. Work such as ZeRO [Rajbhandari et al., 2020] partitions these states on to separate devices. When communicating the gradients for data parallelism, ZeRO folds the communication of the optimizer state into the communication of the gradients to achieve identical computation with no additional data communication volume.

Hybrid Parallelism Recently, combinations of the above forms of parallelism have been investigated and tuned jointly to achieve higher performance [Rasley et al., 2020; Narayanan et al., 2021].

2.5 Generic Partitioning

The main problem we tackle in this work is the problem of partitioning a workload for execution on a compute cluster. This involves producing hardware-optimized executables for each node, and scheduling the communication between them.

Much prior work (Schaarschmidt et al. [2021]; Zheng et al. [2022] as well as TensorFlow's dTensor and JAX's `@pjit`) has been based on the XLA complier [Google, 2017] and the GSPMD partitioner [Xu et al., 2021] for XLA. As we will discuss in the next section, this, and other work leans heavily on manual specification of parallel algorithms for each supported operator. GSPMD, in particular, enumerates these algorithms in over 5k lines of code.

This operator-centric approach is an example of a larger trend in the machine learning framework community to favor manual annotations and kernels over generic solutions [Barham and Isard, 2019]. At first glance, it may seem that generality is not important: after all, all popular workloads are well supported by systems. We believe that this argument is flawed. As argued by Hooker [2021], the model architectures that can be effectively evaluated and investigated are heavily

constrained by contemporary hardware and software. This creates an effect akin to survivorship-bias: models that maximize hardware utilization rise to the top, and models that do not cannot be investigated further.

We aim to avoid these manual approaches to enable partitioning of workloads beyond the set of current popular operators. As we will introduce in the following chapters, our approach generalizes well, and is able to automatically partition and compile high-level Python code to distributed programs that can be executed on large clusters of machines.

3 Schedule Representation

In the following, we outline our chosen model for representing schedules. This is the central choices in our system; the lowering solver, cost model and search all take a schedule as input.

3.1 Prior Approaches

Our representation differs from prior work, and we thus provide a brief survey for context. The large majority of representations in prior work lean towards operator-graph representations of deep learning workloads. This is a natural approach, especially given the fact that similar representations have historically dominated user facing frameworks. (e.g., TensorFlow [Abadi et al., 2016], MXNet [Chen et al., 2015], ONNX [Bai et al., 2019], Theano [Bastien et al., 2012], Caffe [Jia et al., 2014], CNTK [Seide and Agarwal, 2016]).

When using such a representation, operators are initially black-boxed. Since specifying how to distribute an operator requires knowledge about its semantics, most frameworks use manually annotated registries of parallel semantics. In these representations, the only other element apart from the operator nodes are the data nodes. These are of course natural to partition, and as such, a common

theme that emerges is that schemes specify the partitioning of *data rather than partitioning of work*.

Although early work often only partitioned layers (i.e., larger subgraphs of operators), we do not make a distinction to partitioning individual operators in this review, since the granularity of abstraction is relatively similar.

FlexFlow [Jia et al., 2019] and OptCNN [Jia et al., 2018] define a parallelization scheme of an operator as the partitioning scheme of the output tensor of an operator. While this covers many dimensions of parallelism, it misses dimensions where reductions are performed in network (i.e., paralleling over a non-output dimension). AccPar [Song et al., 2020] describes 3 distinct parallelization dimensions and assigns one to each layer.

In Mesh-Tensorflow [Shazeer et al., 2018], Automap [Schaarschmidt et al., 2021] and GSPMD [Xu et al., 2021] a parallelism scheme is similarly defined as the partitioning scheme for each tensor in the graph. However, by permitting specification of input tensor layouts, schemes can induce the selection of algorithms that perform in-network reductions. Alpa [Zheng et al., 2022] uses a similar scheme, but only permits two parallel dimensions per operator.

All work mentioned thus far manually enumerates, implements and/or annotates operator kernels each supported operator, limiting generality.

Tofu [Wang et al., 2019] and Unity [Unger et al., 2022] explored automatic extraction of parallel algorithms for operators. This was done by annotating all supported operators in a DSL using the annotated information to extract parallelizable dimensions.

3.2 Partitioning Work, Not Data

In this work, we explore a novel approach to specifying parallelism in deep learning workloads, aiming to partition the work rather than the data. In this section, we motivate this choice and briefly outline the involved tradeoffs.

Partitioning the work in a deep learning workload is to partition the compute performed. This can be done by specifying how the *parallel maps are partitioned*.

Formally, for a given n dimensional iteration space with ranges $[l_0, h_0], \ldots, [l_n, h_n]$, we provide a list of n partition sizes $[p_0, \ldots, p_n]$. These act as the dimensions of an n dimensional process grid consisting of $p_0 \times \cdots \times p_n$ processors. As a result, each local processor computes a subset of the global iteration space of size

$$\frac{h_0 - l_0}{p_0} \times \cdots \times \frac{h_n - l_n}{p_n}.$$

Making this choice presents several new problems, which we will address in this work. Firstly, this requires information about the iteration spaces that each operator executes. Our workloads will be modeled as SDFGs, which we lower all the way to accelerator code. Naturally, this implies that we have full knowledge of all iteration spaces. Secondly, this scheme makes no specification on the data partitioning. As a result, such as system needs a mechanism to derive the required partitioning schemes from the iteration spaces, and then discover communication routines to marshal the data into the correct locations.

The latter problems are non-trivial; while prior work has included communication solvers to redistribute tensors from one partitioning scheme to another, these possible schemes have always been very limited, since the tensors are always partitioned cleanly along their axes with even block sizes. In our case, depending on the subsets accessed by the operator, any partitioning scheme (even completely unpractical schemes such as partitioning scalars randomly) is possible in principle.

The major advantage of the approach lies in its generality: since there is no reliance on operators or manual annotations, arbitrary parallel programs can be scheduled. As we shall see, in practice, the class of compilable programs will be restricted heavily by the strength of the communication solver.

Secondly, the search space is more compact. In practice, the number of dimensions in the iteration space will be similar to—or larger by one or two contracting dimensions than—the largest of the operand dimensions. Partitioning each of the (multiple) operands then presents a larger search space than partitioning the iteration space.

Finally, we argue that our space is more expressive. For many commonly used operators, both approaches are equally expressive, since each data dimension is typically iterated over independently, yielding a direct correspondence between iteration space dimensions and data dimensions. However, looking beyond these operators, there are many computations where the iteration space is larger than the data. A simple example is Monte-Carlo integration, where the iteration space of iterated sampling is *not* a data dimension, and as a result data-based partitioning will fail to parallelize along this sample axis. As argued in the prior Chapter, we believe that the set of popular operators should not guide the limits of our systems, lest we lose out on innovative operators of the future.

4

Distributing Programs by Analyzing and Solving Communication Constraints

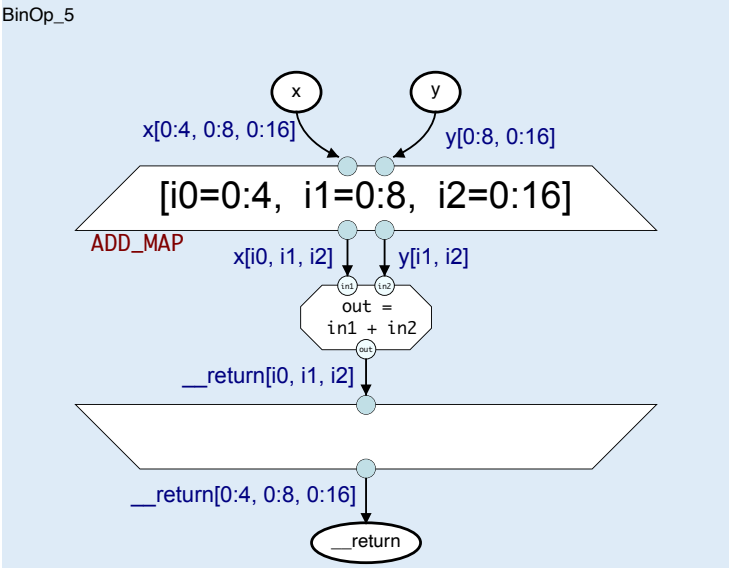
Contents

In this chapter, we describe a method for lowering an SDFG with a given distributed schedule to a single program multiple data (SPMD) executable that communicates using MPI. The goal to produce a program that produces equivalent results (up to floating point inaccuracy) as the original, single-process SDFG.

This proceeds in three stages. First, the iteration spaces of the program are tiled across the available compute nodes. Secondly, the implied communication constraints are extracted using symbolic interval analysis. Finally, these communication constraints are solved to produce a sequence of MPI calls that satisfy the constraints and execute the required communication.

```python
@dace
def program(
      x: dace.int64[4, 8, 16],
      y: dace.int64[8, 16]):
    return x + y
```

(a) Annotated Python/NumPy source code. The lower dimensional array `y` is *broadcasted* up to the dimensions of `x` according to NumPy broadcasting semantics [Harris et al., 2020].



(b) The resulting parsed SDFGs. Memlets have been annotated with blue, and map names with red text.

Figure 4.1: A simple elementwise addition.

4.1 Distributed Schedules

Our goal is to *partition* the work of the program across the compute cluster of p nodes available to us. We will often refer to the computation performed in the original program as *global*, and the computation of the resulting process-local SPMD programs as *local*.

Consider, as a running example, the program shown in Figure 4.1 (left). As can be seen Figure 4.1 (right), the corresponding SDFG consists of a single iteration space of size $4 \times 8 \times 16$, which is implemented with a single parallel map, named `ADD_MAP`. This is a *top-level map*, since it is not enclosed by any other map scope.

Definition 4.1.1 (Distributed Schedule)**.** Given an SDFG, a distributed schedule for the SDFG is a mapping from iteration spaces to process grid dimensions. For each top-level n dimensional map M in the SDFG, the schedule assigns M to a process grid of dimension n. A schedule is *complete* if it contains every top level map.

