XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD

Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro
School of Electrical and Electronic Engineering
Yonsei University
Seoul 120-749, Republic of Korea
{youngjcho, ws.jeong, ohdooh, wro}@yonsei.ac.kr

***Abstract*—Considerable research has been conducted recently on near-data processing techniques as real-world tasks increasingly involve large-scale and high-dimensional data sets. The advent of solid-state drives (SSDs) has spurred further research because of their processing capability and high internal bandwidth. However, the data processing capability of conventional SSD systems have not been impressive. In particular, they lack the parallel processing abilities that are crucial for data-centric workloads and that are needed to exploit the high internal bandwidth of the SSD. To overcome these shortcomings, we propose a new SSD architecture that integrates a graphics processing unit (GPU). We provide API sets based on the MapReduce framework that allow users to express parallelism in their application, and that exploit the parallelism provided by the embedded GPU. For better performance and utilization, we present optimization strategies to overcome challenges inherent in the SSD architecture. A performance model is also developed that provides an analytical way to tune the SSD design. Our experimental results show that the proposed XSD is approximately 25 times faster compared to an SSD model incorporating a high-performance embedded CPU and up to 4 times faster than a model incorporating a discrete GPU.**

I. Introduction

The idea of active disks [1], [2] has been around for several years. By providing processing capability to storage devices, database machines were able to exploit concurrency, and get around the limitations of storage bandwidth in order to improve overall system performance. Despite these benefits, the active disk concept was not adopted widely by the industry because the marginal performance improvement could not justify the associated cost. Recently, however, the active device concept has attracted renewed attention with the emergence of flash-based solid-state drives (SSD).

There are two reasons for this renewed attention to this concept. First, the internal bandwidth of SSD is considerably higher–approximately 2-4x–compared to the bandwidth of a typical host interface. Moreover, it is projected that this gap is likely to widen to more than 10x [3]. Second, the SSD inherently has a computing substrate to support the functions of the flash translation layer (FTL). It is easy to enhance the processing capabilities by just replacing the components of the existing system to more powerful ones.

Several software frameworks [3]–[5] have been developed to leverage the computing power of an SSD. In general, these frameworks are designed for easy-to-parallelize data-centric applications and provide programming interfaces to exploit parallelism in a clustered SSD environment. Some of the frameworks are built around the MapReduce [6] programming model.

The MapReduce programming model has become quite popular since it was introduced by Google researchers in 2004. The primary reason for this success is the fact that the model makes it easy for programmers to write parallel codes as long as the applications can be expressed in the two primitives, map and reduce. Once the program is written in this model, the exploitation of parallelism is automatically done by the MapReduce system and porting the program to a variety of parallel platforms becomes easy.

A study of various SSD frameworks reveals that the performance of tasks running in a conventional SSD is restricted not by inadequate parallelism, but rather by the low computing capabilities of the platform. Thus, even though the multiple inputs are ready to be consumed concurrently, the computing system cannot handle these inputs in a timely manner. In particular, when the workload requires several computations per unit data, the performance is limited by the data processing rate of the embedded CPU typically provided in these systems. Moreover, even when the number of computations per unit data is low, the performance is limited by frequent DRAM accesses that are slowed by the low bandwidth. Thus it is important to supplement the computing power of the embedded CPU in an SSD in order to utilize the SSD in a general-purpose computing environment.

An obvious way to improve the computational capability is to add high-speed embedded CPUs. However, this cannot effectively speed up the data-centric workloads that are typical of in-storage processing. These workloads inherently have high parallelism but few computations per data. Therefore, it is better to add many low-speed cores rather than a few high-performance cores.

Towards this end we propose in this paper the acceleration of MapReduce in an SSD environment through the use of an embedded GPU. The GPU is gaining importance for general-purpose usage beyond its graphic processing roots [7], [8], and we expect that implementation of the MapReduce programming model in a GPU will be attractive to programmers both because of its good programmability as well as its easy exploitation of a high degree of parallelism.

