ProcPIPE: Offloading processing pipelines on Computational Storage

Shashwat Jain
Virginia Tech
shashwatjain@vt.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

Abstract

We propose ProcPIPE - a prototype framework that enables applications to offload processing pipelines to computational storage by providing an interface that manages near storage accelerator resources at device side. The prototype leverages Smart SSD which is near storage computational device with an FPGA-based accelerator. ProcPIPE proposes a near storage Arbiter kernel implementation on FPGA that is capable of scheduling processing tasks, manage memory and establish communication between the host CPU and the near storage FPGA. ProcPIPE allows applications to dynamically compose and offload computation to the storage device; while taking the programming burden from host applications.We evaluate the ProcPIPE framework by integrating it with Tensorflow machine learning framework and training the ResNet50 [2] DL model by offloading the entire image preprocessing steps to the near storage FPGA.

1 Introduction

The idea of moving computation near the storage has been proposed more than two decades ago [19, 12, 26]; but the idea did not gain importance since the CPU performance was scaling really well and the storage was considerably slower than the CPU. There is a change in this pattern with CPU performance saturating and at the same time storage speeds have increased causing, IO bandwidth to be underutilized. This makes the data movement between the storage and the memory a critical performance bottleneck.

The above problem is very evident in the use-case involving Machine Learning/Deep Learning (ML/DL) training models. These models leverage GPUs for training weights but still rely on the CPU to fetch and preprocess the training data. This problem also worsens with the fact that the dataset size for state of the art training models is ever increasing. This stresses the CPU both in terms of 1) fetching the data to CPU memory and 2) preprocessing the data for the training model; leading to stalls in training[24, 20, 15, 31].

This calls for innovation in technology which reduces this data movement and relieve the CPU from becoming a performance bottleneck. Computational Storage Devices (CSDs) are seen as potential solution for reducing the data movement and relieve CPU from becoming a performance bottleneck. Typically, they include a near-storage processing unit (e.g. FPGA, ARM Core, GPU) which enables the device to perform computation on storage data directly without the need to data movement to the CPU memory [11, 1, 25]. These CSDs have a on-device DDR memory called Common Area Memory (CMA) that is used for reading data form the NVMe SSD and then leveraged by the FPGA to perform computation on that data. A major issue with using these Computational Storage Devices has been the lack of availability in flexible programming standards that make it difficult for applications to seamlessly offload computation near storage leveraging these devices. This work is an extension over our team's previous work that proposed a unified Key-Value Store (KVS) [14] framework that assists our computational pipeline framework in seamlessly loading data from storage for the accelerator to operate on. With our work, we claim to have improved the programmability of CSDs by providing the applications with the power to dynamically compose and offload computational pipelines at run time.

We propose the following as part of this work's contribution.

- **Near-Storage Arbiter:** We propose a FPGA Kernel IP implemented on the CSD accelerator that is responsible for offloading compute tasks, managing memory and establish communication with different compute kernel and the host CPU. The main motivation behind having this IP is to transfer the control of managing CSD resources from the host CPU

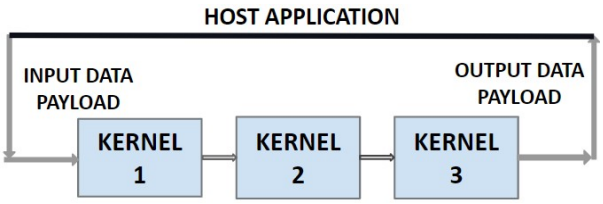

Figure 1: Hardwired Computational Pipeline Example

to the CSD device itself. This ensures that the host application does not have to work with the complexity of managing resources at the device end and at the same time our architecture becomes agnostic to any application leveraging it.

- **Evaluation:** Having recognized ML/DL training models as one of the compelling use-cases for our contribution, we leverage the ResNet50 [2] Deep Learning training model with ImageNet[30] dataset to evaluate our work. The evaluation is done on a Samsung Smart SSD [11] which is equipped with a NVMe SSD tightly coupled with a Xilinx FPGA. We show a considerable improvement in performance along with 65% saving in CPU time. We integrate our implementation directly with TensorFlow in order to have a seamless framework for the use-case in hand.

2 Background and Motivation

Computational Storage Devices include a near-storage processing unit (e.g. FPGA, ARM Core, GPU) which enables the device to perform computation on storage data directly without the need to data movement to the CPU memory [11, 1, 25]. These CSDs have a on-device DDR memory called Common Area Memory (CMA) that is used for reading data form the NVMe SSD and then leveraged by the near storage accelerator to perform computation on that data. We use a Samsung SmartSSD [11] for our work which contains a NVMe SSD interfaced with an FPGA which share a common DDR memory (CMA). Existing frameworks for offloading compute functions on CSDs have shortfalls that motivates the design for our framework.

The legacy existing frameworks (CS APIs [29]) only support hardwired computational pipelines that need to be statically composed before production. Figure 1 demonstrates the basic structure of how a hardwired pipeline looks like. The kernels corresponds to the computational kernels on the FPGA (e.g. JPEG Decoder, Compression Kernel). For simplicity, the term ”kernel” used in this paper inherently corresponds to a computational kernel. The decision on which compute kernel should be executed after which one, is statically determined in the host application code and then run in production. This is highly counter intuitive given that modern applications require to dynamically decide computation orders based on run-time values. This also means that the host application is responsible for handling these compute resources and the memory resources that are needed to assist with the computation. This is the motivation behind proposing a framework that allows applications to be able to compose these computational pipelines dynamically and delegate the power and capability to handle memory, composition of compute function order and scheduling the compute request to the near storage accelerator (e.g. FPGA).

