Harnessing Numerical Flexibility for Deep Learning on FPGAs

(Invited Paper)

Andrew C. Ling, Mohamed S. Abdelfattah, Shane O'Connell, Andrew Bitar,
David Han, Roberto Dicecco, Suchit Subhaschandra, Chris N Johnson,
Dmitry Denisenko, Josh Fender, Gordon R. Chiu
Intel Corporation
andrew.ling@intel.com

ABSTRACT

Deep learning has become a key workload in the data centre and edge leading to an arms race for compute dominance in this space. FPGAs have shown they can compete by combining deterministic low-latency with high throughput and flexibility. In particular, due to FPGAs' bit-level programmability, FPGAs can efficiently implement arbitrary precisions and numeric data types which is critical to fast evolving fields like deep learning.

In this work, we explore minifloat (floating point representations with non-standard exponent and mantissa sizes) implementations on the FPGA, and show how we use a block floating point implementation that shares the exponent across many numbers to reduce the required logic to perform floating point operations. We will show that using this technique we can significantly improve the performance of the FPGA with no impact to accuracy. Using this approach, we show how we can reduce logic utilization by 3x, and memory bandwidth and capacity required by more than 40%.

Keywords

Deep Learning, FPGAs, High-Level Design

1. INTRODUCTION

Deep neural networks have proven to be a powerful means to solve some of the world's most difficult computer vision and natural language processing problems after its first successful introduction into the ImageNet compentition in 2012 by A. Krizhevsky et al. [Krizhevsky et al. 2012] This has led to an explosion of workloads based on deep neural networks in the data centre and edge [Bryant 2016].

One of the key challenges with deep neural networks is their inherent computational complexity, where many deep nets require billions of operations to perform a single inference. To mitigate the computational burden of deep nets three methods are often used:

1. Skipping redundant operations (e.g. multiply by 0) and modifying training algorithms or post-processing weights to lead to sparse connectivity in the network [Gao et al. 2018, Wang et al. 2018].
2. Removing redundancy in the deep net by either trimming layers or connections [Iandola 2016, Han et al. 2017].
3. Reducing the complexity of each operation by reducing their precision and bitwidth [Coussy et al. 2015, Gysel et al. 2016, Gysel 2016, Fu 2016, Chung et al. 2018].

Because of their flexibility, FPGAs are perfect candidates to take advantage of all of these approaches, and in this work we explore the third approach in detail.

Specifically, we will show how to leverage minifloat representations on the FPGA and show how you can efficiently map minifloat operations onto the FPGA fabric leading to a significant reduction in resource utilization with a negligible degradation to accuracy on GoogLeNet, a common network used in image classification. Additionally, we will show how using a block-floating point based approach, we can significantly increase the number of operations that can fit on a single FPGA device and reduce the memory bandwidth and footprint required to store intermediate data and filter weights on the FPGA.

Finally, we will show how our block floating point implementation on Intel's Arria 10 FPGAs have little overhead compared to fixed-point equivalent operations and can be trivially converted to fixed-point if necessary.

2. FPGA DEEP LEARNING ACCELERATOR

We implemented a highly efficient deep learning inference engine in [Aydonat et al. 2017], where a convolutional core, consisting of an array of processing-elements, reads input image and filter data from external (DDR) memory, and stores the data in caches built of on-chip block RAMs. The processing-elements consist of highly efficient dot-product kernels, which can execute several dot-products in parallel: one of the key operations in deep neural nets. In this paper, we will explore how minifloat implementations can significantly reduce both logic utilization and memory bandwidth/usage on the FPGA.

2.1 Compute Precision and Minifloat

Deep Learning applications often have large memory and compute requirements, leading to a lot of exporation in reducing precision and complexity of each individual operation. Fixed-point representation has been employed by NVIDIA [Corporation 2017], Xilinx [Fu 2016], and Google's

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2018) Toronto, Canada, June 20-22, 2018.

DOI: 10.1145/3241793.3241794

TPU [Wu 2016] for CNN and RNN acceleration, while Microsoft has recently announced the use of a reduced-precision floating point on Intel® Stratix® 10 FPGAs in their acceleration of GRUs [Chung 2017].

