Towards efficient mapping of BNNs onto embedded targets using Tensorflow/XLA

Christoph Gratl
Materials Center Leoben Forschung GmbH
christoph.gratl@mcl.at

Manfred Mücke
Materials Center Leoben Forschung GmbH
manfred.muecke@mcl.at

Günther Schindler
Heidelberg University
guenther.schindler@ziti.uni-heidelberg.de

Holger Fröning
Heidelberg University
holger.froening@ziti.uni-heidelberg.de

Abstract

Binarised Neural Networks hold significant promise for efficient inference on embedded devices and have received significant attention recently. We investigate if Tensorflow and its XLA compiler infrastructure are suited for generating efficient embedded code given a trained BNN. We detail the Tensorflow/XLA/LLVM work flow, building on Tensorflow's XLA high-level optimiser and following LLVM infrastructure. We explore XLA generated code for dot products as well as supported data types. We build a binary dot product in LLVM IR and show the efficiency of the resulting code. We identify shortcomings of the current Tensorflow/XLA/LLVM tool chain and suggest future work to yield efficient auto-generated code for binary dot products.

Introduction

Artificial neural networks (ANNs), especially deep ANNs (DNNs) have pushed the limits of machine learning in the past years [9]. ANNs are typically prototyped using single-precision floating-point and are expensive to execute on classical computer architectures. With the increased relevance of DNNs, and their huge computational complexity, there is a fundamental interest in more efficient training and inference. In this work we focus on efficient inference. This means improving the match between DNN representation, inference algorithm and available computer architecture. One example of similar efforts is the half-precision floating-

point number format (16bit). This format is now supported by many GPU and CPU SIMD units [8] and extensively used in DNN training and inference.

Courbariaux et al. [4] have shown how ANNs can be trained such that weights and/or activation functions rely on binary values only, now known as binarised neural networks (BNNs). Recent works like [10, 14] have improved the classification accuracy achievable with BNNs, thereby reducing the gap to ANNs relying on higher-precision data types. Blott et al. [2] have developed a framework for exploration of BNNs and mapping onto FPGAs, yielding excellent performance figures of up to 50 TFLOP/s per device. For CPUs, Courbariaux et al. [4] have suggested the implementation of binary dot products using bit-parallel XNOR and population count – instructions which are available on most CPUs. We have demonstrated the performance advantage of reduced-precision number formats in matrix multiplication on general-purpose CPUs [11], including an implementation using bit-parallel XOR and population count for binary matrix multiplication. Our results include a 21x speed-up of binary matrix multiplication over float32 on an ARM Cortex-A15.

We believe that BNNs are an excellent choice to execute ANNs on constrained devices. In this work, we explore efficient mapping of BNNs onto embedded CPUs (in contrast to ASICs or FPGAs). We are interested in how to best exploit the pervasively available ARM ISA, allowing more intelligent devices based on existing CPU architectures. There exist some frameworks for compressing neural networks. Some target classical reduced-precision [6, 5], some target specifically binarised neural networks [3, 13, 7]. Some target exclusively FPGAs [3]. Changing tool infrastructure is a major investment. We are therefore interested in understanding to what extent the established Tensorflow machine learning framework can accommodate for efficient implementation of BNNs. This is even more relevant as Tensorflow relies on LLVM for many optimisations and final target-specific code generation.

This work aims at (i) sketching how Google Tensorflow/XLA and LLVM could be adapted to support efficient code generation for BNNs and at (ii) identifying related open issues on the way.

Background

Tensorflow [1] is currently one of the dominant software frameworks for machine learning. With Tensorflow/XLA (XLA stands for accelerated linear algebra) there is a dedicated code infrastructure for generating static code.

The Tensorflow/XLA workflow consists of (i) generating a *Tensorflow graph* representing a trained ANN, (ii) converting it to an *XLA graph*, (iii) performing high-level optimisations and (iv) generating architecture-independent *LLVM IR code*, (v) performing target-specific transformations on the IR code (through LLVM) and (vi) generating assembly code for the target architecture (through LLVM).

LLVM IR features a very generic integer data type allowing for any bit width from 1 bit to $2^{23} - 1$. We follow the LLVM notation, where `i1` denotes a single-bit integer and `iWxL` denotes a packed bitstring holding L integer values of W bits each.

