End-to-end Model Inference and Training on Gemmini

Pranav Prakash

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-37
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-37.html

May 9, 2021

End-to-end Model Inference and Training on Gemmini

by Pranav Prakash

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanovic
Research Advisor

4/29/2021

(Date)

* * * * * * *

Professor Yakun Sophia Shao
Second Reader

4/25/2021

(Date)

End-to-end Model Inference and Training on Gemmini

Pranav Prakash
UC Berkeley

Abstract

Gemmini is an open-source generator for systolic-array architectures, allowing for systematic explorations of the accelerator design space. However, the lack of existing software support for Gemmini in machine-learning frameworks (e.g. PyTorch or Tensorflow) poses a significant bottleneck to neural network model evaluation and serving.

To address these limitations we present the design and implementation of a Gemmini backend for Microsoft's ONNX Runtime engine. By extending ONNX Runtime's support for heterogeneous architectures and model graph transformations, the Gemmini backend accelerates the primary computational kernels – matrix multiplications, convolutions, and pooling – while ensuring interoperability between the channel-layouts expected by Gemmini and the rest of ONNX Runtime. We then proceed to benchmark our implementation on a broad-set of networks including ResNet, BERT, and Mask-RCNN – our results show that the Gemmini backend is a performant drop-in replacement for accelerating real-world workloads.

1. Introduction

The surging interest in deep neural networks over the past decade has accordingly led to increased research [25, 43] in the space of hardware accelerators designed to address the computation-heavy workloads that densely connected layers and convolutions pose. Differing use-cases and design constraints – such as training on servers or performing inference at the edge – further increase the diversity of this hardware ecosystem.

Gemmini [15] is an open-source hardware generator explicitly designed with parameterization in mind, to allow for the construction of families of accelerators which can target different workloads but nonetheless share the same instruction set. Coupled with the rest of the Rocket-chip generator ecosystem [2], this allows for rapid exploration of the accelerator design space when trading off power consumption, efficiency, and precision.

However, the benefits of such flexibility in hardware exploration are limited without a means to readily evaluate the performance of the desired workloads (deep neural networks in the most common case). While there exist a variety of training (e.g. PyTorch, Caffe2, TensorFlow) and inference (e.g. TFLite, TVM, Glow) [42, 51, 33] frameworks, the majority support only CPUs and GPUs as first-class citizens – often making use of optimized libraries such as cuDNN or MKL-DNN that corresponding hardware vendors have released. Support for custom, third-party accelerators is less standardized, and can result in various impedance mismatches (in operator granularity) or idiosyncrasies (differing memory layouts) that must be worked around [34].

Thus, one of the significant bottlenecks when working with accelerators without widespread framework support is the need to manually "lower" networks from their graph representation down to the underlying kernels executable by the accelerator – a process that can involve rewriting code, manually performing optimizations, or marshalling data into the correct format. This tedious process not only limits the speed with which hardware designs can be iterated on, but can also hinder the process of hardware-software co-design [29].

In order to embrace the flexibility in hardware diversity made possible by parameterization, it is thus important that there exist a means of easily scheduling such workloads onto the underlying accelerator. In

this paper, we present a Gemmini backend for ONNX Runtime [35] designed to seamlessly allow Gemmini-accelerated inference of any off-the-shelf ONNX network. We evaluate its performance on a wide variety of architectures – such as image recognition, image segmentation, and language models – and demonstrate that its performance is comparable to that of networks implemented by hand. Finally, we show how this work can be extended to allow accelerating neural network training.

2. Background

In this section we first enumerate the features of Gemmini relevant when building out a software stack for inference and training. We then provide some background on the ONNX model format and its surrounding ecosystem – including the ONNX Runtime library on which this project builds.

2.1. Gemmini

Gemmini [15] is a parameterizable generator for hardware accelerators based on a systolic-array architecture. By virtue of its parameterizability, Gemmini supports a diverse set of data types and bit widths – including but not limited to `int8`, `int32`, `float32`, and `bfloat16`; it is accordingly a design goal that the software stack be flexible enough to support all of these as well. While the accelerator *can* be programmed at a low-level by directly issuing instructions according to Gemmini's ISA, Gemmini also generates a header file that provides implementations of DNN-related kernels tuned for a given hardware instance. This header effectively exposes a C API that abstracts away the parameters and configuration of a generated instance, allowing programs to transparently execute on different instances by simply re-linking against the Gemmini-generated library.

In addition to the core primitives of matrix-matrix multiplication and convolution enabled by the spatial array, Gemmini can also support a set of peripheral "compute blocks" to better handle common DNN workloads. These include activation functions such as ReLU, pooling operations such as MaxPool or AveragePool, and transpositions. These peripheral compute units operate on either the input or outputs of the core matrix operations; for instance, a common sequence found in convolution neural networks is "Conv - Pool - Relu" which Gemmini can handle directly on the accelerator without need to shuffle memory between the host CPU and the accelerator's local scratchpad. This ability to handle fused sets of operation on the accelerator boosts performance while reducing power consumption, and is thus crucial to support when designing an inference backend that makes full-use of Gemmini.