We explore a similar approach as Microsoft, where we take advantage of minifloat representations that reduce the mantissa and exponent from IEEE fp32 implementations. In our approach, we explore different mantissa sizes from 2 to 5 bits, which we refer to as fp8 to fp11 respectively. For all our representations, we keep one bit for the sign value and five bits for the exponent.

In Table 1, we show the relative impact of reducing the precision against fp32. Here we show that peak tops can increase up to 8x by moving to lower minifloat representations.

	fp32	fp16	fp11	fp9
Relative Tflops	1.0x	2.0x	3.8x	8.0x

Table 1: Relative Tflops increase of minifloat precisions when compared against fp32 on the Intel Stratix 10 device.

The ability to take advantage of mixed precision networks makes FPGAs particularly attractive for deep networks which have different layers and operations that have varying influence on the final accuracy of the results. Table 2 illustrates this where we compare the accuracy of fp11 against fp32 for GoogLeNet using two approaches: changing the precision to fp11 for the dot-product only in the Convolutional and InnerProduct layers only and changing the entire design to fp11 for all operations in all layers. As Table 2 shows, changing all operations to fp11 has a significant degradation to overall accuracy, while changing only the dot-product operations has little to no impact. The benefit of this is that since over 80% of the resources on the FPGA are dedicated to the dot-products, even changing only the dot-products to lower precision will yield the majority of the logic reduction (hence most of the performance benefits) to the FPGA.

	All	Dot-Product Only
Top-1 Accuracy Drop	2.31%	0.04%
Top-5 Accuracy Drop	1.22%	0.20%

Table 2: Accuracy impact of reducing all operations in DLA to fp11 (*All*) vs *Dot-Product Only*.

2.2 Block Floating Point and Memory Impact

Although, Intel FPGAs support fp32 natively in hard DSP blocks, variable precision minifloating point operations cannot be fully implemented in hard DSP blocks, and take a significant amount of resources to implement each multiply and add in soft logic [Vishwanath 2016]. However, in [Chiu et al. 2018], we illustrated how we can implement the majority of dot-products in block floating point form, where a group of operations are clustered to form a "block" of operations and the exponent is shared across all numbers in the block. An illustration of this is shown in Figure 1, which shows a conversion of fp11 (1-bit sign, 5-bit exponent, 5-bit mantissa) to block fp11 with a block size of 4.

Within each block, the mantissa is shifted such that all numbers in the block will have the same exponent and can be factored out. Any resulting multiplies or adds can then

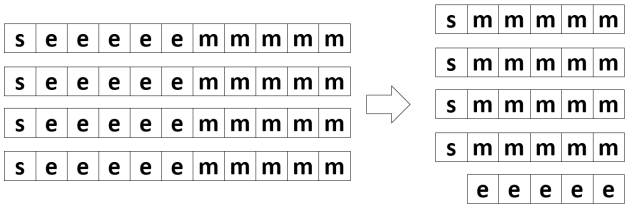

Figure 1: Illustration of block floating point for fp11 with a block size of 4 (s=sign bits, e=exponent bits, m=mantissa bits).

be applied directly on the resulting mantissas, which are equivalent to fixed point operations in terms of operations and cost on the FPGA. This can lead to an over 3x reduction in required logic to implement when a block size of 8 is used, as described in [Aydonat et al. 2017]. In general, the larger the block size, the more resources can be saved, however, this leads to reduced accuracy, since as more numbers are shifted to align to a single exponent value, more bits may be shifted off in the mantissas found within the block. In practice, we find that a block size of 8 or 16 provide a good trade-off between accuracy and resources as described in [Chiu et al. 2018]. However, in the event that accuracy is impacted, previous work has shown that top-up training can successfully recover the accuracy loss incurred by large block sizes and lower precisions [Chung et al. 2018].

In addition to implementing dot-products in block floating point form, we also can store the data in block floating point form. This can lead to a significant reduction in both memory bandwidth to fetch data, and memory capacity to store the data either on or off-chip. To illustrate this, Table 3 shows the compression ratio (the lower the better) achieved by storing fp9, fp11, and fp16 in block-floating point form, with a block size of 2 to 32 versus no blocking (i.e. block size of 1).

