Computational Storage For Big Data Analytics

Balavinayagam Samynathan
Bigstream Inc.
bala@bigstream.co

Keith Chapman
Bigstream Inc.
keith@bigstream.co

Mehdi Nik
Bigstream Inc.
mehdi@bigstream.co

Behnam Robatmili
Bigstream Inc.
behnam@bigstream.co

Shahrzad Mirkhani
Bigstream Inc.
shahrzad@bigstream.co

Maysam Lavasani
Bigstream Inc.
maysam@bigstream.co

ABSTRACT

This paper discusses the advantages and benefits of computation near storage or computational storage in the context of Big Data analytics. SmartSSD computational storage platform from Samsung provides an opportunity for fast data transfers from storage to FPGAs, which can facilitate acceleration of big data processing. In this paper, we discuss our full stack acceleration approach, with zero application code change, for modern open source Big Data environments on accelerators like FPGAs and GPUs, with focus on Apache Spark as our Big Data environment and FPGAs as acceleration devices. We discuss changes that were made to our traditional software and hardware stack in order to incorporate computational storage platforms. The paper describes cross-stack optimizations necessary to achieve high throughput and low latency for SQL query processing for SmartSSDs. Finally, we showcase our results on TPC-DS benchmarks, which are state-of-the-art SQL benchmarks designed for Big Data analytic platforms. The results show up to 6x end to end query run-time speedup for scan-heavy TPC-DS queries, compared to query run-time for the same queries executed by vanilla Spark. The average speedup across all TPC-DS queries is 4x.

1. INTRODUCTION

As Moore's law is slowing down, traditional CPU and transistor scaling no longer translates to performance scaling for data centers and cloud systems. As a solution to this problem, the industry has come up with a number of hardware accelerators to speedup processing in different domains such as machine learning, data analytics and graph processing. A clear indicator of this trend is the fact that accelerators such as FPGAs, GPUs and TPUs are now available in cloud and data centers [7].

Unfortunately, there is a semantic gap that exists between the low-level programming model of the accelerators and the high-level analytics languages used by data scientists and engineers. Data scientists and engineers are not able to easily program and use these accelerators as they need to program using hardware description languages or low-level programming languages such as CUDA/OpenCL. Even if the vendors provide high level software libraries and APIs, the cost of changing analytics code is significant. Also, library calls cannot take advantage of the run-time information associated with the dynamic nature of the workloads as well as the dynamic nature of underlying resources. At Bigstream, we develop hardware-software solutions for enterprise and cloud-based data centers to fill this gap. Our software platform enables accelerated computing for Big Data analytics without requiring any code change. This is especially important for cleansing, managing, and analyzing huge volumes of data that is required for AI/ML solutions. Today's clusters are typically managed using distributed software frameworks on x86-based hardware servers, which include both open source (e.g., Spark, Hive, etc.) [22] as well as closed source (e.g., Redshift, Microsoft SQL Server, Snowflake). In this work we focus on SmartSSD, which is a computational storage platform introduced by Samsung [5] as a hardware accelerator near storage. We utilize our technology on SmartSSDs to process parts of analytic workloads in the in-storage peered device (in this case, FPGA) and show significant performance and throughput improvements.

The rest of the paper is organized as follows: the next section discusses prior work and background. Section 3 discusses SmartSSD components and operational mechanisms. Section 4 describes layers of our software stack and how it enables acceleration without any user code change for analytic applications. Sections 5 and 6 discuss the hardware architecture we used and our hardware/software interfaces respectively. Section 7 and Section 8 present our design and results on row-based data format for TPC-DS [6] benchmarks, respectively.

2. RELATED WORK

The concept of moving processing capability near storage has been explored before in the form of intelligent disks [16] and active disk [18]. However implementation of such systems was not practical due to power and cost issues. Commercially this concept was first explored by adding FPGA near disk in IBM's Netezza product [13]. With regard to SSDs, computation near storage is an emerging technology that can potentially become essential in modern data-

10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19), August 26, 2019, Los Angeles, California, CA, USA.

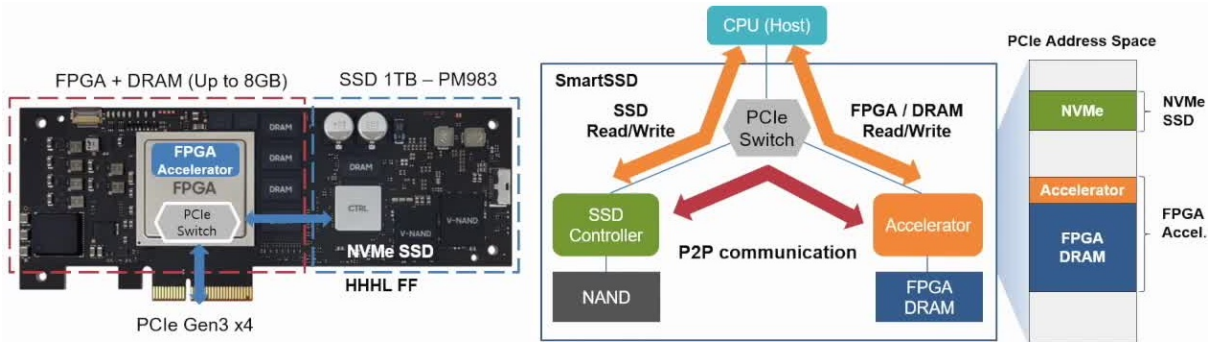

Figure 1: SmartSSD System Architecture[1]