We introduce a concept we call XSD, pronounced as "Access-D", and standing for Accelerator SSD. XSD is a new SSD architecture that basically embeds a GPU in the SSD. We also describe hardware and software support to accelerate the MapReduce-based framework. We also present an analytic model to detect and overcome performance bottlenecks.

Our evaluation shows that introducing a GPU in an SSD improves performance 24.6 times more than a high-speed CPU. Furthermore, the XSD with an embedded GPU is up to 4 times faster than a discrete off-the-shelf GPU for data-intensive workloads. For workloads that have relatively large computational intensity, adding an L2 cache helps in reducing the effective DRAM latency which dominates the overall

performance.

The rest of this paper is organized as follows: Section II presents prior work on in-SSD processing and on MapReduce frameworks for GPUs. Section III describes software support for programmers and hardware solutions to overcome architectural challenges. In Section IV we provide results of evaluation of our XSD architecture and in Section V we provide some concluding remarks.

II. Related Work

A. In-SSD Processing

The embedded CPU in a typical SSD is used typically to execute the flash translation layer (FTL) logic which translates Logical Block Addresses (LBA) into Physical Block Addresses (PBA). However, there has been considerable prior research on techniques to exploit the computing capability of the embedded CPU in the SSD for general-purpose computations.

Cho et al. [4] developed an intelligent SSD (iSSD) that exploits the heterogeneous processing resources inside an SSD. Since current SSDs have insufficient computing power compared to data bandwidth, they propose using the power of the stream processors inside the flash memory controllers (FMCs) to increase the data processing rate. This is particularly suited to the SSD architecture since the computing power scales directly with the number of flash channels. However, the responsibility for managing the limited resources inside the stream processor is completely on the programmer. When the workloads are complex it becomes difficult to program the stream processor with the given resources.Thus the programmability of this solution is limited.

Do et al. [3] ported query processing components into a Smart SSD system. To give more flexibility to programmers, they provided APIs to manage commands from the host, threads for query operation, memory inside the SSD, and data in the DRAM. However, performance improvement was limited by low-performance embedded CPUs. The performance gain was large when the number of computation queries per page was low, but rather small when the number of queries was high. Thus more computing power is required in the SSD to support more complex operations.

Kang et al. [5] introduced a Smart SSD model in which a high-performance host CPU and near-data processing SSD cooperate to compensate for their shortcomings. The host performs overall coordination and offloads the data-oriented tasks to the Smart SSD device. However, the device suffers from low performance for tasks that frequently access the DRAM. This is because the DRAM inside the SSD is not designed to be utilized effectively by the embedded CPU. Therefore, a Smart SSD performs only the task of filtering data before handing over to the host CPU to perform the remaining tasks that access DRAM frequently.

B. MapReduce on GPU

MapReduce is a popular framework that allows programmers to write parallel programs easily. There have been several implementations of the MapReduce framework to exploit the parallelism inherent in GPU platforms.

Mars [9] provides several techniques to overcome challenges that hinder the porting of MapReduce on the GPU. Mars adds two user-implemented APIs, MAP_COUNT() and REDUCE_COUNT(), and executes each function prior to map and reduce, respectively. Thus, the framework can statically prepare memory space for variable-length key/value pairs. Moreover, since the output array locations that are assigned to each thread are deterministic and are free of overlap, multiple threads can write the outputs without synchronization. Furthermore, Mars utilizes coalesced accesses to improve the memory bandwidth and sorting is used for grouping intermediate key/value pairs since hashing is a difficult algorithm to implement in a GPU. However, Mars suffers from performance degradation because of the long grouping phase that is attributed to the execution time of the two additional functions and inefficient sorting.

MapCG [10] provides code-level portability between the CPU and the GPU. To achieve its primary goals, good portability, and ease of use, the framework allows programmers to write a single compatible code that can run on both the CPU and the GPU. Further, a lightweight memory allocation scheme is proposed, which prevents the additional counting overhead. As dynamic memory allocation is available, a hash table is implemented to improve the performance of the grouping phase. However, even though MapCG is profitable for workloads whose data fit into the GPU memory, it is not clear whether it will show the same performance improvement with big data in very large server systems. In this case, the data bandwidth between the storage and the memory is another issue that should be considered.

