EFFICIENT DEEP LEARNING INFERENCE ON EDGE DEVICES

ABSTRACT

The real world recognition applications often require deploying deep learning models on edge devices, which lack optimized deep learning library and are limited in computation resource. In this paper, we propose an optimization pipeline for deploying deep learning models on edge devices, which consists of kernel templates library and kernel tuner for optimized kernels, and graph optimization and quantization for minimizing the amount of computation. With the graph optimization module and quantization module, the computation can be reduced. With the kernels templates and kernel tuner module, users can obtain an optimized kernel library for the specified hardware. We have reduced the inference time of ResNet18 from 1120ms to 369ms, MobileNet from 2429ms to 211ms on Raspberry Pi 3B with this pipeline, which enables running real-time computer vision and other applications on most general purpose devices in real time.

1 INTRODUCTION

Recent years, deep learning approaches have become ubiquitous and have achieved better performance on many recognition tasks, such as image classification [1] and language modeling [2]. Along with the exploration of real world applications like robotics, self-driving car, and image processing, we are seeing a rising need for deploying deep learning models on edge devices such as mobile phones, IoT devices, and embedded devices. However, because deep learning models are inherently computation-intensive, the deployment on such devices is nontrivial. Two primary challenges are posed in those scenarios: no maturely optimized computation libraries on those platforms, and limited resource of those devices.

One way to address these problems is to deploy the models on the cloud servers. However, this solution requires continuous network access, which is hard to guarantee and often unavailable, especially for latency-sensitive applications like self-driving car. It also introduces the privacy issue, as users' sensitive data need to be uploaded to the server.

During my internship, we proposed an optimization pipeline for deep learning model deployment on edge devices. The pipeline reduces computation during inference and well utilizes the computation resource of the hardware, making the efficient inference possible on resource limited general purpose devices. Our contributions are summarized as below:

- We propose several graph optimization algorithms to minimize the computation during inference, based on the graph representation provided by NNVM. We also contribute to combination with TVM and the remote deployment feature in NNVM.
- We provide a cost analysis for common tensor operations in deep learning. We propose **a data-packing algorithm** for computing convolution which outperforms im2col on most convolution workloads.
- We propose and implement optimized deep learning operations as kernel templates with some configurations related to hardware specification to handle the diversity between various hardware targets. We also implement a **kernel tuner** module that can search those configurations to get the best performance on a specific hardware target.
- We implement more aggressive model compression and acceleration algorithms. We make a few improvements for current **quantization** algorithm, making it more suitable for current general purpose devices to gain speedup.

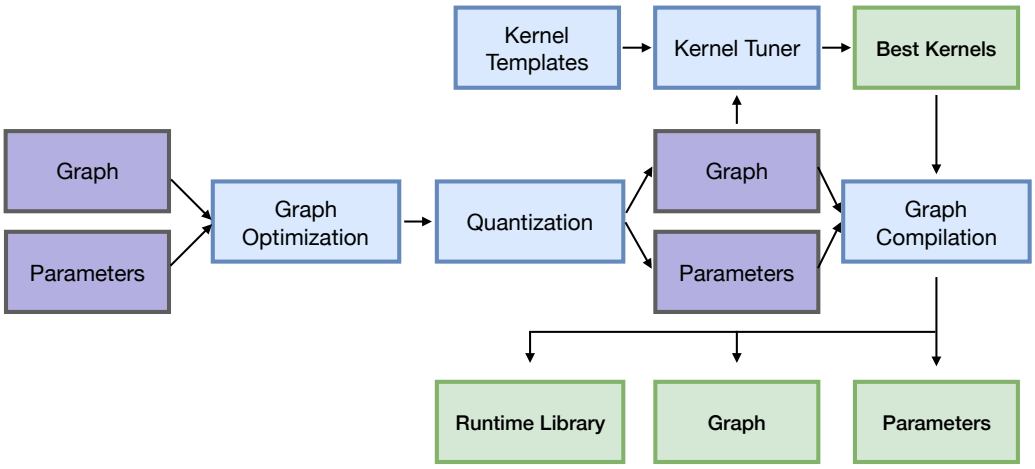


Figure 1: The pipeline of efficient deep learning on edge device.

2 NNVM Compiler

Deep learning workloads can often be represented with a data-flow diagram. **Operators** are the vertexes in the graph, which represent the abstract computation. The operator often binds with a **kernel**, which describes the concrete computational procedure. **Tensors** are the edge between operators, which means a multi-dimensional array with a specific data type.

2.1 Graph Optimization

NNVM provides developers with such data-flow graph abstraction and interfaces for graph manipulation. Developers can implement graph optimization easily based on it.