For the example above, examples of complete schedules include $\{\texttt{ADD\_MAP} \mapsto (2,2,2)\}$ (8 processors), and $\{\texttt{ADD\_MAP} \mapsto (2,2,1)\}$ (4 processors). The fully sequential schedule with only 1 processor is given by $\{\texttt{ADD\_MAP} \mapsto (1,1,1)\}$.

4.2 Rank Tiling

Given an SDFG and a valid schedule, our method proceeds by partitioning the iteration spaces of the SDFG according to the schedule, assigning each process an equally sized subspace of the global iteration space. We call this *rank-tiling* the SDFG. This will induce a set of communication constraints, which we will solve in the later sections.

Definition 4.2.1 (Iteration Space)**.** An n dimensional *iteration space* a set of of n-dimensional tuples. Each map is defined by a *range*, given by n pairs. The i-th pair in the range represents the lower and upper bound for the i-th dimension. Following Python notation, we denote these pairs by representing a range from l to h as $l : h$.

We call an element of an iteration space a *point*, and an iteration space with bounds $[l_0 : h_0, \ldots, l_{n-1} : h_{n-1}]$ contains the points

$$\{(V_0, ..., V_{n-1}) \mid l_0 \leq V_0 < h_0 \wedge \cdots \wedge l_{n-1} \leq V_{n-1} < h_{n-1}\}.$$

Recall that each map in an SDFG is has an associated iteration space, and that the subgraph within the map nodes will be instantiated in parallel once for each point in the iteration space. In our running example, the iteration space of the `ADD_MAP` is $I = [0 : 4, 0 : 8, 0 : 16]$, and the subgraph within the map scope will be instantiated for all points $V \in I$, e.g., `i0=0, i1=0, i2=0` and `i0=0, i1=0, i2=1`.

We process each top-level map M in the SDFG separately. Given a map M with iteration range

$$[l_0 : h_0, l_1 : h_1, \ldots, l_{n-1} : h_{n-1}]$$

and its process grid dimensions $D = [d_0, d_1, \ldots, d_{n-1}]$ assigned by the schedule, [1] the iteration will be split into $d_0 \times d_1 \times \cdots \times d_{n-1}$ subspaces, where the sizes along each dimension of each subspace are

$$b_0, b_1, \ldots b_{n-1} = \frac{h_0 - l_0}{d_0}, \ldots, \frac{h_{n-1} - l_{n-1}}{d_{n-1}}.$$

Each process p with coordinates $(p_0, p_1, \ldots, p_{n-1})$ in the process grid will execute one of these subsets, namely

$$[l_0 + p_0 \times b_0 : l_0 + (p_0 + 1) \times b_0, \ldots, l_{n-1} + p_{n-1} \times b_{n-1} : l_{n-1} + (p_{n-1} + 1) \times b_{n-1}].$$

We now transform the map's range to

$$\left[l_0 : l_0 + \frac{h_0 - l_0}{d_0}, \ldots, l_{n-1} : l_{n-1} + \frac{h_{n-1} - l_{n-1}}{d_{n-1}}\right].$$

Consider again the SDFG in Figure 4.1 (right). Rank tiling the SDFG with the schedule

$$\{\texttt{ADD\_MAP} \mapsto (2, 2, 1)\}$$

results in a local rank-tiled map with iteration space $I' = [0 : 2, 0 : 4, 0 : 16]$.

Notably, the new map range is offset such that each subspace begins at the same coordinates as the original map range, enabling us to execute this same program on all processes in SPMD fashion. This offsetting is possible because the computation performed by the subgraph contained within the map scope is equivalent for each point in the iteration space.[2] As long as the inputs and output data of the map is offset accordingly, the computation will be equivalent to the single-process execution.

Formally, given the tiled iteration space I', a process $p = (p_0, p_1, \ldots, p_{n-1})$ and a point $V' = (v_0, \ldots, v_{n-1}) \in I'$, the semantically corresponding point in the global iteration space I is

$$V = [l_0 + p_0 \times b_0 + v_0, \ldots, l_{n-1} + p_{n-1} \times b_{n-1} + v_{n-1}].$$

[1] For simplicity, we do not discuss edge cases, such as when map dimensions have non-unit step sizes, when the map range is not divisible by the process grid dimensions, or when the process grid dimension is larger than the map dimension. Our method applies in these cases with minor modifications.

[2] Of course, with the difference that the iteration variables are instantiated differently.

In this scheme, we ensure that each 'global' point $V \in I$ has a corresponding 'local' point $V' \in I'$ for exactly one process p: every iteration of the global map will be exactly once in the distributed program. We define the function $f_{I,D}$ such that $f_{I,D}(V) = (p, V')$, and its inverse $f'_{I,D}((p, V')) = V$. In the following, we will omit the parameterization I, D, which will be clear from the context.

In our example, the process at coordinates $(0, 0, 0)$ will execute the sub-iteration space $[0 : 2, 0 : 4, 0 : 16]$. Similarly, process $(0, 1, 0)$ executes $[0 : 2, 4 : 8, 0 : 16]$, $(1, 0, 0)$ executes $[2 : 4, 0 : 4, 0 : 16]$, and process $(1, 1, 0)$ executes $[2 : 4, 4 : 8, 0 : 16]$. Further, consider global point `i0=3, i1=2, i2=12`. The corresponding point in the local iteration space is `i0=1, i1=2, i2=12`, executed by process $p = (1, 0, 0)$.

4.3 Extracting Symbolic Communication Constraints

Due to the offsetting described above, each process will now only read the same, smaller subset of the global array data. In our example, each process will only read `x[0:2, 0:4, 0:16]`, and `y[0:4, 0:16]`, and only write `__return[0:2, 0:4, 0:16]`.

To ensure that this execution remains equivalent to the execution of the global SDFG, we will ensure that each process reads and writes a *different* subset of the global data. We introduce *local containers* for each data container that is read and written by rank-tiled maps, and reroute memlets such that the map reads from the local containers instead of the global. The size of these local subsets is smaller than or equal to the sizes of the global data.

Continuing with notation from the prior section, the subgraph within M will have several *inner* memlets that read and write to the outer arrays. In our example, the two input memlets read the subsets `x[i0, i1, i2]`, `y[i1, i2]`, and the output memlet writes the subset `__return[i0, i1, i2]`.

For a point V in the global iteration space I of the map M, consider $p, V' = f(V)$, where p is a process with coordinates $(p_0, p_1, \ldots, p_{n-1})$ in the process grid, and V' is a point in the local iteration space I' of the rank-tiled map.

Definition 4.3.1 (Memlet propagation)**.** Memlet propagation (as defined in DaCe [Ben-Nun et al., 2019]), is a procedure in which the subsets of inner memlets are propagated through a map scope to determine the subset of the outer, global data that is read and written.

This procedure is used within the framework to compute, for example, the outer memlet `x[0:4, 0:8, 0:16]` given the inner memlet `x[i0, i1, i2]` and its associated map ranges.

Let $i_0, \ldots, i_{n-1}$ be the symbolic variables associated with the map dimensions. We now transform the map iteration space by applying the symbolic α-conversion (effectively undoing the implicit offset that was applied when we tiled the map)

$$\{i_0 \mapsto l_0 + r_0 \times b_0, \ldots, i_{n-1} \mapsto l_{n-1} + r_{n-1} \times b_{n-1}\},$$

where $r_0, \ldots, r_{n-1}$ are fresh symbolic variables. Then, we apply memlet propagation to propagate all inner memlets through this new map iteration space. The resulting outer memlets now hold the subsets of the global data that must be read by the process p with coordinates $r_0, \ldots, r_{n-1}$. These subsets are *exactly* the communication constraints that we need to solve to ensure computational equivalence between the parallel and the single-process program!

We illustrate this with our example: with schedule $\{\texttt{ADD\_MAP} \mapsto (2, 2, 1)\}$, we have $b_0, b_1, b_2 = \frac{4-0}{2}, \frac{8-0}{2}, \frac{16-0}{1} = 2, 4, 16$, and the new, α-converted iteration space is

$$\texttt{i0=r0*2:(r0+1)*2}, \quad \texttt{i1=r1*4:(r1+1)*4}, \quad \texttt{i2=r2*16:(r2+1)*16}.$$

Now, when propagating outward through this new iteration space, the inner memlets `x[i0, i1, i2]` and `y[i1, i2]` become `x[2*r0:2*(r0+1), 4*r1:4*(r1+1), 0:16]`, and `y[4*r1:4*(r1+1), 0:16]`.

With this method, we thus extract symbolic expressions that represent the communication constraints that we need to implement. If each local array contains the process local subset given in the constraint (i.e., the constraints are satisfied), we can guarantee that the execution will be equivalent to the single-process execution. This is because firstly, each iteration of the global map is instantitated exactly

