Modeling and Design Space Exploration of Storage Processing Unit for Energy Efficiency

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ashwin Nagarajan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Prof. David Lilja

February, 2015

Acknowledgements

I express my deeply-felt thanks to my advisor Professor David Lilja, for his continuous support and thoughtful guidance. From offering chance to work in the SPU project to setting research directions whenever it meandered and to complete writing this thesis, his advice and positivity were immensely helpful.

I thank Professor Pen Chung Yew and Professor David Hung-Chang Du for serving on the final committee and providing insightful comments.

I would like to acknowledge Kevin Gomez and Peng Li from Seagate Technology for sharing their knowledge on SPU and encouraging discussions on various related topics.

I wish to express heartfelt gratitude to other members of SPU project: Manas Minglani, Luke Everson and Sneha Deshpande. It was an honor to work with each one of them. I thank Karthi Subbiah, Vivekanandan Seshadri, Vinodh Kumar, Manikandan Palani, Pushkar Nandkar, Harishankar and Soumya Achanta for being great friends and for their support throughout my Masters.

Finally, I thank my family: my parents S.Nagarajan and N.Hemalatha, my brother Kaushik Nagarajan and my sister-in-law Amritha Mani for their support and encouragement.

Dedication

In memory of my grandfather S.Srikantan (1927 - 2011)

Abstract

Computer architectures in the present era of exascale computing and big data face two major challenges - (i) Increasing gap between processor and memory/storage speeds and (ii) Energy Consumption. A compute system that tightly couples data storage and computation is an attractive solution to mitigate these challenges. By implementing processing inside NAND Flash SSDs computation is moved closer to data. In fact, a computational hierarchy, named Storage Processing Unit (SPU), is formed with processing elements in NAND Flash Memory Controller, SSD controller and host general purpose processor. This hierarchy offers unique opportunities to curtail data movement and reduce energy consumption. This thesis models SPU architecture, explores the design space using carefully chosen applications and associated optimizations to understand and evaluate its energy and performance. Sparse BLAS, BFS, K-Means Clustering and K-Nearest Neighbor are used as benchmarks with energy and performance gains observed up to 11x-400x and 4x-66x respectively.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Design of computer architectures has seen a paradigm shift over the past few decades. While technological advances in manufacturing transistors has led this transformation, several limiting challenges have directed the design of present compute systems. Specifically, processor architectures hit the three "walls" : memory wall, power wall and parallelism wall. Memory wall, caused by difference in processor and memory speeds, led to creation of memory hierarchies, out-of-order speculative processors, newer DRAM technologies and interconnects. Increasing power density and lack of effective cooling solutions resulted in power wall. This effectively saturated the maximum operating frequency of microprocessors and led to multi-core and many-core processors. Even with such many core systems, due to power wall, majority of cores will be dormant unless optimized [1]. These issues, combined with ability of applications to utilize the increasing number of cores to redeem processor costs, presents the parallelism wall. In fact, these challenges are not isolated, they coexist to present an *energy wall*. Energy efficiency is of major concern from mobile phones to data centers. The next generation of computing therefore addresses the question - *how do we design architectures to reduce Joules/compute without sacrificing performance ?*

In mobile phones, need for energy efficiency is evident from battery life concerns in present day smart phones. Advanced functionality and superior performance usually come at expense of battery life. On the other hand, in servers and data centers, the problem of energy efficiency is significant. Empirical evaluations report that data centers in USA consume 1.5% of total electricity consumption [2] and is doubling every five years

[3]. These statistics also project cost of data centers to increase to 1600% between 2005 and 2025 [4]. High-end HPC systems consume several megawatts that is enough to power small towns [5]. John et al [6] analyze characteristics of present architectures to identify sources of energy consumption. They show that energy of actual computation is far less compared to energy associated with moving data to perform the computation. For example, while floating point operations consume about 10.6pJ/op, reading from DRAM accounts for 1000pJ [6]. Engineering FLOPS is not a design constraint, data movement presents the most daunting engineering and computer architecture challenge. Thus, it is important to reduce energy cost associated with data transfers in future architectures.

Figure on the left in 1.1 illustrates the concept of conventional model of computation. With compute engine forming the nucleus of the structure, several layers of memory are built with increasing storage capacities. However, this increase also attributes higher time to transfer data to the compute engine. Thus, cost of data transfer is inherent in this architecture as data needs to be moved from the outermost layer to compute engine. Over the years, architecture optimizations have focused on improving locality, spatiality properties of data access and reducing communication between processing elements. While these optimizations have improved overall performance, movement of data through the system is inherent and cannot be avoided. One solution to reduce this movement is to tightly couple compute and storage. This results in model on the right in 1.1 that shows how compute and storage are distributed in the system. As data is moved from outermost layer, it is capable of being processed at different stages, potentially reducing the amount of data transfered. Thus a system of compute and memory hierarchy is formed.

