PreCog: Near-Storage Accelerator for Heterogeneous CNN Inference

Jiyoung An
Department of Computer Science
University of California, Irvine
Irvine, USA
Email: jiyouna2@uci.edu

Esmerald Aliaj
Department of Computer Science
University of California, Irvine
Irvine, USA
Email: ealiaj@uci.edu

Sang-Woo Jun
Department of Computer Science
University of California, Irvine
Irvine, USA
Email: swjun@ics.uci.edu

***Abstract*—Computational Storage Devices (CSD) with near-storage acceleration is gaining popularity for data-intensive applications, by moving power-efficient hardware acceleration closer to data. However, because the power-constrained near-storage accelerator is often not powerful enough by itself to handle all computation requirements of an application, it must intelligently cooperate with other computation and acceleration units in the system. In this work, we explore how a near-storage accelerator can best fit into a larger computer system in the context of CNN inference. We demonstrate that an attractive configuration is using the near-storage accelerator to offload only the first convolution and pooling layers, where the accelerator can achieve almost an order of magnitude better performance compared to a general convolution accelerator. Targeting only the first layer allows some FPGA-specific floating-point computation optimizations such as pre-determine the range of output exponents, performing a costly floating point normalization task only once, as well as pack more input into the datapath to mitigate the performance impact of wide strides of the convolution filter. We package these optimizations into a flexible library we call *Static Range Float (SRFloat)*, and construct a prototype system called PreCog. We evaluate PreCog implemented on the Samsung SmartSSD platform, and demonstrate over 3× performance efficiency compared to conventional convolution accelerators on prominent CNN models without introducing a communications bottleneck.**

I. Introduction

Hardware acceleration is an integral component of neural network infrastructure, for both training [1], [2] and inference portions [3], [4], providing orders of magnitude performance, size, and power efficiency benefits over general-purpose software. As a result, each computer system for neural network workloads is often equipped with multiple powerful accelerators, either GPUs [5] or TPUs [6]. Accelerators are also being explored in various locations within a system, such as the network edge [7], [8], embedded within the network interface card [9], as well as near memory and storage [10].

An attractive configuration which is gaining rapid popularity for accelerator usage is Computational Storage Devices (CSD) equipped with near-storage accelerators. Since input data is often first stored in secondary storage due to their size [11], [12], accelerators embedded in storage can function in a bump-in-the-wire fashion without incurring significant additional latency [13], [14]. Furthermore, storage devices often provision significantly less interconnect resources compared to its peak internal bandwidth due to economic reasons, but the near-storage accelerator can take advantage of the faster internal bandwidth [13]. As a result, commercial offerings such as Samsung SmartSSD are being rapid adopted and evaluated [15].

However, careful design is necessary for efficient near-storage acceleration within a heterogeneous system equipped with multiple accelerators, due to the performance balance against different system components. For example, cost and power constraints limits the performance of computation and on-board memory. It would be detrimental if the power-constrained near-storage accelerator limits the data ingestion rate, causing a potentially more powerful array of standalone accelerators to be idle. It would also be detrimental if near-storage acceleration increases the data volume too much, which may create a new bottleneck at the host-side interconnect. The goal of a near-storage accelerator should be to contribute as much computation as it can, while not introducing a new bottleneck. The question now becomes, when **given an existing cloud machine with near-storage acceleration**, what is **the effective division of work** for the accelerator on a convolutional neural network (CNN) workload?

In this work, we propose that an attractive division of work for a near-storage accelerator is at the very first convolution layer, as well as the first pooling layer, if any. We target models with floating-point weights despite their complexity compared to quantized integer ones. Floating-point models with convolutions as the first layer is a **popular and important** subset of neural network models. Many popular, state-of-the-art neural network models include a convolution layer as the first layer [16]–[21]. Also, floating-point models still represent a very large and important portion of models in use, and commercial accelerators such as the TPU must consider them in their design [22].

We discover that offloading only the first layer has many performance benefits. Not only is a convolution-only accelerator capable of higher performance per chip resource compared to the full model accelerators due to memory performance issues [23], a first-layer accelerator can also take advantage of the fact model input is three 8-bit pixels instead of floating point values, resulting in a narrow range of possible exponent values. Since the number of weights for a single layer

DOI 10.1109/ASAP57973.2023.00021

convolution is quite small, we can also statically hard-code them, so that the compiler can statically pre-compute most of the work required for normalizing floating point numbers, **without accuracy loss.** This is an optimization only available on a reconfigurable accelerator such as FPGAs, and results in an extremely high computation density.

We package these optimizations into a library we call *Static Range Floats (SRFloat)*, and use them to construct a single-layer convolution accelerator.