Modern day applications usually run in a cloud based environment. This might lead to use-cases where these CSDs can be used my multiple applications with different requirements in terms of computational needs. For a framework that allows only static composition of pipelines, this use case is not achievable. Figure 2 showcases a case where two applications 1 and 2; simultaneously require services from a common CSD and hence share resources (compute and Memory) along the way. This kind of setup poses two problems with legacy interfaces. First, applications would have to have some kind of inter communication mechanism to be able to manage the shared resources effectively. Second, in case application 1 requires kernel pipeline execution is order (K1 - K2 - K3) while application 2 requires a pipeline to be executed in order (K3 - K2 - K4) as shown in Figure 2. In order for both the applications to be able to achieve their respective computations to be catered, a flexible dynamic design is the need of the hour.

Hence, we take these motivations as basis for our ProcPIPE design which claims the ability to dynamically compose computational pipelines and allows seamless sharing of computational storage device among multiple applications.

ProcPIPE leverages a cross-layered index structure to manage the data stored in the SSD. SSD. The index structure in the KVS is cross-layered i.e., the internal nodes of the index structure is stored in the host DRAM and it is managed by the host CPU (search layer in Figure 1) while the leaf nodes of the index is stored in the SSD and is managed by the near-storage FPGA (data nodes in Figure 1). The goal of the cross-layered approach is to use the CPU as the control plane (e.g., trigger FPGA calls, concurrency control, etc) and offload the compute to the FPGA which is closer to the data (e.g., perform lookups and inserts on the leaf nodes). This design is effective because the it leverages the high internal P2P bandwidth between the FPGA and the SSD i.e., unlike the traditional systems, the leaf node access does not re-

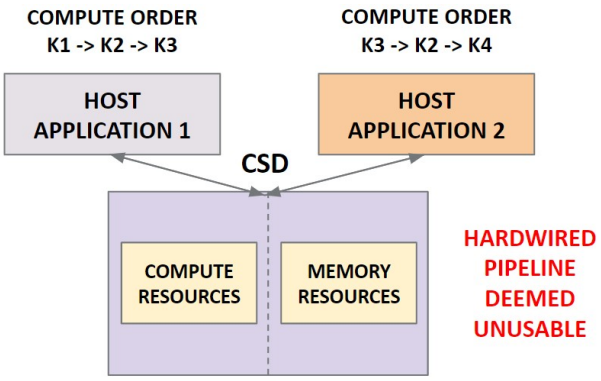

Figure 2: Multiple Applications sharing a CSD

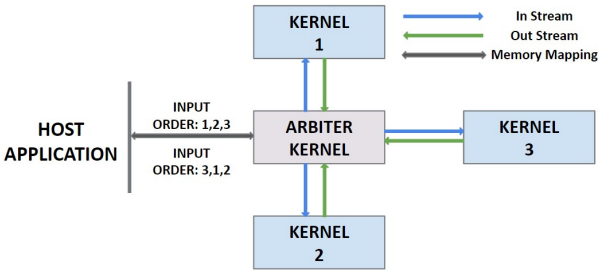

Figure 3: Arbiter Stream connections with Compute Kernels and communication with host.

quire PCIe hops rather it is offloaded to the near-storage FPGA thereby preventing unnecessary data movement between the storage and the CPU memory.

3 ProcPIPE Design

The main idea behind the design is the delegation of power to handle memory management, composing compute functions and scheduling the compute requests, from host CPU to the near storage accelerator. To achieve this, we propose a Arbiter Kernel that is implemented on the CSD accelerator (FPGA in our case). As part of design, we discuss in detail on how Arbiter manages the mentioned three tasks.

Section 3.1 covers the semantics behind how the communication happens between the host and the computational kernels in the FPGA and how arbiter kernel orchestrates this inter-kernel and host communication. Section 3.2 covers the components inside the arbiter's scheduler algorithm and then goes on to discuss the actual arbiter algorithm along with references to a pseudo code and a flowchart. Section 3.3 refers to the logic behind memory management and how our architecture makes it easier for the arbiter kernel to manage memory on the common area memory (CMA) on behalf of host for all computational kernels in the pipeline.

3.1 Inter-Kernel and Host Communication

The Arbiter communicates with each compute kernel via HLS streams which are a optimized implementation of queues (First-In-First-Out). Each instance of a compute kernel is connected to the Arbiter kernel via two streams; one outgoing stream and another incoming stream. Figure 3 showcases this connection. On the other hand, Arbiter kernel is the sole kernel that interfaces and articulates communication between the host and the compute kernel. This communication can be done via stream (given that the underlined CSD supports host-kernel streams) or via memory mapping in the common area memory at the device side.

The inter-kernel communication is achieved via transfer of fixed sized defined packets through streams. We have defined this packet and it comprises of the following fields.

- **Sequence Number (1 Byte):** ID corresponding to the pipeline sequence for the current task/input.
- **Stage Number (1 Byte):** what is the current operation happening within the whole pipeline of multiple compute kernels. Index of operation in the pipeline kernel stage list.
- **Stage IDs (Max Number of Pipeline Stages * 1 Byte):** Sequential list of all the compute kernel IDs in the current pipeline.
- **Sub-Buffer Offset In (1 Byte):** Offset of the input buffer in the CMA that holds the output from the previous compute kernel/ Arbiter in the pipeline.
- **Sub-Buffer Offset Out (1 Byte):** Offset of the output buffer in the CMA that will hold the output from the current compute kernel in the pipeline.
- **Payload (x bytes):** holds the kernel specific details (arguments to kernel, configuration)

3.2 Arbiter Scheduler

The arbiter's main functionality is to be able to efficiently execute the computational pipeline requests from the host in a dynamic manner for which it needs to have a scheduling logic that manages all the compute resources and allocates individual tasks to these resources by utilizing them exhaustively. Before diving into the scheduler logic and algorithm, we need to understand some important components of the scheduler that helps in managing the resources (both compute and memory resources).