III. Accelerator SSD (XSD)

A. Software Supports

There are two main design goals of our software framework. The first is the provision of a simple but sufficient programming interface to users, and the second is the development of a powerful runtime system that can exploit abundant parallelism and a large data bandwidth. Table I shows the list

User-implemented APIs
void XSD_Map(key*, value*)
void XSD_Reduce(key*, value*)
void XSD_Prepare(config*)
void XSD_Hash(key, value)
System-provided APIs
void XSD_StoreKeyVal(key, value)

TABLE I: XSD APIs

of APIs representing user-implemented and system-provided functions. Users implement XSD_Map() and XSD_Reduce() functions as in normal MapReduce frameworks, such as in Hadoop [11]. Further, users can define a hash function, XSD_Hash(), to group a key/value pair when the local and global merge operations are executed. By programming the XSD_Prepare() function, users can change configuration parameters of the framework, such as the input data set and the input format. XSD_StoreKeyValue() is a system-provided API, which should be called inside XSD_Map() or XSD_Reduce(). It is called when the map and reduce function generates the output key/value pairs.

1) Host-Device Communication: The proposed framework is based on communication between the host CPU and the XSD devices. Figure 1 represents the communication protocols in the XSD system. The protocol is implemented as a runtime library for the host CPU and as device firmware for SSD devices. When inputs are ready before the communication begins, the host CPU asks the SSD CPU whether it is ready. The ask signal contains the information about program size, and the SSD CPU returns an acknowledge signal when it

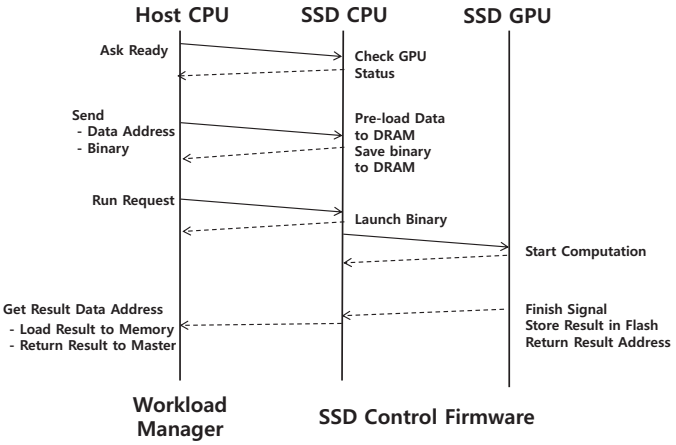


Fig. 1: Communication flow between host and device

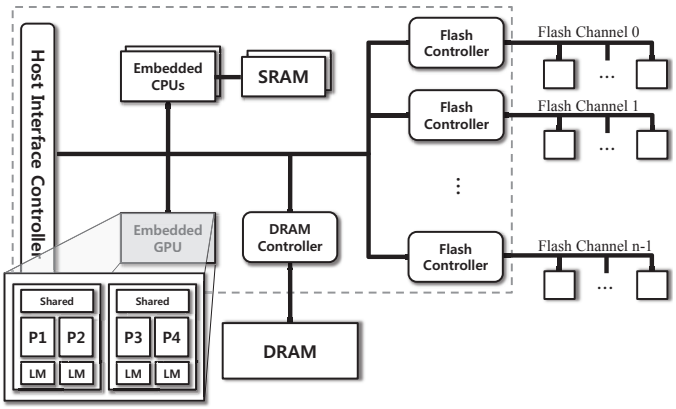


Fig. 2: XSD architecture

finishes assigning memory space for the program. Then, the host CPU sends the page addresses of the inputs and the program codes to the device. The SSD CPU receives the data and returns the device ID to the host as a success signal. When the data and the inputs are ready, the host CPU sends a run signal with the device ID. Controlled by the SSD CPU, SSD GPU executes the MapReduce functions until all processes end. Finally, a finish signal is sent back to the host with addresses of the output pages and the host terminates the communication.