[1]Image obtained with permission from https://youtu.be/OHWzWv4gCTs?t=639

center infrastructures so much so that the Storage Networking Industry Association (SNIA) has created a Computational Storage Technical Work Group to focus on standardizing this technology [1]. The idea of query processing on SmartSSD has been researched [11] where queries were run on Microsoft SQL Server on SmartSSD which had an embedded ARM processor. Studies like [15] and [21] also discuss using the same SmartSSD in the context of applications such as log analysis and document search while our work is focused on SQL Big Data analytics. The newer generation of SmartSSDs has a FPGA instead of an embedded processor. This could lead to better acceleration as custom designs can perform better than general purpose processors. Our work utilizes the FPGA in SmartSSD without application code change to provide speedup. Additionally, our framework is targeted towards Big Data environments, where clusters can extend to hundreds of nodes and data size can range from petabytes to terabytes. Another distinguishing architectural feature between previous generation SmartSSD and the one used in this work is that they were based on SATA or SAS attached SSD while we work with a PCIe attached card which provides higher SSD throughput. Our software interfaces use OpenCL API to access SmartSSD from host application when compared to the OPEN, GET and CLOSE requests [11] in a session based protocol and this leads to a better event model for our applications as explained in 7. There has also been other work that explored part of query execution near SSD [14], where the query planner was modified to offload filter functions to ARM processor in SSD controller. This would work best in the cases where data is plain-encoded since the ARM processor would find parsing a compressed file computationally intensive. Our implementation does scan and filtering for compressed data as well since the FPGA can do decompression, decoding, parsing, and then apply filter functions.

3. SMARTSSD ARCHITECTURE

With the ever-growing amount of data that needs to be processed by data centers, it is critical that servers be able to consume data effectively from storage devices. In this paper, we define computational storage as the ability to do computation or data processing at the storage device level or near storage, before data is moved to host memory. This paper exclusively discusses computational storage with respect to the SmartSSD platform from Samsung [5]. The advantages of having computational storage has been discussed in prior work [17, 16, 10, 11]. The primary advantage is reducing the computational complexity and volume of data reaching CPU's host memory and scaling processing capability to maximize storage bandwidth.

The components of the SmartSSD platform are shown in Figure 1. The SmartSSD platform contains a Xilinx FPGA and NAND Flash Arrays with 1TB capacity on the same board which has the form factor of a PCIe add-in card. The FPGA used in SmartSSD is a Zynq SoC and has ARM cores. The PCIe in this card is Gen3x4-lane with a theoretical maximum throughput of 4 GB/s. The board contains 8 GB DRAM, which acts as an intermediate buffer when data is transferred from SSD to FPGA and from FPGA to host. There is a three-way PCIe switch present in the FPGA, which can transfer data from/to SSD and FPGA and from/to SSD to host memory and from/to FPGA to host memory. SmartSSD supports two modes of data transfer: normal mode and Peer-to-Peer (P2P) mode. When operating in the normal mode, read/write is issued by the host and data is transferred between SSD and host memory through PCIe. A normal read corresponds to how data is read by applications of a normal SSD. It is important to note that under normal reads too, data is transferred through the PCIe switch in the FPGA onto the PCIe Bus. The second mode of operation is the Peer-to-peer (P2P) mode, in which data is transferred from SSD to FPGA DRAM for processing by the local peered devices. For the P2P mode, there is a reserved memory in the FPGA DRAM which we call Global Accessible Memory (GAM) and is accessible by SSD, FPGA, and host. This GAM is exposed to the host PCIe address space. From the software perspective, all access to accelerators and computing on P2P data in the SmartSSD goes through the Xilinx OpenCL framework [12]. An application that would like to use the P2P mode from host software needs to allocate memory in the GAM using OpenCL libraries and extensions provided by Xilinx (*clCreateBuffer* with P2P flag in this case). The allocated memory can then be used by the SSD for direct read/write access into FPGA GAM. This constitutes P2P mode of operation for the SmartSSD.

4. A COMPUTATIONAL STACK FOR BIG DATA ACCELERATION

This section demonstrates layers of our software stack which facilitate the process of using accelerators in Big Data applications seamlessly. Our technology stack consists of

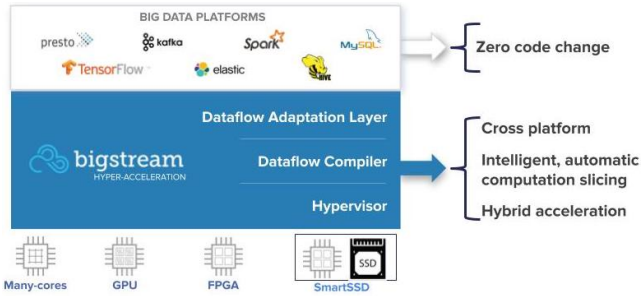

Figure 2: Hyper-acceleration technology

three important layers as illustrated in Figure 2: data-flow adaptation layer, data-flow compiler, and hypervisor. The following section describes these layers in more detail.

1. The **data-flow adaptation layer** converts internal data-flow format of Big Data frameworks like Apache Spark, Hive, Presto, or Tensorflow into Bigstream's canonical data-flow format. In this paper, we focus on Apache Spark [22] which is one of several Big Data application frameworks popular for its higher performance due to in-memory processing. Bigstream canonical data-flow format includes several computation- and communication-related operators that cover analytics and machine learning operations. The implementation of these operators is not tied to any specific platform. (The canonical data-flow format, resulted from this layer, can be considered as an intermediate representation for the next layers.