In an ideal (tool) world, we would wish the above workflow to provide the following in order to allow for proper exploration of number formats used:

1. Users express ANN weights and activation functions in Tensorflow variables approximating the real numbers;

2. Training algorithms can explore finite number spaces for weights and activation functions, annotating the types on the way with the minimal required precision, including the extreme case of 1-bit wide number representation.

3. All Tensorflow graph operations using any data type can be mapped into XLA graph operations

4. XLA generates vectorised IR using arbitrary long vectors of arbitrary precision operators (including i1)

5. LLVM partitions long i1 vectors to i1x32, i1x64 or i1x128, as most suitable for target ISA)

6. Given vectorised IR code using i1x32/64/128 operators, LLVM generates efficient code for the target architecture implementing relevant operations in bit-parallel instructions, using e.g. XNOR and popcount

Experiments

In the following, we report on a set of experiments to evaluate selected steps of the ideal tool flow sketched above in the context of Google Tensorflow/XLA/LLVM. We first use a float32 dot product to demonstrate the behaviour of the existing Tensorflow/XLA/LLVM tool chain (Listings 1-3). Then the effect of using the dedicated `tensordot` operator is shown (Listing 4-6). Then we modify the float32 IR code to accommodate packed 1-bit integer vectors and show the effect of LLVM optimiser passes as well as generated ARMv7 code (Listing 7-9).

All experiments are performed using Tensorflow version 1.12.0 via Python interface. For standalone experiments with IR, LLVM v6.0.0 is used. Tensorflow uses the LLVM optimiser with the following options: `-O3 -inline -loop-vectorize -slp-vectorizer -S`. The disassembly dumps reported in the listings were generated using objdump: `arm-linux-gnueabi-objdump`.

Listing 1 introduces our running example, a Tensorflow graph with two float32 input vectors and two nodes (elementwise multiplication and summation) explicitly formulating a dot product.

Listing 1: Tensorflow graph with explicit specification of vector dot product with float32 operands (TFG1)

```
input_x_t = tf.placeholder(tf.float32, shape = [1024*1024], name='input_x')
input_y_t = tf.placeholder(tf.float32, shape = [1024*1024], name='input_y')
sum_t = tf.reduce_sum(tf.multiply(input_x_t, input_y_t), name='output_x')
```

Listing 2 shows the XLA-generated IR for the graph described in Listing 1. Observe that the elementwise multiplication and summation are executed in separate loops.

Listing 2: LLVM IR of Listing 1 as generated by XLA

```
L8: define internal void @output_x—reduction12([..]) #0 {
[..]
L19:  %6 = fadd fast float %4, %5
[..]
L22: }

L24: define void @entry([..]) #0 {
[..]
L50: multiply.17.5.loop_body.dim.0:
L55:  %11 = fmul fast float %8, %10
[..]
L76: reduce.17.13.inner.loop_body.reduction_dim.0:
L89:    call void @output_x—reduction12([..])
[..]
```

Listing 3 shows the assembly code as generated by XLA+LLVM. Observe that LLVM generates vectorised NEON code (vmul, vadd, operating on four float32 in one 128bit register), yet does not merge the loops.

Listing 3: ARMv7 asm of float32 dot product as generated by XLA/LLVM from Listing 1+2

```
L18: 28: f3420df0 vmul.f32 q8, q9, q8
[..]
L33: 64: f2420de0 vadd.f32 q8, q9, q8
```

Listing 4 constructs an alternative float32 dot product implementation using tf.tensordot (identical to numpy.tensordot). tf.tensordot uses a custom implementation in XLA, relying on Eigen library calls for some cases.

Listing 4: Graph with single-operator vector dot product (TFG2)

```
input_x_t = tf.placeholder(tf.float32, shape = [1024*1024], name='input_x')
input_y_t = tf.placeholder(tf.float32, shape = [1024*1024], name='input_y')
sum_t = tf.tensordot(input_x_t, input_y_t, 1, name='output_x')
```

Listing 5 shows the resulting LLVM IR code of the Tensorflow graph using tensordot when compiled with XLA.