2) Runtime System: To execute computation-intensive workloads with large-volume inputs, it is important to prepare inputs and configure the system appropriately. The runtime system aims to maximize the utilization of the given resources, such as the computing power and the data bandwidth. Initially, when our framework is issued, the runtime system invokes the user-defined XSD_Prepare() function. To utilize all bandwidths and acceleration resources in the SSDs, the preparation is conducted with inter- and intra-SSD redistribution. According to the input size and the distributed patterns, the runtime system determines whether to redistribute the data across the SSDs or not. If the redistribution is issued, the host CPU transfers data from one SSD to another in the most effective manner, which is the inter-SSD redistribution. During this step, intra-SSD distribution is also carried out by the device firmware. It stores the data evenly across flash channels so that the full aggregate bandwidth can be exploited.

The runtime system also manages the control flow of MapReduce. Instead of executing the map and reduce functions step-by-step for all the inputs, XSD_Reduce() is invoked after XSD_Map() in order to generate a certain number of intermediate key/value pairs. This is attributed to the hardware restrictions: small buffer size. If the intermediate data exceeds the buffer size, it should be stored in the flash memory. To decrease the flash access and data size, the framework executes XSD_Reduce() for the buffered data. Then, a local merge is followed by map and reduce that are executed for all the inputs. It merges and sorts the results with the user-defined hash function. Since the execution of local merges is distributed to SSDs, the host CPU can save time for the global merge that may take a considerable amount of time.

B. Strategies of Hardware Optimization

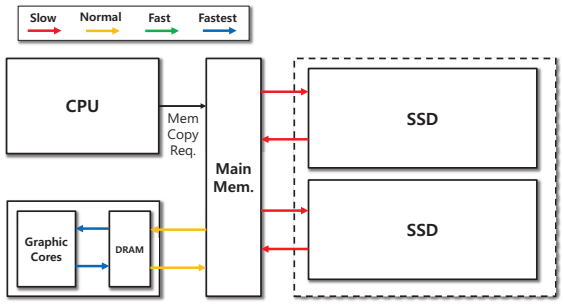
We present an SSD architecture with an embedded GPU. A GPU is composed of many SIMD processors that support thousands of threads. For data-intensive and easily parallelizable workloads, such as the MapReduce framework, GPUs and in-storage computation form a good combination that can improve performance. Our XSD architecture is shown in Figure 2. An embedded GPU shares a memory with CPUs, yet it has its own local memory. In this part, we address the related challenges and provide strategies to overcome them. Moreover, performance is analyzed by modeling each stage of the execution process.

1) Streaming: In general, a discrete GPU follows three steps to process data: the host CPU copies the input data into the GPU memory, the GPU executes, and the host CPU copies back the results to the main memory. Since a discrete GPU has a large memory size, approximately 1 to 6 GB, the input data size fits in the memory in most cases. However, a GPU is unlikely to be suitable for manipulating large-scale data. According to [12], the GPU is idle for 98% of the time when it processes 500 GB of data. This time wasted is due to waiting for data to be copied to the GPU memory.

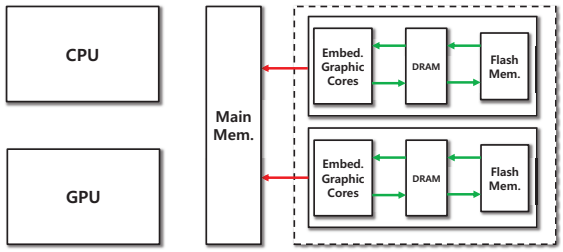
The abovementioned approach can be refined by leveraging a streaming method, such as CUDA streams [13]. A CUDA stream is a sequence of operations executed in the issue-order on the GPU. Each stream undergoes three stages: memory copy to the GPU, execute, and memory copy from the GPU. By overlapping memory copies with the GPU execution time, performance is improved because of the enhanced concurrency. However, if the data size gets considerably larger than the main memory, the GPU idle time will increase because of the I/O access latency of the storage device.