Different solutions have been proposed depending on technology and type of the storage system. Gokhale et al. [7] designed and fabricated Processor-in-Memory (PIM) chip inside main memory. In another attempt, researchers tried to integrate processing inside hard disks to improve performance of niche applications [8]. However, these designs were not widely accepted because of the complexity of solution or high cost. The advent of flash in storage industry has brought a paradigm shift [9], especially for data centers. Flash offers several benefits, including fast random access, high throughput, and low power consumption, over other technologies. Moreover, flash based SSDs offer serial

I/O interfaces, wider buses for media transfer, and high bandwidths for data transfer [10]. Therefore, flash provides a good scope for bringing computation and memory together. Storage Processing Unit(SPU) is an architecture that incorporates processing capabilities inside a NAND Flash Solid State Drive in addition to SSD controller and general purpose processor. SPU creates opportunity to reduce energy by limiting data movement across hierarchy.

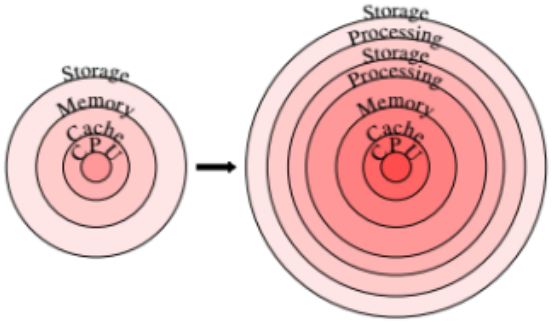

Figure 1.1: Models of Computation

In this thesis, SPU architecture is modeled and performance and energy of applications in conventional and SPU system are measured. A set of applications are identified and main computation kernels in those applications are simulated using this model. Architectural and application space are explored and optimizations to achieve maximum energy reductions are analyzed. Further, a framework to carry out automated design space exploration and multi-objective (performance and energy) optimization of SPU system is presented. Energy savings of up to 11x - 423x and performance gains of up to 4x - 66x for applications including k-means clustering, Sparse BLAS, BFS and K-Nearest Neighbor Search are observed.

This thesis makes following contributions:

- Creation of transaction level model of a compute system based on Storage Processing Unit architecture for measuring performance and energy.
- Identification of applications suitable to be mapped on SPU and different mapping techniques to achieve energy efficiency.

- Propose optimizations within SPU system and investigate its impact on energy and performance.
- Creation of framework to perform automated design space exploration and multi-objective optimization using sparse BLAS as example.

This thesis is organized as follows:

Chapter 2 introduces architecture of Storage Processing Unit. With emphasis on processing inside NAND Flash, characteristics of SPU architecture is portrayed. Feasibility of adding coprocessor and the concept of SPU is extended from previous work by Peng et al.[11].

Chapter 3 demonstrates modeling methodology. Modeling of SPU is explained in detail with data flow verification and analytical validation of the model. Development of entire model is a group effort and my contribution to modeling SSD subsystem is discussed in detail while giving enough background and supported details to understand the model.

Chapter 4 describes mapping of applications on SPU. Data layout, memory requirements, different optimizations and other related properties are elucidated. A classification of applications into bandwidth and compute bound based on these characteristics is also presented. Further, energy and performance profiles of the applications are analyzed in detail. Modeling of K-Means and K-NN include work of other members in the research group and thus mentioned only in brief to complete the discussion.

Chapter 5 presents a summary of all optimizations and discusses importance of multi-objective optimization. Using spmv kernel as an example, framework to obtain Pareto fronts is explained.

Chapter 6 outlines research work related to this thesis. Set of other technologies relevant to coupling processing and memory and the use of NAND flash for processing by other researchers are covered.

Chapter 7 summarizes the thesis and presents thoughts for future work.

Chapter 2

Storage Processing Unit Architecture

Storage Processing Unit is a system with coupled compute and storage components. General purpose processor inside Flash Memory Controllers called coprocessor, SSD controller and host form the compute elements. NAND Flash memory arrays for coprocessor, DRAM, SRAM and register files for SSD controller and host form the memory elements. Since processing in host and SSD controller are prevalent, this chapter emphasizes architecture of coprocessor in the context of SPU system. With a brief introduction to features of SSD relevant to SPU, feasibility and characteristics of coprocessor are discussed in this chapter.

