16-Bit Brain Floating Point for Gemmini

Ryan Lund
University of California, Berkeley

***Abstract*—Pioneered for use on Google's machine learning (ML) specific architecture [1], 16-bit brain floating point (BF16) is a novel format that promises resource benefits with minimal performance impacts compared to the traditional IEEE 32-bit floating point (FP32). With these benefits in mind, BF16 has been slowly trickling into ML oriented processors and accelerators such as Armv8-A, the Habana HL series, and the aforementioned Google Tensor Processing Unit (TPU). This paper explores the process of adding support for BF16 to the Gemmini systolic array generator [2]. Additionally, it discusses the impacts of BF16 on accuracy, area utilization, and power consumption for Gemmini generated systolic arrays coupled with a Rocket Core in-order [3] processor.**

I. Introduction

A. Current Constraints

Within the next five years, the global machine learning market is expected to reach nearly $100 billion in value [4]. A chief segment driving this market is warehouse scale computing, which performs key tasks such as "big data" analysis, generating content recommendations, and user segmentation. In those applications, algorithms are run on massively parallel server farms, with a vast amount of computing resources at their disposal. Yet even warehouse scale computing has constraints, which can can be succinctly broken down into three categories: power, area, and accuracy.

While processor architectures can rapidly change, server-farms are generally buildings with fixed rack volume, power access, and cooling capacity. Often these buildings cannot scale up their supply of resources as quickly as new technology can consume it. This means that power and area are limiting factors on the warehouse scale, and increasing performance for the same area and power is the optimal goal. Beyond the physical limitations of server-farms, accuracy is a primary goal for maximizing user experiences and revenue. Consider the case of Netflix, where 75% of views are driven by recommendations [5]. For the companies that rely on ML algorithms, the smallest change in performance can be the difference between a sale and a lack thereof [6].

B. Gemmini

Today, many ML workloads are run on CPUs or GPUs. Yet, even as network architectures have evolved, the core computational kernels remain largely the same. This has lead to the rise of research into domain-specific hardware accelerators for deep learning. These accelerators can be constructed via FPGA based designs such as those used for Microsoft Asure, or ASIC based designs such as the TPU.

Generally speaking, these different forms of neural network (NN) accelerators share a common base architecture designed around matrix multiplication: the systolic array. This architecture is composed of numerous multiply accumulate cells (MACs), which act as the unit level building block of the larger array. The array itself is laid out in a square, with each individual MAC receiving input from the MACs to the north and west of itself, and passing output to the MACs to the south and east of itself. On the highest level, activation data is passed to the north side of the array and weight data is passed to the west side of the array. After propagation, output is produced on the south side of the array for further accumulation.

Gemmini [2] is a systolic array based matrix multiplication accelerator generator built into the Chipyard [7] framework. Gemmini allows a user to set parameters for a systolic array then elaborate that configuration within the greater Chipyard context. The resulting Gemmini generated systolic array (also referred to as a Gemmini array or just Gemmini) can then perform matrix multiplications in the manner described in the previous paragraph. For the sake of brevity, full details on the architecture and technical specification of Gemmini can be found within the Gemmini documentation [8]. The main purpose of this project is to add support for BF16 as an input and computation type within the Gemmini generator and to explore the characteristics of a BF16 Gemmini array.

C. Benefits of Reduced Bitwidth

The original Gemmini paper demonstrated that increasing the dimensions of the systolic array either by changing the number of MACs or the bitwidth of data can increase power consumption by up to 3.4x and area by up to 2.3x [2]. However, it has also been shown that increasing the number of MACs in a Gemmini array can increase performance by up to 4x for ML inference.

With these metrics in mind, it seems that a logical path to increase performance without increasing resource consumption would be to increase the number of MACs in a systolic array while simultaneously decreasing the bitwidth of data, such that the sum of input bitwidths is kept constant. Alternatively, resource consumption can be improved with minimal impact on performance if a suitable reduced bitwidth representation is chosen. Research has focused on two main approaches to achieve this second result: quantization and 16-bit floating point formats.