Listing 5: XLA-generated LLVM IR from Listing 4

```
[..]
%60 = getelementptr inbounds float, float* %18, i64 %dot.inner.tiled.indvar
%61 = bitcast float* %60 to <4 x float>*
%62 = load <4 x float>, <4 x float>* %61, align 4
%63 = fmul fast <4 x float> %41, %.splat
%64 = fadd fast <4 x float> zeroinitializer, %63
%65 = fmul fast <4 x float> %44, %.splat2
%66 = fadd fast <4 x float> %64, %65
[..]
```

This LLVM IR is transformed to the optimized ARMv7 assembly in listing 6. LLVM emits an efficient vectorised fused multiply-accumulate instruction (`vmla`).

Listing 7 shows hand-optimized IR where the single bits are packed into int32 variables to allow for vectorization. This code has not yet been optimized by LLVM and the operations are done in simple loops.

Listing 6: ARMv7 assembly generated by XLA and LLVM for tensordot

```
[..]
L51:  ac: ed961a00 vldr s2, [r6]
[..]
L54:  b8: ed962a00 vldr s4, [r6]
[..]
L57:  c4: ee020a01 vmla.f32 s0, s4, s2
[..]
L60:  d0: 3affffa4 bcc a8 <entry+0xa8>
[..]
```

Listing 7: handoptimized IR with i32 and bipolar dot product

```
L8: define internal void @output_x_reduction12([..]) #0 {
[..]
L19:   %6 = call i32 @llvm.ctpop.i32(i32 %5)
L20:   %7 = add i32 %4, %6
[..]
L23: }

L25: define void @entry([..]) #0 {
[..]
L53: multiply.17.5.loop_body.dim.0:
L59:   %11 = xor i32 %8, %10
L60:   %12 = xor i32 %11, -1
[..]
L81: reduce.17.13.inner.loop_body.reduction_dim.0:
L94:   call void @output_x_reduction12([..])
[..]
```

Listing 8 shows the code from listing 7 after the optimizations done by LLVM. This code is efficiently vectorized.

Listing 8: LLVM optimized IR with i32 datatype and bipolar dot product

```
L8: define void @entry([..]) #0 {
[..]
L19: vector.body:
L28:  %13 = xor <4 x i32> %wide.load, <i32 -1, i32 -1, i32 -1, i32 -1>
L29:  %14 = xor <4 x i32> %wide.load9, %13
[..]
L42: vector.body10:
L49:  %23 = call <4 x i32> @llvm.ctpop.v4i32(<4 x i32> %wide.load24)
[..]
```

Listing 9 shows the assembly generated from the hand- and LLVM-optimized IR in 8. Both for the XNOR and the

population count, efficient SIMD instructions are used (veor and vcnt, respectively).

Listing 9: ARMv7 assembly for datatype i32 bipolar dot product

```
[..]
L18:  28:             f34201f0            veor    q8, q9, q8
L21:  34:             f3f005e0            vmvn    q8, q8
[..]
L34:  68:             f3f02562            vcnt.8  q9, q9
[..]
```

Discussion

Formulating a float32 dot product in Tendorflow with muliplication and summation operators (see Listing 1), gives vectorised operations, yet executed in separate loops, which also blocks potential fusion of the two arithmetic operations to a single fused-multiply-add. It is surprising to see that the XLA high-level optimiser seems unable to fuse the two operations.

Expressing the float32 dot product with tf.tensordot enables XLA to generate more efficient IR code with a single loop (see Listing 5). This enables LLVM to use fused multiply accumulate in the generated ARMv7 assembly code.

Adapting the IR code manually to implement a bipolar dot product by vector XNOR and population count (see Listing 7) yields a code which, after LLVM optimisation, allows LLVM to generate assembly code using vectorised NEON instructions (see Listing 9). This assembly code corresponds to the manual implementation suggested in [12, 4].

Contrasting the experiments results with the ideal tool flow shows that the core limitation lies in the inability of Tensorflow/XLA to generate 1-bit IR vector code. If Tensorflow/XLA had that ability, LLVM could take up the IR, optimise it and generate efficient bit-parallel and vectorised code for ARMv7 NEON.