In this context, to combine the GPU with the SSD, we should consider that the embedded GPU has two differences compared to a discrete GPU: small DRAM capacity and high data bandwidth. The DRAM in an SSD is smaller in size than in a GPU. Further, since the embedded CPU shares the memory with the GPU, the space that the GPU can use is reduced; therefore, frequent flash access is required. Fortunately, a high aggregate flash bandwidth permits transforming the execution process into data streaming from flash to GPU, and vice versa, by buffering the data in the DRAM.

Figure 3 illustrates the data transfer paths of streaming for (a) discrete GPU and (b) embedded GPU. The color of the arrow represents the relative bandwidth of the inter-node communications. There are three implications of the figure. First, if the data volume is so huge that the data do not fit in the main memory, a discrete GPU is inappropriate for streaming since it involves many slow connections. Second, irrespective



(a) Stream path of discrete GPU



(b) Stream path of embedded GPU in XSD

Fig. 3: Comparision of the stream path of discrete and embedded GPUs

of the data bandwidth, an embedded GPU is easy to scale as the data size increases. On the other hand, the performance of a discrete GPU does not scale well since the performance is bound to the external storage bandwidth. Third, the CPU is free during execution in (b), while the CPU should be kept running in the case of (a) in order to manage the GPU memory, which costs energy.

In streaming, the embedded CPUs control the overall workflow. Embedded CPUs are in charge of managing the input/output buffers and balancing the speed of the three stages. When the data are ready to be written in the buffer, the CPUs allocate a memory space by using memory APIs. Further, they provide a data format to the GPU, which can only use a static array as a data format. To resolve the bottleneck of streaming, the CPUs also regulate the data processing rate of all stages on the basis of the buffer-filling patterns. The regulating strategies are detailed in Section III-B3.

2) Synchronization: Since the GPU and the multiple flash channels share buffers in the DRAM, synchronization is necessary in XSD. If the synchronization operation is based on a lock scheme, the overhead increases as the number of flash channels increases due to contention. Therefore, a lock-free scheme is required to utilize all channels simultaneously.

In an SSD, flash read and write are conducted with page granularity. If the data size is fixed and the base address is given, it is possible to calculate the destination address for all channels. For instance, if the base address of a buffer is 0x1000 and the page size is 0x100, the destination address of the channels will be 0x1000, 0x1100, 0x1200, and so on. Thus, it is possible to avoid conflicts without a lock scheme since there are predetermined locations for the data from each channel. An embedded CPU should calculate these addresses and notify DMAs in the flash memory controllers.

Since the GPU does not have hardware support for atomic operations, embedded CPU support is also required. The CPU

Parameter	Description
α	probability that input buffer is empty
β	probability that input buffer is full
L_{fr}	flash read latency for reading one page
L_{fw}	flash write latency for writing one page
r_{dram}	data bandwidth of SSD DRAM
n_{ch}	number of flash channels in use
b_p	bytes per page
M	GPU parallelism
s_i	average bytes of inputs that GPU accesses during execution
s_o	average bytes of outputs that GPU generates during execution
CPB	execution cycle per byte
f	GPU frequency
b_{insn}	number of bytes per instruction
n_{insns}	number of dynamic instructions

TABLE II: Descriptions of XSD parameters

prevents the GPU from reading a page that is being written to by the flash controller until the write operation finishes. However, simultaneous read and write are allowed for different pages.

3) Performance Models: We develop a performance model by extending the performance model in [4]. Our model includes data read/write patterns of the flash memory, workload characteristics, and the impact of the runtime conditions. Table II lists the parameters that are used in the model and describes what each parameter stands for.