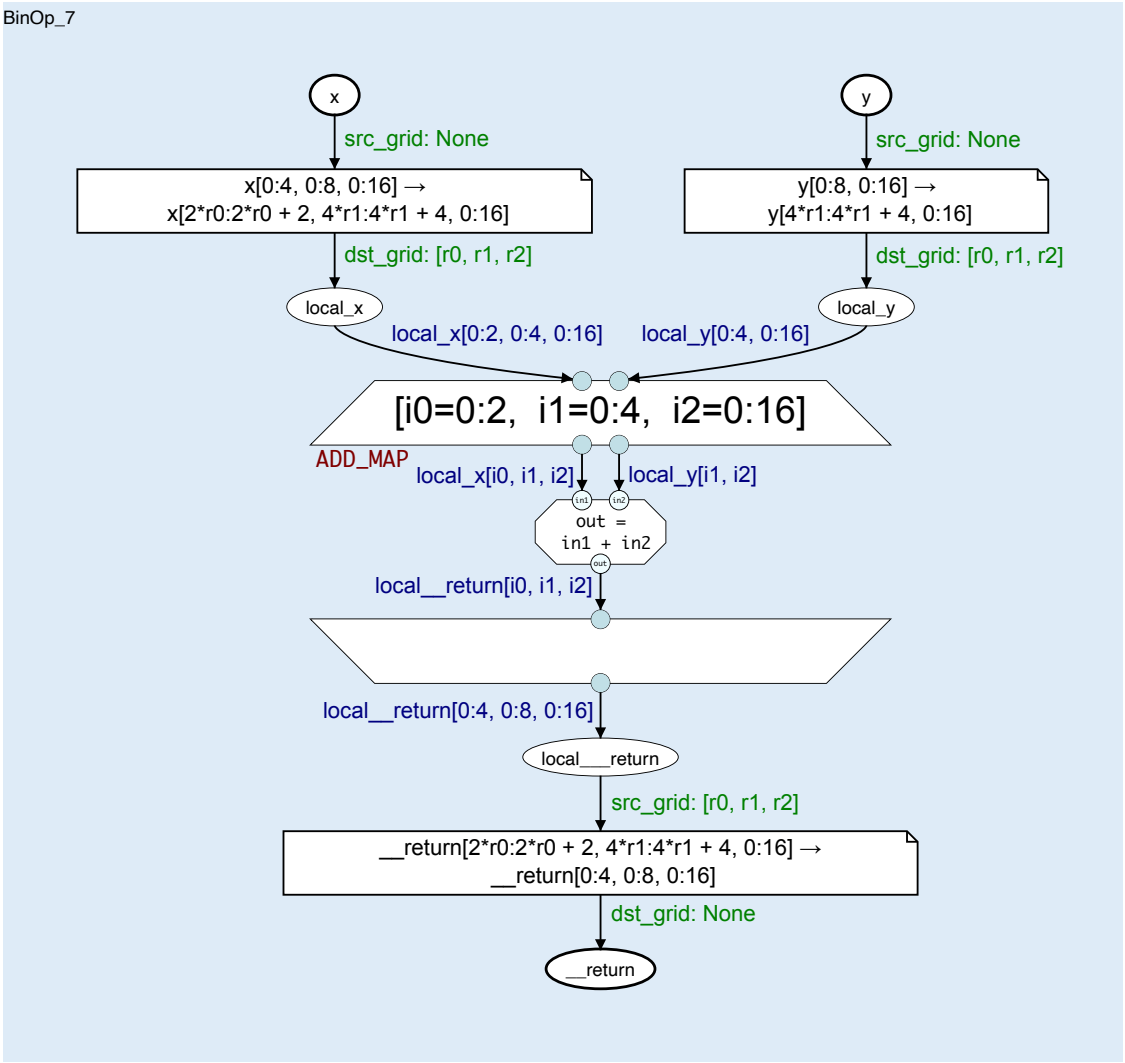


Figure 4.2: The distributed program corresponding to Figure 4.1, after lowering with schedule $\{\texttt{ADD\_MAP} \mapsto (2, 2, 1)\}$. Selected memlets have been annotated with blue, map names with red, and process grids with green text. The library nodes are all `DistributedMemlet` nodes. The free variables `r0` and `r1` represent the first two coordinates of the executing process in the process grid with dimension $(2, 2, 1)$.

once in our distributed program, and secondly, each instantiation of an iteration point in the distributed program reads and writes data to the same locations in the global array as the single-process program.

4.4 Solving Communication Constraints

Next we discuss how these constraints can be implemented. We first introduce a new library node that acts as an abstraction for these communication constraints.

Definition 4.4.1 (Distributed Memlet)**.** A *distributed memlet* is a library node that models and implements data movement *between processes*. It has a single input and a single output. For both the source and destination, the library node has the following attributes: the process grid along which the data is distributed, the symbolic subset (communication constraint) that must be satisfied, and a list of symbolic variables for each process grid dimension (these variables are used as atoms in the subset expression).

Figure 4.2 shows an example of the simple program we have been following, distributed with schedule $\{\texttt{ADD\_MAP} \mapsto (2, 2, 1)\}$. We briefly recap and point out the changes were made by our transformation thus far when compared to the starting SDFG shown in Figure 4.1 (right).

- A process grid of dimensionality $[2, 2, 1]$ has been allocated (not visible in the figure).
- The `ADD_MAP` implementing the element-wise addition has been tiled, reducing its iteration ranges from $[0 : 4, 0 : 8, 0 : 16]$ to $[0 : 2, 0 : 4, 0 : 16]$.
- The reads and writes to the global arrays `x` and `y` have been replaced with reads and writes to the process-local arrays `local_x` and `local_y`.
- `DistributedMemlet` library nodes have been inserted to communicate the data `x`→`local_x` and `y`→`local_y`, as well as `local__return`→`__return`. These communication nodes ensure that each the process-local data containers on the rank with coordinates `r0, r1,` `0` hold the subset of the global array given by the communication constraint in terms of the symbolic variables `r0` and `r1`.

Initially, distributed memlets are insert with a special source process grid shape of `None`. This sentinel value is used to indicate that the value should not reside distributed across a process grid, but rather exists only on the root process (rank 0 in MPI terms). This is useful for scattering input values at the start of a workload, and finally gathering the output values at the end.

To proceed with lowering, these distributed memlet library nodes can be solved and then *expanded* with an implementation that satisfies the communication constraints. A theoretically simple implementation of this node can lower to an `MPI_Alltoallv` call, instantiated with appropriate parameters.[3] This powerful primitive allows processes to communicate any data from any process to any other process. Another possible implementation, which we discuss in the next chapter, attempts to solve the constraints using the less expressive, but more performant `MPI_Scatterv` and `MPI_Gatherv` calls.

4.5 Fusing Distributed Communication

Initially, the rank-tiled graph will always communicate through the root process: Inputs to a kernel first be scattered from the root process across the grid, and gathered again to the root process after the kernel computation is complete. This is clearly wasteful when kernels composed or called sequentially. To eliminate this wasteful communication, a fusion transformation detects the pattern

local array → distributed memlet with `dst_grid=None`
→ global array
→ distributed memlet with `src_grid=None` → local array,

and fuses them into a single distributed memlet by combining the constraints. This is a cleanup pass that runs directly after rank tiling. Usually, schedules with low costs (according to the cost model) induce constraints such that this fused distributed memlet can be solved without communicating to repartition the data.

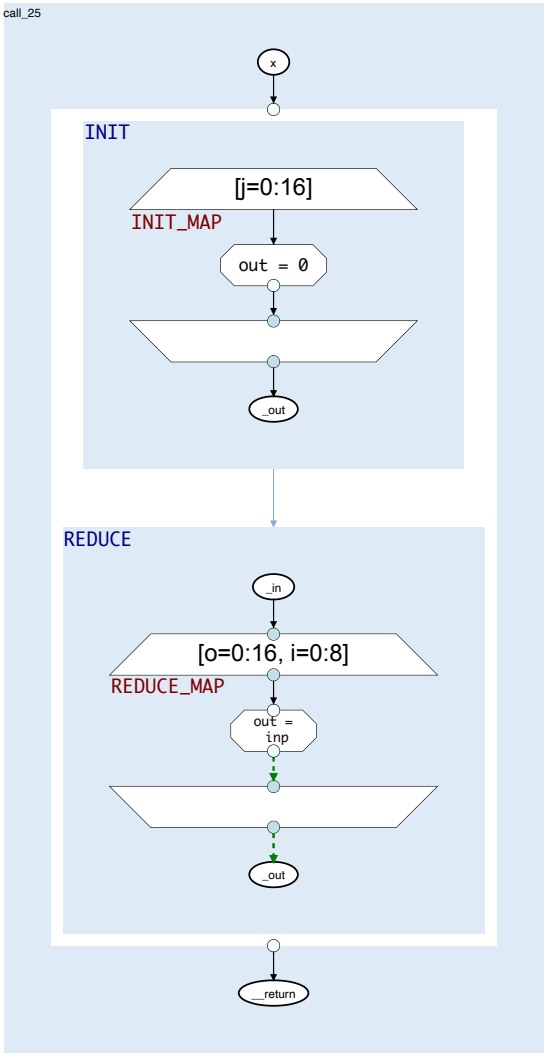
4.6 Nested & In-place Computation

In-place computation is rare in deep learning workloads, and is typically avoided by compilers by choosing novel variable names for each operator output, similar to static single assignment form [Aycock and Horspool, 2000].

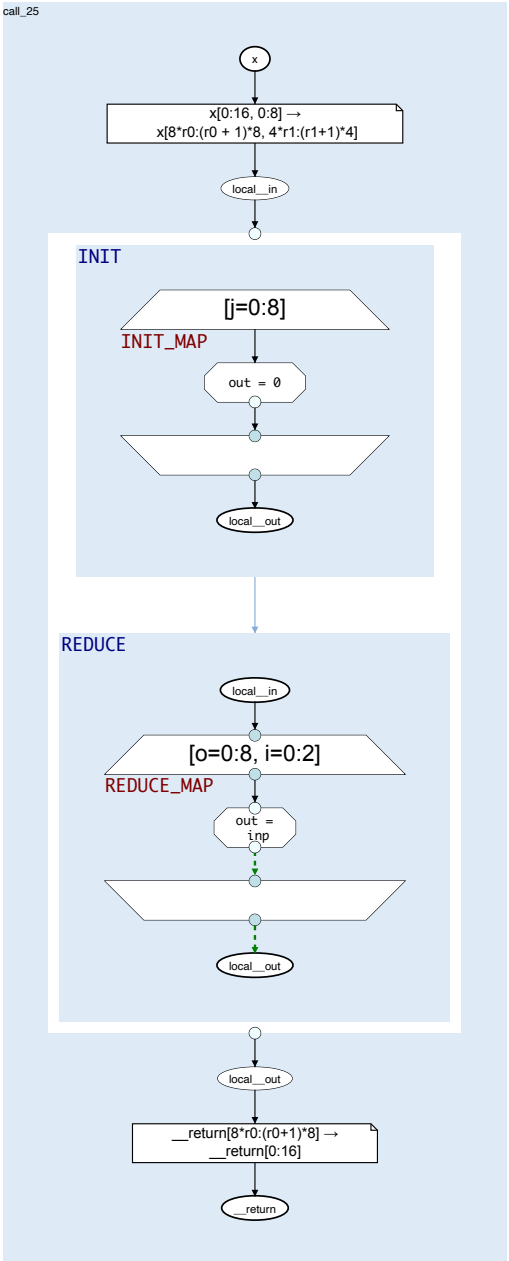
[3] A detail here is that, similar to the implementation discussed in the next chapter, overlapping reads/writes need to be appropriately resolved using an MPI broadcast/reduction respectively.

```python
@dace
def program(x: dace.int64[16, 8]):
    return np.add.reduce(x, axis=1)
```

(a) Annotated Python/NumPy source code. The `x` array is reduced along the last axis, resulting in an array of shape $(16,)$



(b) The SDFG obtained by parsing the NumPy code. The SDFG contains a nested SDFG with two states. The first state (named `INIT`) initializes the reduction output buffer `_out`. The second state (named `REDUCE`) performs the reduction. It does so using the green marked memlets leading to the `_out` array. These are write-conflicted memlets, with the write conflict resolution set to sum.



