Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product

Brian Hickmann*, Jiasheng Chen†, Michael Rotzin‡, Andrew Yang‡, Maciej Urbanski§, and Sasikanth Avancha¶
Intel Corporation
*Hillsboro, OR, USA, Email: brian.j.hickmann@intel.com
†Folsom, CA, USA, Email: jiasheng.chen@intel.com
‡Santa Clara, CA USA, Email: michael.rotzin@intel.com, andrew.yang@intel.com
§Gdansk, Poland, Email: maciej.urbanski@intel.com
¶Bangalore, India, Email: sasikanth.avancha@intel.com

***Abstract*—Intel's Nervana Neural Network Processor for Training (NNP-T) contains at its core an advanced floating point dot product design to accelerate the matrix multiplication operations found in many AI applications. Each Matrix Processing Unit (MPU) on the Intel NNP-T can process a 32x32 BFloat16 matrix multiplication every 32 cycles, accumulating the result in single precision (FP32). To reduce hardware costs, the MPU uses a fused many-term floating point dot product design with block alignment of the many input terms during addition, resulting in a unique datapath with several interesting design trade-offs. In this paper, we describe the details of the MPU pipeline, discuss the trade-offs made in the design, and present information on the accuracy of the computation as compared to traditional FMA implementations.**

***Index Terms*—Machine learning, deep learning, tensor, matrix multiplication, floating point dot product**

I. Introduction

The rapid growth of artificial intelligence and deep learning applications has led to an increased interest in accelerating these workloads with hardware accelerators. The neural networks used by deep learning rely heavily on matrix operations, especially the matrix multiplication operation, to perform both training of and inference from these networks [1]. Currently the training of deep learning workloads is commonly processed using the IEEE Standard 754-2008 binary32 (single precision or FP32) format [2]. However, due to the immense computational requirement of deep learning workloads, there has been an increased focus on using 16-bit floating point formats such as IEEE 754 binary16 (half precision or FP16) to reduce memory overhead and increase training performance. As an example, NVIDIA's Tensor Core design uses the FP16 format with an FP32 accumulator to provide a major increase in performance for deep learning applications [3].

Intel's Nervana Neural Network Processor for Training applications (NNP-T) is a hardware accelerator targeted at AI applications. Each Matrix Processing Unit (MPU) on the Intel NNP-T contains an advanced floating point dot product design that can process a 32x32 matrix multiplication every 32 cycles, accumulating the result in FP32. To improve performance and reduce hardware costs, this advanced design uses the BFloat16 (BF16) 16-bit floating point format optimized for deep learning and AI applications [4]. In addition, the design exploits the implementation-specific flexibility given by the IEEE 754-2008 standard's definition of the reduction summation, using a block alignment of the many input terms during addition to reduce hardware costs. While such a design has been previously demonstrated for 4 input terms [5], this unit is novel in that extends the design to 32 input terms. In this paper, we describe the details of the Intel NNP-T MPU pipeline, discuss trade-offs made in the unique dot-product design, and present information on the accuracy of the computation as compared to traditional FMA implementations.

II. Intel NNP-T Overview

Intel's Nervana Neural Network Processor for Training (NNP-T) is a standalone PCIE-based accelerator for deep learning and artificial intelligence training acceleration, designed from the ground up to accelerate the training of larger models and datasets with a power efficient custom architecture. Fabricated on TSMC's 16nm process, the chip utilizes a single large 680mm^2 die with over 27 billion transistors and typical workload power ranging from 150-250W. As shown in Figure 1, the heart of the design is an array of 24 Tensor Processor Cores (TPCs) used to accelerate tensor matrix processing, supported by four HBM2 controllers with 8 GB of HBM2-2400 on-package memory each, 64 lanes of SERDES communication for scale-out support, and a fourth generation PCIE x16 interface with the host processor. All of the these components are connected by an advanced bidirectional 2-D mesh network supporting a total cross-section bandwidth of 2.6 TBps.