Data processing in XSD consists of three stages: read inputs, execute, and write outputs. We assume that the size of the inputs is extremely large, and calculate performance as the number of bytes each stage can process per second. Since the stage with the lowest performance determines the overall performance in a stream, the minimum value among the three stages is the final performance. Hence,

$$Perf = min\{Perf_*\}. \tag{1}$$

The performance of the first step in the read-input stage is modeled as Equation 2. The data size is $n_{ch}b_p$ since data granularity in the flash memory read/write is a page and there are n_{ch} channels that can read simultaneously. The latency of flash read is expressed with the probability term α, because the latency only applies when the buffer is empty. That is, α implies the runtime condition of the buffer state. If the buffer is empty, the flash read latency cannot be hidden by the other stages.

$$Perf_{f2d} = \frac{n_{ch}b_p}{\alpha L_{fr} + \frac{n_{ch}b_p}{r_{dram}}}. \tag{2}$$

In the second stage, the same instructions are executed with all the key/value pairs. Therefore, M pairs are read and written from the DRAM, while only one instruction set is read. The term CPB, which stands for cycles per byte, represents the computation intensiveness of the workloads. As a result,

$$Perf_{exe} = \frac{Ms_i}{t_{exe}}, \tag{3}$$

$$t_{exe} = \frac{Ms_i + b_{insn}n_{insn} + Ms_o}{r_{dram}} + \frac{CPBs_i}{f}. \tag{4}$$

The last stage writes back the data to the flash memory. As in Equation 5, we add β to reflect the case when the output

buffer is full. If the buffer gets full, the data in the buffer must be stored back to the flash memory. Therefore,

$$Perf_{d2f} = \frac{n_{ch} b_p}{\beta L_{fw} + \frac{n_{ch} b_p}{r_{dram}}}. \quad (5)$$

According to this model, the ideal performance is achieved when the three stages have equivalent performance. To realize this goal, we present three control strategies that can balance each stage either statically or dynamically. The first strategy is to control the channel number when a flash read/write is requested simultaneously. If simultaneous flash read/write is not allowed, either read or write operations must wait until the other has finished. Therefore, read and write operations should be arbitrated by allocating appropriate bandwidth to each of them. To control the bandwidth, the firmware divides the flash channel into two. Then, it calculates α and β values whenever read and write commands are issued. If the performance imbalance is severe between the read and the write stages, the firmware controls the number of channels to regulate performance. The reading and writing channels rotate in round-robin manner to prevent reading and writing to the same channels repeatedly. Thus, it is possible to achieve balance between reads and writes.

The second strategy is to control the DRAM bandwidth. Since every stage shares DRAM, many contentions for DRAM occur during streaming. The DRAM controller must prioritize the request from the lowest performance stage over others. Furthermore, if the performance of the GPU execution stage is low because of frequent memory access, SSD manufacturers can consider the addition of a dedicated SRAM or last-level cache (LLC) to the embedded GPU.

Lastly, SSD manufacturers can estimate a rough specification of the embedded GPU by exploiting our performance model. With the performance model of GPU execution, the maximum value of *M* and *f* in the GPU can be analyzed by assuming an ideal read-input and write-output performance. Using such value, the manufacturers can choose the number of GPU threads (*M*) and frequency (*f*) of the embedded GPU in the most cost-effective manner possible.

IV. Evaluation

A. System Configuration

We implement the proposed system on the basis of the simulator introduced in [14]. The simulator integrates two state-of-the-art simulators, gem5 [15] and GPGPU-Sim [16], to simulate the interaction between CPU and GPU in a shared memory environment, which fits our architecture. We modified the GPU interface to be controlled by the CPU and added DMA, SRAM, and a controller in the storage model to simulate the operation of SSD.

In our experiment, we compare the performance of the discrete GPU and XSD. We configure the discrete GPU to model NVIDIA GeForce GTX 580. Further, the embedded GPU is modeled as a GPU in NVIDIA Tegra 4. There are two Embedded CPU specifications. One is a high-performance CPU to compare the performance with the embedded GPU, and the other is a CPU to support the normal XSD operation. We configure the former CPU as the one in Tegra 4. For the latter CPU, DRAM, and flash memory model, we use the parameters introduced in [2], which reflects the technology trend of the SSD. The detailed configurations are summarized in Table III.