2.2. ONNX

ONNX [3] is an open model-interchange format that has gained widespread adoption in the machine learning community as a means of alleviating the interoperability issues that can arise from the multitude of frameworks [42]. The ONNX specification is composed of two parts: a file format encoding the graph of the computation dataflow and an associated set of standardized operators.

The ONNX model format encodes a network in terms of its representation as a directed acyclic graph of operations on tensors – with each high-level operation such as "convolution" or "gemm" conventionally corresponding to a layer in the network. Each operator is recorded as a "node" that includes the operator type, the set of input and output edges, and any additional "attributes" that may be needed to fully specify the behavior of the operator. For instance, a convolution operator might include attributes specifying the stride lengths on each of the image dimensions. While most node inputs can be computed only at run-time – often depending on the output of a previous node which is ultimately derived from user input – others such as the weights of a convolution are fixed during training and must be included in the graph encoding. Such node inputs with fixed tensor values can be specified as "initializers" of the graph, whose values will be directly encoded into the final model file. This information (along with optional metadata such as human-readable documentation strings or type/shape information for non-terminal nodes) is serialized as a protocol buffer (protobuf) [16], with schema provided by the corresponding [13] proto definition files. The resulting `.onnx` file can be imported into nearly all of the previously mentioned inference engines [12, 11] and such files can be natively exported by training frameworks such as PyTorch [40].

Operator types are represented as strings and hence

unconstrained by the protobuf schema; however, to provide a degree of standardization, the ONNX specification additionally consists of a standardized set of operators [14] that are expected to be well-supported by inference engines consuming ONNX models. "Custom" operators not part of this set may be used at the risk of losing interoperability – for instance, ONNX Runtime defines and implements a number of experimental operators such as `Gelu`. (As models containing this operator are not generated or recognized by other tools in the ONNX ecosystem, ONNX Runtime thus also provides a set of scripts to rewrite "standard" ONNX networks into ones containing `Gelu` nodes.)

ONNX groups operators into versioned *operator sets* (opsets) distinguished by *domains* that correspond to different vendors; standardized operators fall under the `ai.onnx` domain, and Microsoft's custom extensions fall under the `com.microsoft` domain.

2.3. ONNX Runtime

ONNX Runtime is an inference engine developed by Microsoft with support for ONNX models as a first-class citizen. Namely, its internal graph representation matches the ONNX graph model closely, allowing for serialization of a post-transformation graph back into an ONNX model. Moreover, with an explicit goal of supporting inference on heterogeneous architectures, its codebase is divided into loosely coupled "provider-dependent" and "provider-independent" modules to provide flexibility and extensibility.

ONNX Runtime uses the notion of an *execution provider* to abstract the notion of the hardware on which a given ONNX node will be run. Each execution provider can register implementations – termed "kernels" – for operators that it supports. The interface between ONNX Runtime and a given execution provider is standardized: each execution provider implements a method `GetCapability()` that accepts a view into a graph of nodes and returns a list of sub-graphs that can be handled. These sub-graphs are then internally assigned to the execution provider that claimed them in a process known as *partitioning*. In most cases, each sub-graph consists of only a single node, and so the nodes of a graph will be divided according to the execution providers that can handle them. Currently, this assignment is done in a greedy manner: searching through execution providers in the order that they were registered, and assigning each node to the first execution provider that can handle it. By convention, the default CPU-based execution provider has lowest priority and is the fallback for any node that has not been claimed by another provider.

Execution providers must separately register a mapping from operator types to kernel implementations. These implementations can be wholly provider-dependent, but they must make use of the pre-defined `Tensor` and `Node` APIs for reading/writing node inputs/outputs as well as accessing associated attributes.

ONNX Runtime additionally supports graph transformations that can be performed both before and after the partitioning process. These graph transformations can arbitrarily rewrite the graph – removing, modifying, or adding nodes – allowing for common optimizations such as constant folding or identity elimination. Furthermore, during partitioning a *subgraph* of more than one node that is claimed by a provider is transformed into a single "fused" node assigned to that provider – allowing graph transforms for the common case of fused-operators to be handled by execution providers themselves.

3. Related Work

In this section, we survey the landscape of frameworks related to inference and discuss their relation to our work. Broadly speaking, inference frameworks can be divided into two classes: graph compilers and graph interpreter engines.