2.1.1 Pre-Compute

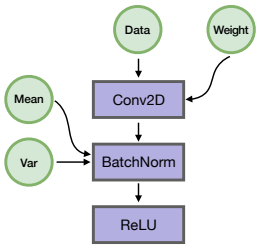
Not all the operations involve input data in the data-flow graph, which means we can pre-compute those operators only related with parameters ahead of inference. It is essential for other graph optimization that transforms the parameters, to avoid putting the burden on inference. The algorithm can be implemented as a graph traversal algorithm on the data-flow graph, starting from input data. Vertexes that can be visited from data cannot be precomputed, since they are dependent on the input data, and other vertexes in the graph can be pruned from the graph and precomputed.

2.1.2 Layout Transformation

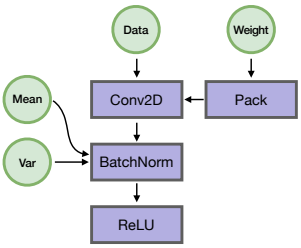
Some kernel requires special layout format to make the best use of memory locality during computation. In the layout transformation stage, we replace those operators with more efficient equivalents with packed data, also insert the corresponding pack operator between its inputs. And after PreCompute, the data packing procedure can be computed before inference.

2.1.3 Inference Simplification

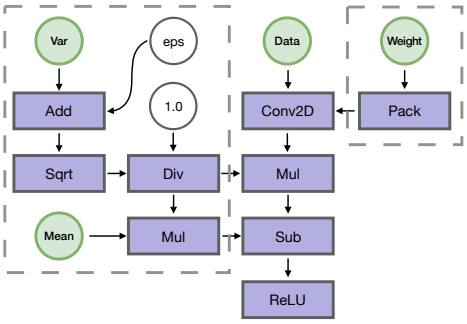
Some operators behave differently during inference, and may provide opportunity for simplification. For example, we can remove *dropout* safely, since it behaves the same as an identity function during



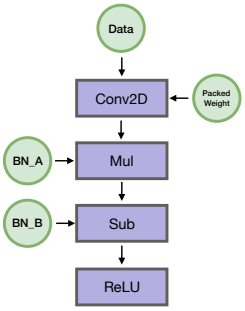
(a) The original dataflow graph.



(b) The graph after layout transformation.



(c) The graph after simplify inference.



(d) The graph after precompute.

Figure 2: Graph Optimization

inference. This saves memory in most deep learning framework with static memory allocation. And the behavior of *batch norm* during inference time is like below:

$$\begin{aligned} \hat{x} &= \frac{x - Mean}{\sqrt{Var + \epsilon}} \\ &= \frac{1}{\sqrt{Var + \epsilon}} x - \frac{Mean}{\sqrt{Var + \epsilon}} \end{aligned} \tag{1}$$

Thus, *batch-norm* operator can be viewed as a broadcast multiplication and broadcast subtraction. With PreCompute, the most time-consuming computation part for $\frac{1}{\sqrt{Var+\epsilon}}$ and $\frac{Mean}{\sqrt{Var+\epsilon}}$ can be executed ahead of inference. This can speed up models with frequent *batch norm* operations such ResNet.

2.2 Remote Deployment

The embedded device often has limited disk space, and we may not want to put all the compilation stack on the target device. The remote-deployment feature in NNVM allows to deploy and run models remotely, which is especially helpful during the kernel-tuning procedure and to achieve minimum runtime in the remote devices. The pipeline is showed in figure 3.

3 High Performance Kernel

The performance gap between naive kernel and well-optimized kernel can be very large. In this section we provide some analysis and guidelines on how to obtain high-performance kernels.

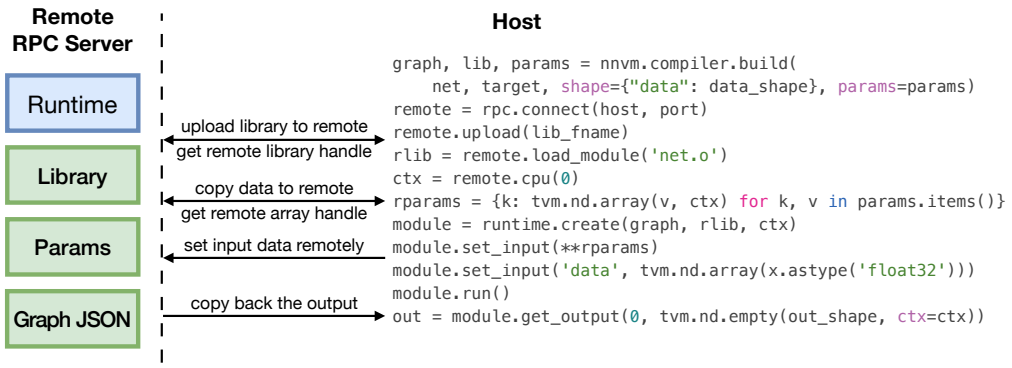


Figure 3: The remote deployment pipeline.