The Tensor Processor Core (TPC) is shown in detail in Figure 2. Each TPC contains a memory unit with 2.5MB of on-die memory, a control unit with a programmable micro-controller, an on-die mesh router, and two Matrix Processing Units (MPU). Each MPU is able to compute a full 32x32 BF16 matrix multiplication operation every 32 cycles, accumulating the result using FP32 into a local partial result memory. In addition, each MPU can perform two input vector operations and one output vector operation per cycle on the input tensor data and partial result, respectively, in either FP32 or BF16 format. The MPU also has support for deep learning specific optimizations, including matrix wide reduction summation,

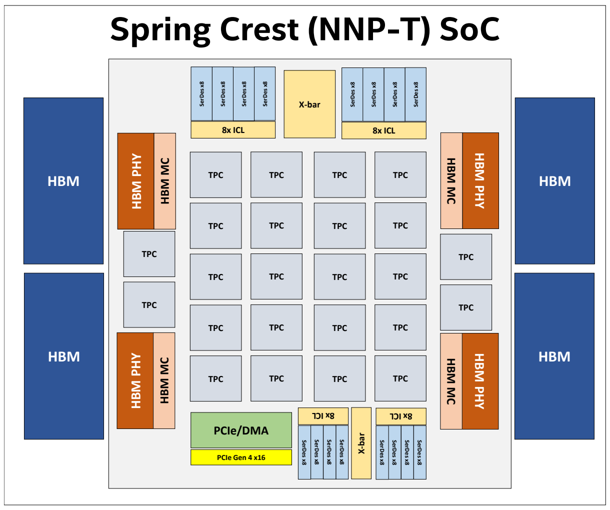

Fig. 1. Intel NNP-T Block Diagram

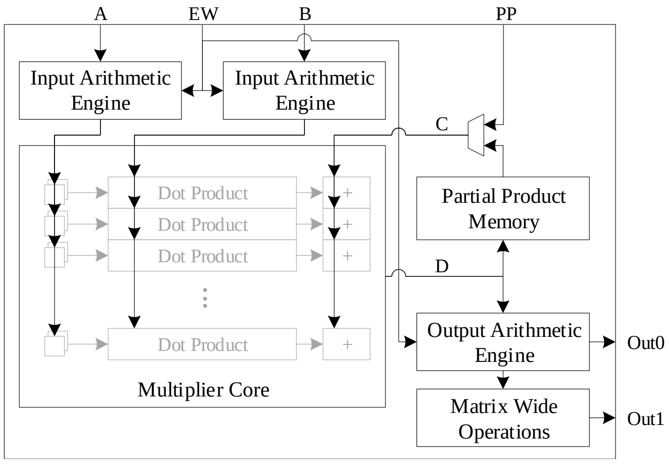

Fig. 3. Intel NNP-T MPU

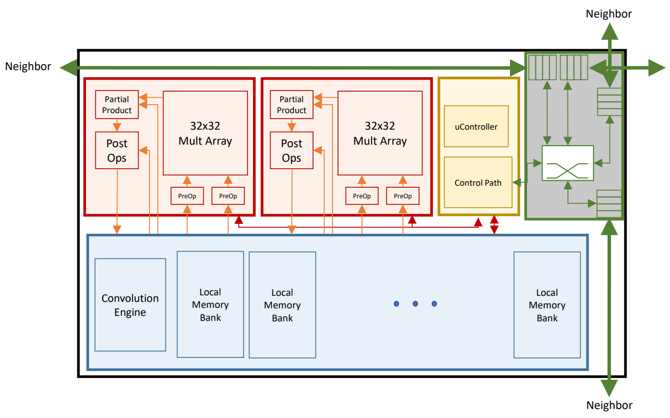

Fig. 2. Intel NNP-T Tensor Processor Diagram

random number generation, and programmable FP32 look-up tables to allow emulation of arbitrary complex functions for use as activation functions or other needs. The whole MPU is fully pipelined to enable high utilization and throughput with large workloads and datasets.