3.1. Graph Interpreters

We define the class of "interpreted inference engines" as those that roughly share a common execution pattern: parsing a model graph at runtime to produce an execution DAG, subsequently performing optimizations via graph transformations, mapping subgraphs onto a predefined set of kernel implementations, and finally calling into each of those kernels as a "black box." Significantly, the specific choice of model affects only the sequence of kernels that are invoked; the granularity of optimizations is limited to that of individual graph nodes, limiting opportunities to specialize kernels against a given model. (Of course, an execution provider may still dispatch between multiple kernel implementations at run-time based on e.g.

input dimensions or attributes.) Along with other popular frameworks including Tensorflow/TFLite (without XLA) [1], Caffe2 [24], and MxNet [6], ONNX Runtime belongs to this class. For a fixed hardware backend (e.g. GPU) most of these frameworks provide similar performance as they often rely on identical underlying kernel implementations (e.g. MKL-DNN on CPUs or CuDNN on GPUs) [48, 6]. A large benefit of this style of framework is their simplicity and ease of extensibility – adding new backends or operators only requires implementing the corresponding kernel as a standard function.

As machine learning models have increased in complexity, however, simple graph transformations have proven insufficient or unwieldy in producing optimized models, resulting in the rise of domain-specific compilers for these model graphs.

3.2. Graph Compilers

Frameworks in the class of "graph compilers" compile a given model down into a fixed binary (either ahead of time or just-in-time), with the generated code implementation optimized to target fixed architectures. Such frameworks have the potential to generate highly efficient code as they are not limited to working with graph transformations and fixed kernels, but can perform optimizations both on underlying tensor operations as well as the generated instructions. These graph compilers are often closer to compilers for traditional languages – having intermediate representations (IR) whose instructions are progressively lowered to target backends – and indeed many leverage existing projects in this space such as LLVM [30] and MLIR [31]. Frameworks in this class include ONNC [34], Glow [44], Tensorflow with XLA, and TVM [7].

This potential increase in performance comes at the cost of complexity in porting new operators and backends; operators must be lowered down into IR rather than expressed directly as functions, and backends require appropriate code-generation support. These can pose a variety of impedance mismatch issues – for instance, a coarse-grained operator that can map onto a backend directly rather than needing to be broken up into fine-grained IR instructions – which must be worked around via e.g. a post-lowering pass.

3.2.1 Halide

Halide [41] by itself is *not* an inference framework. However, the Halide IR which was designed for optimizing the highly parallel tasks of digital image processing has found new applications in graph compilers such as TVM. The core idea of Halide – separation between the high-level specification of the algorithm being computed and the underlying means by which it is scheduled onto the hardware – is in some sense analogous to the idea of scheduling operators of computational graphs onto different providers. In both cases, compilers have freedom to explore various schedules – optimizing for memory locality, performing fusion, tiling, reordering as necessary, etc.

However, Halide's IR is best suited for vector-based architectures (such as SIMD or GPUs) given that it represents operations in terms of loop nests. On the other hand, as Gemmini operates on matrices as primitives, the optimizations that Halide is capable of performing are not directly applicable.

3.3. Relation to Our Work

Our decision to base our work on an interpreted inference engine was due to both its greater flexibility in implementing operators and its ability to generalize to use-cases such as training. Moreover, given that Gemmini can directly handle the majority of compute-heavy bottleneck kernels in most networks, benefits from graph compilers' optimizations would be realized only for those few kernels executed on the CPU. (Note however that the primitive matrix-multiplication operation Gemmini exposes must still be tiled via an outer software loop to act on larger matrices; we consider the problem of finding efficient schedules for Gemmini primitives to be orthogonal to our work.)

Even with an interpreted inference engine, it is possible to take advantage of the optimizations offered by graph compilers by using a graph compiler as an execution provider onto which subgraphs will be scheduled. In such a manner, one effectively JIT compiles subgraphs which are then invoked by the inference engine. This technique can be extended to compute kernels for subgraphs *ahead of time* as well [1], further

[1] As a proof of concept, we have been able to accomplish this by combining pre-existing ONNX graph compilers [18, 52] with ONNX Runtime's support for kernels dynamically loaded at run-

lowering any performance drawbacks and allowing interpreted inference engines to take advantage of automated scheduling.

4. Implementation

In this section we detail the design of our Gemmini backend for inference with ONNX Runtime. We then show how by re-using much of the same underlying infrastructure, our backend can be extended to support training of neural networks as well.