(c) The result of distributing the SDFG with schedule $\{$`REDUCE_MAP` $\mapsto (2, 4)$, `INIT_MAP` $\mapsto (2,)\}$. When constructing the SDFG, the system asserts that the communication constraints involving `_out` are consistent.

Figure 4.3: Distributed Lowering of a simple reduction operation. State names are annotated with blue, and map names with red text.

However, since we desire to be able to distribute a wider class of programs, including partially lowered and fused kernels, we need to be able to handle in-place operations. A common situation is shown in Figure 4.3, where lowering a reduction operation results in a nested SDFG with two states, where the first state initializes the output buffer, and the second state performs the reduction.

To support these, we support scheduling top-level maps even when they are nested. As such, the schedule used in Figure 4.3 must also contain process grid sizes for the inner maps `REDUCE_MAP` and `INIT_MAP`.

However, in these cases, we assert additional constraints. The basic principle we follow is that inter-process communication nodes (i.e., distributed memlet library nodes) are kept *outside* any nested SDFGs. This is essential for communication optimizations such as the described distributed memlet fusion. Further, we wish to avoid differing communication constraints for the same local array, inducing additional communication costs within the kernel. In the shown example, zero-initializing a buffer on some processes and then repartitioning the zero-initialized values is a nonsensical schedule.

Following this principle of not communicating within nested SDFGs, the lowering process must therefore assert that the communication constraints within the nested SDFG are consistent. This is slightly more complicated than a simple equality check. To see why, recall that each map is assigned a process grid of dimension equal to its iteration space. In the example of Figure 4.3, the `INIT_MAP` has dimension 1, and the `REDUCE_MAP` has dimension 2. As a result, they initially have different variables assigned to their axes, and their communication constraints will use different variables. Schedule consistency must thus be checked up to renaming, defined more formally as follows.

Definition 4.6.1 (Schedule Consistency)**.** A schedule is consistent for a set of maps $\{M_1, \ldots, M_n\}$ if there exists an α-renaming of the dimension variables of all maps, such that for each array x, and its set S_x of communication constraints induced by a read or write from a map, it holds that all constraints in S_x are equivalent after applying the renaming.

This can be checked through symbolic manipulation and pattern matching on constraint expressions. We omit the details of this procedure, but refer the interested reader to the accompanying section in the source code.[4]

The lowering process proceeds by rank-tiling the top-level maps as before. Contrary to the standard case described above, we don't introduce new buffers for the local views of arrays, but rather resize the existing buffers in the nested SDFG and introduce the local data container access nodes, as well as the distributed memlet nedes, outside the nested SDFG. Given a consistent schedule, this is possible because all local views of the global data are partitioned in the same way, and can thus be implemented as a single distributed memlet for each read/write before/after the kernel respectively.

4.6.1 Library Nodes

A trivial lowering of library nodes would be to expand them to their 'pure' SDFG implementation. In this expansions, the node is replaced with a (typically) nested SDFG and lowering can proceed as above.

However, we would often like to avoid lowering library nodes to allow the compiler to choose implementations when compiling the SPMD program. A common case here is matrix multiplication. Lowering to the pure implementation results in an unoptimized SDFG using atomic writes, requiring further optimization: lowering to an optimized BLAS library like MKL or cuBLAS is often preferable.

To avoid lowering, our method expands the library node to the pure implementation, but *does not* retain this expansion in the graph. The rank-tiling and communication analysis proceeds as normal, and once the outer communication constraints are derived, the distributed memlets are connected to the unexpanded library node, and the pure expansion is discarded.

This works for the simple reason that all implementations of a library node must be functionally equivalent. Thus, using the pure SDFG as a proxy for analysis allows us to derive communication constraints *even for opaque binary blobs* like cuBLAS.

[4] Refer to `daceml/distributed/schedule.py`, function `rank_tile_nested`.

```
procedure DistributedLowering(SDFG, schedule)
    for each state in topologically ordered states do
        for each node in top level nodes of state do
            if node is a library node then                                  ▷ §4.6.1
                Expand node to pure impl., proceed with its nested SDFG
            end if
            if node is a nested SDFG then                                   ▷ §4.6
                Assert that schedule is consistent within the nested SDFG of node
                RankTileNested(SDFG, state, node, schedule)
            else                                                            ▷ §4.2
                RankTileMap(SDFG, state, node, schedule)
            end if
            Initialize process_grid size according to schedule
            for each global read or write to array a do                     ▷ §4.3
                Extract communication constraints
                Insert distributed memlet node between global and local a
            end for
        end for
    end for
    Fuse distributed memlets                                                ▷ §4.5
    Solve and lower distributed memlets          ▷ §4.4, and next chapter
end procedure
```

Listing 1: A high-level overview of the process of lowering a single-process SDFG to a SPMD distributed SDFG, with references to individual sections where the annotated part is discussed.

4.7 Summary

This concludes the discussion of lowering single-process SDFG to SPMD distributed programs that can be executed across a cluster of distributed nodes. The proposed novel method is, to the best of our knowledge, the first method capable of distributing such low-level programs given no additional annotations except for the process grid dimensions.

Listing 1 summarizes the procedure outlined in this chapter. The implementation of this procedure is available in the accompanying source code.[5]

We conclude with a brief comparison to other work. Prior work on DL workloads (discussed in Chapter 3) has relied on data-dependent schedules and the manual definition of parallel algorithms for each supported operator, failing to generalize to

[5]Refer to `daceml/distributed/schedule.py`.

unseen operators. Other work, such as Ziogas et al. [2021], has further matched specific data movement patterns such as element-wise sums or reduction and distributed these, failing to generalize beyond these simple patterns.

By comparison, our method is capable of handling a wide range of programs (i.e., 'nearly' single state SDFGs). Rank-tiling the maps is possible without restrictions, and the only remaining constraints are the strength of the solver,[6] and the discussed constraint on schedule consistency within nested SDFGs. Even in cases where available solvers cannot find an efficient communication schedule, the lowering can succeed by over-approximation of the constraint subsets.

The technique thus generalizes the various forms of intra-operator parallelism introduced in Chapter 2, and as we shall show in evaluation, can even distribute complex fused and tiled operator kernels. In the next chapter, we discuss further details of solving a commonly encountered classes of communication constraints.

[6] An `MPI_Alltoallv`-based solver is very expressive, and can even handle irregular accesses.

5 Solving for Grid Mapped Arrays

Contents

In this chapter, we outline the functionality of one instantiation of a communication constraint solver. The grid mapped array solver solves an important class of communication constraints that arise from affine array accesses in rank-tiled parallel loops.

5.1 Grid Mapped Arrays

A grid mapped array is a partitioning scheme of an array that partitions an array onto a process grid. The scheme is parameterized by: (1) the array we are partitioning, (2) the process grid we are mapping the array on to, and (3) the axis mapping.

This maps each dimension of the process grid. For each dimension i of the process grid, there are three possible schemes:

PARTITION(j) The j-th axis of the array is evenly partitioned along the i-th axis of the process grid.

REPLICATE(j) The j-th axis of the array is will be replicated along the i-th axis of the process grid.

BROADCAST The whole array will be broadcast along this axis. This is typically required when an array is of lower dimensionality than the process grid.

Each array axis can only be mapped to at most one dimension in the process grid, and any array axes not mentioned in the axis mapping will be implicitly broadcasted to permit communication of arrays of higher rank than the process grid.

As an example, consider the following array X of shape $(4, 4)$, and process grid of shape $(2, 2)$:

$$X = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}, \quad P = \begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix}.$$

`[REPLICATE:0, REPLICATE:1]` The array is fully replicated, and each process receives $X[:, :]$.

`[PARTITION:0, REPLICATE:1]` The array is split along the first dimension (data-parallelism). Processes p_{00} and p_{01} receive $X[0:2, :]$, while processes p_{10} and p_{11} receive $X[2:4, :]$.

`[PARTITION:1, REPLICATE:0]` The array is split along the second dimension (column-parallelism). Processes p_{00} and p_{01} receive $X[:, 0:2]$, while processes p_{10} and p_{11} receive $X[:, 2:4]$.

`[PARTITION:1, PARTITION:0]` The array is fully partitioned onto the process grid, but with swapped axes. Process p_{00} receives $X[0:2, 0:2]$, while processes p_{01}, p_{10}, and p_{11} receive $X[2:4, 0:2]$, $X[0:2, 2:4]$ and $X[2:4, 2:4]$ respectively.

Broadcasting is used when the grid rank and the array rank are different. For example, with the same process grid of size $(2, 2)$ and a smaller array of shape $(4,)$, the mapping `[PARTITION:0, BROADCAST]` will broadcast the array to the process grid such that p_{00} and p_{01} receive $X[0:2]$, while p_{10} and p_{11} receive $X[2:4]$.

5.1.1 Relation to Programs & Distributed Schedules

This class of partitions arises very naturally in array programs and is of central importance to deep learning workloads. Let us elaborate on this class of programs by discussing when each of the iterations can occur.

Consider a two-dimensional map, which can be distributed using a two- dimensional process grid.

```
A = np.random.rand(M, N)
B = np.random.rand(M)
for i, j in dace.map[M, N]:
  ...
```

Partitioned schedules arise when inner accesses index each axis with one loop variable. For instance, `A[i, j]` would induce `[PARTITION:0, PARTITION:1]` and `A[j, i]` would induce `[PARTITION:1, PARTITION:0]`.