We use the Samsung SmartSSD [15] near-storage acceleration platform to implement a prototype near-storage single-layer convolution accelerator using Static Range Floats, which we call **PreCog**, and evaluate its performance efficiency using four popular CNN models. We demonstrate that our single-layer accelerator achieves almost an order of magnitude performance efficiency compared to accelerators with conventional floating point operators such as Float16 and BFloat16 [24]. It also aids in better power efficiency, as our accelerator has over 4x the power efficiency of an optimized GPU implementation of CNNs.

The rest of this paper is organized as follows: We present some background and existing work on this topic in Section II. We describe the architecture of our PreCog design in Section III, and present our evaluation and analysis results in Section IV. We conclude with future work in Section V.

II. Background and Related Work

A. Neural Network Acceleration

Neural Networks are widely used in a number of different applications ranging from image and speech recognition [25] to recommendation systems [26] and Natural Language Processing [27]. Many studies [17], [28], [29] have suggested that larger models is an effective way to improve performance, which puts more emphasis on high-performance accelerators.

While training neural network models is definitively more complex than using them for inference, inference still accounts for 90% of total ML operational costs [30], and their acceleration is important in industry, not only for quantized fixed-point models [31], [32], but for new and existing floating-point models as well due to the large number and importance of existing models [22].

For a full-model neural network accelerator, memory bandwidth requirements of fully connected layers is a prominent bottleneck during inference, prompting research into efficient caching and reuse [33]–[36].

B. Near-Storage Acceleration

Fast modern storage devices often under-provision the host-side PCIe bandwidth compared to the peak attainable internal bandwidth. For off-the-shelf SSDs, the gap can be 2× or larger [37]. Embedding fast accelerators on the storage device itself can be a practical way to utilize this performance gap, while also incorporating power-efficient acceleration. While some systems explored using general-purpose processors for this purpose [38], [39], they are not always sufficient for intensive computation such as CNNs. Embedding reconfigurable hardware accelerators such as FPGAs is an attractive solution thanks to their high performance within the low storage power budget [40]–[42]. Many near-storage accelerator platforms are being developed and explored, either located between the host and storage [15], [43], or as peer-to-peer entities on the PCIe fabric [44].

C. Floating point optimizations for neural networks

Parallel to research efforts regarding quantized, fixed-point arithmetic for neural network models [31], [32], there is active research in neural network optimized floating point formats as well. For example, BFloat16 [24] modifies the IEEE half-precision floating point format to have a larger exponent and smaller mantissa fields, as a larger exponent was discovered to be more beneficial for accuracy. The Microsoft Float (MSFP) takes this approach further [45], encoding 4x4 blocks of mantissas with a shared 8-bit exponent. MSFP suffers a slight loss of accuracy from needing to align mantissas to the shared exponent, but it has demonstrated virtually no loss of machine learning model accuracy for popular models.

III. PreCog Architecture

PreCog targets a computer system equipped with PCIe-attached storage coupled with near-storage accelerators, as well as one or more powerful machine learning accelerator such as GPU, TPUs, or more FPGAs. This is the typical expected deployment scenario for CSDs as the power-constrained CSD by itself can rarely provide sufficient performance. The PreCog near-storage accelerator is programmed onto the FPGA fabric of the computational storage device, as seen in Figure 1.

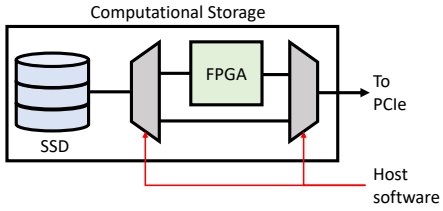

Fig. 1. PreCog implements accelerated and a non-accelerated paths to avoid stalling the storage.

PreCog contains two separate data paths; an accelerated path and a non-accelerated path. The non-accelerated path is the normal file system interface through `open` and `read`. The accelerated path exposes an accelerator-specific interface to the host software: `precogread` to route a variable number of images from storage to the accelerator and then read the results, as well as `precogqueuecnt`, which returns the number of images currently in the accelerator queue. In the accelerated path, PreCog will perform the first convolution as well as max-pooling layers of the CNN and emit its result. Based on the number of images currently waiting in the accelerator queue, the host software can decide to use the non-accelerated path to avoid waiting for the accelerator to finish.

A. Convolution Filter Architecture

PreCog's convolution filter design takes advantage of various optimizations made possible by its focus on only the first convolution layer, coupled with the reconfigurable FPGA characteristics.