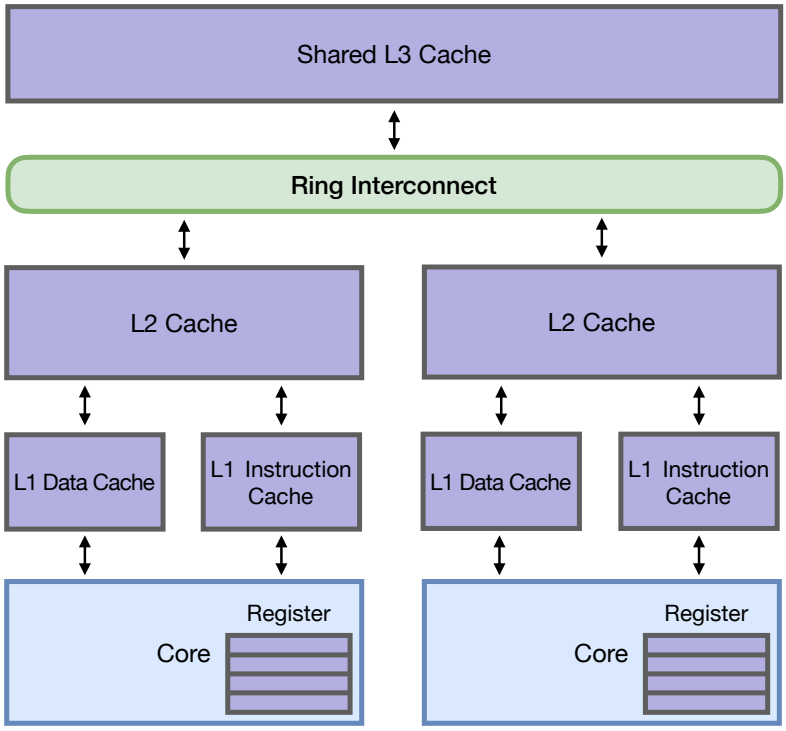


Figure 4: The typical CPU memory hierarchy.

3.1 Cost Analysis

To analyze the approximate cost-savings of the high-performance kernel, we need to take into account of the latency difference between register and cache at all levels. Figure 4 shows a typical CPU-memory hierarchy. The latency of L1 cache reference is 0.5ns, and L2 cache reference is 7ns, which means the computation often be bound by the L2 cache. For example, when we analyze the time-complexity of matrix-matrix-multiplication with sizes $n \times n$ and $n \times n$, the brute-force approach takes time complexity of $O(n^3)$. This actually refers to the number of floating points arithmetic needed in the model. However, a modern architecture goes beyond the FPU(floating point units), and we need to consider the different latencies between different cache levels we have introduced above. Specifically, we need to consider the following costs:

- Time complexity of ALU computing(this is the easiest one, for square gemm it is $O(n^3)$
- Time complexity of register loading
- Time complexity of memory to cache

Let us take a simple example of computing $C = dot(A, B.T)$ to demonstrate how to use this method.

```
float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    register float c = 0;
    for (int k = 0; k < n; ++k) {
      register float a = A[i][k];
      register float b = B[j][k];
      c += a * b;
    }
    C[i][k] = c;
  }
}
```

Register load time cost of A: n^3
Register load time cost of B: n^3
Register number requirement for A: 1 (since we use one register)
Register number requirement for B: 1 (since we use one register)
Register number requirement for C: 1 (since we use one register)
Total time cost:

$$n^3 t_{ALU} + 2n^3 t_{RegLoad} \tag{2}$$

The reason is that for every compute operation, we need to load one element from A and another element from B, thus results in this time cost. Sometimes register loading cost might still be larger than simply compute from registers, so this cost can become bottleneck. This is a common problem for the optimizing kernels within memory hierarchy.