III. Matrix Processing Unit Design

The high level diagram of the MPU unit is shown in Figure 3. The MPU core matrix multiplication unit at the center of the diagram must perform a fully pipelined 32x32 matrix multiplication D=A*B+C every 32 cycles, where A and B are input 32x32 tensors, C is the intermediate partial result 32x32 tensor, and D is the final 32x32 result tensor. The A and B input tensors, stored in BF16, are streamed in over two separate 512-bit (32x16-bit) busses from the memory subsystem. Optionally, input arithmetic operations can also be performed on A and B input tensors as they are loaded from memory in BF16 format. The intermediate result C tensor is either streamed from the memory unit on the dedicated ”PP” 512-bit bus or loaded from a software-managed partial product array, to allow efficient support of matrix multiplications larger than 32x32. The final result tensor D is then stored to the partial product array and optionally streamed out to memory on the output bus. As the result is streamed to memory, it can be optionally down converted to BF16 and additional arithmetic vector operations performed on the result. Note that both the input and output arithmetic units share the single ”EW” input 512-bit bus from the memory unit. Finally, the result tensor D can also be sent to a matrix-wide operation unit to perform reduction summations or to find matrix-wide maximum or minimum values.

In order to perform a matrix multiplication operation, the TPC controller begins by first streaming in the A input tensor over 32 cycles, which is stored in a MPU internal storage array. This storage array contains two storage contexts, allowing the A tensor to be sent even while another operation is happening within the multiplier core. Once the A tensor has been pre-loaded, the B input tensor is then streamed in for 32 cycles. In each cycle, the 32 elements of B are multiplied against the entire stored value of the A and accumulated into a single row or column of the result. This result is then added with single row or column of the C tensor, either read from the local partial result array or streamed in from the memory unit. The final result D is then stored to the partial result array or optionally streamed out to the memory unit on the output bus.

To ensure that the design had excellent numerical properties, The first major design decision was to select BFloat16 (BF16) as the numerical format over several alternatives such as FP32, FP16, and various integer formats, as it offered the power and area efficiency of a 16-bit format with convergence characteristics similar to that of FP32 [6]. The BF16 format utilizes a single sign bit, a 8-bit exponent, and a 7-bit mantissa, encoded in the same manner as existing IEEE 754 formats. To reduce the hardware cost, several IEEE 754 features not used by AI applications are removed from BF16, including support for subnormals, rounding modes other than Round to Nearest Even (RNE), and precise exception handling [4]. During matrix multiplications, the accumulation of BF16 products is always done with a FP32 accumulator to improve precision.

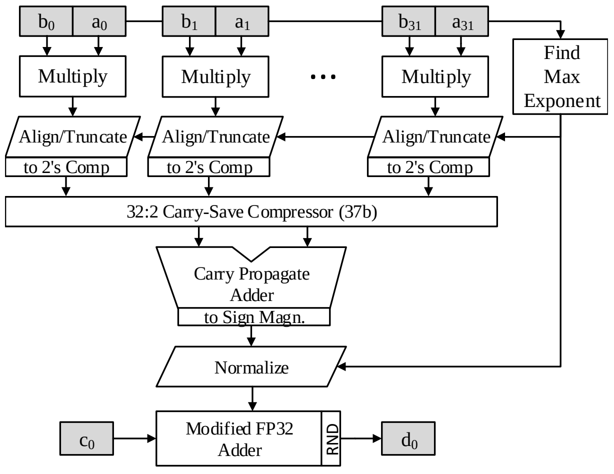

Fig. 4. Intel NNP-T Dot Product Design

The second critical design decision was the microarchitecture of matrix multiplication core. The multiplier core is split into 32 dot product units, each of which calculates the result of one ”row” of the A tensor from the local stored copy multiplied with one ”column” of the B tensor which is broadcast from the value being streamed in from memory. For each dot product unit, the equation describing the computation is show in Equation 1, where $a_{0\text{-}31}$ are the elements of the A tensor, $b_{0\text{-}31}$ are the elements of the B tensor, c is the element of the input partial result, and d is the final result.

$$d = a_0 * b_0 + a_1 * b_1 + ... + a_{31} * b_{31} + c \qquad (1)$$

When designing the dot product unit, we investigated several different microarchitectures, including a chain of floating point fused multiply add (FMA) units, a tree of floating point adders to sum the products, and a fused design using a block alignment scheme with a tree of integer adders. With all of the possible architectures, we had the option of early or late addition of the partial result to the global accumulator. The floating point FMA and adder designs had the benefit of matching the existing IEEE 754 compliant CPU software instructions. However these designs incurred several additional alignment, normalization, rounding, and full carry propagate addition steps which added latency, power, and area to the design, especially due to the large number of products (32) in this design. Therefore these options were dropped in favor of the fused design with a block alignment scheme as shown in Figure 4. In this design, all of the 32 products are first computed independently and then aligned in a single step to the maximum product exponent, truncated to an internal datapath width of 37b, and then converted to 2’s complement form before being added using a large 32:2 carry save adder. A final carry propagate adder computes the intermediate result and converts back to signed magnitude form, which is then normalized and then added to the partial product in a traditional floating point adder to produce the final result.