1) Model-Specific Architecture: PreCog can afford to use a highly efficient row-buffer structure [46] tailored to each individual model, instead of constructing a fixed-sized accelerator and trying to map models to it. This is an approach made available thanks to reconfigurable FPGAs. The filter dimensions are parameterized to adapt to different model requirements. For example, if a new model requires 7×7 convolutions, PreCog will instantiate 7×7 filter implementations and reconfigure the near-storage accelerator. models.

This convolution module can be replicated according to the filter count requirement of the model layer. If the FPGA chip's resource limitations prevent the required number of filters from being instantiated, weights of more than one filters can be hard-coded into each convolution module instance. This way, a small number of hardware modules can be time-shared between multiple filters to satisfy model requirements.

2) Wide Input Bus for Wide Strides: A straightforward row-buffer architecture for convolutions can suffer performance degradation if the model requires a wide stride. The most straightforward way input can be delivered to the convolution accelerator is a simple stream of one input tuple per cycle, which is not very efficient if the convolution filter has a wide stride. For a stride of S, on average S^2 cycles need to be spent ingesting and computing convolutions, only to emit a single valid convolution output. To be more specific, when the filter moves through a column, S cycles need to be spent ingesting and calculating S convolution operations, even though $S-1$ of the operations are discarded. When the filter moves through a row, $(S-1) \times W$ cycles is wasted to fill $S-1$ row-buffers, where W is the width of image.

To efficiently support convolution filters with a wide stride, PreCog co-optimizes the data layout with the convolution filter architecture. The input data layout is re-organized into blocks of S rows and S columns, adding up to $3S^2$ bytes. The value 3 represents the R, G, B pixel bytes. One block is ingested every cycle by the convolution filter. This way, there is enough data being inserted into the filter module every cycle, to support a single valid convolution operation output every cycle.

Figure 2 shows the convolution filter architecture with stride size of $S = 3$ and a filter size of $N \times N$. Input is inserted into S rows at every cycle, and S data elements are dequeued from each filter row at every cycle, allowing a stride of S to happen in a single cycle. The top module consists of M convolution modules, where M is the number of output channels. Each convolution module is responsible for performing the calculations on each output channel and accumulating the resulting partial sums.

We emphasize that this approach is made feasible thanks to the exclusive focus on the first layer, which has 8-bit pixel inputs. For example, the first convolution layer of AlexNet [28] has a stride size of 4, meaning $3 \times 4^2 = 48$ input elements needs to be ingested per cycle. For 8-bit pixel inputs, this requires a wide but manageable 384-bit datapath. For 32-bit floating point inputs, the requirement becomes 1,536-bits, which is a lot more difficult to route on the FPGA at high clock speeds.

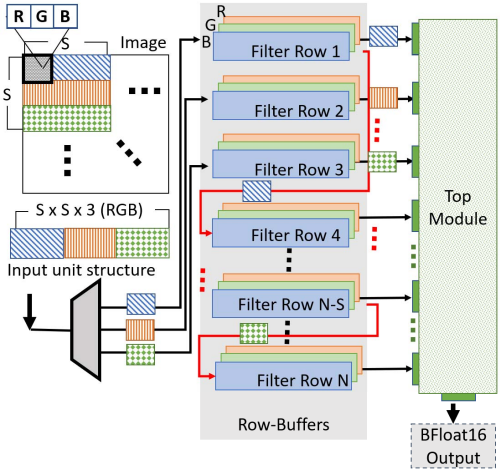

Fig. 2. Convolution Filter Architecture with Stride size of S and a filter size of $N \times N$ ($S = 3$)

B. Pooling Layer Architecture

The *pooling* layer implementation is tightly coupled and pipelined with the convolution layer in order to reduce overhead and utilization. The *pooling* operations, including max-pooling, min-pooling, and average-pooling, can be divided into two steps; column-wise and row-wise. The column-wise portion of the pooling operation is actually integrated into the convolution module itself. The column-wise pooled results are moved to the Top module, where each column-wise pooled result is inserted into the M buffers to perform row-wise pooling.

C. Static Range Float Optimization

Static Range Float (SRFloat) takes advantage of the narrow range of possible exponent values from three 8-bit pixels, being multiplied to the compile-time provided model weight values. The potential exponent value of 8-bit pixel inputs, after normalization, will be between zero and positive seven, and since exponents of the weights are known at compile time, the range of the output exponents can be determined down to this range.

To take advantage of this characteristic, SRFloat encodes floating point values for 8-bit pixel inputs in a slightly different way from the IEEE floating point format. SRFloat encodes the 8-bit inputs with an 8-bit mantissa and a fixed exponent value (127 to represent integers). Figure 3 compares the SRFloat intermediate encoding to BFloat16, which also has a bit budget of 16 bits. Instead of adjusting the exponent to align the MSB of the mantissa, SRFloat simply stores the integer value in the mantissa without normalizing the exponent. This approach is

possible because input to the first layer are 8-bit pixel values, which is small enough to directly encode in the mantissa without too much resource overhead. For example, if each pixel input had a very wide range of valid values similar to Float32, it would not be possible to encode them in a fixed-exponent format. Eventually this encoding needs to go through exponent normalization before emitting it as output, but in the best-case scenario it has to be done only once, before the output is emitted outside of PreCog.