To solve this problem, we need to reuse the data we have already loaded. This is a common technique called **tiling (blocking)**. The general idea is that we divide the matrix by submatrices and perform the computation with the submatrices.

```
float A[n/v1][n/v3][v1][v3], B[n/v2][n/v3][v2][v3],
    C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
  for (int j = 0; j < n/v2; ++j) {
    register float c[v1][v2] = 0;
    for (int k = 0; k < n / v3; ++k) {
      register float a[v1][v3] = A[i][k];
      register float b[v2][v3] = B[j][k];
      c += dot(a, b);
    }
    C[i][k] = c;
  }
}
```

Register load time cost of A: $(n/v1)(n/v2)(n/v3)(v1v3) = n^3/v2$
Register load time cost of B: $n^3/v1$
Register number requirement for A: $v1v3$
Register number requirement for B: $v2v3$
Register number requirement for C: $v1v2$
Total time cost:

$$n^3 t_{ALU} + n^3 (\frac{v1 + v2}{v1v2}) t_{RegLoad} \tag{3}$$

Notice that the total time-cost can be reduced when we increase $v1$ and $v2$. When $v1 \geq 2$, $v2 \geq 2$, we can easily get:

$$Eq.(2) \geq Eq.(3)$$

which demonstrates that tiling can improve the time cost of gemm under this condition. This is because we only load a[v1][v3] once, and reused it multiples times when we actually computing the dot(a, b). The implementation of dot(a, b) vary, but the general idea is that from that point one, a, b already sits in register.

3.2 SPATIAL PACKING

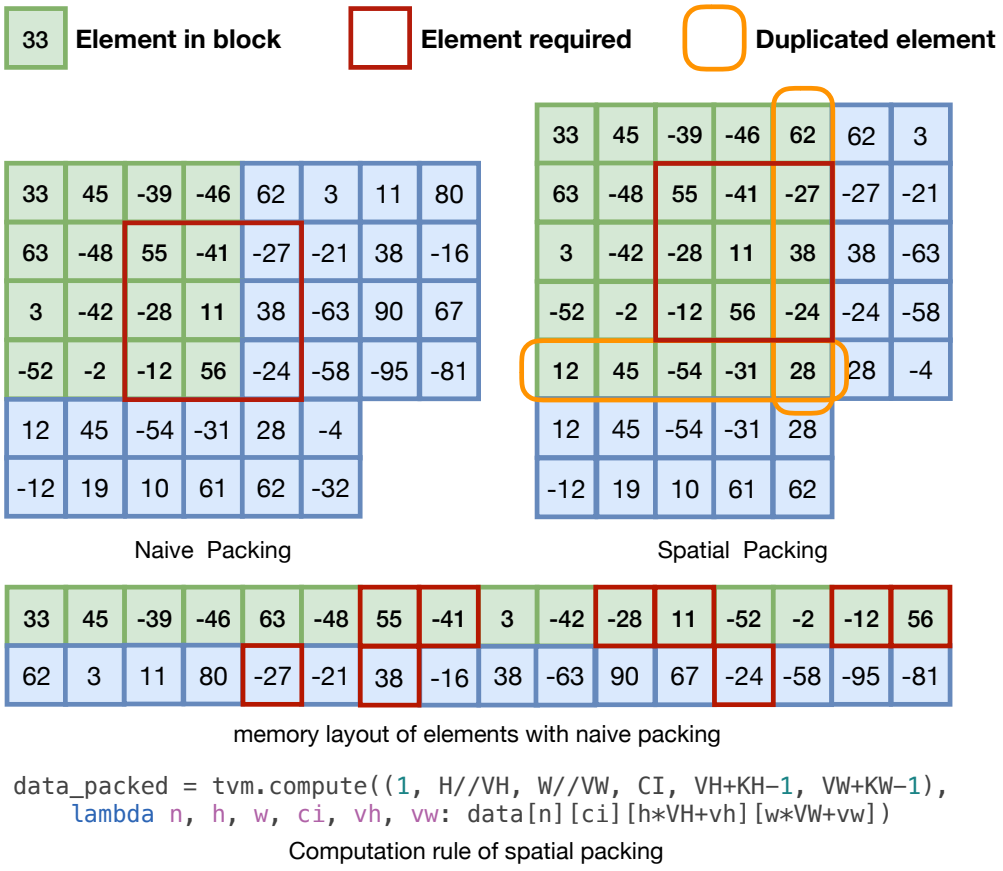


Figure 5: Spatial packing method. The elements are located in different block under naive packing. It may require to load more cache entries.

When we applied the same **tiling** technique on conv2d workload, we split height and width axises of the data for the block. After packing, all the elements in the block are contiguous in memory, so can be loaded to L1 cache with less cache load time. However, in the conv2d workload, the straightforward blocking approach can lead to noncontiguous memory layout. As the figure 5 shows, the computation for the edge element requests the element of the other block, which means more load times to L1 cache. To address this problem, we pad the data with the redundant elements for contiguous access as figure 5 shows. From table 1 we can find this packing method out-performs traditional Im2Col packing on many conv2d workloads.