D. Bitwidth Reduction Methods

1) Quantization: Quantization is the processing of taking a set of floating point values (typically FP32) and converting them into a lower bitwidth integer representation. This process can be thought of as generating a step function that maps the

range of values for a given dataset to the range of the smaller representation. Research has shown that neural networks can be quantized to 8-bit integers (INT8) with minimal accuracy loss [9]. Current state-of-the-art is focused on achieving the same result with 4-bit integers [10].

There are dual benefits to quantization into small integers. First, by reducing the bitwidth of data values, less hardware is required for each MAC, which means that computation can run faster and consume lower power. The second benefit to quantization is that MACs can implement integer-only arithmetic [9]. This results in additional resource savings and speed benefits as integer arithmetic hardware is significantly smaller than floating point hardware.

The downside of these savings is that quantized networks often suffer from accuracy degradation due to the loss of expressible range for weights and activations. However, for most networks this loss can be minimized to single digit percentage points through careful consideration of outlier weights if converting an already trained network, or by training using a quantization-aware network.

2) IEEE Half Precision Floating Point: IEEE 754 half-precision floating point (FP16) [11] is the official standard for 16-bit floating point, and most systems that support a 16-bit floating point type use FP16. While full-width FP32 has 8 exponent bits and 23 mantissa bits, half-width FP16 only has 5 exponent bits and 11 mantissa bits. This means that the FP16 is limited to a maximum range of ± 65504 which is significantly smaller than FP32's range of $\pm$ 3.4028235 $\times$ 10^{38}. This trade-off is made in order to maintain a high level of precision with FP16.

Due to the ubiquity of FP16 and the benefits of bitwidth reduction, some research has focused on the use of this format for NN training and inference. This work has shown that with sufficient optimizations, mixed precision FP16 networks (master weights stored in FP32 for training) can achieve accuracy as good or better than the accuracy achieved by a FP32 baseline across a variety of network architectures [12]

A majority of these optimization are used to address the limited range of FP16, which can lead to over- and under-flow errors during computations. For example, loss scaling is required to attain parity accuracy to FP32 baselines [13]. This method ensures that critical values are scaled within the range of FP16 to preserve accuracy. In effect, this approach addresses the numerical instability of using FP16 for compounding calculations. Due to this instability and the additional work required to address it, the domain-specific architecture industry has favored more stable formats for computation in NNs.

3) BF16: To address the issues of FP16, the BF16 format was developed with the specific needs of ML in mind. BF16 preserves the original 8 exponent bits from FP32 and thus only has 8 mantissa bits. By maintaining 8 exponent bits, BF16 can cover the same range as FP32 but with significantly lower precision. In a way, this means that BF16 makes a trade-off opposite to FP16, sacrificing precision for range. However for ML workloads, precision is not necessarily a needed thing, especially considering that quantization can achieve minimal accuracy losses with little weight precision.

Because BF16 promises the benefits of bitwidth reduction with the same ease of use as FP32, recent research has attempted to evaluate the performance of networks that perform calculations using this new datatype. Work using BF16 emulation through the zeroing out of the lower two bytes of FP32 values has shown that similar accuracy can be achieved between networks using BF16 and those using using FP32 for both convolutional neural networks and recurrent neural networks [14]. Moveover, this work showed that accuracy parity can be achieved using both rounding and truncation to convert from FP32 to BF16.

Due to the accuracies that it is able to achieve without the need for loss scaling techniques because of its wider range, BF16 has quickly become a popular 16-bit floating point representation for use in ML workloads. This paper will attempt to further explore the benefits and impacts of integrating BF16 into the existing Gemmini framework.

II. Adding BF16 to Gemmini

A. Hardware Modifications

Gemmini uses the Hardfloat [15] library as the source for the floating point arithmetic units (FPU) within each MAC. A key benefit of Hardfloat is that it allows for the parameterized generation of FPUs based on an input type's defined exponent and significand width. This meant that by creating a new Gemmini configuration with input and output types of Float(8, 8), Hardfloat was able to automatically generate FPUs for BF16. All told, this made adding hardware support for BF16 to Gemmini relatively trivial. The same approach was also used to add FP16 support to Gemmini hardware.