Accessing a lower dimension array, in this case `B`, induces broadcasting. For instance, `B[i]` induces `[PARTITION:0, BROADCAST]`, and `B[j]` induces `[BROADCAST, PARTITION:0]`.

Loading whole axes requires replicating the axis. For instance, reading `A[i, :]` requires the mapping `[PARTITION:0, REPLICATE:1]`.

5.1.2 Lowering to MPI

The grid mapped array abstraction is implemented as two library nodes: `ScatterOntoGrid` and `GatherFromGrid`. Given the mapping, the nodes are lowered to at most two MPI calls. Communication of a partitioned axis is implemented by creating an `MPI_Subarray` type for the subarray (the array that is partitioned), and the offsets for each rank are computed statically at initialization time. Computation of these offsets correctly requires some care in handling array strides, dimensions and implicitly broadcasted axes. The offsets are used together with `MPI_Scatterv` (or `MPI_Gatherv`) to communicate the whole subarray.

Replication and broadcasting is handled with a second communication call. We construct two subgrids: one `partition` grid that handles the partitioned dimensions and one `replicate` grid for the broadcasted and replicated dimensions. For scatter, the array is first scattered according to all partitioned axes (using the `partition` grid and `MPI_Scatterv` as described above). Every process in the `partition` grid now has a different subset of the array. Next, these processes broadcast their unique subset of the array as required using the `replicate` grid and `MPI_Bcast`. Each unique subset is broadcast in parallel.

For gather, the process is reversed, and `MPI_Bcast` must be replaced with `MPI_Reduce` according to the write conflict resolution set on the memlets. The processes in the replica-group reduce their conflicted subsets onto the root process for that subset, which is finally gathered onto the global root process.

5.2 Solver

The task of the Grid Mapped Array solver is, given a process grid and a communication constraint, to output, if possible, a mapping such that the communication constraint is fulfilled.

Recall that the communication constraint is a symbolic expression with free variables corresponding to process indices in the given process grid. The solver observes each axis in the subset and attempts to symbolically match the axis expression to an evenly distributed subset and a grid axis. For instance, the subset axis $[i * 2 : i * 2 + 2]$ can be matched to a grid axis with variable i and dimension s, but only if the axis size divided by s is equal to 2. Replication can be matched in cases where the expression covers the whole axis, i.e., [:], and in those cases, there is no constraint on the process grid dimension.

We now construct the grid mapping scheme by assigning each matched partitioned axis to the matched process grid dimension. Next, we assign replicated axes: for these we have some freedom since we can choose any remaining unmapped dimension. Lastly, any remaining process grid dimensions are filled with broadcast.

6 Framework Engineering Contributions

As is typical for systems research, the work in this thesis also includes various contributions to the frameworks it uses, notably the DaCe compiler framework and the DaCeML machine learning optimization framework. In the following, we provide a brief overview of these contributions made during the work on this thesis.

6.1 DaCe

Syntactic sugar for scheduling maps in the Python Frontend This change enables explicitly annotating the `ScheduleType` of a map and the `StorageType` of data containers using the `@` operator. For example the following code can now be parsed:

```
def runs_on_gpu(a: dace.float64[20] @ StorageType.GPU_Global,
                b: dace.float64[20] @ StorageType.GPU_Global):
    # This map will become a GPU kernel
    for i in dace.map[0:20] @ ScheduleType.GPU_Device:
        b[i] = a[i] + 1.0
```

This bypasses the need to run the `GPUTransformSDFG` transformation and enables finer-grained control over the schedule and storage in the SDFG.

Improved support for compile-time constants in SDFG Previously, the compiler backend would not be aware when an input to a `NestedSDFG` is constant, causing compiler errors related to `const` qualifiers on arguments. With this change the evaluation of constants is front-loaded, and the information about is propagated through into nested SDFGs before code generation.

Bug fixes relating to torch tensors This change improved support for passing torch tensors into SDFGs, and fixed a related regression.

Improvements to InlineSDFG Previously, `InlineSDFG` would fail to correctly determine the viewed array name when inlining an SDFG that had view arguments. Also fixed errors relating to computation of subsets on unsqueezed memlets.

Bug fixes related to ufunc reductions Fixed bugs in the NumPy frontend relating to ufunc reductions along `axis=0`.

Change initialization behavior Changed SDFG initialization to only pass symbol values. Previously, SDFGs would receive scalar values alongside the symbols during initialization.

Web serving the SDFG viewer The SDFG viewer can now served on a local port. This is useful for invoking it on a remote host (together with SSH port forwarding).

Improved symbolic broadcasting support Broadcasting, aswell as broadcasted assignment now performs symbolic checks when parsing broadcasted statements. This means, that for example, the following data containers can be broadcasted together:

```
A: dace.float32[N,]
B: dace.float[N * (j + 1) - N * j]
```

6.2 DaCeML

Automatic Differentiation Engine: updated support for views The semantics of viewed data containers were changed with dace versions, and this fixed the engine in light of these changes.

Remove ONNX Runtime dependency ONNX Runtime has historically existed as a hard dependency for DaCeML, namely to execute kernels for which no SDFG implementations exist. With increased maturity, this was no longer necessary, and this change removed the dependency. This involved writing a new constant folding pass that didn't call into ONNX Runtime.

Add automated testing using ONNX framework tests The ONNX library includes basic test cases for almost all of its operators. This change added these test cases to the DaCeML test suite, running all tests for SDFG-based kernels. Besides uncovering a few bugs, this also now provides reports on what subset of operators, attributes and inputs the DaCeML implementations support.

Improved kernel authoring interface The `@python_pure_operator_implementation` decorator was improved to now support computation of AST values that are present when the program is parsed. This enables porting operators like `ReduceMin` to this interface, saving tens of lines of code:

```
@python_pure_op_implementation(axes=lambda node: node.axes)
def ReduceMin(data, reduced):
    reduced[:] = np.min(data, axis=axes)
```

7 Capturing Full Graphs

Contents

Prior work on the DaCeML framework has focused on model compilation for training. To this end, the system converted PyTorch `nn.Module`s to SDFGs, generating code both for the forward and backward passes of the modules. These were then compiled as PyTorch C++ extensions that could then be called from Python code.

Notably, with intended use, the system did not capture any behavior outside of the model, including important parts of the workload such as the loss computation, optimizer delta computation, gradient buffer clearing and parameter updates. These parts of the workload are natural targets for fusion due to their element-wise nature and the excessive data movement cost that this implies.

In the context of distributed scheduling, optimizer computations and state have recently received attention. For instance, ZeRO [Rajbhandari et al., 2020] has proposed partitioning schemes specialized to the optimizer computation.

In chapter, we seek to extend the data-flow captured by the DaCeML system to capture the computation of a training step fully. This will allow our cost model and

```
module = torch.nn.Sequential(
  torch.nn.Linear(10, 10),
  torch.nn.ReLU(),
  torch.nn.Linear(10, 3)
)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(module.parameters(), lr=0.01)

def train_step(x, y):
    # reset gradient buffers
    optimizer.zero_grad()

    # compute the model forward pass
    logits = module(x)
    # compute the loss
    loss = criterion(logits, y)

    # invoke the autograd engine
    loss.backward()
    # perform an optimization step
    optimizer.step()

    return loss
```

Listing 2: A typical training step invoking the backward pass of a module and an optimizer.

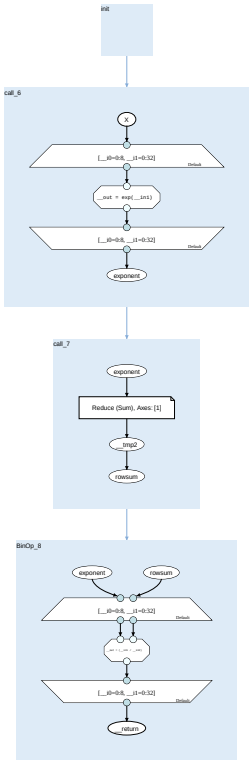
search strategies to transparently consider partitioning schemes for the optimizers jointly with the scheduling of the rest of the workload.

While systems exist for performing automatic differentiation on data flow graphs, these systems typically perform the differentiation given the complete data-flow graph. We wish to parse PyTorch training step code (e.g., Listing 2) into the DaCeML intermediate representation. Listing, the challenge is that the specification that drives the differentiation engine, such as what gradients to compute, and when, are interleaved within the control-centric PyTorch code.

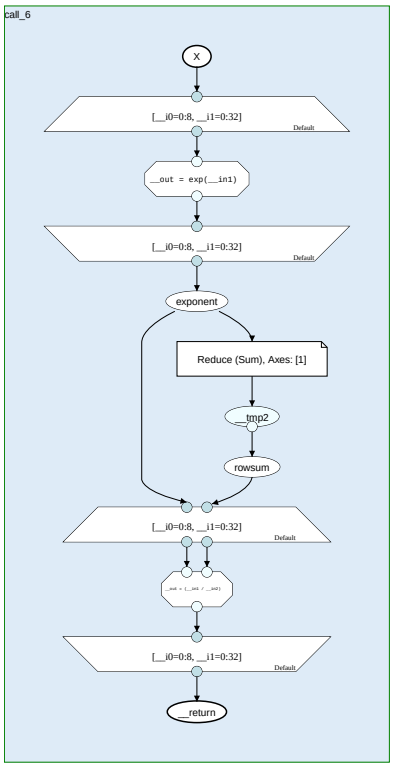
In the following, we propose an approach to translate the control-centric PyTorch API into our data-centric intermediate representation, while retaining the enough semantics of the original API to satisfy the various use-cases.

```python
@dace
def unstable_softmax(X: dace.float32[8, 32]):
    exponent = np.exp(input)
    rowsum = np.add.reduce(exponent, axis=1, keepdims=True)
    output = exponent / rowsum
    return output
```

(a) Annotated Python/NumPy source code.



(b) Control-centric parsed SDFG.