4 KERNEL TUNER

The register number, memory hierarchy, cache line size are often vary among different hardwares, which can influence the vector lanes, and block size in our kernel implementation. So, instead

H/W	IC	OC	K	S	Im2Col/(ms)	Spatial/(ms)
224	3	64	7	2	1.243	**1.381**
56	64	64	3	1	1.134	**1.539**
56	64	64	1	1	1.162	**1.164**
56	64	128	3	2	1.186	**1.362**
56	64	128	1	2	1.012	**1.078**
28	128	128	3	1	1.084	**1.432**
28	128	256	3	2	1.104	**1.239**
28	128	256	1	2	**1.415**	1.316
14	256	256	3	1	1.095	**1.285**
14	256	512	3	2	**1.245**	0.754
14	256	512	1	2	**1.293**	0.934
7	512	512	3	1	**1.104**	0.758

Table 1: Comparison of Im2Col and Spatial on Conv2D operators of ResNet on Raspberry Pi 3B. H/W for height and width, IC for input channels, OC for output channels, K for kernel size, S for stride size.

of writing the concrete kernels, we implement kernel templates with configuration placeholders. Different hardwares can thus share the same template, but with specified configuration to achieve the best performance.

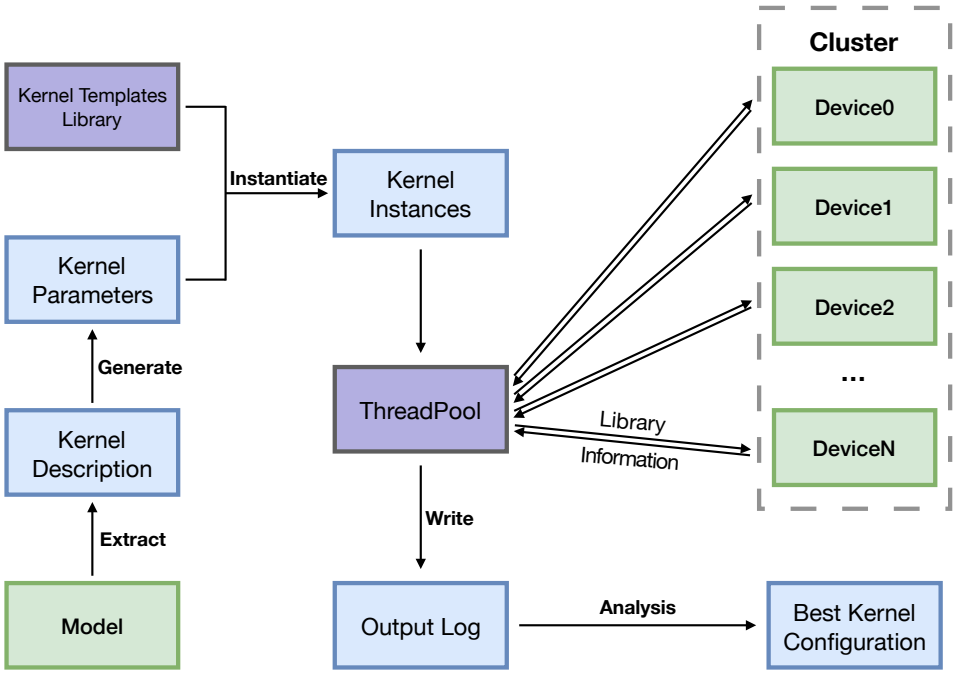


Figure 6: The workflow of kernel tuner.

The kernel tuner can extract the workload description from the model, and generate a set of candidate kernel configuration. Then the real kernels can be instantiated by filling the configuration into the kernel template. Due to the space of configuration can be large, so it is better to have a cluster with multiple devices to explore the space concurrently. We use a thread-pool to manage the communication with the cluster, to compile and distribute the library to the remote devices, and

to retrieve the running log. After the exploration, we analyze the log for the most efficient kernel configurations.

The cluster can be built with local devices or cloud machines. For common devices like mobile phones, there exist some cloud services like AWS Device Farm, which provides the remote access to various phones. For less common devices like embedded devices, you may need to build the cluster with several local devices. In our experiments, we set up the cluster with 10 raspberry pi, and it takes several hours to get the best performance. The hardware and time costs are very reasonable for an optimized kernel library for specialized devices.

We compare the time cost of NNPack and kernels generated by our tuner in table 2. In our experiments, kernel tuner can generate kernels that out-perform NNPack on most conv2d workload.