The use of a block alignment scheme, while hardware efficient, has major implications on the value of the numerical result as compared to traditional IEEE 754 compliant floating point FMA and adder based designs. While deep learning and AI applications are less sensitive to numerical accuracy loss by their nature, we attempted to minimize the impact of the block alignment scheme with two optimizations. First, we selected a design which adds the partial result at the end of the datapath as opposed to including it as a 33rd input to the alignment and reduction tree. This improves accuracy in cases where the accumulated value grows large in comparison to the input data, as is seen in some stages of deep learning algorithms. Secondly, we carefully chose the internal datapath width of 37b, which is significantly wider than the 24b mantissa width of the FP32 result, to ensure that average accuracy across many different random input data distributions met or exceeded that produced by traditional software matrix multiplication implementations. Additionally, these extra bits help protect against corner cases where cancellation due to subtraction in the upper mantissa bits causes a large normalization shift. In these cases, low order mantissa bits which would otherwise only participate in rounding are shifted into the final result mantissa. By keeping extra mantissa bits in the datapath, we reduce the large error that results in these corner cases. Finally, a 37b datapath allows easy future support for a 16b integer data format inside the dot product, as 37 bits are required to sum 32 32b-products without overflow.

During the design of the datapath, several important optimizations were made to further improve the power and area of the design. First, one major difference in this design as compared to traditional floating point multiplier and FMA designs is that early in the pipeline, the timing is actually dominated by the maximum exponent and algnment shift amount calculation. This allowed for the design of a mantissa multiplier optimized for area and power as opposed to minimum timing throughput. Similarly, the additional timing margin was used to implement fine grain clock gating to disable the multiplier during zero and exceptional cases, helping to reduce power in the event of sparse input matrices. Second, to minimize the impact of converting back to signed magnitude form after the reduction, we implemented a specialized completion adder which speculatively computes both positive and negative results to reduce the conversion latency, along with a custom leading zero anticipator (LZA) to ensure the normalization shift amount was computed with minimum delay. Finally, we opted to skip rounding the intermediate reduction result and instead created a customized floating point adder with support for an input with a 37b mantissa. Both of these optimizations allowed us to remove extra pipe stages from the design, saving the associated sequential area and power costs without affecting the timing of the final design.

The resulting design has a 9 cycle pipeline. Cycle 1 is spent staging and decoding the input operands. Cycle 2 and 3 perform the mantissa multiplication and the exponent logic, including finding the maximum exponent using a tree of traditional adders/muxes and calculating the alignment shift amount. Note that an optimized design based on the work present in [7] was considered, but was dropped due to schedule considerations. Cycle 4 performs the mantissa alignment shifts, conversion to 2's complement, and the first half of the carry save addition. Cycle 5 completes the carry save addition and peforms the final carry-propagate addition, LZA, and conversion to signed magnitude form. Cycle 6 performs the result normaliztion and then cycles 7 through 9 perform a floating-point addition with the single precision accumulator using our customized floating-point adder with support for the wider 37b mantissa of the intermediate result.

IV. Intel NNP-T Accuracy Comparison

We performed a numerical precision analysis by looking at the convolution operation during Resnet-50 training with the BF16 format. We sampled all the convolution layers' data around epoch60, with particular interest paid to the weight gradient calculation step because it illustrates two challenging problem for the block aligned adder design. Firstly, it performs very long chains of dot products which require numerical stability during accumulation. We have found the weight gradient calculation to be like an ill-defined accmulation problem with the final result close to zero but with a relatively large intermediate accumulator. The Intel NNP-T design addresses this problem with the late summation of the partial results with the global accumulator to minimize the accumulator error. Secondly, one of the inputs is an activation gradient which shows more frequent exponent changes [8], which could require the block alignment adder to hold larger dynamic range of inputs. The Intel NNP-T design addresses this problem with our large 37-bit adder width.