2. **Data-flow compiler** is responsible for compiling the canonical data-flow representation, which is generated for each application, and mapping to pre-compiled accelerator templates that slice the computation between different heterogeneous devices. The features of this layer are illustrated in Figure 2.

3. **Hypervisor**[1] is a high performance C++ library that interacts with heterogeneous devices like FPGAs, GPUs, multi-core CPUs, and SmartSSDs, some of which may exist on the cluster nodes that the users are running their application on. The pre-compiled accelerator templates generated by our data-flow compiler layer, along with the application binary, are broadcast to all nodes of the cluster. The modified Spark platform, at run time, executes the accelerated version of the task. The accelerated task interacts with the hypervisor to execute the pre-compiled templates on the accelerator like GPU or FPGA. The hypervisor layer chooses templates that can run on FPGA, GPU or CPU based on cost-functions for operators of that stage. More details are discussed in the following sub-sections.

To make the process of Spark acceleration more clear, the rest of this section first discusses how Spark executes user applications on clusters. The section then explains how Spark execution is seamlessly accelerated during run-time on

[1]Please note that this is our acceleration stack hypervisor and is different from the hypervisor used in VMs [8]. Our computational stack can work across all containers and hypervisor environments.

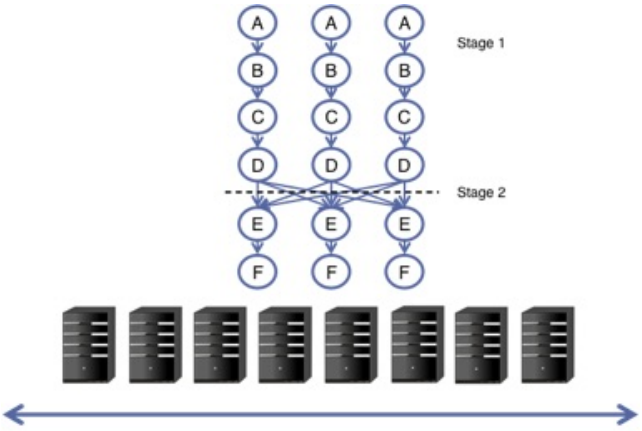

Figure 3: Spark Execution Illustrated With 3 Tasks and 2 Stages

different types of accelerator devices across the three layers described here.

4.1 Spark Execution Overview

The scope of this article is limited to Spark SQL. In Apache Spark, a user application is executed by a *driver* program and one or more *executors* [22]. The driver program takes user code and dispatches it to executors across multiple *worker nodes*. First, the driver breaks down the user code into a DAG (Directed Acyclic Graph) of *stages*. In this DAG, the operators that have linear dependency (such as file scan, filter, or map) are grouped in one stage. However, if the operators have more complex dependencies (such as groupBy or join), they will end up in different stages. When running a SQL code, Spark SQL compiler, named Catalyst [9], converts the code into an optimized *query plan*. A query plan describes SQL operators and their dependencies in the query. Eventually, Catalyst generates a DAG and determines which nodes are responsible to run which stages(s) of the DAG.

Spark divides the data for each stage into multiple *data partitions* across the cluster. When running a stage, Spark executors run the operators in that stage as a set of *tasks*. Each task is associated with an independent data partition. This enables the tasks to be executed in parallel. As an example, Figure 3 shows an application with two stages. The first stage has four operators (A, B, C, and D). At run-time, the stages are executed in the order of DAG dependencies. In this case, Stage 1 gets executed before Stage 2 and each stage runs three tasks on different partitions of data. For a large number of data partitions, a stage might consist of thousands of tasks. When all tasks in stage 1 finish, the results are re-distributed across executors. This re-distribution is known as *shuffling* in Spark terminology. Shuffling acts as a synchronization barrier for all executors and upon completion of shuffling, executors move to Stage 2 of the DAG.

4.2 Spark Acceleration

Figure 4 shows how Spark queries get accelerated using our hyper-acceleration layers. Two of the layers (data-flow compiler and hypervisor), described in Section 4.1, are

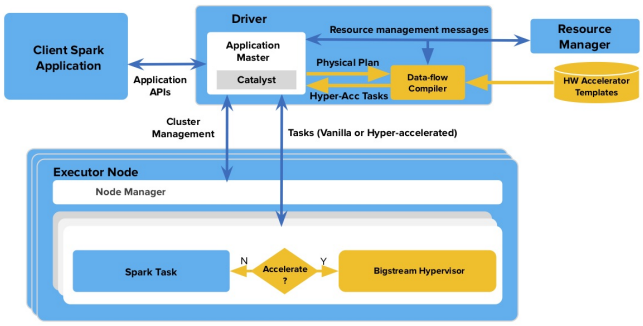

Figure 4: Apache Spark (with and without acceleration)

shown in yellow boxes in this figure. The *physical plan* arrow signifies the data-flow adaptation layer.