4.1. Inference

At a high level, we add support for Gemmini to ONNX Runtime by implementing a custom execution provider (EP) that schedules convolutions and matmuls on Gemmini. By ensuring that all interaction with the hardware is mediated by calls into the auto-generated Gemmini headers mentioned in 2.1, we can thus immediately support the entire range of design parameters that can be varied in Gemmini-generated accelerators. However, due to layout and granularity mismatches between standard ONNX operators and the primitives exposed by Gemmini, implementing a Gemmini execution provider by naively converting corresponding CPU kernels would pose several challenges:

- Whereas CPU operator kernels can assume 32-bit floating-point data-types, the most common configuration of Gemmini involves a parameterization that uses 8-bit signed integers for inputs and outputs (with an internal 32-bit accumulator). Such a configuration is thus limited to performing inference on neural networks that have been quantized [17], and explicit care must be taken to ensure that such quantized operators are handled by the Gemmini EP.

- Whereas ONNX Runtime traditionally schedules each operator in an ONNX graph independently, Gemmini supports Relu and Max-Pooling as functions that can be performed post-matmul/convolution. For full performance, these must be taken advantage of by adding appropriate graph-fusion transforms.

time via shared-libraries.

- Standard ONNX operators use an `NCHW` memory layout for tensors, where the first two tensor dimensions correspond to batch-size and number of channels, while the remaining dimensions are spatial dimensions (height and width in the most common case of networks using 2D convolutions). Gemmini's convolutions, however, expect tensors to be in an `NHWC` format – with the channel dimension innermost.

In the following subsections, we discuss how our backend addresses each of these points.

4.1.1 Quantized Operators

For `FP32` Gemmini configurations, support for operators that rely primarily on general matrix multiply (`GEMM`) can be implemented by directly converting CPU-based kernels (most of which call directly into BLAS [4] after setting up inputs and outputs) to instead call into the Gemmini headers. Designs parameterized with `int8` inputs, however, are limited to accelerating *quantized* operators; such operators take as inputs not only tensors but also associated scalar "scale factors" which are used to reconstruct the original dynamic range of the tensor from values in the quantized 8-bit range.

Version 10 of the ONNX operator set introduced support for such quantized operators by adding schemas for `QLinearConv`, `QLinearMatMul`, `QuantizeLinear`, and `DequantizeLinear` to the standard set of operators. These operations can be conceptually mapped onto Gemmini's functions, and thus be used to accelerate inference of models that have been appropriately converted to make use of these operators (see Section 5.1).

4.1.2 Operator fusion

Gemmini primarily supports two types of fusion: a ReLU that can be performed post-matmul or post-convolution as part of `mvout`, or max-pooling that can applied after convolution. We handle these two cases via two types of graph transformations.

The former case – fusing a sequence of matmul or convolution followed by ReLU – is handled directly during model partitioning. Before the Gemmini execution provider returns the list of nodes it can handle,

it attempts to pattern-match the operator DAG against the desired `MatMul/Conv + ReLU` sequence. If found, such a two-node sequence is wholly claimed as a *subgraph*, instead of as two individual nodes. As described in Section 2.3, claimed subgraphs of more than one node are internally converted into a single fused node, for which an implementation can be appropriately defined.

We discuss the case of fused pooling – which requires that the preceding convolution be converted to `NHWC` format to be run on Gemmini – in the subsequent subsection.

4.1.3 NHWC Transformations

Unlike the `NCHW` data layouts assumed in standard ONNX models, Gemmini requires inputs for convolutions to be in `NHWC` format. Thus, to allow for hardware acceleration of convolution (and pooling), we first define our own schema for `NHWC` versions of these operators then implement a custom graph-transform pass that converts operators into this `NHWC` format. Note that in addition to converting convolution inputs into an `NHWC` format, convolution weights must also be converted to an `HWIO` format – with the first two dimensions corresponding to spatial height and width of the kernel, the third dimension corresponding to the number of input channels, and the fourth corresponding to the number of output channels.

Transposes are inserted before or after operators as needed to preserve correctness in the case of edges from our custom `NHWC`-aware to standard ONNX `NCHW` operators. While such a transformation may potentially lead to slowdown in the pathological case of alternating `NHWC` and `NCHW` operators (resulting in transposes after each layer), for convolution neural networks whose layers consist mostly of stacked convolution $\rightarrow$ pooling $\rightarrow$ ReLU operations, the bulk of the network can be converted to `NHWC` format with transpositions needed only at terminal nodes. (Note that as Gemmini supports a hardware transposer unit, the case of exotic networks that result in pathological transposes can be addressed by performing transpositions in hardware.) Finally, as part of `NHWC` conversion we fuse any MaxPool nodes into the preceding convolution, if it exists.

Because Gemmini handles only a subset of the convolutions permitted in the ONNX spec (e.g. Gemmini does not support depthwise-convolutions), any `NHWC` convolution that cannot be handled directly by Gemmini uses a fallback path that runs `im2col` [23] on the CPU followed by a matrix-multiplication on Gemmini. In the case of a fused operator, pooling is subsequently performed on the CPU as well.