Value	BFloat16 {Sign(1), Exponent(8), Mantissa(7)}	Our Approach {Sign(1), Exponent(7, Fixed), Mantissa(8)}
1.0	{0, 127, 0}	{0, 127, 1}
2.0	{0, 128, 0}	{0, 127, 2}
3.0	{0, 128, 64}	{0, 127, 3}

Fig. 3. 8-bit pixel input format example for our approach

To avoid introducing accuracy degradation without having to do normalization after every arithmetic operation, SRFloat adjusts the bit width of the mantissa for each arithmetic operation. For example, multiplying two, each with 8-bit mantissas, will result in a 16-bit mantissa to avoid mantissa overflow. On the other hand, simply taking 8 most significant bits may cause underflow if the multiplied mantissas are small. Maintaining the same number of mantissa bits without introducing new data loss requires costly exponent normalization, and we avoid this by simply adding more bits to avoid overflow.

Figure 4 shows an example SRFloat architecture for a 1×3 convolution filter. There are nine 8-bit input values corresponding to 3 input color pixels. In this example, weights are encoded in BFloat16 format with a 7-bit mantissa, 8-bit exponent, and a sign bit. Multiplying each 8-bit input to a 7-bit static weight value results in 15-bit mantissa values. These values are inserted into adder tree, and summed into the final output. In this example, we use a tree of 3-ary adders since because this was the maximum fan-in our adder could support without signal timing issues.

By focusing only on the first layer, SRFloat supports addition without intermediate normalization even without the block-aligning approach taken by block operators such as MSFP [45].This is possible because the exponents of the weights are known at compile time, and the SRFloat intermediate format stores mantissas without normalization. As a result, the exponent of each addition operands, according to the un-normalized SRFloat intermediate encoding, are also known at compile time. So aligning exponents between addition operands can be done with a static, compile-time determined shifting operation based on the statically known input exponents.

In the adder tree, the bit width of the resulting partial sum may increase for each addition, because sum of three values can require a bit width increase of up to 2 to avoid overflow, if the exponents of the adder operands are the same. If we used 2-ary adders, the bit width increase would have been up to 1 bit per adder.

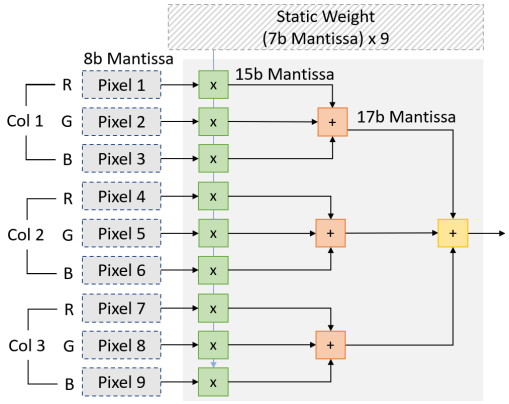

Fig. 4. SRFloat single row: 1×3 filter example

The resulting partial sum of each row in Figure 4 are be summed up by another 3-ary adder, as shown in Figure 5. Next, we add the (also statically encoded) bias to the resulting 19-bit value, potentially increasing the bit width to 20. Lastly, we apply the costly floating-point normalization operation only once, to the final mantissa. The normalized final results are then trivially encoded to BFloat16 or other floating-point formats by truncating the mantissa, and then emitted for use by other components. The default PreCog design emits BFloat16 values to reduce data amplification, and due to the ease of translating BFloat16 to IEEE Float32, but it is trivial to support other formats in the hardware.

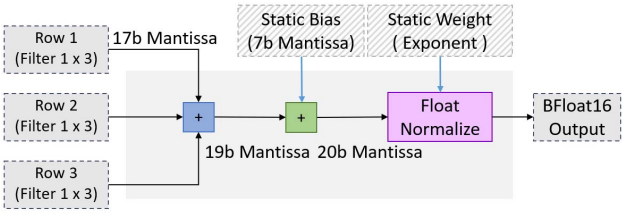

Fig. 5. SRFloat: 3×3 filter example

IV. PreCog evaluation

A. Evaluation Environment

We evaluate PreCog implemented on the *Samsung SmartSSD* platform [15], which augments a 4 TB PM1733 SSD with a Xilinx Kintex Ultrascale+ KU15P FPGA and 4 GiBs of DDR4 SDRAM. The current SmartSSD device has a bandwidth of 3 GB/s both between the FPGA and SSD, as well as between FPGA and the host over PCIe. All versions of our PreCog prototype implementation operate at the standard kernel frequency of 300 MHz.