In Figure 4, the *HW accelerator template database* consists of several templates which are associated with a linear set of operators referred to as Linear Stage Trace (LST). In other words, LST can be considered as a subset of operators that need to be executed sequentially within a stage. In case of FPGAs, a template includes a partially re-configurable bit file, which is pre-synthesized and optimized to accelerate each operator in the LST, along with meta-data for necessary run-time configurations. In case of GPUs, a template includes CUDA/OpenCL binaries that implement the operators in the LST. In case of CPUs, our templates consist of native (C++) code which links to our optimized libraries. There is a cost function that determines which LSTs need to be implemented on accelerators. This cost function is tuned by offline profiling of different operators in production ETL pipelines and SQL pipelines and it is tuned to be more effective by adding new profiling data over time.

The adaptation layer interacts with Spark query compiler and converts the output of Spark query compiler, known as *physical plan* to a canonical data-flow intermediate representation. This canonical representation is given to our data-flow compiler, and based on accelerator template availability, it generates accelerator code which can communicate with available accelerators (e.g., FPGA and GPU) and will be running on each executor. If data-flow compiler decides not to accelerate a stage, that stage will be executed through the original Spark execution path.

Figure 5 shows more details on the steps taken by data-flow compiler:

- Slicing and mapping: Based on LSTs in each stage of the query plan and matching accelerator templates in template database, our compiler maps each part of the query plan to different computational resources (e.g., FPGA, GPU, CPU, etc.). The cost function for each operator helps choose the best accelerator match for each LST in a physical plan.
- Control plane generation: The part of the code generated for transitioning the control flow between LSTs and the Spark task.
- Data plane generation: The part of the code generated for moving the data between LSTs and the Spark task.

As a result of data-flow compiler, an accelerator code is generated and it is ready to be executed by hypervisor layer.

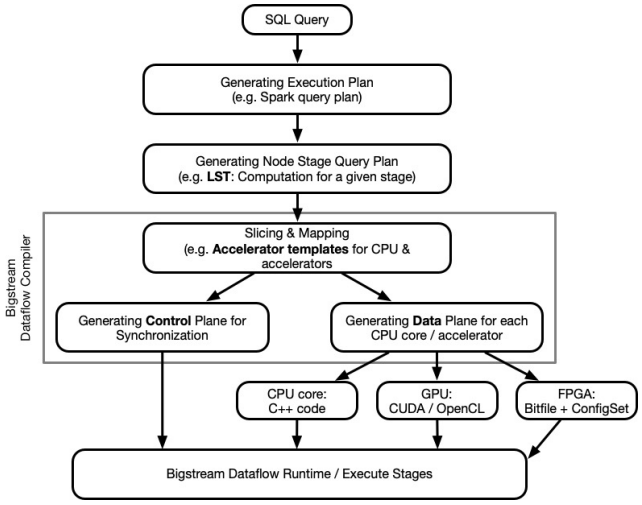

Figure 5: Bigstream data-flow compiler

At high-level, hypervisor is responsible for picking LSTs in a stage, loading the right accelerator code for it (if exists), preparing the environment for task execution if necessary (e.g., FPGA partial programming), and executing the code on the accelerator device.

Now we will describe run-time components in hypervisor for the same example shown in Figure 3. Figure 6 illustrates the acceleration of the first stage of this sample query. As shown in Figure 3, the first stage includes operators A, B, C and D. There can be multiple LSTs associated with a stage since there are multiple combinations of subsets available for a linear set of operators. For example, if a stage has three operators X, Y and Z that need to be executed sequentially, then any possible sequential combination of LSTs can provide the optimal performance. For example, based on data size, operator type, any combination of $LST_{\{X,Y\}}$ or $LST_{\{X\}}$ or $LST_{\{X,Y,Z\}}$ can provide best performance in a Stage. In our example, Stage 1 can be represented by ($LST_{\{A,B\}}$, $LST_{\{C,D\}}$), or ($LST_{\{A\}}$, $LST_{\{B,C,D\}}$), or ($LST_{\{A\}}$, $LST_{\{B\}}$, $LST_{\{C\}}$, $LST_{\{D\}}$), etc.

Assume our data-flow compiler finds FPGA accelerators for $LST_{\{A\}}$, $LST_{\{C\}}$, and $LST_{\{D\}}$. It also finds a native accelerator for $LST_{\{B\}}$. This means that operators A, C, and D will be executed on FPGA, while operator B will be executed on a CPU by native code. In Figure 6, a list and order of LSTs that are going to be executed at each time is determined by *Execute Stage Task* step. In our example, it picks $LST_{\{A\}}$ first. Since this LST can be executed on FPGA, the hypervisor programs the bit file from the template and it configures the FPGA with specific parameters from the template. Then it executes $LST_{\{A\}}$ on the FPGA. According to data-flow compiler, the next LST would be $LST_{\{B\}}$. Since this LST has a native template, it will be executed on the CPU with our native libraries. $LST_{\{C\}}$ is the next to be picked. Based on operator functionality, the current programmed template (in this case, template for $LST_{\{A\}}$) cannot be reused for $LST_{\{C\}}$. Therefore re-programming the bit file and parameter configuration need to be done for this LST. The next in the list is $LST_{\{D\}}$. Let's assume that the operator functionality for current programmed template (i.e., $LST_{\{C\}}$) can be re-used for $LST_{\{D\}}$. For example, both