4.2. Training

ONNX Runtime additionally supports training of neural networks: by augmenting a standard ONNX graph with gradients of operators, it becomes possible to re-use much of the underlying graph execution engine to perform not only the forward pass but also the backward gradient-computation passes and weight updates. Thus assuming an `FP32` Gemmini configuration, the inference backend described in the previous section is sufficient to accelerate both the forward-propagation phase as well as gradient computations for certain operators such as `Gemm`, whose gradients can be expressed directly in terms of acceleratable operators.

Gradients of convolutions, however, require more care. The gradient with respect to weights (dW) can be computed as a unitary-stride convolution of the input against the dilated output gradient [5] (preceded and followed by appropriate transpositions to match tensor shapes). Thus with minimal hardware changes to support filter dilations, computation of dW can be mapped onto a convolution accelerated by hardware. (In the case that Gemmini is configured with an `im2col` block, dW can also be computed by an application of im2col and transposed matrix-multiplication [50].)

Likewise, computation of the gradient with respect to inputs, dX, (an operation known in the literature as "transposed convolution" or sometimes "deconvolution" [10, 45]) can be expressed as a unitary-stride convolution of the dilated output gradient against the filter rotated by $180°$ (reversal in both height and width axes) [5]. Note that unlike in the computation of dW, here dilation is performed on the *input* as opposed to the *filter* of the convolution.

As some operators exhibit different behavior in training versus inference modes, certain transformations that were applicable during inference no longer apply. In particular, during training `MaxPool` outputs an additional tensor containing the indices of elements

from the input that were transferred to the output. As this is currently unsupported by Gemmini, `MaxPool` must be run on the CPU during the forward-pass of training. Finally, care must be taken to add support for `NHWC`-data layouts to all gradient operators, to match the `NHWC` operators added for inference.

5. Evaluation

In this section we discuss how the backend detailed in the previous section can be put into practice to perform inference or training on a variety of real-world networks. We first describe our methodology for producing quantized ONNX networks via automated post-training quantization. Using FireSim [27], we then benchmark these networks on our fork of ONNX Runtime.

5.1. Post-training Quantization

While our inference backend can be used to directly run exported ONNX models, most networks are exported in a floating-point format. Running networks on an `int8` Gemmini configuration, however, requires an ONNX model that makes use of quantized weights and the associated set of quantization-aware operators.

In general, neural-network quantization manifests in two forms [28]: quantization-aware training/fine-tuning – where weights are projected to lower-precision during training and can thus be optimized to retain accuracy – and post-training quantization – where the precision of weights is reduced according to its distribution on sample data but is otherwise not optimized.

As the ONNX ecosystem currently lacks support for exporting quantization-aware trained models, we have opted to make use of post-training quantization in the models used for our evaluation, performed via a version of ONNX Runtime's quantization scripts [36] that were modified to support `int8` data-types and additional quantized operators.

Given a floating-point ONNX model and a set of sample inputs, the aforementioned quantization scripts first record the ranges [2] of inputs/outputs for each intermediary layer, then compute appropriate scale factors; weights and operators such as `Conv` can thus be converted into their quantized equivalents. To improve accuracy in the case of a `Conv` node followed by a pooling or activation, we compute the scale factor of the convolution's output based on the output of that pooling/activation node.

[2] More robust post-training quantization schemes [38, 37] could also be implemented.

Table 1. Comparison of ResNet-50 performance. FPS calculated assuming a 1GHz clock speed.

Execution Mode	Cycles ($\cdot 10^6$)	FPS
Baremetal WS	75	13.33
ORT WS	113	8.85
ORT CPU	131690	0.01

6. ResNet-50

To evaluate the performance of ResNet-50 [20] on our backend, we first performed post-training quantization on the `ResNet50-caffe2` model downloaded from the ONNX Model Zoo [39]. This model was used as input to an "imagenet runner" program written to perform inference on a variety of models designed to be used with the imagenet dataset [8] (e.g. MobileNet [21] or SqueezeNet [22]); it first performs appropriate pre-processing on an input image then calls into the ONNX Runtime library to perform inference. As a benchmark reference, we used a version of ResNet-50 that was hand-implemented in C and compiled to a standalone bare-metal binary.

Our results comparing the inference times from ONNX Runtime using the Gemmini backend, ONNX Runtime running only on CPU, and the baremetal reference are provided in Table 1. Note that results for all models evaluated were obtained from a FireSim simulation of Gemmini (parameterized with its default `int8` configuration) and an in-order Rocket core as the host CPU.

As seen above, the performance obtained using the Gemmini ONNX Runtime backend is within a factor of two of the performance obtained from a hand-coded bare-metal implementation (both of which are three orders of magnitude faster than the CPU-only run). The remaining gap in performance between the two can likely be attributed to three factors:

- When performing post-training quantization on ResNet-50, it was found that quantizing the fully connected layer at the end of the network severely impacted accuracy. As such, this `Gemm` was left

unquantized and performed on the CPU. More robust post-training quantization procedures may ameliorate this, allowing for automated quantization of the fully-connected layer as well, bringing performance up to parity with the hand-rolled implementation.