We evaluate PreCog on the four popular CNN models listed in Table I. Dimensions of the convolution layer spanned widely

TABLE I
EVALUATED CNN MODELS

Model		Conv 1	Max pooling 1
VGG19	[25]	64 3x3, Stride 1	None
YOLOv3-Tiny	[16]	16 3x3, Stride 1	2x2, Stride 2
Resnet50	[17]	64 7x7, Stride 2	3x3, Stride 2
Alexnet	[28]	64 11x11, Stride 4	3x3 Stride 2

from 3x3 to 11x11, and also the output channel count spanned widely between 16 to 64. While most models also included a max pooling layer after the first convolution layer, the VGG19 model did not, which resulted in significant data amplification.

For each of these models, we have explored the efficiency of a wide range of floating point formats and operators, including the standard IEEE 754 single and half-precision floating-points processed with standard Xilinx floating point cores, BFloat16 [24] format via a custom-configured Xilinx floating-point core, MSFP.12, and SRFloat, which is used in our custom floating-point library.

B. Resource Utilization Analysis

We first evaluated the number of convolution modules of each type which could fit into the *Samsung SmartSSD* platform by incrementally increasing the number of filters until the kernel could not fit. Table II shows the maximum amount of processing that could fit on chip for each combination of model and data type. The column **FilterN** represents the number of filter that can fit, and data types which support the highest attainable filter count numbers are marked with an asterisk (*****). If the number of filter modules instantiated is the same or larger than the number of filters in the model layer, it is marked with a plus (**+**).

TABLE II
RESOURCE UTILIZATION OF PRECOG WITH VARIOUS OPERATOR TYPES.

Model	DataTypes	FilterN	CLB	BRAM	DSPs
VGG19	Float16	18	60,827	314	1,953
	BFloat16	18	63,830	314	981
	Float32	14	63,680	310	1,899
	MSFP	69	64,370	365	9
	+* SRFloat	152	61,789	546	9
YOLOv3-Tiny	Float16	18	61,356	323	1,953
	BFloat16	18	65,071	323	981
	Float32	14	64,328	317	1,899
	MSFP	63	65,013	391	9
	+* SRFloat	142	61,780	607	9
Resnet50	Float16	3	53,929	305	1,773
	BFloat16	3	57,988	305	891
	Float32	2	54,275	299	1,479
	MSFP	6	64,734	407	9
	* SRFloat	26	61,471	296	9
Alexnet	Float16	1	49,520	300	1,461
	BFloat16	1	50,306	300	735
	Float32	1	61,098	300	1,824
	MSFP	3	64,417	466	9
	* SRFloat	11	61,562	294	9
SAMSUNG SmartSSD resource			65,340	984	1,968

From Table II, we can see that our SRFloat implementation is consistently the most efficient design. Conventional floating point operators such as Float16, BFloat16 and Float32 have significantly worse efficiency, so much that they sometimes even had trouble fitting a single convolution filter, with models with large-size filters such as Alexnet. In all cases, Float32 consumed the highest number of CLBs, followed by BFloat16, and then Float16. MSFP does second best, training behind our SRFloat implementation. DSP usage of BFloat16 accounts for half of the one seen for Float16 and LUT usage of BFloat16 is higher than that of Float16. This is because the BFloat16 module, created via specifying a custom floating point encoding in the Xilinx core generator, does not use DSPs in the custom floating-point addition module. We also note that our SRFloat implementation does not use DSPs, and the 9 DSPs used by the SRFloat configuration is by the platform infrastructure and not the SRFloat kernel.

C. Performance analysis

1) Peak throughput of convolution accelerator: Figure 6 depicts the peak throughput of each accelerator configurations on the SmartSSD platform in terms of GFLOPS, using the maximum number of PEs as describe in Table II. We note that our Float32 serves as a good baseline for performance comparisons, since it shows competitive performance density compared to existing floating-point neural network accelerators normalized across target FPGA resources [23], [47]. Furthermore, our Float32 and Float16 designs use almost all of the available DSPs, per Table II, and keeps them busy, further assuring we have a competitive baseline implementation.