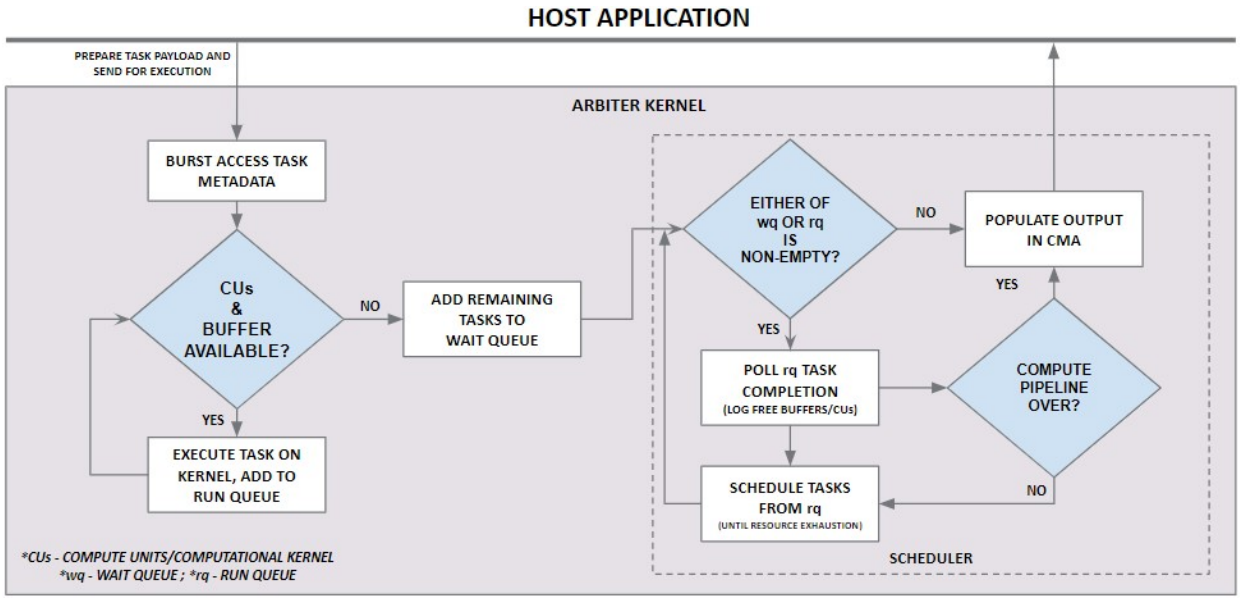

Figure 4: Arbiter Scheduler Flowchart

- **Run Queue:** This queue stores entries (Kernel ID, Next Pipeline Stage, Buffer Offset etc.) for all tasks that are currently running on compute kernels. This queue is periodically polled to check completion of any task that is currently scheduled. We use non-blocking reads from HLS streams to make sure that arbiter does not waste time in polling by stalling the scheduler until the completion of the task it polls.

- **Wait Queue:** This queue stores entries with respect to tasks that are yet to be scheduled and waiting due to lack of current availability of resources (memory buffer or compute kernel instance). The tasks in this queue are scheduled in a FIFO order based on availability of blocking resource. If a polling operation on run queue leads to availability of resources, then the first entry in wait queue that can be executed using this newly available resource, is scheduled to run.

- **Buffer Usage State Array:** As depicted in the previous section, it maintains the usage details of the sub buffers. If a corresponding bit is set for a sub buffer it means it is in use.

- **Compute Kernel State Array:** It maintains the usage mapping of various instances of different compute kernels. Each type of kernel has a unique ID associated with it and each type can have multiple compute instances available which is determined statically. The kernel state array maintains the list of kernel numbers that are not running currently for each type of kernel. This information is leveraged by the scheduler for optimum resource utilization.

These are the most important structures that form the basis for the scheduler algorithm.

Algorithm 1 shows the logic behind the the arbiter kernel IP. The host application sends in metadata information to the arbiter via shared memory in the Common Memory Area(CMA). This information includes the order of execution for each pipeline along with input data location references and any arguments that might be needed for each compute kernel execution. All this information is accessed to arbiter's local memory in burst mode to save CMA memory access time. Initially the arbiter exhausts all available compute or memory resources by scheduling the tasks and adding corresponding entries to the run_queue. Once all available compute resources are exhausted, the remaining tasks in the queue are added to the wait_queue. After this step, we start our scheduler which is a loop that runs until all tasks in the run_queue and wait_queue are finished. Each iteration of the loop, the scheduler polls for completion of running tasks in the running_queue using non-blocking API calls to the stream connection to that specific compute kernel. For all completed tasks, we move the pipeline to next stage and add the entry to wait queue. The scheduler again exhausts the available resources by allocating any pending pipeline requests. After that, we iterate over the wait queue and based on the availability of resources, run tasks from the wait queue and move them to running queue. The process goes on until all available tasks in

the running and wait queue are finished. Arbiter, provides host with the references to the output data for each pipeline and the execution is transferred to the host application. Figure 4 showcases the algorithm in a flowchart.