Conclusion

Binarised neural networks are likely a perfect fit for an efficient execution of many ANNs on embedded architectures. We have documented the lacking support for 1-bit data types in Tensorflow/XLA, but also the ability of LLVM to generate efficient bit-parallel code – given it receives proper IR code (i.e. expressing the binary or bipolar vectors as <i1xN>).

Core insights:

- Tensorflow does not support custom-precision floating-point or integer data types, including 1-bit integer.
- LLVM IR knows custom-precision integer types including 1bit integer and can operate on it.
- LLVM IR can encode vector operations on <i1x32> vector code.

Recommendations:

- Tensorflow should support custom-precision integer and floating-point data types to support a wider design-space exploration of ML models. Given the interest in QNNs/BNNs, we expect Tensorflow to incorporate this feature soon.
- The Tensorflow high-level optimiser rules should be reviewed and improved to allow for fusion of operators (rather than relying on library calls).
- ARMv7 bipolar product throughput is limited by the NEON bitcount instruction (VCNT), which can only operate on groups of 8bit. The ARM Scalable Vector Extension (SVE) introduced in ARMv8.2 should be screened for an improved instruction.

Acknowledgements

The project SemI40 is co-funded by grants from Austria, Germany, Italy, France, Portugal and the ECSEL Joint Undertaking and is coordinated by Infineon Technologies AG. The authors gratefully acknowledge the financial support under the scope of the COMET program within the K2 Center "Integrated Computational Material, Process and Product Engineering (IC-MPPE)" (Project No 859480). This program is supported by the Austrian Federal Ministries for Transport, Innovation and Technology (BMVIT) and for Digital and Economic Affairs (BMDW), represented by the Austrian research funding association (FFG), and the federal states of Styria, Upper Austria and Tyrol.

REFERENCES

1. Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899
2. Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. 2018a. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. (2018).
3. Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. 2018b. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *arXiv:1809.04570 [cs]* (Sept. 2018). http://arxiv.org/abs/1809.04570 arXiv: 1809.04570.
4. Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]* (Feb. 2016). http://arxiv.org/abs/1602.02830 arXiv: 1602.02830.
5. Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. 2018. QUENN: QUantization engine for low-power neural networks. In *Proceedings of the 15th ACM International Conference on Computing Frontiers - CF '18*. ACM Press, Ischia, Italy, 36–44. DOI: http://dx.doi.org/10.1145/3203217.3203282
6. Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. 2018. Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 29, 11 (Nov. 2018), 5784–5789. DOI: http://dx.doi.org/10.1109/TNNLS.2018.2808319
7. Yingjian Ling, Kan Zhong, Yunsong Wu, Duo Liu, Jinting Ren, Renping Liu, Moming Duan, Weichen Liu, and Liang Liang. 2018. TaiJiNet: Towards Partial Binarized Convolutional Neural Network for Embedded Systems. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, Hong Kong, 136–141. DOI: http://dx.doi.org/10.1109/ISVLSI.2018.00034

8. Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2017. Mixed Precision Training. (2017).

9. Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2018. A Survey on Deep Learning: Algorithms, Techniques, and Applications. *Comput. Surveys* 51, 5 (Sept. 2018), 1–36. DOI:http://dx.doi.org/10.1145/3234150

10. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016).

11. Günther Schindler, Manfred Mücke, and Holger Fröning. 2018. Linking Application Description with Efficient SIMD Code Generation for Low-Precision Signed-Integer GEMM. In *Euro-Par 2017: Parallel Processing Workshops*, Dora B. Heras, Luc BougÃ©, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer (Eds.). Vol. 10659. Springer International Publishing, Cham, 688–699. DOI: http://dx.doi.org/10.1007/978-3-319-75178-8_55

12. Yaman Umuroglu and Magnus Jahre. 2017. Streamlined Deployment for Quantized Neural Networks. (2017).

13. Tianli Zhao, Xiangyu He, Jian Cheng, and Jing Hu. 2018. BitStream: Efficient Computing Architecture for Real-Time Low-Power Inference of Binary Neural Networks on CPUs. In *2018 ACM Multimedia Conference on Multimedia Conference - MM '18*. ACM Press, Seoul, Republic of Korea, 1545–1552. DOI: http://dx.doi.org/10.1145/3240508.3240673

14. Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR* abs/1606.06160 (2016).