In all cases, SRFloat significantly outperforms all other configurations, regularly outperforming conventional floating point operators by **over 9**$\times$ on average, and even the upper bound projection of unmodified MSFP by **over 2**$\times$. This shows that as long as the accelerator resources can always be kept busy, PreCog can contribute a much better performance benefit compared to conventional convolution accelerators using general-purpose floating point cores. We emphasize that PreCog also encodes its static weights via BFloat16 by default, and does not introduce any additional accuracy loss compared to the BFloat16 design. However, the incremental optimizations of lightweight floating-point operators, utilizing 8-bit input pixels and hard-coded weights, and limiting to only the first layer, results in the substantial improvements seen in SRFloat.

Furthermore, we emphasize that the GFLOPS numbers in Figure 6 can be contributed to the overall system **regardless of the performance of additional accelerators** such as GPUs on the host. Specifically, PreCog will never become a new performance bottleneck no matter how fast the host-side GPU is. This is because PreCog dynamically routes data to the accelerated path only when neither the PCIe or the accelerator is busy, never creating a new bottleneck.

2) End-to-End Performance: We evaluate the end-to-end performance benefits of PreCog by comparing the performance of a GPU-only system with a single host-side GPU (NVIDIA GeForce RTX 2080ti), and a PreCog+GPU system where PreCog implemented on the SmartSSD processes only the first convolution layer, and the GPU processes the subsequent

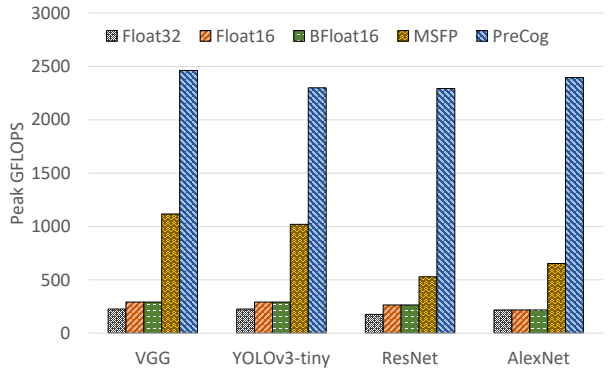

Fig. 6. Peak throughput of SRFloat significantly exceeds all other evaluated configurations.

layers. The GPU-only system also includes a Samsung SSD 860 evo for fast access to input data. We use the term *Offloaded GPU* to indicate when a GPU is processing the layers other than the first convolution.

We evaluate GPU Performance using ONNX Runtime [48], a cross-platform inference and training machine-learning accelerator on multi-node NVIDIA GPU, developed by Microsoft. We choose this system for GPU evaluations because it showed consistently higher performance compared to many other platforms such as Keras. Batch size of 64 was chosen as best-performing after exploration, and matches published suggestions as the optimal batch size [49].

Table III shows the performance of individual system components: SSD, GPU, PreCog, and Offloaded GPU performance, in terms of Images/s. The fastest system component (excluding the SSD) is emphasized with bold. In all cases, the PCIe-attached NVMe SSD can supply more images per second than GPU can process, suggesting that for best performance balance, it may be beneficial to provision more than one GPU per each SSD. We emphasize that the throughput difference between GPU and Offloaded GPU is proportional to the FLOPs requirement of the first convolution layer relative to the whole model, meaning the approach of offloading the first layer does not incur additional processing overhead.

TABLE III
SSD, GPU, PRECOG AND OFFLOADED GPU PERFORMANCE

Model	SSD (Images/s)	GPU (Images/s)	PreCog (Images/s)	Offl. GPU (Images/s)
VGG19	22,112.83	287.29	**1,905.89**	221.61
YOLOv3-Tiny	6,411.41	718.48	**2,064.97**	794.65
Resnet50	22,112.83	1,472.00	1,304.03	**1,633.45**
Alexnet	22,112.83	7,958.58	2,373.32	**20,266.66**

Accelerated vs. Non-Accelerated Paths: For VGG19 and YOLOv3-Tiny, the PreCog accelerator has higher throughput that the host GPU, meaning that all images should be routed through the accelerator for best performance, provided that it does not cause a PCIe bottleneck from data amplification. We also present measurements on data amplification below. For Resnet50 and Alexnet, the offloaded GPU performance is higher than the accelerator performance, meaning the host software should direct images through the accelerator only when the accelerator is idle and can accept new work. This way, PreCog can never cause a performance loss, even when the accelerator is slower than the data rate. Furthermore, this performance gap results in the PreCog accelerator being kept busy at all times, meaning the overall system can fully benefit from the performance benefit of SRFloat compared to conventional floating point operators, as presented in Figure 6.

An interesting case study is VGG19, which cases a very high data amplification due to a large number of filters (64) in the first convolution layer without an immediately subsequent pooling layer. Despite the large gap in SSD and GPU performance as seen in Table III, such a large amplification causes the PCIe to become a new bottleneck, and the performance of the offloaded GPU instance is actually *slower* for VGG19 compared to the original GPU system, as seen in Table III. PreCog considers this information and always chooses the non-accelerated path for VGG19, causing the computational storage to behave like a regular SSD, and resulting in no performance improvement from PreCog.