Algorithm 1 Arbiter Scheduler Algorithm

```
Arbiter Start:
load tasks:

    burst access host data to kernel memory;
    while available CUs > 0 and pipelines > 0 do
        if pipeline[i] != SCHEDULED then
            if pipeline[i].stages[0].CU then
                schedule pipeline[i];
                increment CU-Counter[current CU];
                mark pipeline[i] as SCHEDULED;
                i++;
                pipelines–;
            else
                add-to-wait-queue(pipeline[i]);
            end if
        end if
        available-CUs = calculate-cus();
    end while

start scheduler:

    while run-queue or wait-queue not empty do
        poll-running-queue ();
        exhaust-available-cus:
        for pipelines from host do
            if pipeline[i] != SCHEDULED then
                add-to-wait-queue();
            end if
        end for
        for tasks in wait-queue do
            if CU-available and buffer-available then
                schedule tasks[i];
                add-to-running-queue();
                remove-from-wait-queue();
            end if
        end for

    end while

return:
```

3.3 Arbiter Memory Management

Arbiter is responsible for managing a set of I/O buffers on the Common Memory Area (CMA) used by compute kernels. Host CPU initializes a large pool of buffer during initialization and provides a starting reference pointer to the buffer pool to each kernel. Every kernel uses this start reference to access subsections within this large pool of memory and this information on which sub-buffer is to be accessed at what point in the pipeline execution is completed articulated and communicated by the Arbiter kernel. An interesting thing to notice here is that this avoids unnecessary transfer of input data through streams. Regardless of the size of input that any computation kernel requires; arbiter-kernel communication bandwidth is kept to minimum since only the reference to the input location is shared which is a constant. This idea is similar to passing a pointer in host code implementation rather than transferring the complete input parameter to a function.

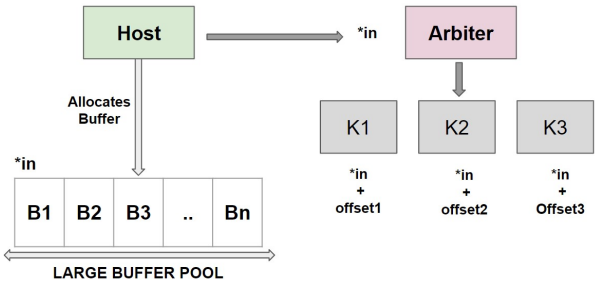

Figure 5: Memory management by Arbiter Kernel with sub-buffers

Figure 5 showcases the division of the large pool of buffer in equal sub-buffers, with arbiter maintaining the usage of each sub-buffer with regards to its relative offset.

4 Implementation

The host side implementation of the architecture is responsible for allocating the big chunk of shared CMA memory on the CSD and generation of instances of required compute kernels. The host code is also interfaced with TensorFlow for our evaluation scenario via a shared library. Our host side interface is compiled as a shared library that provides APIs for the TensorFlow framework to call instead of legacy CPU based APIs. We intercept legacy TensorFlow APIs by leveraging the creation of our own file system in the TensorFlow codebase. Tensorflow framework is very closed in the sense of direct integration with external frameworks. We realized that during training, tensorflow disables a specific mode called “eager execution”; that runs everything in a highly protected environment and blocks all calls to APIs external to the tensorflow namespace. The solution for being able to call external library APIs in tensorflow was to somehow run application with “eager execution” enabled

which can only be done in debug mode. We enabled debug mode for the training to enable us to call our framework's APIs while training and launch ResNet50 preprocessing pipeline through our framework. Host code intefaces witht he CSD using the OpenCL framework. The KVS host code provides an application with standard KVS style APIs which can be leveraged to access data on SSD and compose compute functions dynamically.

Implementation on the device side has the omnipresent Arbiter kernel which has already been discussed in detail. The arbiter has stream connections with each compute kernel that are provided statically before building the xclbin binary (specifically for Xilinx based Smart SSD) to be loaded on the accelerator. Due to static nature of kernel configuration; arbiter is aware of all compute kernels and hence enumerates each kernel/instance with unique IDs that is leverages while scheduling. We also implemented three compute kernels as part of the ResNet50 preprocessing pipeline.

- **JPEG Decoder:** We leveraged the Vitis Library[7] implementation of the JPEG Decoder as the base implementation. The library version was not usable directly to fit our architecture requirement: no intermediate communication with the host until the whole preprocessing pipeline is completed. The library implementation just gives color conversion metadata (YUV metadata) and not a raw RGB image. We had to implement the logic to convert this YUV metadata to a RGB image at the kernel side.
- **Random Crop:** We leveraged the Vitis Library [9] implementation as a base with some tweaks to make sure that the next kernel in pipeline is able to use the output. Also, implemented a random area (Size: 224 x 224) crop logic using the current time as the random seed.
- **Flip:**We leveraged the Vitis Library [10] implementation as a base with some tweaks to make sure that the TensorFlow library accept it in a proper format.