H/W	IC	OC	K	S	NNPack/(ms)	Kernel Tuner/(ms)	**Speedup**
224	3	64	7	2	44.06	21.93	2.01
56	64	64	3	1	24.42	21.45	1.14
56	64	64	1	1	7.69	3.56	2.16
56	64	128	3	2	16.06	12.62	1.27
56	64	128	1	2	3.09	1.86	1.66
28	128	128	3	1	28.78	21.36	1.35
28	128	256	3	2	14.47	15.36	0.94
28	128	256	1	2	2.2	2.15	1.03
14	256	256	3	1	56.44	23.48	2.4
14	256	512	3	2	23.22	28.56	0.81
14	256	512	1	2	3.37	2.54	1.33
7	512	512	3	1	179.81	62.48	2.88

Table 2: Comparison of NNPack and kernels generated by tuner on Conv2D operators of ResNet on Raspberry Pi 3B. The columns are: H/W for height and width, IC for input channels, OC for output channels, K for kernel size, S for stride size.

5 QUANTIZATION

The parameter of deep learning models can take up a lot of space on disk. Many past works convert the parameter of model with low bit representation to compress the size of parameters, which quantization means. The low bit representation also increases the throughput of the kernels, so it can potentially accelerate the inference speed.

5.1 REPRESENTATION

Although Han's work [3] demonstrates that nonlinear quantization method can achieve better accuracy than linear quantization, it may incur more computation cost on general devices. So we focus the discussion on linear quantization method.

v denotes original value, q represents quantized value, k represents the order of magnitude of v, and s represents the scale (power of 2) representation.

$$v_i = 2^{s_i} * q_i$$

$$|v_i| < 2^{k_i}$$

Since the value ranges in different operators often vary, we assign different scale factors for every tensor instead of using a global scale factor. This improves accuracy.

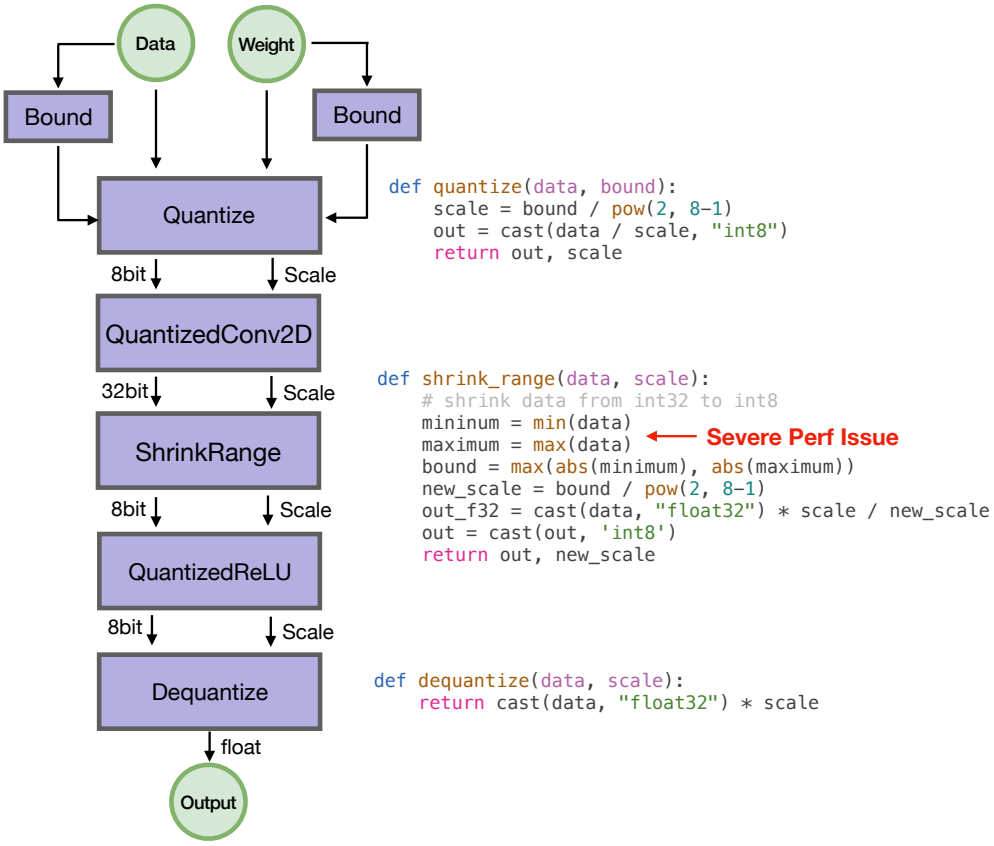


Figure 7: The dynamic quantization approach.