(c) Resulting SDFG after data-flow coarsening (simplification).

Figure 7.1: Parsing a naive, numerically unstable Python/NumPy softmax implementation.

7.1 The DaCe Python Parser

For context, we briefly describe the DaCe python parser [Ziogas et al., 2021], which operates on annotated Python/NumPy programs. Figure 7.1 shows the stages in parsing a simple program. The `@dace` decorator wraps the function as a `DaceProgram`, which will JIT parse and compile the program when it is called.

Upon parsing, the program abstract syntax tree (AST) is traversed in python execution order and decomposed into simpler steps. A separate SDFG state is

inserted for each step, and similar to static single assignment form [Aycock and Horspool, 2000], temporaries are inserted for intermediate results of sub-expressions. Control expressions such as loops or if statements are parsed into a state machine representation. The system parses function calls, such as `np.exp`, by looking for a matching pure implementation in its extensible operator repository.

To extract a data-flow graph from this control-centric SDFG (see Figure 7.1b), the system applies a set of data-flow coarsening (also referred to as simplification) transformations, including inlining SDFGs, fusing states and removal of redundant copies. The resulting SDFG (see Figure 7.1c) represents a data-centric view of the program.

In the following, we will extend the parser by providing replacements for the involved torch functions, notably `torch.autograd.backward`. However, it is noteworthy that any replacements we provide will be called during the single, execution-order AST traversal. This has two important implications. Firstly, the parsed graph will only contain parser output for the AST nodes that would have been executed in a python execution of the program. Secondly, the data-flow graph will be in its control-centric form, and we thus cannot rely on a data-flow graph that is fully connected.

7.2 Extracting the computation to differentiate

We now turn to the DL workloads we wish to parse. Our first task is the extraction of the subgraph to which shall be differentiated. The subgraph is delineated by input 'parameter' arrays, which require gradients to be computed, and the output arrays for which gradients are provided by the user.

To mark parameters, we introduce a new data descriptor class, the `ParameterArray`. This class is a subclass of `dace.data.Array`, and contains the name of a gradient buffer to be used for the parameter. We promote regular `Arrays` to these during parsing via two cases.

Explicit Parameters The first is an explicit call to `Array.requires_grad_()`, which we hook into as a method replacement in the dace parser. Since the parser traverses the AST in python execution order, the parent arrays already exist when parsing the method call, and we can thus promote descriptors directly.

Torch Modules `torch.nn.Module`s are, as in normal DaCeML use, required to be wrapped with the `DaceModule` decorator. To support parsing these modules, we provide an implementation of the `SDFGConvertible` mix-in. When calling `SDFGConvertible`s, the parser inserts a nested SDFG rather than a python callback. Our implementation simply invokes the existing DaCeML PyTorch parser, and inserts the resulting SDFGs.

The `torch.nn.Module` holds references to its parameter `torch.Tensor`s. When converting a module, we construct a closure that wraps pointers to these arrays so that they are passed to the nested SDFG when the program is executed. Notably, parameters that require gradients are promoted to `ParameterArray`s, and gradient buffers are allocated.

When parsing a backward call, we are given the output arrays on which backward is called. We must now determine all input arrays requiring gradients which were used to compute the given output arrays. Notably, when the backward call is parsed, the SDFG is still in its control-centric form. At this stage, we cannot rely on the data-flow simplification passes. To compute this, we introduce a dependency analysis pass that traverses the SDFG state machine, and for each array, computes which arrays were read to produce its value. We again rely on the execution-order parsing which ensures that when parsing the backward call, the input graph has already been parsed. The required input arrays are then simply all `ParameterArrays` on which the output arrays depend. Given the inputs parameters and outputs arrays, the computation we must differentiate is fully determined.

7.3 Parsing backward calls

To parse backward calls, we provide a parser replacement for the `torch.autograd.backward` and `torch.Tensor.backward` methods. These are given the output arrays of the subgraph to differentiate, and with the methods described above we can extract the inputs to the subgraph.

We now introduce a new `BackwardPass` library node that is inserted into the SDFG during parsing. It takes the given gradients of the output arrays as input, and writes to the parameter array's gradient buffers as an output. The library node will only be expanded after parsing is complete, which allows us to delay the invocation of the DaCeML autodiff engine until after parsing and simplification, which is necessary since it only operates on full data-flow graphs.

To ensure that the post-parsing data-flow coarsening does not fuse too many states of the graph, all outputs of the subgraph are initially added as inputs to the library node, modeling conservative control dependencies. Once we have differentiated the graph, and know what outputs are required, we remove these conservative dependencies.

Upon expansion of this node, several tasks are performed.

Subgraph extraction & differentiation Given the input parameters and outputs that are arguments to the backward call, the subgraph is extracted and differentiated using the DaCeML autodiff engine, producing a backward pass SDFG. This SDFG is inserted as a nested SDFG node.

Forwarded Values The produced backward pass may require certain intermediate values from the forward pass. These values are forwarded from the forward pass to the backward pass.

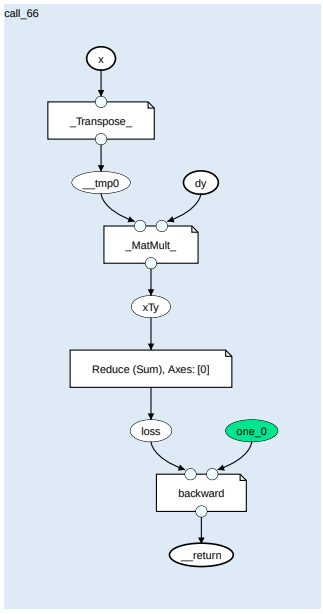
Clearing Parameter Buffers According to PyTorch semantics, the backward pass should accumulate gradients onto a parameter buffers that are initialized to zero. After computing gradients, the PyTorch autodiff engine thus always launches a separate accumulation kernel that adds the gradients onto the

buffer. In our implementation, we have full view over both the forward SDFG and the generated backward SDFG. By performing static analysis on both SDFGs, we can determine when gradient buffers are only written by a single backward pass, which allows us to fully elide the accumulation kernel in some cases.

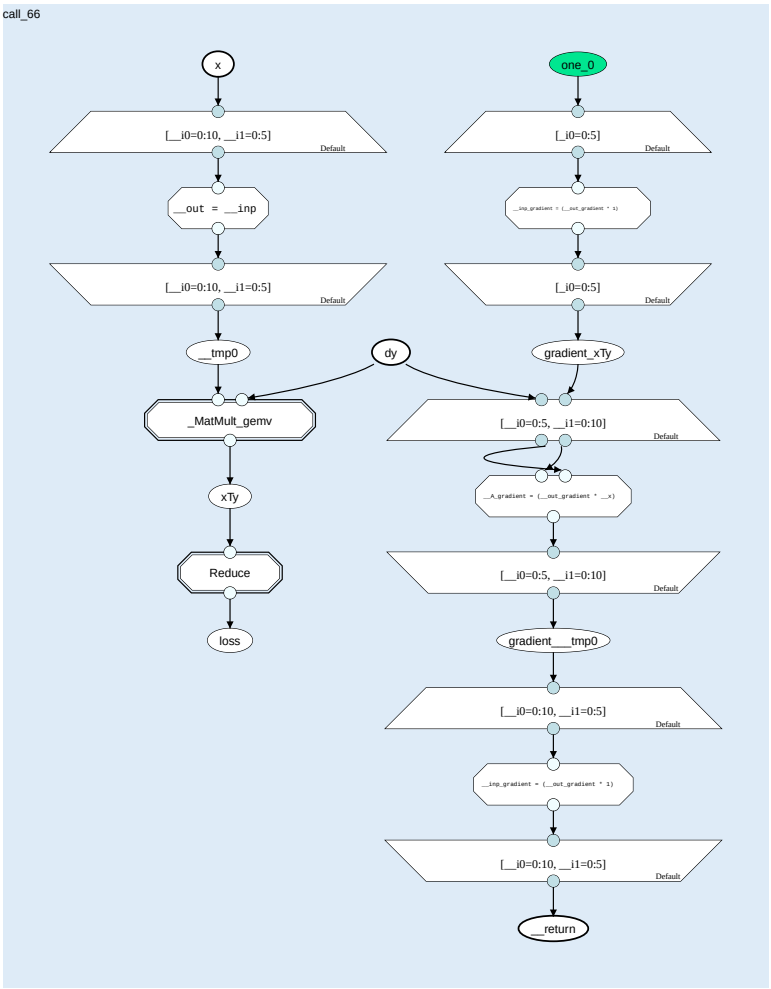
With the proposed design, we fully support functions such as those including multiple backward calls. As an example of the output consider Figure 7.2. Upon backward pass expansion (Figure **??**), the graph is correctly connected to the forward pass, including forwarding of the `dy` value. Furthermore, parts of the backward pass that are independent of the forward pass do not exhibit any unnecessary dependencies, allowing for independent scheduling.

```python
@dace.program
def step(x: dace.float32[10, 5], dy: dace.float32[10]):
    x.requires_grad_()
    xTy = x.T @ dy
    loss = np.add.reduce(xTy)
    loss.backward()
    return x.grad
```

(a) A simple program that calls `torch.autograd.backward`.



(b) The SDFG after parsing. The backward pass library node has been inserted, as well a the data container for the gradient of `x`.



(c) Resulting SDFG after expanding the backward pass library node.

Figure 7.2: Parsing a simple backward call.

8 Evaluation

Contents

We evaluate the correctness of our automated distributed compiler by demonstrating execution of various programs on a distributed cluster. We compile and execute programs on the CSCS Piz Daint supercomputer.[1] Each compute node is a Cray XC50, with a 12-core Intel E5-2690 v3 CPU @ 2.6GHz and an NVIDIA P100 GPU with 16GB device memory. The nodes are connected through a Cray Aries system interconnect in a Dragonfly topology.

All results are parsed from high-level Python/NumPy code using the DaCe Python frontend.[2] The single-process SDFGs are then automatically partitioned and lowered using the process described in Chapter 4, and all communication

[1] See https://www.cscs.ch/computers/piz-daint/ for more information.