5 The Complete Picture

Figure 7 shows the complete architecture of the framework we aim to achieve. Applications use the ProcPIPE KVS APIs to access the CSD (0.). First, ProcPIPE traverses the search layer using the input key to find the offset of the data node where the key-value pair exists (1.). Then it checks the mirror cache to see if the data node is already cached (2.) if not then the data node is fetched from the SSD to the data cache (3.). If the data node is already cached then ProcPIPE KVS skips step 3

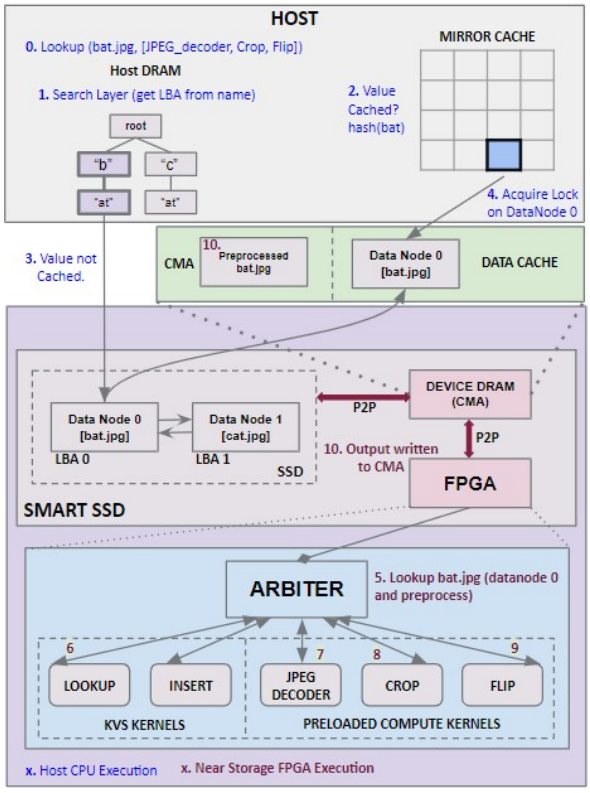

Figure 6: The Complete Architecture: Showcases the integrated architecture when the KVS is fetches input image for the arbiter kernel to execute computation.

and goes on to acquire the lock on the DataNode-0 (4.). ProcPIPE invokes the Arbiter on the FPGA (5.) which then executes the lookup operation on DataNode-0 to retrieve the image (6.). Arbiter performs the preprocessing steps in the application specified order (7., 8., 9.) and saves the output to the CMA (10.).

A portion of the CMA region is reserved for caching (data cache) the frequently accessed data nodes and to know if a particular data node is present in the data cache, KVS uses a mirror cache in the host memory. Mirror cache prevents additional PCIe hops to check the data cache for the data node. Mirror cache just stores the LBA of a data node and its corresponding offset in the CMA.

6 Evaluation

As stated in the introduction section we aim to evaluate our work by completely offloading ResNet50 model training pre-processing on ImageNet dataset from CPU to the near storage computation device. We also analyze timings and CPU utilization for running other independent common kernels like JPEG Decoder and Snappy

Compression. We aim to study the performance benefits if any, on offloading computation to the CSD. We also aim to analyze any the difference in CPU utilization that is achieved by our infrastructure.

6.1 Experimental Setup

As part of our experimental setup we leverage a server equipped with a Samsung SmartSSD which integrates a Samsung NVMe SSD (PCIe Gen3) of 3.84TB capacity with a Xilinx KU15P FPGA [4] . The server has 8-core Intel Xeon Gold 6152 CPU with two NUMA nodes, 1 NVIDIA Tesla V100 GPU and 263GB DRAM memory. SmartSSD provides 4GB of accelerator dedicated DRAM which is mapped to the PCIe bar address (CMA). We use Vitis Accel [6] toolchain to configure and trigger FPGA kernels. The CPU uses OpenCL API calls to communicate with the FPGA. Data transfers between the CPU and the FPGA occurs across the PCIe bus and the same between the FPGA memory (CMA) and the SSD occurs via the P2P interconnect within the SmartSSD.

We implement and evaluate 5 compute kernels (JPEG decoder, Snappy file compression, flip, crop, and resize). cache. In all our evaluations the FPGA(kernel side) concurrency is set to 1 i.e., ProcPIPE maintains only one instance of each FPGA kernels. This is because the KU15P FPGA board is relatively smaller and it runs out of logic cells when creating more than one instance. For instance, 1 instance of each lookup, insert, split, Arbiter, and JPEG decoder kernels altogether consumes about 73% of the available FPGA resources, so increasing the kernel instances results in FPGA build failure. Consequently, we also limit the host side concurrency to 1 thread as there is no benefit to have more host side concurrency when multiple threads will end up waiting for one instance of FPGA kernel to be available.

6.2 Near-storage vs CPU Compute

To study the benefits of moving compute near the storage, we offload the Snappy compression and JPEG decoding to the near-storage FPGA and compare its performance against the CPU execution. For FPGA, we use the Snappy compression and the JPEG decoder in the open-sourced VITIS library [8, 7]. For CPU, we use the Google's Snappy implementation [5] and the JPEG decoder in the OpenCV [3] library.

Figure 7 shows the performance of executing Snappy compression and JPEG decoder on FPGA and CPU for different file sizes. Both FPGA and CPU uses the same KVS framework to insert/lookup the data in the SSD. Essentially, this means that the performance delta between the FPGA and CPU comes only from performing snappy compression/JPEG decoding on the CPU and FPGA respectively.

Size (KB)	Snappy Compression		JPEG Decoder	
	CPU (ms)	FPGA (ms)	CPU (ms)	FPGA (ms)
64	0.4	0.65	10	27
128	0.7	0.7	37	30
256	1.2	0.8	40	31
512	2.8	1.4	55	36
1024	4	2.3	96	45

Table 1: Execution time for CPU- and FPGA-side Snappy compression and JPEG decoder for different input sizes.