- The bare-metal implementation makes use of recently added support in Gemmini for accelerating global average pooling. Support for this feature was not present in our ONNX Runtime backend at the time of evaluation.
- The presence of unquantized operators caused by the previous two factors leads to the need to convert between `float` and `int8` when passing tensors between these floating-point operators and quantization-aware ones.

The above is corroborated by Table 2, which breaks down the total inference time by the time taken for each operator type. Though the dominant operator is the `QLinearConv_nhwc` which is performed on Gemmini, both `Gemm` and `QuantizeLinear` nonetheless contribute significantly – accounting for about 25% and 12% of the total time respectively – and the impact of `AveragePool` closely trails at 9%.

Also note that aside from the initial – `QuantizeLinear` and `NHWC Transpose` – and final – `DequantizeLinear`, `NCHW Transpose`, `Gemm`, and `Softmax` – layers, all other nodes have been mapped onto Gemmini. In particular, `MaxPool` and ReLU activation have been fused into the convolution as a single accelerated macro-op.

Table 2. ResNet-50 ORT-WS inference time breakdown per operator. Durations based on the default FireSim base frequency of 3.2 GHz.

Operator	Time (µs)	%
Reshape	14	0.04
Softmax	54	0.16
Transpose	443	1.35
DequantizeLinear	563	1.71
AveragePool	3007	9.16
QLinearAdd	3210	9.77
QuantizeLinear	4071	12.40
Gemm	8037	24.47
QLinearConv_nhwc	13442	40.93
Total	32841	100%

Table 3. Comparison of BERT performance on a 9-token input. Tokens/sec based on assumption of a 1GHz clock speed.

Execution Mode	Cycles ($\cdot 10^6$)	Tokens/sec
ORT WS	238	37.82
ORT CPU	34564	0.26

7. BERT

The performance of our backend on NLP models was evaluated against a BERT model [9] pretrained against the objective of masked language modeling. To procure a suitable model, we first exported the `bert-base-uncased` model from the Huggingface [49] model hub into the ONNX format before performing ahead-of-time model optimization and post-training quantization. As the default ONNX opset lacks a specification for the self-attention operator used heavily in BERT, `Attention` operators are conventionally unrolled into their constituent primitive operators when transformer models are exported into ONNX format; the aforementioned AOT optimization scripts provided by Microsoft fuse such unrolled operators back into a custom `Attention` operator defined in the `com.microsoft` domain.

Within our backend, support for the `Attention` operator (quantized or unquantized) is currently implemented by accelerating the first step of self-attention calculation. That is, the computation of queries, keys, and values [46] vectors (performed via multiplication of input and weight tensors) is mapped onto Gemmini. Note that while the subsequent query-key dot products used in computation of the final attention scores can also be accelerated, this has not yet been added.

Our results on a 9-token input are provided in Table 3. Based on the breakdown of time per operator as shown in Table 4, we see that when using Gemmini, computation of self-attention is no longer the primary bottleneck; instead, these costs are shifted to computation of `BiasGelu` (performed on the CPU) and the associated quantization/dequantization costs of running a floating-point operator after a quantized operator. As we note in the Section 9, these costs could potentially be mitigated by offloading `QuantizeLinear` and `Gelu` computation onto a vector accelerator such as Hwacha [32].

Table 4. BERT ORT-WS inference time breakdown per operator. Durations based on the default FireSim base frequency of 3.2 GHz.

Operator	Time (µs)	%
Unsqueeze	6	0.01
Shape	9	0.01
ReduceSum	13	0.02
Cast	16	0.02
Gather	101	0.13
Slice	159	0.20
LayerNormalization	165	0.21
Add	1829	2.30
DequantizeLinear	4685	5.90
SkipLayerNormalization	4894	6.16
QLinearMatMul	13256	16.68
QAttention	15451	19.45
QuantizeLinear	16858	21.20
BiasGelu	22007	27.70
Total	79449	100%

8. Mask-RCNN

The final network we evaluate is the Mask-RCNN object detection and segmentation network [19]. Because the sub-networks for bounding-box recognition and mask-prediction make heavy use of data-shuffling operators such as `Flatten` and `Squeeze`, to avoid the introduction of redundant `DequantizeLinear` nodes it is important that the quantizer be capable of "passing-through" the scale factors of quantized inputs to such data-shuffling operators.

After exporting [3] and quantizing the pre-trained Mask-RCNN model from the Torchvision library, we can perform inference on a sample image using a runner nearly identical to that which we used for running `ResNet-50` networks.