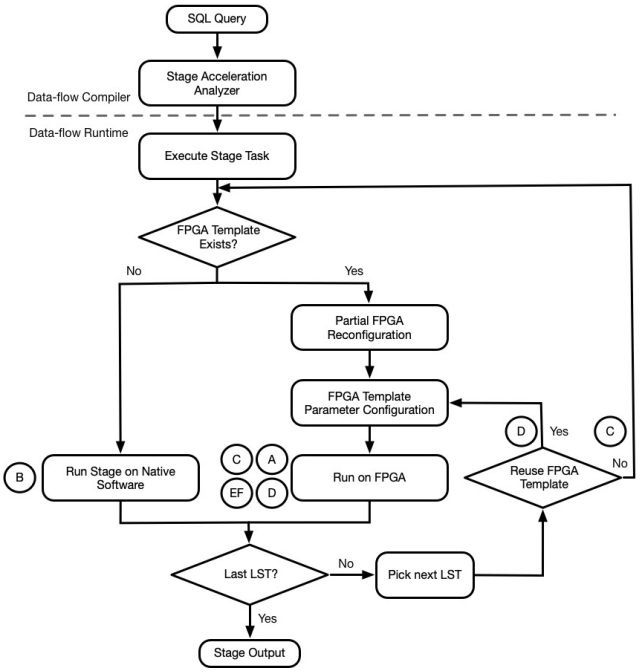

Figure 6: Bigstream hypervisor run-time flow

$LST_{\{C\}}$ and $LST_{\{D\}}$ could be filter operators and therefore we could reuse the template. Therefore, hypervisor skips bit file programming step and only performs parameter configuration and runs $LST_{\{D\}}$ on FPGA. Since $LST_{\{D\}}$ is the last one, the result will be considered as Stage 1 result. The same flow is repeated for Stage 2 of Figure 4. For this stage, the data-flow compiler finds a template for $LST_{\{E,F\}}$, which can be executed on the FPGA, as the most cost-optimized option. The hypervisor takes $LST_{\{E,F\}}$ as the first LST on the list. Then it checks if the current programmed bit file can be re-used by this LST. In this example, it cannot be re-used and the bit file for $LST_{\{E,F\}}$ is programmed into the FPGA and the parameters are configured for it. Then the hypervisor executes operators E and then F on the FPGA. Since this LST is the last LST in the list of Stage 2 LSTs, the results will be considered as the results for Stage 2. All of the above hypervisor steps, discussed in the above example, are executed as a Spark task running on a Spark executor.

This section focused on the flow of accelerated query execution in Spark. In the next section, we will focus more on the architecture of our FPGA accelerators.

5. FPGA SOLUTION ARCHITECTURE

As shown in Figure 7, the hardware components of FPGA can be split into three sections logically: shell, shim, and core. The shell region has fixed IP components that interact outside of FPGA, such as PCIe controller, DMA controller, DRAM controller, Ethernet controller, etc. It is typically provided by FPGA vendor. The next layer in question is our proprietary shim layer, which converts external bus interfaces to compatible interfaces for our core. An example of this is converting from memory mapped interface to streaming interface which is the most common interface in our templates. The shim layer also collects error and performance metrics from the core.

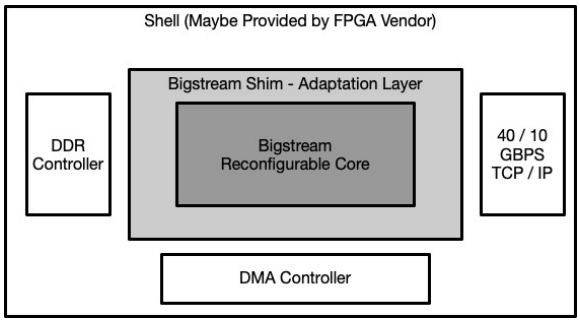

Figure 7: Bigstream Hardware Abstraction Layers

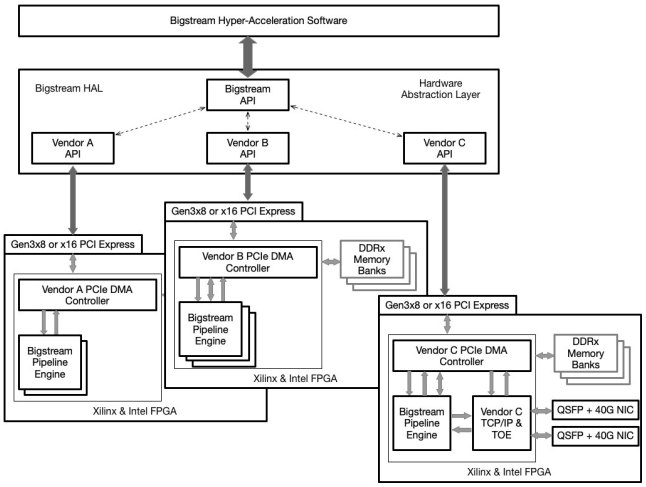

Figure 8: Bigstream hardware - software interface

The partially re-configurable component of the design is the core region. The core corresponds to SQL, machine learning and deep-learning operators or a subset of operators that can be mapped and accelerated in FPGA. The core region, which includes RTL IPs, is converted to RTL Kernels in Xilinx SDAccel Development Environment [4]. This enables us to use the same software interface model as OpenCL instantiated kernels. Based on the area constraints of the FPGA, our core region can consist of multiple RTL kernels. The independent RTL kernels operate on mutually exclusive partitions of data. These RTL Kernels can be considered as logically equivalent of having multiple independent cores in a processor.