B. Software Modifications

In spite of the ease with which Gemmini hardware could be adapted to support BF16, the software side of things was not as simple. Thus adding BF16 to Gemmini became largely a software challenge, of which the key steps are detailed below.

1) Softfloat Extension: As it is not an IEEE standard format, neither C nor RISC-V contains support for BF16. Luckily, Gemmini was once again well designed and does not require values to be any specific type to be passed in to and out of a generated systolic array. However, the Gemmini Rocc Tests and CIFAR-10 inference programs perform calculations on layer inputs and outputs in the CPU and thus require the ability to handle BF16.

In order to allow the Rocket Core to speak BF16 in RISC-V, an extension was made to the Softfloat [16] library. The library typically allows systems without native support to use IEEE standard floating point representation by defining custom types. These types are wrappers around similarly sized unsigned integer types (i.e. uint32_t for float32_t) that have specially defined operations for arithmetic operations, comparisons, and conversions.

To add support for BF16, I defined a custom type (bfloat16_t) and the accompanying helper functions to handle magnitude operations and special value handling. Using those helper functions, a barebones extension to Softfloat was established with the minimal operation set needed for ML computations. This set included 1) the arithmetic operations of addition, subtraction, multiplication, and fused multiply-add 2) the comparison operations of equals, less than, and less than or equal to 3) conversion operations to and from 32-bit integers and 32-bit floating point.

Rather than using a more complex rounding mode for conversion or overflowed operations (i.e. multiplies), the extension uses rounding by truncation. This choice may explain some of the discrepancies seen later between the results produced by software BF16 those produced by a BF16 Gemmini systolic array.

2) gemmini_params.h Generator Modifications: A key component of insuring Gemmini hardware and software compatibility is the gemmini_params.h file that is generated as the hardware is built within Chipyard. Within this file, the parameters of the generated systolic array are detailed. Programs and tests that use Gemmini are written in such a way that the compiler will produce code that works properly with the systolic arrays datatype. To insure compatibility with the BF16 systolic array, a new flag was added to the automatically generated parameters file, ELEM_T_IS_BFLOAT. Tests were modified to use the functions defined in the Softfloat extension in the place of typical arithmetic, comparison, and cast operations when the BF16 flag was defined.

3) Gemmini Rocc Tests Modifications: As mentioned above, the tests within Gemmini Rocc Tests [17] required the addition of Softfloat operations to handle BF16 when indicated by the new BF16 flag. Additionally, modification was required to handle network weights and input activations. As a non-standard format, BF16 lacks compiler support for implicit conversion from a value, say -127 or 1.2345, to the equivalent BF16 hex value. In order to avoid this complication, when the BF16 flag is set, weights and input activations were stored in float and int format respectively. A new function was defined that could be called at the start of any program requiring access to a given vector and would perform element-wise conversion to populate a corresponding BF16 vector for use. Finally, for the accuracy testing below, the cifar_quant test provided in EE290's Lab 3 [18] was modified to perform computation on a variable number of size four batches. This allowed inference to be run on more than just four images per test.

III. Experimental Methodology

After adding support for BF16 to Gemmini, the next research tract was to examine the impact of this specialized floating point format on a generated systolic approach. Experimentation was focused on two categories of metrics: resource utilization and accuracy. Each category had its own experimental methodology for testing.

A. Power and Area

To gather power and area statistics, seven different Gemmini configurations were run through the Hammer [19] VLSI flow until the end of place and route (PAR) backed by Cadence Genus and Innovus tools. The first configuration used was a baseline FP32 configuration with 256 KB of scratchpad memory and 64 KB accumulator memory. This configuration acted as a baseline for comparison, as most common ML workloads are based on FP32. The next two configurations tested were FP16, one with the same sized memories as FP32 and one with half-sized memories, such that the number of entries matched those of FP32. These configurations also were used as a baseline to explore specific benefits of BF16 compared to the FP16 standard. Next, two BF16 configurations were tested with the same memory sizing as the FP16 configurations. Finally, two INT8 configurations were tested, one with the same size memory as the FP32 configuration and one with fourth-sized memories, such that they held the same number of entries as the previous configurations. The INT8 configurations were used as a comparison between floating-point and the typical quantization datatype. All configurations used an 8x8 systolic array with a tile size of 1x1.