Block Size	2	4	8	16	32
fp16	0.84	0.77	0.73	0.71	0.70
fp11	0.77	0.66	0.60	0.57	0.56
fp9	0.72	0.58	0.51	0.48	0.46

Table 3: Compression ratio of different block size storage requirements, Block size vs no blocking used to store weights and intermediate data (feature maps).

2.3 Block Floating Point vs Fixed Point

When implementing operations in block-floating point, the majority of the operations are applied directly to the mantissas, which effectively converts the floating point operations into fixed-point leading to an implementation that is as efficient as fixed-point. For example, in fp11, the mantissa plus sign is 6-bits in width. This allows us to map two fp11 multiplies as two INT6 operations in the native 18x18 multiplier as illustrated in Figure 2.

In the event that fixed-point is desired, the block floating point dot-products in our architecture can be trivially converted to fixed-point by simply removing the exponent and shifts required to convert to the block-floating point format.

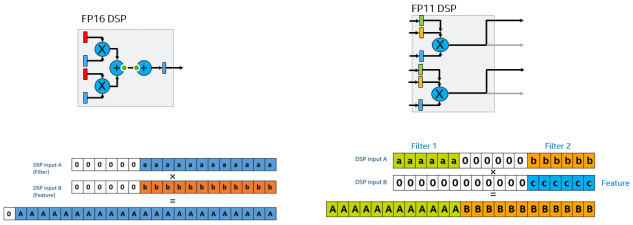

Figure 2: DSP packing technique of fp16 and fp11 block floating point multiplications into 18x18 integer multiplier DSP block.

Once converted to naive fixed-point, scaling and quantization of weights would be required to account for the loss in dynamic range by moving to true fixed-point operations.

3. CONCLUSION

In this work we have demonstrated how using minifloat representations can have a significant impact to the overall performance of the FPGA for deep learning inference applications. Using block floating point, we show how we can both reduce the logic utilization and memory footprint of the design. Additionally, we describe how block floating point efficient is similar to fixed-point implementations with little overhead over fixed-point designs.

4. REFERENCES

Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 55–64.

Diane M. Bryant. 2016. Keynote at Intel Developer's Forum 2016, San Francisco. (August 2016). `https://newsroom.intel.com/chip-shots/2016-idf-keynotes-innovation-drives-technology-\future-artificial-intelligence/`

Gordon R. Chiu, Andrew C. Ling, Davor Capalija, Andrew Bitar, and Mohamed S. Abdelfattah. 2018. Flexibility: FPGAs and CAD in Deep Learning Acceleration. In *Proceedings of the 2018 International Symposium on Physical Design (ISPD '18)*. ACM, New York, NY, USA, 34–41. `https://doi.org/10.1145/3177540.3177561`

Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. 2018. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. `https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/`

Eric et. al Chung. 2017. Accelerating persistent neural networks at datacenter scale. HotChips.

NVidia Corporation. 2017. NVidia TensorRT. (2017).

Philippe Coussy, Cyrille Chavet, Hugues Nono Wouafo, and Laura Conde-Canencia. 2015. Fully Binary Neural Network Model and Optimized Hardware Architectures for Associative Memories. *J. Emerg. Technol. Comput. Syst.* 11, 4, Article 35 (April 2015), 23 pages. `https://doi.org/10.1145/2629510`

Yao et. al Fu. 2016. Deep Learning with INT8 Optimization on Xilinx Devices. *white paper of Xilinx* (2016).

Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, New York, NY, USA, 21–30. `https://doi.org/10.1145/3174243.3174261`

Philipp Gysel. 2016. Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks. *CoRR* abs/1605.06402 (2016). arXiv:1605.06402 `http://arxiv.org/abs/1605.06402`

Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *CoRR* abs/1604.03168 (2016). arXiv:1604.03168 `http://arxiv.org/abs/1604.03168`

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 75–84. `https://doi.org/10.1145/3020078.3021745`

Forrest N et al. Iandola. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS '12)*. 1097–1105.

Amulya et. al Vishwanath. 2016. Enabling High-Performance Floating-Point Designs. *white paper of Intel* (2016).

Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling Efficient LSTM Using Structured Compression Techniques on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, New York, NY, USA, 11–20. `https://doi.org/10.1145/3174243.3174253`

Yonghui et al. Wu. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).