Measured Performance Improvements: Table IV compares the achieved end-to-end performance of a conventional SSD + GPU system and a PreCog + GPU system, and Table V shows the data amplification after the first convolution+pooling layer. The results show that PreCog can often significantly improve the end-to-end performance of the whole system. Looking at individual models, the performance of YOLOv3-Tiny was improved by 11%, Resnet50 by 9%, and Alexnet by 18% compared to the conventional SSD + GPU system. This is an attractive addition to the overall system, especially considering the power efficiency of PreCog we present in the following section.

For models like VGG19 with high data amplification, we cannot use the only CONV1 approach we used for PreCog, and must instead use the accelerator to implement a conventional convolution accelerator using conventional, performance-inefficient operators such as Float32 or BFloat16. We do not explore this additional approach in this work, to focus on the PreCog approach. For all other models, the small amount of data amplification did not have a negative impact on performance due to the performance gap between SSD and GPU, as well as the high computation contributions of the accelerator.

3) Per-FLOP Power Efficiency: The performance contributions of PreCog also comes with very high power efficiency. We could not measure dynamic power since we did not have physical access to the SmartSSD power input. As an alternative, we analyzed power efficiency based on the advertised maximum power consumption of each device, since most components are always working at maximum performance in our evaluation scenarios. The maximum power consumption collected in each device specification is as follows:

TABLE IV
END-TO-END SYSTEM PERFORMANCE

Model	SSD + GPU (Images/s)	PreCog + OffloadedGPU (Images/s)	PreCog Improvement Rate
VGG19	287.29	221.61	1.00
YOLOv3-Tiny	718.48	794.65	1.11
Resnet50	1472.00	1600.89	1.09
Alexnet	7,958.58	9,399.91	1.18

TABLE V
DATA AMPLIFICATION RATE OF EACH MODEL.

	VGG19	YOLOv3-Tiny	Resnet50	Alexnet
SRFloat	42.7	2.7	2.7	0.62

- Samsung SSD 860 evo : $4W$
- Samsung SmartSSD : $25W$
- NVIDIA GeForce RTX 2080ti : $250W$

We note that current estimated results did not reflect dynamic power management, one of the key features of SmartSSD. Thus, the actual end-to-end power efficiency may be higher.

Figure 7 shows power efficiency comparisons **per FLOP** between computation engines including GPU, meaning the power efficiency of individual components while they may be working together in the overall system. Measured power consumption for the FPGA implementations accounts for one SmartSSD, and the GPU implementations accounts for one GPU and one SSD. We observe that power efficiency of SRFloat significantly exceeds all other evaluated configurations. On average, SRFloat contributes $4.5\times$ more GFLOPS per Watt to the heterorgenous system compared to the GPU.

V. CONCLUSION

In this paper, we present PreCog, a near-storage accelerator which achieves superior performance efficiency by offloading only the first convolution and max pooling layers of a CNN. We demonstrate that SRFloat can contributes almost an order of magnitude higher FLOPS compared to conventional floating-point encodings such as Float16, Float32, BFloat16, and MSFP. As a result, programming a near-storage accelerator with PreCog was able to significantly improve the performance of a GPU-accelerated heterogeneous inference system, at a minimal additional power consumption.

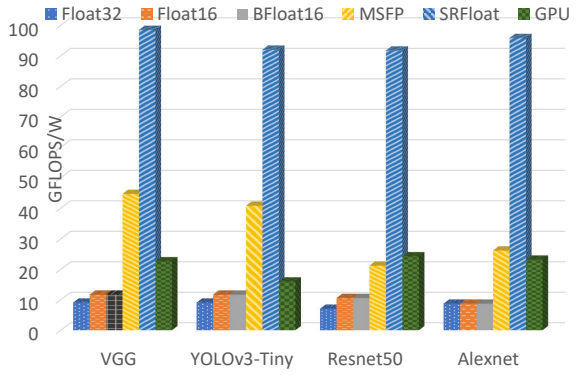

Fig. 7. SRFloat can calculate $4.5\times$ more GFLOPS per Watt than GPU on average.

For near-storage accelerated systems, PreCog is an efficient approach for extracting the maximum performance for neural network applications. We plan to continue exploring variations of this approach, including similar optimizations for multi-layer convolution accelerators, as well as modifying neural network models for this architecture. Furthermore, we note that the current SmartSSD system does not have a higher internal bandwidth compared to its PCIe. Benefits of PreCog may become more pronounced with future computational storage products which plan to open its higher internal bandwidth to the accelerator.