6. SOFTWARE INTERFACE FOR FPGA BASED ACCELERATORS

As discussed in Section 4.1, the data-flow compiler is responsible for generating codes for specific accelerators like FPGAs and the hypervisor is responsible for executing the generated codes on the accelerators. The hypervisor layer communicates to low level device-specific drivers through an interface class which has a fixed set of APIs that are called at run-time by any operator that wishes to use FPGA. The translation of these API calls to device-specific drivers is handled through Bigstream's HAL or *Hardware Abstraction Layer*. As shown in Figure 8, these API calls that abstract away device-specific drivers are being standardized through

an open-API initiative, or OHAI (Open Hyper Acceleration Initiative) that will be published in the near future. These API calls belong to an interface class called OHAI class interface. Each Spark executor can potentially have a SQL operator in every stage that instantiates the OHAI class interface to accelerate on FPGA. As shown in Figure 8 for *Vendor C*, we also support accelerating Spark streaming applications by processing TCP/IP based packets in FPGA. All such interfaces contend to access resources of FPGA. We have implemented a priority round-robin scheduling to grant access to FPGA resources. The software and hardware interfaces were originally targeted for offload mode of processing, in which data is brought to host memory and then send to FPGA for compute acceleration. In this mode, FPGA can be logically considered to be a co-processor to CPU cores, sharing host memory with them.

7. DEVELOPING TEMPLATES FOR SMARTSSD

This section discusses adapting our templates for SmartSSD and associated optimization necessary for high performance. With reference to the hardware logical layers in Figure 7, the shell layer for SmartSSD is provided by Xilinx. Our shim layer, designed for offload mode, needed a minor modification on the DRAM interfaces to adapt to P2P shell. Our kernels consume Spark data partitions from FPGA GAM and send the processed data back to host memory in a format similar to Tungsten [19] rows, which is an in-memory format used in Spark. For interoperability, all our row based hardware templates process data internally in a similar format. We ported multiple of these row based templates to SmartSSD so that based on the query being processed we can reconfigure the core region. All templates in the core receive data to process from FPGA GAM and send the result back to FPGA DRAM. After porting our design, the configurable logic blocks (CLB) utilization was 52% to 64% for our designs with the core region operating at 200Mhz for the row-based templates with three kernels. As explained in Section 4, a query can pick different templates based on application requirements and the cost function of the template. On the software side, changes are necessary to make sure that data partitions generated by Spark are read into P2P GAM as opposed to host memory in the offload implementation. This required adding a Spark partition file receiver specifically targeted for P2P transfer. In the case of accelerated Spark, for the scan operator, we issue a P2P read to transfer the data from SSD to FPGA GAM and once the data transfer is complete we enqueue an OpenCL compute task to operate on the data in GAM.

7.1 Performance Optimization

An important optimization that is necessary for good performance is to decouple the P2P command queue from compute command queue. This is necessary because we can have data transfers initiated to the FPGA GAM region through P2P command queue, while the compute command queue is being processed on another GAM region. A further optimization is to use asynchronous read for P2P as opposed to synchronous read so that a single thread can operate on both P2P and compute command queue. Since Xilinx drivers support POSIX API for P2P read, we used *aio_read* as opposed to *pread* or *read* [3]. This also enables us to have all our OpenCL calls as non-blocking and asynchronous, in order to issue multiple commands to both command queues on a single thread effectively. This also reduces CPU utilization by avoiding polling/busy-waiting. Another optimization tool provided by Xilinx is to use the Embedded Run Time (ERT) option which reduces polling on the CPU side by moving the polling to the ARM processor in the FPGA. We will discuss our results with respect to JSON data format which is one of the predominant row based formats used in big data analytics.

8. EVALUATION

Our testbed consists of 5 servers with 1 driver node and 4 worker nodes with SmartSSD attached to the PCIe slot. Each server contains Intel Xeon Gold 6152 CPU operating at 2.10GHz with 22 physical cores in dual sockets. Each server contains 128 GB DRAM and the servers are connected through 10Gbps network. Figure 10 compares performance of scan heavy TPC-DS queries between vanilla Spark and Bigstream accelerated Spark in SmartSSD for a single node. These measurements were made after all the aforementioned hardware optimizations and the results presented are end-to-end query time for 200SF(200GB) data. As is observable from Figure 10, most queries achieve around 6x acceleration with respect to Spark. The number of Spark executors was set to 6 with a total executor memory of 100GB. We decided on the executor count being 6 for one SSD as a comparable equivalent to one SmartSSD. The top-15 queries are presented for both cases separately since they are not the same subset for both of them and the baseline Spark performance also varies between SmartSSD and SATA connected SSD. Figure 11 shows results for a 4 node cluster with each node having 100GB of TPC-DS data totaling 400GB for the cluster. The acceleration numbers here are similar to that of a single node cluster. In cluster mode, data is distributed across SmartSSD of each node and scan operation and operators succeeding scan are processed in SmartSSD. Once the scan stage is complete, shuffle data is also placed in SmartSSD so that it can be redistributed to other nodes. The average acceleration across all TPC-DS is 4.08x with respect to vanilla Spark.

Another important advantage of SmartSSD is lower CPU utilization which is illustrated through time-line graph of user utilization obtained through Grafana [2] in Figure 9 for Query 69. The left side of the graph shows CPU utilization of executors in our accelerated Spark. The right side shows CPU utilization of vanilla spark executors. The peak CPU utilization of our version is less than 30% while that of Spark is around 90% for the cores executing the query. Across all TPC-DS scan heavy queries, we observed that CPU utilization was similarly low for SmartSSD. In production environments, with multi-tenancy support, lower CPU utilization could lead to more workloads being run on the same machine through different containers.