B. Accuracy

1) Tensorflow-Only Accuracy: Initially, this paper relied onmodels generated and tested using Google's Tensorflow (TF) [20] as the only source of accuracy metrics for the discussed datatypes. However, feedback suggested that the measured accuracies reflected software and hardware characteristics of TF and the backing compute device rather than what Gemmini could accomplish. Specifically, the use of an 8x8 systolic array in Gemmini would lead to the build-up of precision errors as results were accumulated into and converted out of FP32. These errors would be dependent on the size of the systolic array, and thus the 256x256 TPU backing TF would produce different results than the 8x8 Gemmini. Moreover, the TF uses several different rounding modes to optimize performance, while Hardfloat does not. This again would lead to discrepancies between Gemmini accuracy and TF accuracy. This all being said, the results from TF are still included to provide a frame of reference for the level of accuracy state-of-the-art and highly optimized systems can achieve with each datatype.

2) Comparing Gemmini to Tensorflow: With discrepancies in mind, additional testing was done to explore the differences between the results produced by BF16 models running on TF and those produced by BF16 models running on Gemmini using the same weights. For these experiments five networks based on a modified LeNet-5 [21] were trained using BF16 in Tensorflow on the CIFAR-10 [22] dataset. Additional complications arose from the fact that TF saves weights in FP32 for BF16 calculation while Gemmini uses BF16 weights for BF16 calculation. To address this, an additional 5 models were generated by taking the original 5 models and sending their weights on a round-trip from FP32 to BF16 to FP32, such that

the resulting weights were FP32 values where the last 16 bits were 0 (thus the FP32 and BF16 values should be equal).

These models were then run through a processing function to produce header files containing their weight values, as well as the predictions that the model would make for a given subset of the CIFAR-10 dataset, measured in size four batches. A separate processing function also generated a header file for a variable number of images and their labels. Using these two functions, header files for the 10 models, images, and labels were generated for batch sizes of 1, 2, 25, 125, 250, 1250, and 2000. Ultimately the largest batch size used in testing was 125 (corresponding to 500 test images) due to the limitations of the memory of a FireSim FPGA image.

As hinted to above, due to the inordinate amount of time that a VCS binary would take to perform simulation, FireSim [23] was used to run inference for the 10 generated weight sets. In order to do so, a custom AGFI was generated from a BF16 config, and a set of 10 tests were created to run batched inference with each of the different weight sets. These tests calculated the number of correct predictions, the number of predictions that matched the TF model, and in the case that the predictions did not match, if either TF or Gemmini made a correct prediction.

IV. Measured Results

A. Constant Memory Size (Bytes)

Fig. 1: Power (mW) for Constant Memory Size (Bytes)

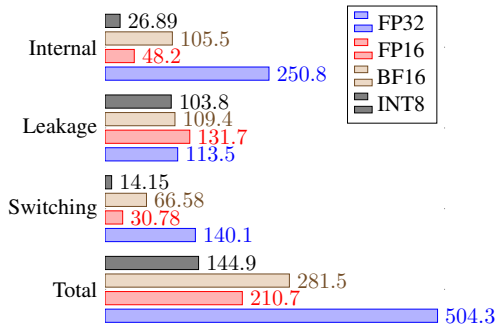
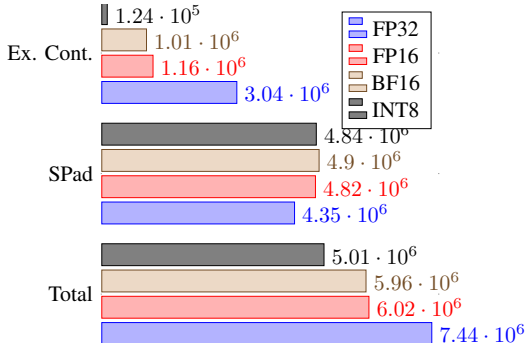