Our results are summarized in Table 5. The per-op timing breakdown shown in Table 6 reveals that with hardware acceleration of all convolutions and matrix-multiplies, the majority of inference time is instead spent on the CPU, in the form of three operators: `RoiAlign`, `ScatterElements`, and `QuantizeLinear`.

[3]Reference implementations of Mask-RCNN use a `FrozenBatchNorm` operator in the Resnet backbone; as PyTorch does not map these onto ONNX's `BatchNormalization`, before export it is necessary to modify the network to make use of standard `BatchNorm` operators.

Table 5. Comparison of Mask-RCNN performance on an 800x800 image. Duration calculated assuming a 1GHz clock speed.

Execution Mode	Cycles ($\cdot 10^6$)	Sec/image
ORT WS	11572	11.6
ORT CPU	1647055	1647.1

Table 6. Mask-RCNN ORT-WS inference time breakdown per operators. Top-15 operators shown. Durations based on a 1GHz clock speed.

Operator	Time (s)	%
Mul	0.12	1.05
QLinearAdd	0.15	1.31
Slice	0.16	1.39
Loop	0.17	1.48
Div	0.20	1.74
Concat	0.22	1.92
NonMaxSuppression	0.26	2.26
If	0.27	2.35
DequantizeLinear	0.30	2.61
Expand	0.40	3.48
Transpose	0.52	4.53
QLinearConv_nhwc	0.79	6.88
QuantizeLinear	1.35	11.76
ScatterElements	2.37	20.64
RoiAlign	3.53	30.75
Total	10.81	94.15%

The `RoiAlign` operator – which is used in the prediction of masks based on outputs of the region proposal network – is known to pose performance issues even on GPUs due to its control-flow heavy logic [47]. On the other hand, we believe that the chains of `ScatterElements` (and preceding `QuantizeLinear` nodes) in the exported ONNX network are a result of operator unrolling during PyTorch export; more careful analysis and tweaked export procedures could eliminate these. Finally, it should be noted that the Mask-RCNN network contains a single `ConvTranspose` operator used for upsampling. Though this is among the dominant operators in CPU-only runs, when accelerated with Gemmini (via an implementation identical to that in `ConvGrad`) the impact of this operator during runtime becomes negligible.

9. Future Work

While the presented Gemmini backend is reasonably comprehensive in making full-use of Gemmini's features, there are several avenues by which this work can be extended to result in increased performance or power-efficiency.

9.0.1 Cross-accelerator Interaction

The most compelling improvement is exploration of cross-accelerator integration, such as with the Hwacha [32] vector accelerator. While an independent Hwacha execution-provider would allow for the acceleration of bottlenecking operators such as `QuantizeLinear`, `BatchNormalization`, or depthwise convolutions, the combined use of Hwacha and Gemmini backends allows for much richer forms of interaction. For instance with a scratchpad shared between Hwacha and Gemmini, redundant memory-movement could be avoided when passing tensors between operators run on different accelerators. Moreover, given that many operators such as `Attention` require certain parts of the computation to be carried out in floating-point format, Hwacha/Gemmini integration could result in better performance and power-efficiency than could be achieved with either backend independently.

9.0.2 Parallelization

The existing Gemmini backend could also be modified to scale-out inference across multiple Gemminis. ONNX Runtime currently supports three notions of parallelism: task-level parallelism, where for a given network inference on different batches is performed independently across several threads; pipeline-parallelism, where the execution of a given DAG is parallelized rather than performed serially; and data-level parallelism, where the implementation of a single operator can be accelerated by use of multiple threads. Note that as Gemmini does not support context-switching, multi-Gemmini configurations must use one Gemmini per rocket tile. Accordingly, ONNX Runtime will need to be modified to pin each thread in its threadpool to a separate core.

Multiple Gemmini accelerators could also be used in a heterogeneous arrangement of different data-widths, some supporting `int8` and others supporting `fp32`/`bfloat16`. Such an arrangement allows both training and quantized inference on the same chip, which might be useful for edge-computing applications.

9.0.3 BFloat16 Support

The current Gemmini backend supports only Gemmini parameterized with `int8` and `fp32` types. A natural extension of this is to support `bfloat16` datatypes, which has gained increasing popularity [26] in training due to its power efficiency and performance. Given that `bfloat16` kernels are nearly identical to their `fp32` counterparts, this addition would require minimal changes to the Gemmini execution provider itself, with the majority of changes occurring in the bridge layer between ONNX Runtime and the Gemmini hardware primitives exposed via `gemmini.h`.

9.0.4 Bare Metal Binaries

The ONNX Runtime library requires a Linux host environment due to its direct use (e.g. memory allocation & threading) and indirect use (via the protobuf library) of various syscalls. However, we have experimentally found that the required syscall surface is minimal, such that binaries linked against ONNX Runtime run unmodified using Spike's proxy kernel [2] (with the `futex` syscall noop'd, as Spike does not support threading). Given this, it is conceivable that such binaries could be made to work directly on bare-metal platforms by both compiling a minimal subset of ONNX Runtime and providing equivalent implementations of needed syscalls.