9. FUTURE WORK

We have presented results that show significant run-time speedup over Apache Spark, based on our row based design. In our final product, which is under quality control and testing, we are using more SSDs in the system and we are observing that the performance scales across executors

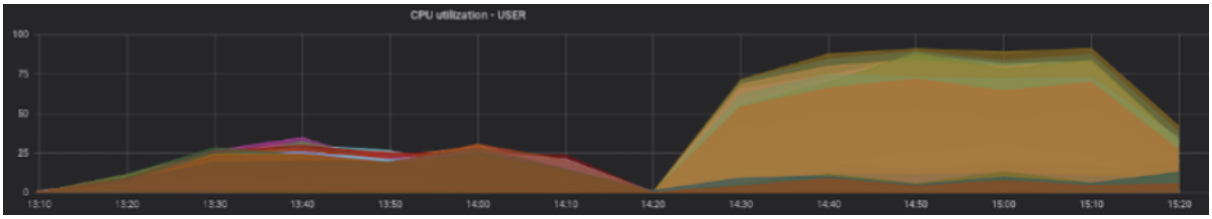

Figure 9: CPU utilization - Spark vs XSD

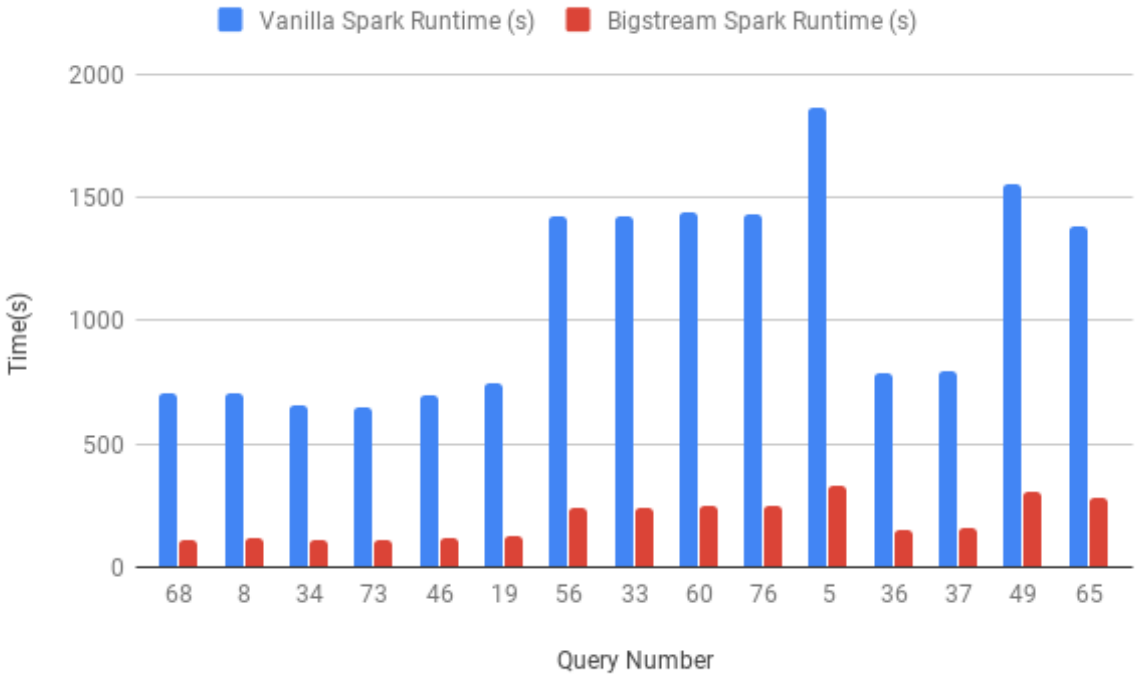

Figure 10: Row Based acceleration results for Top 15 TPC-DS scan heavy queries with SmartSSD

and SmartSSDs. In our future work, we will present comprehensive results on columnar formats such as Parquet and ORC. On the hardware side, we are actively working on increasing the number of templates available in SmartSSDs by including solutions such as DNN and machine learning pipelines [20], cryptographic and hashing templates, etc. We are also working on pushing the frequency of our kernels along with the number of kernels. On the software side, we are working on adding support for frameworks like Hive and Kafka KSQL to take advantage of our hyper-acceleration layer which now supports SmartSSD as well as traditional FPGA architectures.

10. CONCLUSION

In this paper, we have explained our framework for accelerating Big Data platforms on computational storage. We chose Samsung SmartSSD which enables performing computation close to storage and have shown significant query runtime speedup for TPC-DS benchmarks for row-based format on SmartSSD, compared to Apache Spark. Our results also scale when we move from single node to multi-node. Apart from higher performance, we also gain total cost of ownership (TCO) savings when using our accelerated platform on SmartSSD. Two main reasons for TCO saving are: 1) CPU utilization drops dramatically during our accelerated Spark with SmartSSD, specifically during scan stages. This drop leads to shorter tenant run-time and enables more containerized processes in a multi-tenant environment. 2) SmartSSD consumes lower power than adding more cores to a processor which adds up to a significant amount of energy saving in modern data-centers.