Fig. 2: Area (um^2) for Constant Memory Size (Bytes)



The above figures display key metrics collected from post-PAR analysis of the Gemmini configurations with scratchpad memory kept constant at 12KB and accumulator memory kept constant at 4KB. As seen in Figure 1, switching from FP32 to BF16 resulted in a 1.79x reduction in total power yet switching from FP16 to BF16 resulted in a 1.34x increase in total power consumption. In the case of FP16 to BF16, the result was driven by a 2.19x increase in internal power and a 2.16x increase in switching energy. Compared to INT8, BF16 consumed 1.94x more total power, which is roughly proportional to the change in bitwidth.

Across all four configurations, leakage power remained largely the same, with FP16 consuming the most leakage power and INT8 consuming the least leakage power. This indicates that the largest generator of leakage power in Gemmini arrays is the scratchpad and accumulator memories.

The trends in area for fixed memory size in bytes are illustrated in Figure 2. Given that scratchpad size is fixed, the magnitude of total savings are less dramatic, so the key metric here is reduction in execute controller area. Regardless, switching to BF16 resulted in a 1.25x reduction in total area and a 3.02x reduction in execute controller area. Switching from FP16 to BF16 resulted in a negligible 1.01x reduction in total area but a 1.15x reduction in execute controller area. The changes in execute controller area line up with what is expected, as area should scale roughly with the square of bitwidth. However, the changes in total area are significantly lower due the dominance of fixed scratchpad area.

Compared to INT8, BF16 consumed 1.19x more total area and 8.16x more execute controller area. This is much more than the expectation of squared bitwidth proportionally, likely due to the lack of extra floating point hardware area within the INT8 Gemmini array.

Note that in spite of scratchpad area remaining as a fixed number of bytes across all configurations, its area slightly increased for non-FP32 variants. This is something that may be worth exploring further.

B. Constant Memory Size (Entries)

Fig. 3: Power (mW) for Constant Memory Size (Entries)

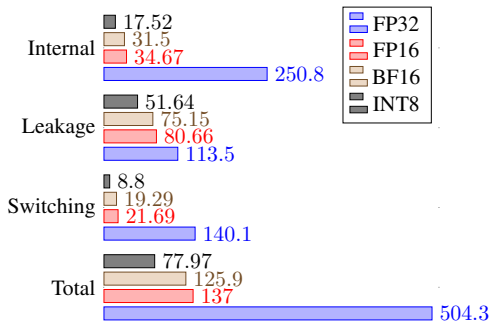
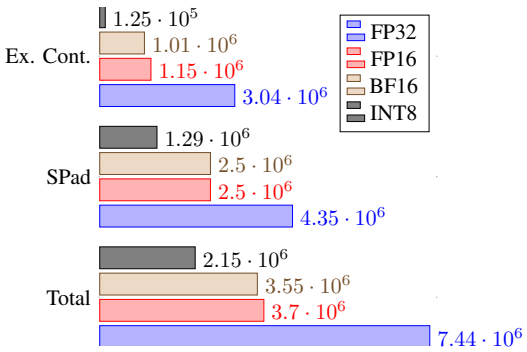


Fig. 4: Area (um^2) for Constant Memory Size (Entries)



With the number of entries held by the scratchpad fixed at 65536 and those held in the accumulator fixed at 16384, the differences between the various configurations becomes more apparent. As shown in Figure 3, switching from FP32 to BF16 resulted in a 4.01x reduction in total power. Switching from FP16 to BF16 resulted in a 1.09x reduction in total power, due to small improvements across all categories. Of note is that with memory size fixed in entries, there is not the same spike in internal energy that had led to BF16 consuming more energy than FP16 when the scratchpad sized was fixed in bytes. Compared to INT8, BF16 consumed 1.61x more power.