ACKNOWLEDGMENT

This work has been made possible in part by grants from Samsung Electronics, as well as funding from the National Science Foundation (CNS-1908507).

REFERENCES

[1] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards efficient deep neural network training by fpga-based batch-level parallelism," *Journal of Semiconductors*, vol. 41, no. 2, p. 022403, 2020.

[2] H. Nakahara, Y. Sada, M. Shimoda, K. Sayama, A. Jinguji, and S. Sato, "Fpga-based training accelerator utilizing sparseness of convolutional neural network," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 180–186.

[3] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019.

[4] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[5] Amazon AWS, "Recommended gpu instances - deep learning ami," https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html, 2021, [Online; accessed 2022-03-20].

[6] Google Cloud, "Tpu types and zones," https://cloud.google.com/tpu/docs/types-zones, 2022, [Online; accessed 2022-03-20].

[7] J. Chen, S. Hong, W. He, J. Moon, and S.-W. Jun, "Eciton: Very low-power lstm neural network accelerator for predictive maintenance at the edge," in *Field Programmable Logic and Applications (FPL)*. IEEE, 2021.

[8] R. A. Cooke and S. A. Fahmy, "Quantifying the latency benefits of near-edge and in-network fpga acceleration," in *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, 2020, pp. 7–12.

[9] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 36–43.

[10] P. Das and H. K. Kapoor, "Clu: A near-memory accelerator exploiting the parallelism in convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 2, pp. 1–25, 2021.

[11] R. Kaplan, L. Yavits, and R. Ginosar, "Prins: Processing-in-storage acceleration of machine learning," *IEEE Transactions on Nanotechnology*, vol. 17, no. 5, pp. 889–896, 2018.

[12] S. Xu, G. Yang, Y. Mu, and X. Liu, "A secure iot cloud storage system with fine-grained access control and decryption key exposure resistance," *Future Generation Computer Systems*, vol. 97, pp. 284–294, 2019.

[13] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1221–1230. [Online]. Available: http://doi.acm.org/10.1145/2463676.2465295

[14] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*. IEEE, 2013, pp. 1–12.

[15] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," *IEEE Computer architecture letters*, vol. 19, no. 2, pp. 110–113, 2020.

[16] P. Adarsh, P. Rathi, and M. Kumar, "Yolo v3-tiny: Object detection and recognition using one stage improved model," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. IEEE, 2020, pp. 687–694.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[18] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[21] K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, and C. Xu, "Ghostnet: More features from cheap operations," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 1580–1589.

[22] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[23] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *ieee Access*, vol. 7, pp. 7823–7859, 2018.

[24] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2019, pp. 88–91.

[25] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[26] H. Wang, N. Wang, and D.-Y. Yeung, "Collaborative deep learning for recommender systems," in *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, 2015, pp. 1235–1244.

[27] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[30] AWS re:Invent, "Machine learning inference with amazon ec2 inf1 instances," https://www.youtube.com/watch?v=yQU7MDCXnoU, 2020, [Online; accessed 2022-03-18].

[31] J. H. Kim, J. Lee, and J. H. Anderson, "Fpga architecture enhancements for efficient bnn implementation," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 214–221.

[32] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[33] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 13–19.

[34] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[35] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 16–25. [Online]. Available: https://doi.org/10.1145/2847263.2847276

[36] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 45–54. [Online]. Available: https://doi.org/10.1145/3020078.3021736

[37] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[38] G. Koo, K. K. Matam, H. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram *et al.*, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 219–231.

[39] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.

[40] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu *et al.*, "Bluedbm: An appliance for big data analytics," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 1–13.

[41] M. Torabzadehkashi, S. Rezaei, V. Alves, and N. Bagherzadeh, "Compstor: An in-storage computation platform for scalable distributed processing," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1260–1267.

[42] Z. Ruan, T. He, and J. Cong, "{INSIDER}: Designing in-storage computing system for emerging high-performance drive," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 379–394.

[43] BittWare, "250s+ pcie card with xilinx kintex ultrascale+ ku15p fpga," https://www.bittware.com/fpga/250s/, 2021, [Online; accessed 2022-03-27].

[44] ——, "250-u2 u.2 computational storage module with xilinx kintex fpga," https://www.bittware.com/fpga/250-u2/, 2021, [Online; accessed 2022-03-27].

[45] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner *et al.*, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," *Advances in Neural Information Processing Systems*, vol. 33, pp. 10 271–10 281, 2020.

[46] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 101–108.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[48] O. R. developers, "Onnx runtime," https://onnxruntime.ai/, 2021, version: x.y.z.

[49] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 5–14.