2.1 Overview of SPU

NAND Flash memory is a persistent data storage medium used in devices such as Solid State Drives (SSDs). It is designed using Floating Gate MOSFETs [12] that can be electrically programed to store digital information. This is in contrast to magnetic storage in hard drives (HDDs) that use rotating platters with magnetic heads to access data. NAND Flash memory contains data logically organized in a specific form - it is divided into *planes, pages and blocks*. A page is the smallest addressable unit of read and write. Page sizes are usually in the order of several KBs such as from 4KB to 128KB [13]. A set of pages, in multiples of 16 ranging from 16 to 1024, form a block.

A NAND device encompasses a set of blocks, usually in powers of 2 (e.g. 4096, 8192). Blocks are grouped to planes. For example, 512 blocks form a plane and 4 such planes are present in a NAND device. An 8-bit (sometimes 16 bit) bus, called NAND Flash bus connects the NAND die to external systems. Pages read from NAND Flash arrays are temporarily stored in DRAM inside SSD controller before being forwarded through PCIe interface to host processor.

Parallelism inside SSD stems from organization of NAND device into *dies and channels*. Multiple dies of NAND flash are present in a Flash package. Each die connects through NAND Flash bus to transfer pages of data. A Flash Memory Controller manages command/data requests to and from dies. Dies contend for the bus as data is read from Flash memories. The use of multiple dies helps interleave read and write operations between dies improving overall performance. These multiple NAND dies on a flash package connect to a *channel* in the SSD. Channels are usually referred in this thesis in the context of NAND Flash Package with dies and coprocessor attached to it. Multiple such channels operate in parallel and connect to SSD controller. Typical values of number of dies and channels are 4, 8 and 4, 8, 16, 32 respectively. A timing diagram of read operation utilizing multiple dies and planes highlighting interleaving and parallelism is shown in Figure 2.2 . Plane level parallelism refers to simultaneous read and write on pages/blocks located in different planes of the same die. While a page from multiple planes are read to registers in parallel, they are transfered serially over NAND Flash bus. A Quad-plane die is modeled in this study. Also shown in Figure 2.2, multiple read requests are interleaved to different dies within single flash package. This enables dies to read pages to respective registers in parallel. Only after a complete page from one die is transfered can another die use NAND interface for its transfer. It is important to note that these operations happen in parallel in other channels.

SSD controller implements a Flash Translation Layer (FTL) that performs logical to physical address mapping of pages. Upon receiving commands from host processor, it initiates requests for read/write to appropriate Flash packages. Page allocation scheme used in this study is detailed in Chapter3. This scheme results in a balanced distribution of pages to different flash packages. Due to various reasons like garbage collection, wear leveling, page updates/writes, it is possible that pages are moved from their initial address. Such a movement of pages is not taken into account for this study and is a

part of future work.