5.2 Dynamic Quantization vs. Static Quantization

In the beginning, we have investigated the dynamic quantization approach, which TensorFlow adopts. Besides replacing all the individual ops with quantized equivalents, this approach also maintains the scale of the current quantized value. For operators such as *QuantizedConv2D*, *QuantizedMatMul*, it persists the results in int32 in order to avoid overflow. However, this approach requires another 32-bit to 8-bit *ShrinkRange* operator after this op, since the real range only occupies a small part of the domain that int32 can represent. Although this method can achieve good accuracy, it actually has severe performance issue. The *ShrinkRange* requires calculating min and max, which is the bottleneck on both CPU and GPU.

To address the performance issue, we adopt the quantization method described in [4]. There are two major differences compared to the dynamic quantization mentioned above:

- Since the magnitude of every tensor is often the same among different samples, we can estimate the range in advance with a small dataset, instead of calculating it during runtime.
- Relax the scale factor to the nearest exponent of two, so that we can use shift to replace unnecessary float multiplication, which is more efficient, as well as friendly for FPGA.

In our experiments, we also found stochastic rounding introduced in [4] is very helpful for accuracy.

$$Round(x) = \begin{cases} \lfloor x \rfloor & \text{w.p.} 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{w.p.} \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}$$

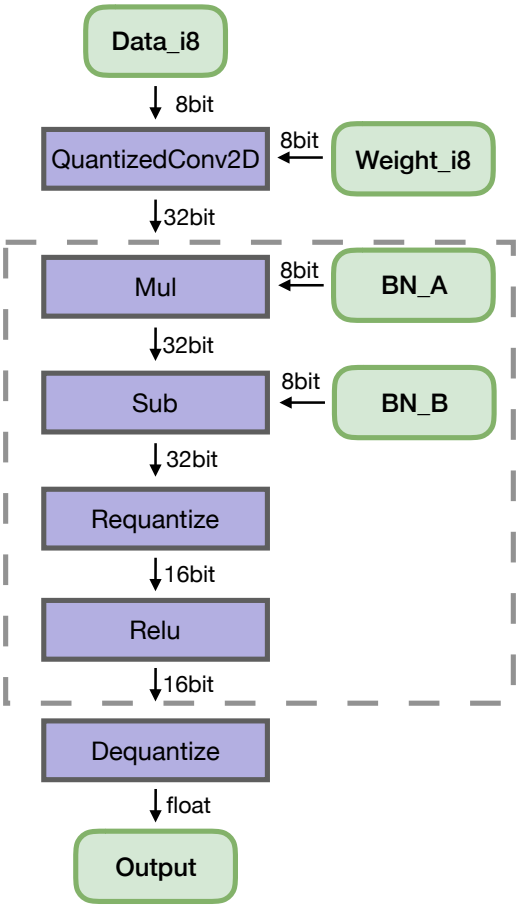


Figure 8: A residual block example with static quantization approach, which shows good performance on CPU. Operators in the grey box can be fused together to speedup further.

In stochastic rounding, the probability of rounding x to $\lfloor x \rfloor$ proportional to the proximity of x to $\lfloor x \rfloor$. Compared with the normal round-to-nearest, stochastic rounding is unbiased and has the desirable property that the expected rounding error is zero: $E(Round(x)) = x$.

5.3 Specialize to Edge Devices

While the original quantization is often designed for specialized hardware, we also make alternation to make it more suitable for general purpose devices:

- On general CPU, the common SIMD instruction is often int16 to int32 FMA instead of int8 to int32, and the performance difference can be observed from table 3, so we store the intermediate value with 16bit, while saving the parameters with 8bit.
- Re-quantize only when necessary. For example, in a residual block, the quantized conv2d, multiplication, and subtraction are all faced with overflow risk. Instead of re-quantizing int32 intermediate value to int8 every time. We only re-quantize once in one residual block.

In our experiments, this configuration would lead to the best speed-up while keeping compression rate.

H/W	IC	OC	K	S	F32/(gflops)	I8_I32/(gflops)	I16_I32/(gflops)	I8_I16/(gflops)
224	3	64	7	2	5.30	5.80	6.59	11.16
56	64	64	3	1	6.55	5.83	8.78	14.60
56	64	64	1	1	3.48	5.14	4.63	6.62
56	64	128	3	2	4.87	4.86	7.32	13.59
56	64	128	1	2	3.69	4.34	4.33	6.01
28	128	128	3	1	6.75	6.05	8.83	15.31
28	128	256	3	2	4.34	4.83	6.32	14.38
28	128	256	1	2	3.37	3.86	5.12	7.38
14	256	256	3	1	6.55	6.15	8.79	15.50
14	256	512	3	2	3.99	4.86	4.81	9.86
14	256	512	1	2	2.55	4.58	3.37	5.61
7	512	512	3	1	3.94	4.64	4.59	8.85

Table 3: GFLOPS with different data type on Conv2D operators of ResNet on Raspberry Pi 3B. The columns are: H/W for height and width, IC for input channels, OC for output channels, K for kernel size, S for stride size.