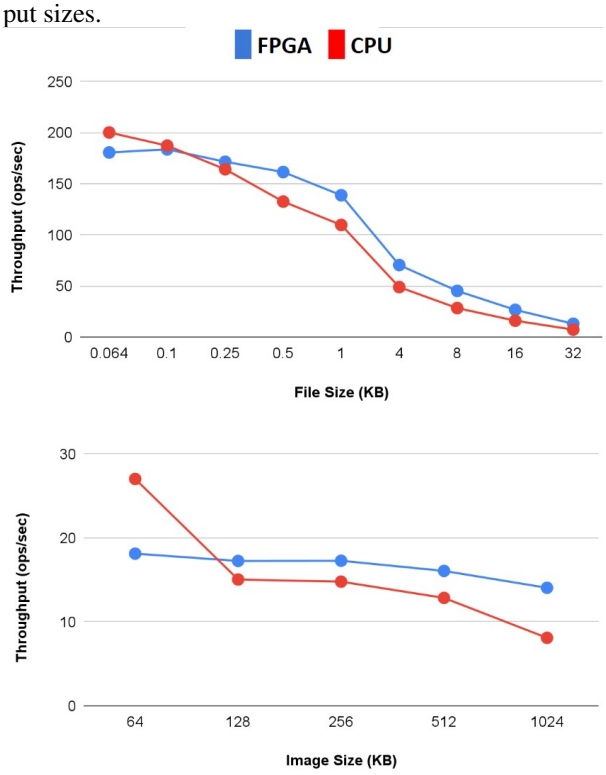

Figure 7: Performance comparison of Snappy compression and JPEG decoder executed on the CPU and the FPGA.

FPGA performs up to 70% faster than CPU for both Snappy compression and JPEG decoder. The benefit of offloading compute to the storage is more pronounced for larger file sizes. Unlike the CPU execution, the large files does not have to be moved from storage to the CPU memory for the offloaded FPGA execution. Further, FPGA is more suited for performing such compute intensive decoding and compression operations. This is corroborated by Table 1 which shows the execution time of Snappy compression and JPEG decoding on CPU and FPGA. FPGA takes up to 2× less time as the input size increases from 64KB to 1MB.

Figure 8 shows the average execution times for dif-

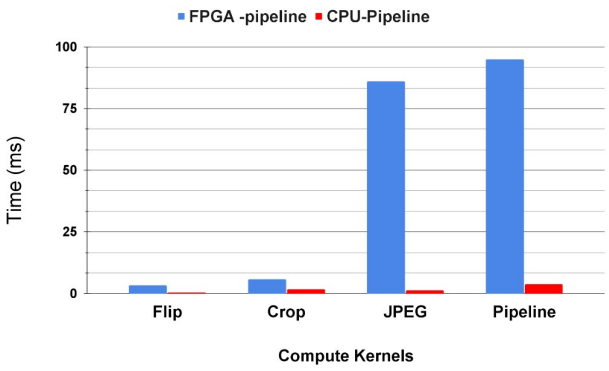

Figure 8: Preprocessing Kernels Execution Times averaged over 1 Million Images on CPU and FPGA

ferent preprocessing kernels on the CPU and the FPGA. JPEG kernel is responsible for about 85% of the total pipeline execution time on the FPGA, which means that the JPEG kernel is the main bottleneck in the whole pipeline execution. The results show that we are no where near to the CPU latency for the whole pipeline and in order to match the CPU latency we need 7 instances of JPEG decoder, 2-3 instances of flip and crop each. This is something that we could not test due to lack of available powerful FPGA on the SmartSSD that supports enough resources to accommodate multiple kernel instances. We leave this as a future work to match CPU latency times which is supported by our current design.

Also FPGA consumes nearly 65% less CPU time than CPU. This is because, the role of CPU in the FPGA based setting is just to orchestrate the data movement between the storage and the CMA and to trigger the compute function calls on the FPGA.

6.3 Deep Learning Preprocessing Pipeline

We evaluate ProcPipe by offloading the image preprocessing steps in the ResNet50 DL model training to the CSD. We use Tensorflow [19] framework as it allows to add custom filesystem support [1]. The ResNet50 model includes three preprocessing steps wiz., JPEG decoding, crop, and flip before feeding the images to the GPU for training. Usually, the images are retrieved from the storage and preprocessed using the CPU. In ProcPIPE, the images are retrieved, preprocessed at the storage using the near-storage FPGA, and the preprocessed image can be directly fed to the GPU. Generally, Tensorflow allows input images to be fetched in a batch (e.g. 128 images) and creates a pipeline of preprocessed images to be fed to the GPU. Tensorflow uses multiple CPU cores to enable parallel preprocessing. Since ProcPIPE can not create more than one instance of FPGA kernels we set the