For reference, we calculated the dot product ground truth by using 500-bit floating point math and then compared the NNP-T numerical error against two other BF16 software implementations. The first is SEQ-FMA which uses a single precision FMA to emulate BF16 as suggested by the Intel BF16 white paper [4]. The other is TC4-24bT which is a generic 4-way block aligned design with a 24bits internal datapath width, early summation for global accumulator, and a final result that is truncated to FP32 while output, modeled after the design shown in [9]. For all the three models, we perform the dot product sequentially by calling the primitive functions without any order changes of the input data.

Table I shows the average Mean Square Error (MSE) aggregated by layer resolutions. The NNP-T offers about an order of magnitude more precision than SEQ-FMA across all of the Resnet50 layers when calculating weight gradient, while the TC4-24bT design is 3 to 6 of orders of magnitude worse.

TABLE I
Average MSE by Layer

	SEQ-FMA	NNP-T	TC4-24bT
HxW=56x56, C=64	9.4E-16	1.4E-17	1.9E-11
HxW=28x28, C=128	9.2E-18	3.8E-19	1.5E-13
HxW=14x14, C=256	5.0E-19	3.9E-20	1.6E-15
HxW=7x7, C=512	7.4E-21	1.3E-21	1.5E-18

The chart in Figure 5 shows the histogram error for weight gradient calculation of one 1x1 convolution layer with NxCxHxWxK=54x512x28x28x128, we use the number of bits of error counted from LSB for the histogram as horizontal axis, which is defined as:

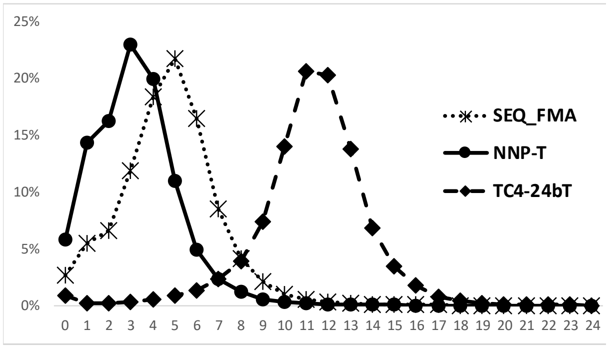

Fig. 5. Histogram of the Number of Bits of Error

$$\text{\# Bits of Error} = \begin{cases} 0 & \text{if } (\log_2 ulp < 0) \\ round(1 + \log_2 ulp) & \text{otherwise} \end{cases}$$

If the error is over 15 or 16 bits it could impact the rounded BF16 result used for weight update and if the error is over 23 or 24 bits, it means the calculated gradient could be 2x off. From this chart, we observed that NNP-T is about 2 to 3 bits more accurate than SEQ-FMA, while TC4-24bT is up to 9 bits worse. We also have data shows that 3x3 convlution layer's weight graidient calculation error is normally 1 to 3 bits better than 1x1 layer, and the layer's error decrease when the feature map increase.

V. Conclusion

The Intel NNP-T accelerator uses a novel fused floating point dot product design with 32 input terms that is highly area and power efficient. We described several unique design optimizations due to its block alignment scheme. In addition, we described design decisions made to enhance numerical precision and showed that this resulted in better average accuracy than IEEE 754 FMA based software implementations.

References

[1] I. Goodfellow *et al.*, *Deep Learning*. The MIT Press, 2016.
[2] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.
[3] (2017) NVIDIA Tesla V100 GPU Architecture. [Online]. Available: http://tinyurl.com/volta-architecture-whitepaper
[4] (2018, Nov.) BFLOAT16 - Hardware Numerics Definition. [Online]. Available: https://tinyurl.com/bf16-white-paper
[5] J. Sohn and E. E. Swartzlander, "A Fused Floating-Point Four-Term Dot Product Unit," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 3, pp. 370–378, 2016.
[6] D. D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019. [Online]. Available: http://arxiv.org/abs/1905.12322
[7] H. Kaul *et al.*, "Optimized fused floating-point many-term dot-product hardware for machine learning accelerators," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 84–87.
[8] V. Popescu *et al.*, "Flexpoint: Predictive Numerics for Deep Learning," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, pp. 1–4.
[9] B. Hickmann and D. Bradford, "Experimental Analysis of Matrix Multiplication Functional Units," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, 2019, pp. 116–119.