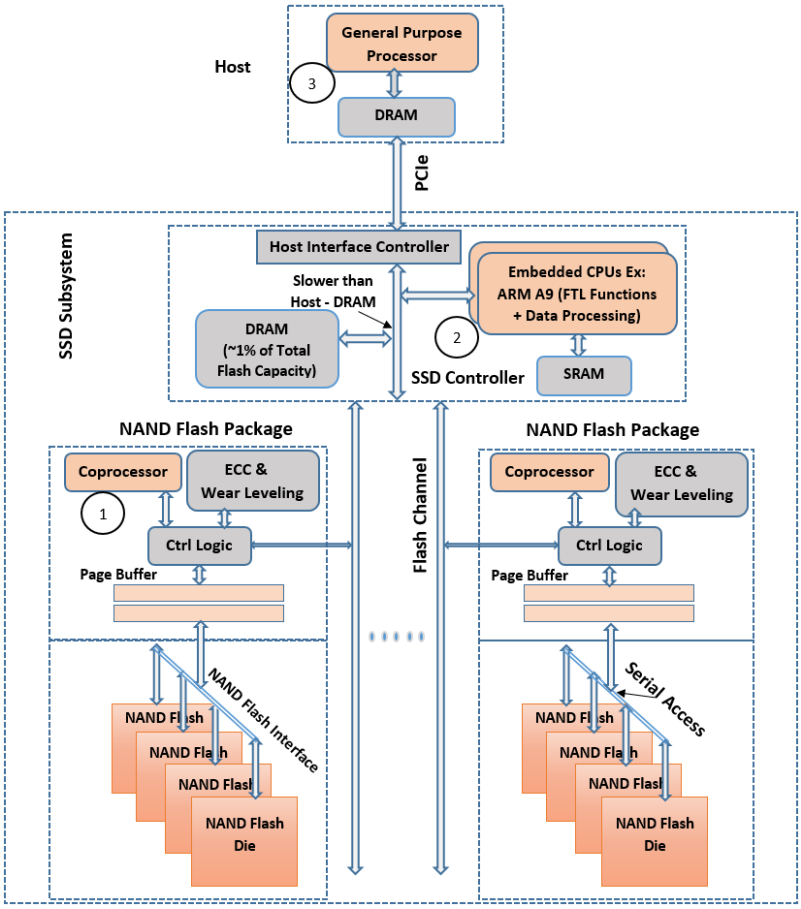

Figure 2.1: SPU Architecture

Feasibility of adding a coprocessor inside NAND Flash package is introduced in previous work [11]. Briefly, die area in a flash package outside NAND dies that implements hardware (Flash Memory Controller) to support ECC and FTL functions such as block management is pad-limited. The die area is driven by number of pins rather than gate counts. This "wasted" silicon is used to implement functional units to process pages of data fetched from NAND dies. Since die area does not change, addition of coprocessor is possible with no additional costs. This functional unit is envisioned as a coprocessor that aids host compute system to implement energy efficient processing. Gate counts of coprocessor is expected to be up to few millions. The coprocessor is assumed to be

implemented using Low Operating Power (LOP) technology (as opposed to HP technology in host) defined by ITRS [14]. At about one-fifth the frequency of a HP technology, the speed of LOP technology matches the transfer rates of NAND Flash interface(2.5 ns per byte). This is important because clocking the coprocessor at higher frequencies without sufficient bandwidth leads to energy-inefficient designs. Further, dynamic and leakage power of LOP technology is only 60% and 5% of HP technology. While reduced frequency of coprocessor is compensated by rich parallelism inside SSDs, extremely low power design can potentially deliver impressive energy gains.

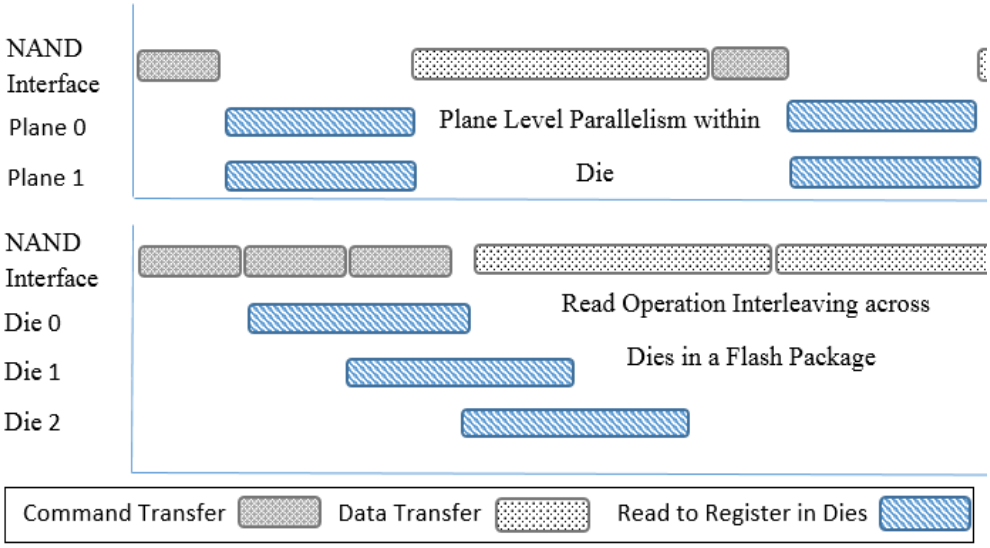

Figure 2.2: Plane Level Parallelism and Die Interleaving for Read Operation

Coprocessor offers earliest opportunity to begin data analysis. Second stage of processing data (if needed) happens at SSD controller Figure 2.1. SSD controller is implemented in current SSDs using embedded microprocessors such as the ARM Cortex [15] or even multi-core controllers (e.g. a 4-core 780MHz controller on the OCZ RevoDrive X2 [16]). Software ecosystem enabling computation in these processors have already been discussed and related works are pointed out in Chapter6.They are attached to RAM modules to store information necessary for implementing FTL, garbage collection, wear leveling etc and to store data temporarily before transferring to Host/NAND Flash channels depending on read/write operations respectively. The size of SSD controller's

DRAM is roughly 1% of the total capacity of NAND Flash memory. The coprocessor and embedded CPUs in SSD controller form two layers of computation in SPU system. Third stage of processing (if needed) happens at host compute system. This model of computation as in Figure 1.1 offers an opportunity to map applications utilizing the distributed compute and memory hierarchy enabling energy efficient processing.