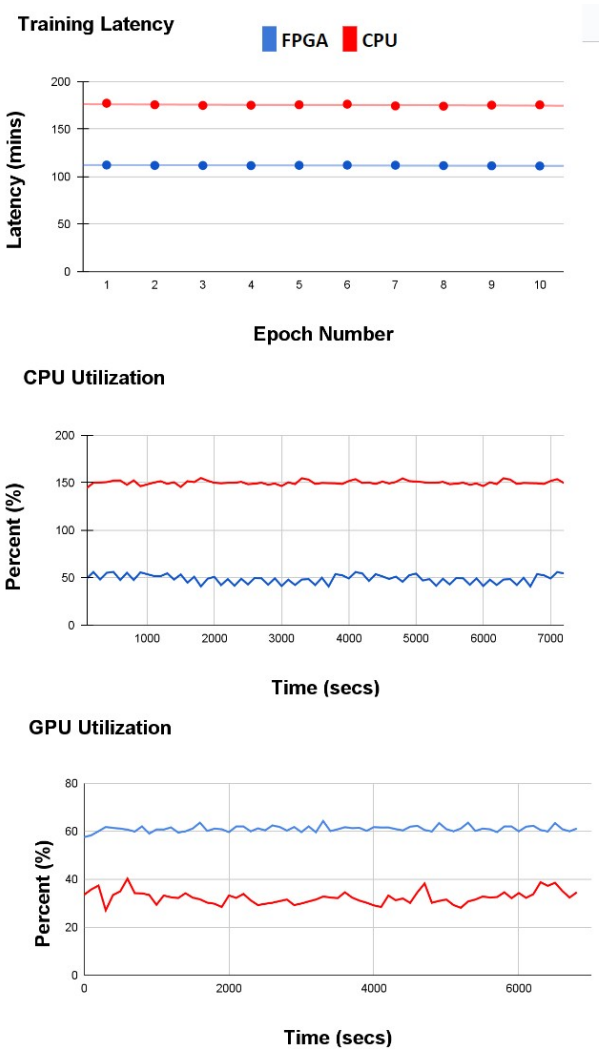

Figure 9: Performance and resource utilization comparison for ResNet50 DL model training using ImageNet dataset when image preprocessing done on the FPGA (ProcPIPE) and the CPU.

batch size to 1 and force the Tensorflow to use only one CPU core for preprocessing for a fair comparison.

Integrating ProcPIPE with Tensorflow: Tensorflow provides a virtual wrapper class for filesystem and the developers can overload the virtual class functions to execute their own filesystem logic during the Tensorflow runtime. To this end, we overloaded the Tensorflow filesystem layer to call KVS APIs. For instance, read and write filesystem function in the Tensorflow will call KVS's lookup and insert functions respectively. Essentially, when the Tensorflow calls image fetch KVS would lookup the particular image, perform the preprocessing steps on the SSD, and returns the preprocessed image

which the Tensorflow directly feeds to the GPU. Note that we load the full ImageNet dataset to the KVS before the training begins.

Figure 9 shows the time take per epoch for ResNet50 training when the preprocessing is done using ProcPIPE and compares it against the CPU side preprocessing. ProcPIPE consumes about 60 minutes less than the CPU per epoch and for the whole training (10 epochs) ProcPIPE is nearly 36% faster than the CPU. Data movement is significantly reduced in the ProcPIPE; the ResNet50 model requires the size of a preprocessed image to be exactly 256*256 regardless of its original size. This means, in ProcPIPE only the final 256*256 image moves from the storage to CPU memory at all times. Whereas in the traditional CPU based training the unprocessed large JPEG images are constantly moved from storage to memory. In addition, ProcPIPE also consumes 100% less CPU time as the preprocessing is offloaded to the FPGA and consequently it reduces the GPU idle time by up to 25% during the training time as illustrated in Figure 9.

7 Related Work and Possible Future Work

Prior works on CSDs focused mainly on ad-hoc compute offloads to the CSDs such as SQL processing [17, 16, 32], graph analytics [21, 18, 23],filesystem [34, 22, 13], and few other works focused on addressing the programmability of CSDs [28, 27, 33, 35].MetalFS [28] and Insider [27] propose CSD programming framework using UNIX pipes and virtual filesystem respectively. Both the works focus on offloading compute to the CSD but suffers from high performance overhead. Whereas, ProcPIPE provides a better programmability and achieves high-performance with its cross-layered index and caching architecture.

Although we designed ProcPIPE with host and kernel side scalability in mind, we were unable to test the design due to current hardware limitations. A more powerful hardware will enable us to validate the scalability of ProcPIPE design.For instance, the next-generation of SmartSSD [8] is expected to have a dedicated embedded ARM Soc that runs petalinux in addition to a more powerful Versal series FPGA [14]. We believe this would allow ProcPIPE to have multiple instance of FPGA kernels which is key to test the scalability. Further, advancements in the hardware will enable us to test ProcPIPE for more complex applications such near-storage erasure coding. Another interesting direction is to test ProcPIPE in distributed setting where we run applications on multiple computational storage devices. New CSD hardware such as the one from Scaleflux's CSD 3000 [3] supports out-of-box device virtualization using SR-IOV will be an interesting platform to evaluate ProcPIPE.

SNIA has recently released a standard programming model for CSDs called the CS APIs [2]. Rewriting ProcPIPE using CS APIs would make it portable across different CSD architectures. Although ProcPIPE is evaluated on SmartSSD, we designed it to be used across different CSD architectures. But it is unclear at this point whether ProcPIPE's design would work out-of-box on other architectures; it would be interesting direction to make ProcPIPE portable across different CSD architectures as our end goal is to make ProcPIPE a de-facto programming framework for any applications that needs to adopt CSD in its storage stack.

8 Conclusion

We propose ProcPIPE - a prototype framework that enables applications to offload processing pipelines to computational storage by providing an interface that manages near storage accelerator resources at device side. ProcPIPE proposes a centralized Arbiter; a near storage IP that enables applications to compose compute pipeline on-fly. We evaluated ProcPIPE on a real CSD drive (SmartSSD) and our evaluation showed that ProcPIPE performs up to 75% better, saves up to 65% less CPU time, and significantly reduces the data movement as compared to the traditional CPU only compute.