Looking at area, Figure 4 shows that switching from FP32 to BF16 resulted in a 2.1x reduction in total area with a 1.74x reduction in scratchpad area. Slightly better than in the previous section, switching from FP32 to BF16 resulted in a 1.04x reduction in total area with nearly identical scratchpad size.

Compared to INT8, BF16 consumed 1.65x more total area, driven mainly by a 1.93x increase in scratchpad area. These changes in total area are more linear due to a linear scale of memory area with bitwidth when accounting for a fixed number of entries.

Across all four configurations, execute controller area logically remained constant when compared to the test with a fixed scratchpad size in bytes and thus the improvements with regards to execute controller area between datatypes were also identical.

C. Accuracy Comparisons to Tensorflow

Table 1: Tensorflow Best Accuracy

	FP32	FP16	BF16	INT8
Accuracy	53.11%	54.75%	53.38%	52.21%

Fig. 5: Accuracy for 500 CIFAR-10 Images

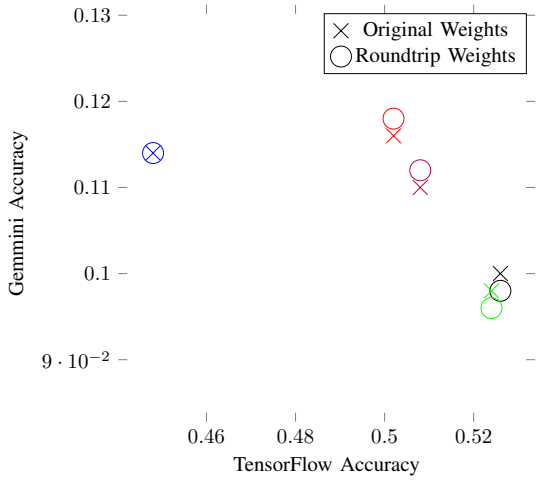


Fig. 6: Matches to TF Predictions

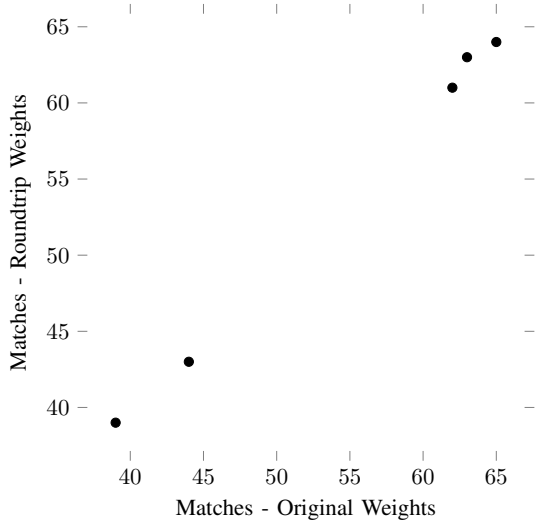
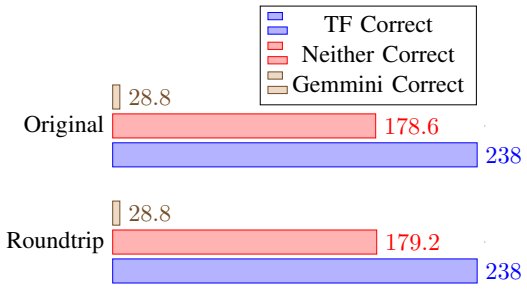


Fig. 7: Average Classification of Mismatches



The above figures display the results of inference carried out on Gemmini in FireSim using the weights from the ten TF models (five with original weights and five with roundtrip weights). Figure 5 shows TF accuracy vs Gemmini accuracy for both the original and roundtrip weights of the five different models, grouped by color. The groupings clearly demonstrate

that the results of this experimentation were very negative for the portability of TF weights to Gemmini. While the best TF model achieved 54.2% accuracy on the first 500 images of CIFAR-10, the best Gemmini model achieved only 11.8%. Moreover, there is no indication that either the original weights nor the roundtrip are best for use in Gemmini arrays. In experimentation, the roundtrip weights achieved better accuracy on 2 of the 5 models, the original weights on 2 of the 5 models, and there was a tie on 1 of the 5 models.