[2] Although many workloads are deep-learning related and more concisely expressed in PyTorch, we avoid the DaCeML frontend here to ease understanding.

constraints are automatically solved using the grid mapped array solver described in Chapter 5. Lowering a program with a given `schedule` is as simple as executing:

```
sdfg = program.to_sdfg() # parse to SDFG
daceml.distributed.lower(sdfg, schedule) # partition to SPMD program
```

After partitioning, we compile to machine code using the DaCe SDFG compiler and GCC 9.3.0, finally linking to CUDA-aware Cray MPICH 7.7.18 for communication.

Note: in this chapter, where symbolic array sizes are used in the Python programs (e.g., `N` in `my_kernel(A: dace.float32[N])`, these array sizes were statically fixed at program-parse-time. Currently, our communication solvers only have very limited support for symbolic shapes.

8.1 Lowering

We evaluate our lowering technique by using it to compile several distributed programs, and describe the solutions obtained by the communication solver, verifying their optimally (in terms of communication volume, given the schedule).

8.1.1 Matrix Multiplication

Matrix multiplication is a central primitive in deep learning and other HPC applications. We begin with the follow NumPy code, yielding a three dimensional iteration space.

```
@dace
def matmul(a: dace.float32[M, K], b: dace.float32[K, N]):
    return a @ b
```

Lowering the program with a given `schedule` is as simple as executing:

```
sdfg = matmul.to_sdfg()
daceml.distributed.lower(sdfg, schedule)
```

The program can then be executed on a computing cluster. The compiler successfully compiles this with all tested parallelism schemes. We test using sizes `M=4096, K=4096, N=4096`. All schemes are solved successfully using the grid mapped array solver.

Fully replicated $(1, 1, 1)$ The trivial schedule of full replication only 'communicates' with the root process, since only rank zero participates in the process grid of size 1. The communication is a simple `MPI_Bcast` over the size-one process grid which can be implemented as a simple copy.

Data-Parallel $(n, 1, 1)$ With this scheme, the `a` matrix is partitioned horizontally, and the `b` matrix is fully replicated. Each process computes a data-parallel, horizontal subset of the output, and the output is partitioned the same as `a`. Due to the partial partitioning in `a` and the output, the solver emits two stage communication (`MPI_Scatterv, MPI_Bcast`) for `a` and `MPI_Gatherv` for the output.

Reduction-parallel $(1, n, 1)$ With this scheme, `a` is partitioned row-, and `b` column-wise, communicated in two steps as before. Each process computes a full-sized 4096×4096-sized output, and the output is reduced across all processes in-network using `MPI_Reduce`.

Hybrid $(\sqrt[3]{n}, \sqrt[3]{n}, \sqrt[3]{n})$ The hybrid scheme is partitioned along all three dimensions, combining the approaches explained above: the inputs are fully partitioned using `MPI_Scatterv`, and the output is accumulated using two stage `MPI_Reduce` and `MPI_Gatherv`.

8.1.2 Sparse Matrix Vector Multiplication

With the rising popularity of graph neural networks (e.g., Kipf and Welling [2016]; Hamilton et al. [2017]) and growing interest in sparsification of neural networks [Hoefler et al., 2021], sparse tensor operations are becoming increasingly important. We evaluate lowering a sparse matrix vector multiplication (SPMV), starting with the following NumPy code.

```
@dace
def spmv(A_row: dace.uint32[M + 1], A_col: dace.uint32[nnz],
         A_val: dace.float64[nnz], x: dace.float64[N]):
    b = np.empty([M], dtype=np.float32)
    for i in dace.map[0:M]:
        b[i] = 0
        for j in dace.map[A_row[i]:A_row[i + 1]]:
```

```
            b[i] += A_val[j] * x[A_col[j]]
    return b
```

When lowering with schedule $(2,)$, we notice that this case begins to push the limits of our grid mapped array solver in a subtle way: the subset read from `A_row` is `[0:M + 1]`, which is one-larger than the iteration space size. The rank-tiled subsets are correctly lowered to `[0:M//2 + 1]`, but the grid mapped array solver does not match the pattern correctly since it is limited to solving non-overlapping subsets. We fall back to a more primitive `Scatterv` solver that computes the displacements statically at initialization time by manually instantiating the subset expression for each process in the process grid, and can thus handle overlapping reads without issue.

8.1.3 Warp-tiled Softmax

We now demonstrate that our method is capable of analyzing even hardware specific, low level kernels. We begin with simple NumPy code, and we test on BERT-large [Devlin et al., 2018] input sizes.

```
@dace.program
def softmax(inp: dace.float32[8, 16, 512, 512]):
    rowmax = np.maximum.reduce(inp, axis=-1, keepdims=True)
    exp_arr = np.exp(inp - rowmax)
    rowsum = np.add.reduce(exp_arr, axis=-1, keepdims=True)
    return exp_arr / rowsum
```

Next, we optimize the SDFG for CUDA code using warp tiling (described in Rausch et al. [2022]). The resulting SDFG that is then passed to our lowering procedure is highly specialized to CUDA hardware: it is fully fused, warp-tiled along the last dimension, and contains warp-reduction instructions. Despite this, our framework is able to automatically partition the workload along the first 3 dimensions (corresponding to data-parallelism).

Although not relevant for contemporary architectures, in theory, partitioning along the reduction dimension is possible, which would be useful with a very large reduction dimension. Our framework is able to lower with a schedule that partitions that dimension, but not in the fully-fused operator. Solving that case would require communication within the kernel, which we currently do not automatically handle.

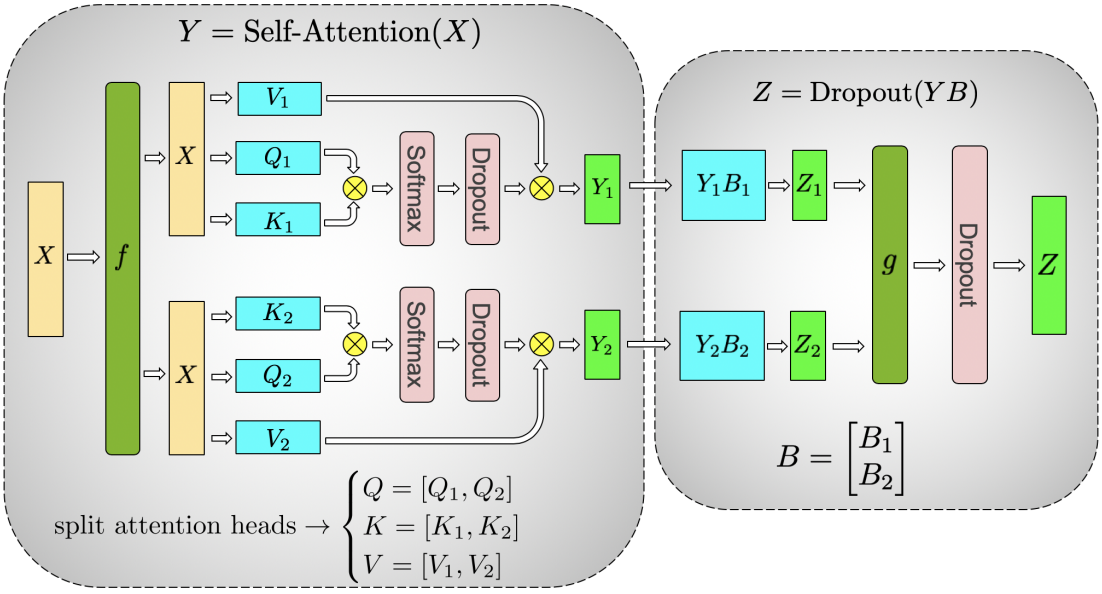


Figure 8.1: The attention-head parallel partitioning scheme from Megatron-LM (Figure reproduced from [Shoeybi et al., 2019]).

8.1.4 Megatron-LM

We now demonstrate the capability to express the popular Megatron-LM tensor parallelism scheme [Shoeybi et al., 2019]. Illustrated in Figure 8.1, the scheme partitions the self attention computation along the head dimension. We automatically partition the following NumPy code, using sizes from BERT-large `B=8, S=512, E=1024, H=16`

```
@dace
def projection(x: dace.float32[B, S, E],
        W_Q: dace.float32[E, H, E_H],
        W_K: dace.float32[E, H, E_H],
        W_V: dace.float32[E, H, E_H]):
    # projection bias ommitted for brevity
    Q = np.einsum('bik,khj->bhij', x, W_Q)
    K = np.einsum('bik,khj->bhij', x, W_K)
    V = np.einsum('bik,khj->bhij', x, W_V)
    return Q, K, V

# we fully fuse softmax as before
def scaled_softmax(x: dace.float32[B, H, S, S]):
    scaled_x = x / SQRT_H
    rowmax = np.maximum.reduce(scaled_x, axis=-1, keepdims=True)
    exponent = np.exp(scaled_x - rowmax)
    rowsum = np.add.reduce(exponent, axis=-1, keepdims=True)
    return exponent / rowsum
```

```
def self_attn(Q, K, V):
    scores = np.einsum('bhik,bhjk->bhij', Q, K)
    norm_scores = scaled_softmax(scores)
    return np.einsum("bhik,bhkj->bhij", norm_scores, V)
```

The projection, softmax and self attention modules above are successfully partitioned along the head dimension. `x` is replicated to all ranks, and projection matrices are partitioned according to head. With this scheme, no inter-process communication is necessary throughout the projection and self-attention computation.

```
def mhsa(x: dace.float32[B, S, E],
         W_Q: dace.float32[E, H, E_H],
         W_K: dace.float32[E, H, E_H],
         W_V: dace.float32[E, H, E_H],
         W_O: dace.float32[E, H, E_H]):
    Q, K, V = projection(x, W_Q, W_K, W_V)
    values = self_attn(Q, K, V)
    # unpermute
    values_permute = np.einsum('bhse->bshe', values)
    values_reshaped = values_permute.reshape([B, S, E])
    return np.einsum('bik,kj->bij', values_reshaped, W_O)
```

Following Megatron, we partition the output projection in data-parallel fashion. As in Megatron, this requires communication, and the solver emits an `MPI_Allreduce` operation for the redistribution after the self-attention and before the output projection.