We choose three benchmarks to test the performance of our implementation: MatrixMultiplication (MM), PageViewCount (PVC), and PageViewRank (PVR). They represent the applications in web data analysis, such as web log analysis and web document searching. We use benchmarks provided by Phoenix [17] for CPU execution and by Mars [9] for GPU execution.

Discrete GPU	
cores	512
clock frequency	772 MHz
memory bandwidth	192 GB/s
XSD	
CPU frequency	1.6/0.4 GHz
GPU frequency	600 MHz
GPU L1/L2 cache	32/192 KB
CPU/GPU cores	4/48
DRAM/Flash bandwidth	3.2/0.4 GB/s
Flash channels	8

TABLE III: Key configuration parameters of discrete GPU and XSD

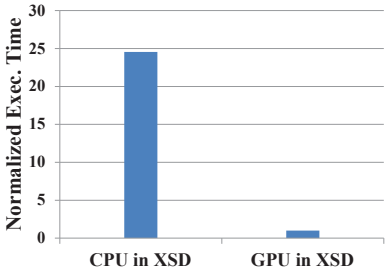


Fig. 4: Performance comparison between CPU and GPU in XSD

B. Performance Analysis

Figure 4 depicts the elapsed time of the embedded GPU and CPU normalized to that of the GPU. Since MM has a relatively larger number of computations among the three benchmarks, we choose it so that the embedded CPU can benefit from its high computational speed. The result shows that the embedded GPU is 24.6 times faster than the embedded CPU. This implies that the GPU is considerably more capable of accelerating data-centric workloads even though it requires a considerable number of computations.

Figure 5 illustrates the elapsed time of the discrete GPU and XSD with and without L2 cache for the three benchmarks. All elapsed times are normalized to the elapsed time of XSD with the L2 cache. We express the performance of XSD only in terms of the kernel execution time, because it is difficult to classify the elapsed time of each stage in the XSD due to the streaming scheme.

Even though the kernel execution time of XSD has increased for all benchmarks, the overall performance of XSD for PVR and PVC is higher than that of the discrete GPU, which is attributed to XSD's high internal bandwidth. The kernel execution of XSD takes more time due to the low frequency and parallelism of GPU, contention for DRAM and flash bandwidth, and control overhead of the embedded CPU. Further, XSD does not benefit from the L2 cache; instead, it suffers from the overhead caused by the cache operation.

On the other hand, XSD without the L2 cache is slower than the discrete GPU when executing MM. This can be attributed to the dominance of the increase in the kernel execution time over the decrease of the storage access time.

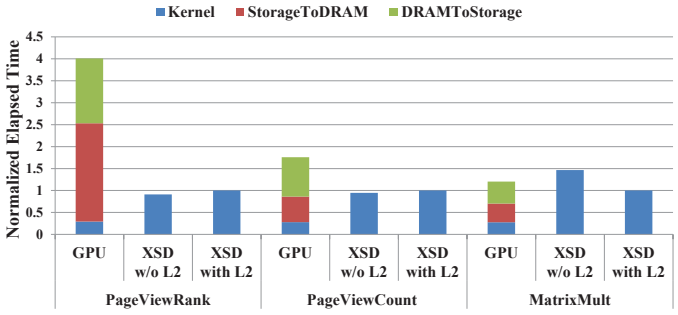


Fig. 5: Elapsed time of discrete GPU, XSD, and XSD with L2 cache

However, the L2 cache improves the performance of XSD by almost 50%, which implies that the elapsed time of MM heavily depends on the DRAM latency. Although the addition of the L2 cache has a reverse effect in the case of PVR and PVC, it is a beneficial trade-off with respect to performance since speed-up in computation-intensive workloads, such as MM, is fairly high while the performance degradation in data-intensive workloads, such as PVR and PVC, is rather small.