Figure 6 plots the number of matches between the predictions made by Gemmini and those made by TF, comparing the round trip weights to the original weights. There was a nearly 1:1 relationship between matches for the original weights and matches for the roundtrip weights, but interestingly the original weights achieved one match for three of the models. Regardless, in the best case, Gemmini only achieved a 13% match rate for predictions when compared to the TF baseline.

Finally, Figure 7 examines the average trend in predictions where the Gemmini predictions did not match the TF predictions. In a majority of cases, for both the original and round trip weights, when the models did not agree TF made a correct prediction, or neither model made a correct prediction. There were only a few rare cases (on average 5.76% of total predictions made) where Gemmini produced a correct label and TF did not.

V. Discussion

A. Reflection on Area and Power

The results in area and power savings under both sets of testing conditions largely aligned with the expectations of bitwidth reduction. Each case also had a unique set of benefits when considering the larger design goals of a system. Keeping memory sizes fixed in bytes would be more ideal for latency reduction, in a way analogous to adding larger caches to a system. Conversely, keeping memory sizes fixed in entries would provide less latency benefit but would allow for a much larger reduction in overall energy usage.

In both cases, switching from FP32 to BF16 resulted in a reduction in power and area, the reduction being larger when the memories were also allowed to shrink. Curiously, the comparison to FP16 was split across the two test cases. With a fixed memory in bytes, BF16 consumed 1.34x additional power but 1.01x less area. With a fixed memory in entries, BF16 consumed 1.09x less power and 1.04x less area. Further research could be done to explore this discrepancy and find the switching point in memory size where BF16 overtakes FP16 in memory usage. Finally, compared to INT8, BF16 consumed more power and area in both cases.

Considering the case of variable sized memories only, switching from FP32 to BF16 resulted in a change in energy proportional to the square of the reduction in bit-width. It was surmised in the fixed memories in bytes experiment that leakage power is largely tied to the scratchpad, so halving the scratchpad size roughly halved leakage energy. However, there was an enormous 7.96x reduction in internal energy and a 7.26x reduction in switching energy that led to the 4.01x decrease in total energy. This result hints that the systolic array itself is the largest driver of switching and internal energy, hence the decrease in energy proportional to the square of bitwidth reduction

Interestingly, switching from BF16 to INT8 did not result in this same reduction behavior and instead led to an overall 1.61x decrease in energy reduction. For INT8, leakage power dominates, and thus the total savings are limited. This indicates that there is a minimum amount of power savings that can be achieved due to bitwidth reduction.

Looking at area for variable sized memories shows that area scales roughly linearly with the reduction in bitwidth but again with diminishing returns as your move from FP32 to BF16 and from BF16 to INT8.

All told, the findings of these experiments conclusively indicate that BF16 is superior to FP32 with regards to area but do not provide a clear indication if FP16 or BF16 is the best 16-bit floating point type for resource reduction. These results also confirm the logical finding that INT8 consumes less power and area than any of the 32 or 16-bit floating point types.

B. Reflection on Accuracy

With regards to accuracy, the findings of this paper are largely negative for BF16. For both original weights and roundtrip weights, Gemmini achieved around one fifth of original TF accuracy, with a best accuracy of 11.8% compared to a best TF accuracy of 54.2%. Moreover, our experimentation indicated that there was no difference between using the original weights for a model and the weights converted through BF16 and then back to FP32 (thus harnessing TFs build in rounding techniques). Finally, Gemmini achieved a best match rate to TF predictions of 13% and a worst match rate of only 7.8%. Moreover, for a majority of the mismatched predictions, either TF was correct or neither system was correct.

In summary, the findings from our accuracy experiments indicate that model weights cannot be ported over from one BF16 environment for training to another BF16 environment for testing. Future work might want to examine accuracy for models trained on Gemmini, or want to explore the layer-by-layer discrepancies between Gemmini calculations and TF (backed by a TPU) calculations.