It should be noted that this is made simpler by the fact that the head dimensions are explicit in the data and the einsums. Popular single-node implementations of self attention often 'fuse' the projection matrices into a single matrix-multiplication followed by splitting the output into `Q`, `K` and `V`. Our automatic partitioning fails on those cases because the head dimension is not explicit in the iteration space. This is similar to the constraint in data-layout-annotation systems (e.g., GSPMD [Xu et al., 2021] and derivatives) to have the parallel dimension explicit as a tensor axis. Extending our framework with the ability to partition the process grid in cyclic distributions is an interesting avenue for future work that would remove this constraint, but increase complexity in the communication solvers.

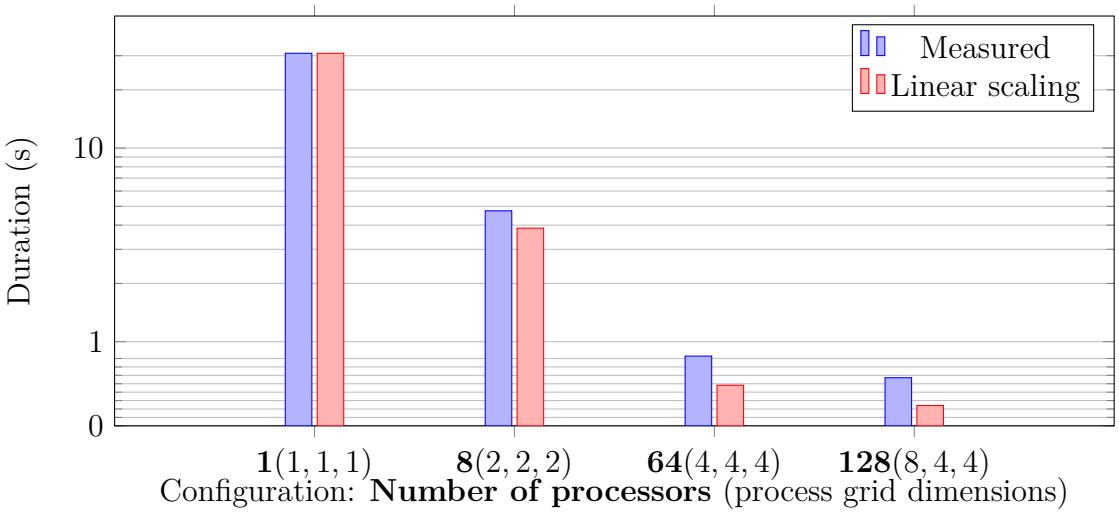


Figure 8.2: Execution duration (seconds) of a $8192 \times 8192 \times 8192$ matrix multiplication with different process grid sizes. Measured durations are median of 5 runs.

8.2 Scalability

A critically important ability of a distributed execution framework is clearly to compile and execute programs on large clusters. Our choice of MPI for communication is particularly helpful in this case for traditional HPC-style clusters such as Piz Daint. We partition, compile and execute a $8192 \times 8192 \times 8192$ matrix multiplication on up to 128 nodes, demonstrating that our framework is capable of operating at large node counts. Figure 8.2 illustrates these results. Although we observe sub-linear scaling, this is in large part due to the fact that the measured workload includes scattering inputs from, and gathering outputs to the root rank.

9 Conclusion

In this work, we proposed a system for automatic partitioning of generic array programs, including deep learning workloads. Contrary to prior work, our approach partitions workloads based on work rather than data, enabling partitioning along non-data dimensions that data-based approaches fail to capture.

Rather than relying on manual operator-based annotations, we propose a system for automatic extraction of communication constraints, which are then solved using one of several communication solvers. In this manner, the system automatically discover parallel algorithms for the workloads, and can lower the workload to an executable that communicates using MPI. This makes it, to the best of our knowledge, the first such system that operates completely without operator annotations and generalizes beyond the popular deep learning operators of today.

The parameterization exposed to the user is simple yet general: the user simply specifies the degree of parallelism on any iteration space dimensions they would like to tile. This generalizes various forms of parallelism, such as data-, model-, spatial-, optimizer- and tensor-parallelism.

The transformation is a transformation over generic SDFGs, and can be freely composed and interleaved with other transformations. As a result, we can automatically derive parallel schedules even for optimized, fused and hardware specialized kernels.

We implement this system as an extension to DaCeML [Rausch et al., 2022], and we demonstrate correctness on several workloads. We also demonstrate the system's ability to compile and execute programs on 128 machines on the Piz Daint supercomputer.

9.1 Future Extensions

We currently plan to extend the system in several ways. The first is GSPMD-like heuristic propagation of partitioning annotations is an important feature for usability. Using several simple propagation heuristics, we can greatly to reduce the number of annotations the user needs to specify.

A holistic cost model for our system is a natural next step, since all communication and data-movement is explicitly modeled in the intermediate representation. A cost model can be as simple as iterating over all edges in the graph and summing their data-movement volumes. This would enable rapid feedback for more effective performance tuning.

Finally, we plan to extend introduce fully-automated search over the schedule space. The current system can automatically extract all parallel dimensions in the workload, and together with a cost model and a search procedure, we can enable fully-automatic scheduling of array programs.

Bibliography

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. (1989). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236. Springer.

Walker, D. W. and Dongarra, J. J. (1996). Mpi: a standard message passing interface. *Supercomputer*, 12:56–68.

Aycock, J. and Horspool, N. (2000). Simple generation of static single-assignment form. In *International Conference on Compiler Construction*, pages 110–125. Springer.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y. (2012). Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

Neyshabur, B., Tomioka, R., and Srebro, N. (2015). In search of the real inductive bias: On the role of implicit regularization in deep learning. In *ICLR (Workshop)*.

Ronneberger, O., Fischer, P., and Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283.

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

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Seide, F. and Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 2135–2135.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Google (2017). XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Unat, D., Dubey, A., Hoefler, T., Shalf, J., Abraham, M., Bianco, M., Chamberlain, B. L., Cledat, R., Edwards, H. C., Finkel, H., et al. (2017). Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020.

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

Yarotsky, D. (2017). Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103–114.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. (2018). Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, pages 2279–2288.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. (2018). Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31.

Bai, J., Lu, F., Zhang, K., et al. (2019). ONNX: Open neural network exchange. https://github.com/onnx/onnx.

Barham, P. and Isard, M. (2019). Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183.

Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.

Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T., and Hoefler, T. (2019). 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*.

Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., and Mansell, D. (2019). Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91. IEEE.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

Jia, Z., Zaharia, M., and Aiken, A. (2019). *Proceedings of Machine Learning and Systems*.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. (2019). Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Sutton, R. (2019). The Bitter Lesson.

Wang, M., Huang, C.-c., and Li, J. (2019). Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17.

Ziogas, A. N., Ben-Nun, T., Fernández, G. I., Schneider, T., Luisier, M., and Hoefler, T. (2019). A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

Advani, M. S., Saxe, A. M., and Sompolinsky, H. (2020). High-dimensional dynamics of generalization error in neural networks. *Neural Networks*, 132:428–446.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*.

Harris, C. R., Millman, K. J., van der Walt, S., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with numpy. *Nat.*, 585:357–362.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. (2020). Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506.

Song, L., Chen, F., Zhuo, Y., Qian, X., Li, H., and Chen, Y. (2020). Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355. IEEE.

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., et al. (2021). Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. (2021). Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(241):1–124.

Hooker, S. (2021). The hardware lottery. *Communications of the ACM*, 64(12):58–65.

Ivanov, A., Dryden, N., Ben-Nun, T., Li, S., and Hoefler, T. (2021). Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732.

Li, S. and Hoefler, T. (2021). Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. (2021). Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. (2021). Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*.

Schaarschmidt, M., Grewe, D., Vytiniotis, D., Paszke, A., Schmid, G. S., Norman, T., Molloy, J., Godwin, J., Rink, N. A., Nair, V., et al. (2021). Automap: Towards ergonomic automated parallelism for ml models. *arXiv preprint arXiv:2112.02958*.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., et al. (2021). Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*.

Ziogas, A. N., Schneider, T., Ben-Nun, T., Calotoiu, A., De Matteis, T., de Fine Licht, J., Lavarini, L., and Hoefler, T. (2021). Productivity, portability, performance: data-centric python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

Ben-Nun, T., Groner, L., Deconinck, F., Wicky, T., Davis, E., Dahm, J., Elbert, O., George, R., McGibbon, J., Trümper, L., Wu, E., Fuhrer, O., Schulthess, T., and Hoefler, T. (2022). Productive Performance Engineering for Weather and Climate Modeling with Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*.

Calotoiu, A., Ben-Nun, T., Kwasniewski, G., de Fine Licht, J., Schneider, T., Schaad, P., and Hoefler, T. (2022). Lifting C Semantics for Dataflow Optimization. In *Proceedings of the 2022 International Conference on Supercomputing (ICS'22)*.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Du, N., Huang, Y., Dai, A. M., Tong, S., Lepikhin, D., Xu, Y., Krikun, M., Zhou, Y., Yu, A. W., Firat, O., et al. (2022). Glam: Efficient scaling of language models

with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR.

Rausch, O., Ben-Nun, T., Dryden, N., Ivanov, A., Li, S., and Hoefler, T. (2022). A data-centric optimization framework for machine learning. In *Proceedings of the ACM International Conference on Supercomputing*.

Sevilla, J., Heim, L., Ho, A., Besiroglu, T., Hobbhahn, M., and Villalobos, P. (2022). Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al. (2022). Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. (2022). Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Unger, C., Jia, Z., Wu, W., Lin, S., Baines, M., Narvaez, C. E. Q., Ramakrishnaiah, V., Prajapati, N., McCormick, P., Mohd-Yusof, J., et al. (2022). Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. (2022). Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578.

Ziogas, A. N., Kwasniewski, G., Ben-Nun, T., Schneider, T., and Hoefler, T. (2022). Deinsum: Practically I/O Optimal Multilinear Algebra. In *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22).