10. Conclusion

Given the flexibility in design-space exploration made possible by Gemmini's parameterizations, the ability to easily evaluate neural network workloads is critical when iterating and tuning hardware designs. However, Gemmini currently lacks a high-level software stack, requiring one to manually implement networks – a tedious process which limits the classes of workloads that can be readily run. To address this deficiency, in this paper we have presented a Gemmini backend for Microsoft's ONNX Runtime inference engine. Our backend transparently supports different pa-

rameterizations while taking full-advantage of Gemmini's exposed primitives. Evaluating this backend on a broad set of networks – ResNet-50, BERT, and Mask-RCNN – shows that our backend's performance is comparable to that of hand-written implementations; our backend can thus serve a seamless drop-in replacement for running real industry workloads.

11. Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) through the Real-Time Machine Learning (RTML) Program under award FA8650-20-27006, as well as partially funded by ADEPT Lab industrial sponsors and affiliates.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*, 2019. `https://github.com/onnx/onnx`.

[4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[5] Jake Bouvrie. Notes on convolutional neural networks. 2006.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[11] Open Neural Network Exchange. *Convert ONNX models into Apple Core ML format*, 2019. `https://github.com/onnx/onnx-coreml#convert-onnx-models-into-apple-core-ml-format`.

[12] Open Neural Network Exchange. *Converting Models from ONNX to TensorFlow*, 2019. `https://github.com/onnx/onnx-tensorflow#converting-models-from-onnx-to-tensorflow`.

[13] Open Neural Network Exchange. *ONNX IR*, 2019. `https://github.com/onnx/onnx/blob/master/docs/IR.md`.

[14] Open Neural Network Exchange. *ONNX Operators*, 2019. `https://github.com/onnx/onnx/blob/master/docs/Operators.md`.

[15] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[16] Google. *Protocol Buffers*, 2008. `https://developers.google.com/protocol-buffers/`.

[17] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[18] Halide. *Halide ONNX Converter*, 2019. `https://github.com/halide/Halide/tree/master/apps/onnx`.

[19] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[22] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

[23] Yangqing Jia. Learning Semantic Image Representations at a Large Scale. 2014.

[24] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[26] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[27] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[28] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[29] Kiseok Kwon, Alon Amid, Amir Gholami, Bichen Wu, Krste Asanovic, and Kurt Keutzer. Co-design of deep neural nets and neural net accelerators for embedded vision applications. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[30] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[31] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[32] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and K Asanovic. The Hwacha vector-fetch architecture manual, version 3.8. 1. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262*, 2015.

[33] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[34] Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, and Luis Hsu. ONNC: A compilation framework connecting ONNX to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218. IEEE, 2019.

[35] Microsoft. *ONNX Runtime*, 2019. `https://github.com/microsoft/onnxruntime`.

[36] Microsoft. *Quantize ONNX Models*, 2019. `https://www.onnxruntime.ai/docs/how-to/quantization.html`.

[37] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1325–1334, 2019.

[38] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. Loss aware post-training quantization. *arXiv preprint arXiv:1911.07190*, 2019.

[39] ONNX. *ONNX Model Zoo*, 2019. `https://github.com/onnx/models`.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch:

An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf and https://pytorch.org/docs/stable/onnx.html.

[41] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1):106–115, 2017.

[42] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[43] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. In *2019 IEEE high performance extreme computing conference (HPEC)*, pages 1–9. IEEE, 2019.

[44] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[45] Wenzhe Shi, Jose Caballero, Lucas Theis, Ferenc Huszar, Andrew Aitken, Christian Ledig, and Zehan Wang. Is the deconvolution layer the same as a convolutional layer? *arXiv preprint arXiv:1609.07009*, 2016.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[47] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A unified optimization approach for cnn model inference on integrated gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[48] Zhaobin Wang, Ke Liu, Jian Li, Ying Zhu, and Yaonan Zhang. Various frameworks and libraries of machine learning and deep learning: a survey. *Archives of computational methods in engineering*, pages 1–24, 2019.

[49] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. HuggingFace's Transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[50] Jianxin Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 5:23, 2017.

[51] Yanzhao Wu, Ling Liu, Calton Pu, Wenqi Cao, Semih Sahin, Wenqi Wei, and Qi Zhang. A comparative measurement study of deep learning as a service framework. *IEEE Transactions on Services Computing*, 2019.

[52] Jerry Zhao. *ONNX-Halide: A Halide backend for ONNX*, 2018. https://github.com/jerryz123/onnx-halide.