VI. Conclusion

In brief, this paper detailed the process of adding support for the BF16 datatype to Gemmini. This process included minor configuration changes and the creation of a custom Softfloat extension to support BF16 in C. Additionally, modifications were made to the Gemmini Rocc Tests and CIFAR-10 classification tests to integrate the newly defined Softfloat functions. Finally, the CIFAR-10 classification test was expanded to perform inference on a variable number of batches in order to perform accuracy testing on FireSim.

After adding support for BF16 to Gemmini, testing was performed to demonstrate the impacts on a generated systolic array. These tests showed that BF16 outperformed FP32 in

area and power, but were inconclusive in calling the battle between FP16 and BF16.

Finally, FireSim was used to explore the accuracy of BF16 Gemmini when using weights from a TF trained model. This experiment produced negative results for Gemmini, with a best case accuracy less than one fifth of the TF baseline. Future work could be done to explore the discrepancies between TF inference and Gemmini inference, and to develop a NN training library for Gemmini.

VII. Acknowledgements

I'd like to quickly thank the GSIs for EE290 their help on this project. Thanks to Hasan for helping me to fix errors with the floating point implementation in Gemmini and other miscellaneous Gemmini rated questions. Thanks to Alon for guiding me through my numerous FireSim issues. I'd also like to acknowledge my EE241B partner, Anson Tsai who provided some assistance with my Softfloat extension and ran the FP32 configuration through PAR as part of our related project for EE214B.

VIII. Code

All code for this project can be found at: https://ryan-lund.github.io/gemmini-bfloat.

References

[1] S. Wang, and P. Kanwar, "BFloat16: The Secret to High Performance on Cloud TPUs," cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, 2019.

[2] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolić, I. Stoica, and K. Asanović, "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures," arXiv:1911.09925, 2019.

[3] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, Henry Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo and A. Waterman, "The Rocket Chip Generator," http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html, 2016.

[4] "Machine Learning Market Size Worth $96.7 Billion by 2025," https://www.prnewswire.com/news-releases/machine-learning-market-size-worth-96-7-billion-by-2025–cagr-43-8-grand-view-research-inc-300985428.html, 2020.

[5] X. Amatriain and J. Basilico. "Netflix Recommendations: Beyond the 5 stars (Part 1)," https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429, 2012.

[6] J, Brutlag. "Speed Matters," https://ai.googleblog.com/2009/06/speed-matters.html, 2009.

[7] "Chipyard," https://github.com/ucb-bar/chipyard.

[8] "Gemmini," https://github.com/ucb-bar/gemmini.

[9] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," arXiv:1712.05877, 2017.

[10] R. Banner, Y. Nahshan, and D. Soudry, "Post Training 4-bit Quantization of Convolutional Networks for Rapid-Deployment," arXiv:1810.05723, 2019.

[11] "754-2019 IEE Standard for Floating-Point Arithmetic," https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html, 2019

[12] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, H. Wu, "Mixed Precision Training," https://arxiv.org/pdf/1710.03740.pdf, 2018.

[13] P. Micikevicius, "Mixed-Precision Training of Deep Neural Networks," https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/, 2017.

[14] D. Kalamkar, D. Mudigere, D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A Study of BFLOAT16 for Deep Learning Training," arXiv:1905.12322, 2019.

[15] "HardFloat," https://github.com/ucb-bar/berkeley-hardfloat.

[16] "SoftFloat," https://github.com/ucb-bar/berkeley-softfloat-3.

[17] "Gemmini-Rocc-Tests," https://github.com/ucb-bar/gemmini-rocc-tests.

[18] S. Shao, A. Amid, H. Genc, "Lab 3: Tiling and Optimization for Accelerators," http://www-inst.eecs.berkeley.edu/ ee290-2/sp20/assets/labs/lab3.pdf, 2020.

[19] "Hammer," https://github.com/ucb-bar/hammer.

[20] "Tensorflow," https://www.tensorflow.org/.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition," Proceedings of the IEEE, 1998.

[22] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10," https://www.cs.toronto.edu/ kriz/cifar.html.

[23] "Firesim," https://fires.im.