Another interesting point is that the execution time of XSD can be further decreased by leveraging more XSDs. For instance, if there are two XSDs in a host machine, the elapsed time can be reduced almost half because each XSD exploits its own internal bandwidth and does not affect the other. On the other hand, the discrete GPU has limited scalability in one host system since most of the execution time of the discrete GPU is occupied by the storage access time. The problem is that the storage bandwidth is unlikely to be scaled along with the increasing number of discrete GPUs. As a result, the external storage bandwidth limits the utilization of multiple discrete GPUs, which leads to low scalability.

V. Conclusion and Future Works

In this paper, we introduced a new SSD architecture that has a more powerful brain, namely embedded GPU. We illustrated that the addition of a powerful CPU is not an appropriate solution as the new brain to accelerate data-centric workloads. Furthermore, the discrete GPU has a limitation with respect to the processing of large-scale data sets, since frequent storage accesses lead to low utilization of the GPU. On the other hand, by exploiting the high internal bandwidth of the SSD and the streaming scheme, the embedded GPU can alleviate the performance dependence on the storage bandwidth. Our experimental results show that the embedded GPU is 24.6 times faster than a high-performance embedded CPU, and up to 4 times faster than discrete GPUs. Moreover, combining an SSD with a GPU is advantageous from the perspective of scalability. Even though the data size increases, system scaling is rather easy upon the addition of SSDs that have the appropriate computing power to process the growing data sets.

However, much work in this field is still left. We did not consider the perspective of energy efficiency. Energy efficiency is always an issue in large server systems. However, we project that energy consumption may decrease by distributing the computations of the power-hungry host CPU to the low-power embedded GPUs. Further, sparse access to memory and storage can be one energy-saving factor of the proposed SSD model.

We did not evaluate the cost to place a GPU inside SSDs. The current SSD lacks a high processing rate because of economical and technological costs. However, we expect that, in the near future, these costs can be justified by providing an effective direction to utilize high-performance processing units in an SSD.

Acknowledgment

This work is partly supported by the Flash Software Development Team of Memory Division, Samsung Electronics Co., and by the Industrial Strategic technology development program (10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea). The authors would like to express their sincere gratitude to fellow members of the ESCAL research group and Moon Sang Kwon and Myung Hyun Jo of Memory Division, Samsung Electronics Co., for their time, suggestions, and valuable feedback.

References

[1] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, vol. 27, no. 3, pp. 42–52, 1998.

[2] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proc. of VLDB*. Morgan Kaufmann Publishers Inc., 1998, pp. 62–73.

[3] 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 *Proc. of SIGMOD*. ACM, 2013, pp. 1221–1230.

[4] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active disk meets flash: a case for intelligent ssds." in *Proc. of ICS*. ACM, 2013, pp. 91–102.

[5] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in *Proc. of MSST*. ACM, 2013, pp. 1–12.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] NVIDIA. (2006) CUDA. [Online]. Available: http://www.nvidia.com/cuda

[8] Khronos Group. (2008) OpenCL. [Online]. Available: http://www.khronos.org/opencl/

[9] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 4, pp. 608–620, 2011.

[10] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: writing parallel program portable between cpu and gpu," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 217–226.

[11] T. White, *Hadoop: the definitive guide*. O'Reilly, 2012.

[12] O. Trelles, P. Prins, M. Snir, and R. C. Jansen, "Big data, but are we ready?" *Nature reviews Genetics*, vol. 12, no. 3, pp. 224–224, 2011.

[13] S. Rennich, "Cuda c/c++ streams and concurrency," *NVIDIA,[online]*, 2012.

[14] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, "Workload and power budget partitioning for single-chip heterogeneous processors," in *Proc. of PACT*. ACM, 2012, pp. 401–410.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[16] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proc. of ISPASS*. IEEE, 2009, pp. 163–174.

[17] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proc. of IISWC*. IEEE, 2009, pp. 198–207.