The hardware, that we are working with, is a pre-production part and we are confident of improving performance with our next generation of software and hardware IPs targeted for the production part. The production version of SmartSSD is a PCIe U.2 attached card, which is more tightly integrated with improved bandwidth and SSD capacity.

11. ACKNOWLEDGMENTS

We would like to thank Xilinx and Samsung for providing us access to pre-production parts of Xilinx DSA and SmartSSD. We would like to thank Gopi Jandhyala, Sebas-

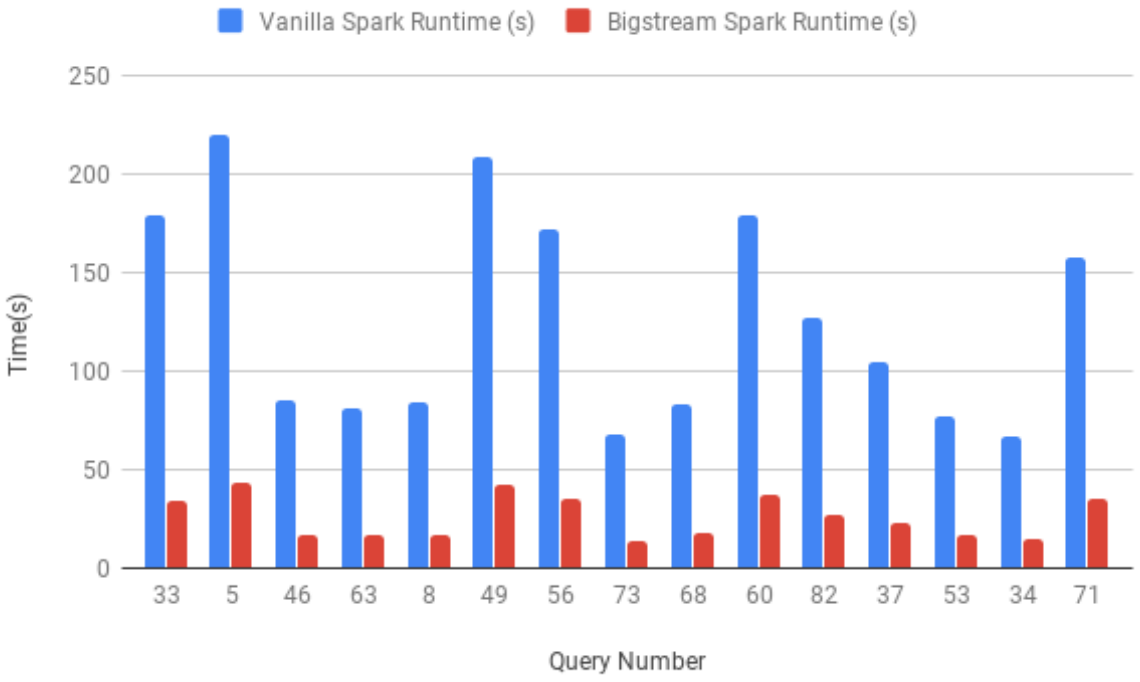

Figure 11: 4 node cluster results for Top 15 TPC-DS with 400 GB data with SmartSSD

tian Turullols from Xilinx and Vish Maram, Pankaj Mehra, Fred Worley and Bob Napaa from Samsung. We would also like to thank Weiwei Chen, John Davis, Gowtham Chandrasekaran, Brian Hirano, Roop Ganguly, Chris Ogle, Chris Both and Danesh Tavana for support in this work.

12. ADDITIONAL AUTHORS

13. REFERENCES

[1] Computational storage technical work group. `https://www.snia.org/computational`, visited 2019-06-02.

[2] Grafana labs. 2018. grafana – the open platform for analytics and monitoring. `https://grafana.com`, visited 2019-06-02.

[3] The open group base specifications issue 7. `https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/`, visited 2019-06-02.

[4] SDAccel development environment help for 2018.3. `https://www.xilinx.com/html_docs/xilinx2018_3/sdaccel_doc/creating-rtl-kernels-qnk1504034323350.html`, visited 2019-06-02.

[5] SmartSSD, faster time to insight. `https://samsungatfirst.com/smartssd/`, visited 2019-06-02.

[6] TPC-DS. `http://www.tpc.org/tpcds/`, visited 2019-06-02.

[7] An update on data center fpgas. `https://www.forbes.com/sites/moorinsights/2018/07/20/an-update-on-data-center-fpgas/#4fcff99346c2`,visited 2019-06-02.

[8] What is a hypervisor. `https://www.vmware.com/topics/glossary/content/hypervisor`,visited 2019-06-02.

[9] M. Armbrust, Y. Huai, C. Liang, R. S. Xin, and M. Zaharia. Deep dive into spark sql's catalyst optimizer. `https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html`, visited 2019-06-02.

[10] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. Active flash: Out-of-core data analytics on flash storage. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[11] 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*, pages 1221–1230. ACM, 2013.

[12] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu. Optimizing opencl applications on xilinx fpga. In *Proceedings of the 4th International Workshop on OpenCL*, page 5. ACM, 2016.

[13] P. Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.

[14] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 153–165. IEEE Press, 2016.

[15] 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)*, pages 1–12. IEEE, 2013.

[16] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[17] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[18] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[19] J. Rosen. Deep dive into project tungsten: Bringing spark closer to bare metal. jun. 16, 2015. `https://databricks.com/session/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal`, visited 2019-06-02.

[20] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[21] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 4. ACM, 2016.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.