Model	MXNet/(ms)	NNVM/(ms)	NNVM GOpt/(ms)	NNVM GOpt INT/(ms)	**Speedup**
ResNet18	1120	566	512	369	3.04
MobileNet	2429	289	211	TODO	11.51

Table 4: The speedup with graph optimization and quantization. GOpt for graph optimization, INT for quantization model.

6 Conclusion

With graph optimization, kernel tuner and quantization, we can achieve extremely efficient inference on edge devices. With 3.04x speed-up on ResNet18 and 11.51x speed-up on MobileNet, it makes the real-time deep learning application possible.

We are working on expanding the result to more models and more devices like mobile phones. Also, the kernel tuning can be viewed as an online machine learning problem. We will investigate more heuristic learning based method to search the configuration in the future.

7 Acknowledgments

This report has been prepared for my internship that has been done in the Amazon AI in order to explore and implement how to deploy deep learning models on edge devices. I have successfully completed the project and compiled this report as the summary and the conclusion that have drawn from the internship experience.

I would like to express my sincere gratitude to my internship mentor Mu Li for providing this opportunity to work in Amazon AI, and his great guidelines for my internship despite having his busy schedule. I am also thankful to Tianqi Chen for giving me precious instructions which were extremely valuable for my project. I am also grateful to the thorough advice for the report from Sheng Zha and Yida Wang. Lastly, I would like to thank all member of MXNet Research providing documents, papers, data, figures and services as well as sharing their experience with me and teaching me different techniques to help me complete my project effectively and efficiently. I choose this moment to acknowledge their contribution gratefully.

The time in Amazon AI is very audacious and supportive to my career through which I have gained valuable work experience. Therefore, I consider myself as a very lucky individual to be a part of it.

I am also grateful for having a chance to meet so many wonderful people and professionals who led me though this internship period.

8 APPENDIX

8.1 AN EXAMPLE KERNEL TEMPLATE

```
def gemm_template(N, M, L, UNROLL_FACTOR, VEC_FACTOR, BLOCK_FACTOR,
    L1_FACTOR, L2_FACTOR):
   P, Q, R = UNROLL_FACTOR, VEC_FACTOR, BLOCK_FACTOR
   N, M, L = N / P, M / Q, L / R

   A = tvm.placeholder((N, L, R, P), name='data')
   B = tvm.placeholder((M, L, R, Q), name='weight')

   l = tvm.reduce_axis((0, L), name='l')
   r = tvm.reduce_axis((0, R), name='r')

   C = tvm.compute((N, M, P, Q), lambda i, j, p, q: \
      tvm.sum(A[i][l][r][p] * B[j][l][r][q], axis=[l, r]),
      name='gemm')

   s = tvm.create_schedule(gemm.op)

   CC = s.cache_write(C, "global")
   AA = s.cache_read(A, "global", [CC])
   BB = s.cache_read(B, "global", [CC])

   ci, cj, cp, cq = C.op.axis
   ck, cr = C.op.reduce_axis

   cio, cii = s[C].split(ci, L2_FACTOR)
   cjo, cji = s[C].split(cj, L3_FACTOR)
   s[C].reorder(cjo, cio, cji, cii)

   s[C].parallel(paxis)
   s[C].pragma(cjo, "parallel_launch_point")
   s[C].pragma(cji, "parallel_stride_pattern")
   s[C].pragma(cio, "parallel_barrier_when_finish")

   s[CC].compute_at(s[C], paxis)
   cci, ccj, ccp, ccq = CC.op.axis
   cck, ccr = CC.op.reduce_axis
   s[CC].reorder(cci, ccj, cck, ccr, ccp, ccq)

   s[CC].unroll(ccp)
   s[CC].vectorize(ccq)

   return s
```

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] Tomas Mikolov, Martin Karafit, Luks Burget, Jan Cernock, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH*, pages 1045–1048. ISCA, 2010. URL http://dblp.uni-trier.de/db/conf/interspeech/interspeech2010.html#MikolovKBCK10.

[3] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015. URL http://arxiv.org/abs/1510.00149.

[4] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015. URL http://arxiv.org/abs/1502.02551.