Acknowledgement

I want to thank Dr. Changwoo Min for consistently providing me with guidance and motivation to work on this research project at such an intricate level. This project has enabled me to deep dive in understanding the basis of how to build a system architecture, work on different stacks from application side to device level and the importance of basic OS understanding in building systems architectures. A special thanks to Dr. Tam Chantem and Dr. Lynn Abbott for taking their crucial time and serving on my project committee.

References

[1] CSD 3000. https://scaleflux.com/products/csd-3000/.

[2] Deep Residual Learning for Image Recognition. https://arxiv.org/abs/1512.03385.

[3] Image file reading and writing. https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html.

[4] KINTEX UltraScale+. https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale-plus.html.

[5] Snappy, a fast compressor/decompressor. https://github.com/google/snappy.

[6] Vitis unified software platform documentation: Application 238 acceleration development.

[7] Xilinx JPEG Decoder. https://github.com/Xilinx/Vitis_Libraries/tree/main/codec/L2/demos/jpegDec.

[8] Xilinx Snappy Compression and Decompression. https://github.com/Xilinx/Vitis_Libraries/tree/main/data_compression/L2/demos/snappy.

[9] Xilinx Vision: Crop. https://github.com/Xilinx/Vitis_Libraries/tree/main/vision/L2/examples/crop.

[10] Xilinx Vision: Flip. https://github.com/Xilinx/Vitis_Libraries/tree/main/vision/L2/examples/flip.

[11] Product Brief Samsung SmartSSD Computational Storage Drive, 2018. https://semiconductor.samsung.com/resources/brochure/Samsung/SmartSSD/Computational/Storage/Drive.pdf.

[12] ACHARYA, A., UYSAL, M., AND SALTZ, J. H. Active disks: Programming model, algorithms and evaluation. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998* (1998), ACM Press, pp. 81–91.

[13] BIJLANI, A., AND RAMACHANDRAN, U. Extension framework for file systems in user space.

[14] BIKONDA, N. S. Retina: Cross-layered key-value store using computational storage.

[15] CHENG, Y., LI, D., GUO, Z., JIANG, B., LIN, J., FAN, X., GENG, J., YU, X., BAI, W., QU, L., SHU, R., CHENG, P., XIONG, Y., AND WU, J. DLBooster: Boosting End-to-End Deep Learning Workflows with Offloading Data Preprocessing Pipelines. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019* (2019), ACM, pp. 88:1–88:11.

[16] DO, J., KEE, Y.-S., PATEL, J. M., PARK, C., PARK, K., AND DEWITT, D. J. Query processing on smart ssds: Opportunities and challenges.

[17] JO, I., BAE, D.-H., YOON, A. S., KANG, J.-U., CHO, S., LEE, D. D. G., AND JEONG, J. Yoursql: A high-performance database system leveraging in-storage computing.

[18] JUN, S.-W., WRIGHT, A., ZHANG, S., XU, S., AND ARVIND. Grafboost: Using accelerated flash storage for external graph analytics.

[19] KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. A case for intelligent disks (idisks).

[20] KUMAR, A. V., AND SIVATHANU, M. Quiver: An informed storage cache for deep learning.

[21] LEE, J., KIM, H., YOO, S., CHOI, K., HOFSTEE, H. P., NAM, G.-J., NUTTER, M. R., AND JAMSEK, D. Extrav: Boosting graph processing near storage with a coherent accelerator.

[22] LU, Y., SHU, J., AND ZHENG, W. Extending the lifetime of flash-based storage through reducing write amplification from file systems.

[23] MATAM, K. K., KOO, G., ZHA, H., TSENG, H.-W., AND ANNAVARAM, M. Graphssd: Graph semantics aware ssd.

[24] MOHAN, J., PHANISHAYEE, A., RANIWALA, A., AND CHIDAMBARAM, V. Analyzing and Mitigating Data Stalls in DNN Training. *arxiv preprint arxiv:2007.06775* (2020).

[25] NGD SYSTEMS. NVMe Computational Storage, 2021.

[26] RIEDEL, E., GIBSON, G. A., AND FALOUTSOS, C. Active storage for large-scale data mining and multimedia.

[27] RUAN, Z., HE, T., AND CONG, J. INSIDER: designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (2019), D. Malkhi and D. Tsafrir, Eds., USENIX Association, pp. 379–394.

[28] SCHMID, R., PLAUTH, M., WENZEL, L., EBERHARDT, F., AND POLZE, A. Accessible near-storage computing with fpgas. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (2020), ACM, pp. 28:1–28:12.

[29] SNIA. Computational Storage APIs, 2021.

[30] STANFORD VISION LAB. ImageNet dataset, 2020.

[31] TREDAK, P., AND LAYTON, S. S8906: Fast Data Pipelines for Deep Learning Training, 2018. https://on-demand.gputechconf.com/gtc/2018/presentation/s8906-fast-data-pipelines-for-deep-learning-training.pdf.

[32] WOODS, L., ISTVÁN, Z., AND ALONSO, G. Ibex: An intelligent storage engine with support for advanced sql offloading.

[33] YANG, Z., LU, Y., LIAO, X., CHEN, Y., LI, J., HE, S., AND SHU, J. Lamda-IO: A unified IO stack for computational storage.

[34] YANG, Z., LU, Y., XU, E., AND SHU, J. Coinpurse: A device-assisted file system with dual interfaces.

[35] ZHANG, J., REN, Y., AND KANNAN, S. FusionFS: Fusing I/O operations using CISCOps in firmware file systems.