Chapter 3

Modeling of Storage Processing Unit

This chapters begins with a brief introduction to modeling using Intel CoFluent touching the idea of application and platform models. Various *functions* in the model and their interactions are then explained in detail. The chapter continues with a brief account of instruction set and analytical validation of model. Finally, an introduction to kernels modeled in thesis is included.

3.1 Introduction to Modeling Using CoFluent

SPU system is modeled using Intel's CoFluent Studio [17]. Intel CoFluent Studio is a system modeling and simulation environment. Systems are modeled at a transactional level abstraction and represented using graphical user interface and C++. Modeling and simulation library based on SystemC kernel generates simulation traces.

Modeling in CoFluent starts with creation of a Timed-Behavioral Model (TBM) of the system. This step is used to represent different components of system as *functions* . A function's behavior is an application-oriented viewpoint of internal structure of the architecture. It is the logical technology independent representation of the system. For example, floating point add unit, bit shift unit, page allocation are functions in TBM . Several functions can be embedded within a function to create desired functionality.

Functions contain *operations, events, control structures* that define its intended operation. Operations can be thought of different tasks performed by the function. Every operation has an associated *algorithm* that defines its functional or behavioral characteristics. Apart from algorithm, several other attributes can be defined for operations such as time taken to complete the operation, dynamic power and dynamic resource load. Attributes such as cost, power, memory etc can also be defined for functions. It is important to note that functions are technology-independent; their attributes may vary with technology. Several such functions form a system. Functions communicate with each other through *transactions* represented as message queues. In fact, functions communicate in three ways: (i) A shared variable, (ii) A synchronization or event relation and(iii) Data transfer through message queues. Functional structure and its associate behavioral representation is the Timed-Behavioral Model of a system.

Once TBM is verified for correct functionality, a Platform Model is created. The purpose of platform model is to define a physical architecture or a hardware platform that implements functions defined in TBM. Platform model comprises of processors, communication modes that manage relationship between processors and memory. For example, Processor, DRAM, SSD are components in a platform model. Attributes or properties of these components are our main focus - these are the design parameters that decide the overall performance of the system. Examples of design parameters include CPU Frequency, Dynamic Power, Memory Bandwidth, Number of Dies etc . It is always possible to create parameters within TBM (implementation specific) that need not be exposed to architecture model. The Timed Behavioral Model is then *mapped* on to the Platform Model to create an Architecture Model of the system. This step maps functions in application model to corresponding components in platform model. All the functions inherit the properties of respective components. For example, clock frequency of operations inside a function in application model is the CPU frequency assigned to the 'Processor' component in platform model. Further, operations inside functions can be assigned delays in terms of this frequency. Assigning 1 cycle to an operation will inherit the value of one cycle period (e.g. 1 ns) from CPU frequency(e.g. 1 GHz) defined in platform model. When design parameters are duplicated in TBM and platform model, values in the latter assume higher precedence. Thus, once TBM is designed and verified for correctness, design parameters defined in architecture model

can be varied to understand system performance. Modeling of message queues/buses follow a similar procedure. A *send time* or transfer time attribute is used to define time taken for transfer over the bus. Since this transfer time is a measure of amount of data transfered it is necessary to use this amount while defining time attribute. CoFluent allows this by letting user to pass the value using a variable called *USERDATASIZE* which is accessed while setting attributes. Thus transfer time on buses is modeled from bandwidth values defined in designed parameters and from actual amount of data transfer. As a special case, since mapping multiple instances of message queues in TBM to buses in Platform Model created unintended behavior, properties of those buses were modeled on processor modules. It can be noted that NAND Flash bus is a multiple instance message queue that gets mapped to a single bus instance in platform model. While intended behavior is to allow dies in different channels to use its respective NAND Flash bus in parallel, single instance bus in platform model serialized the process. In such instances, either application model is used or components in platform model are used to model bus behavior. Table 3.1 outlines technological and other design parameters used in the model carried over from [11]. CoFluent simulation process explained above only include content necessary to understand SPU model. Detailing all capabilities of CoFluent simulator exceed the scope of this thesis. Interested readers are referred to CoFluent Methodology Guide and other examples provided by Intel [17].

3.2 Developing SPU Model

This section includes details on modeling of SSD relevant to SPU and other components of SPU system. A set of instructions developed for simulating applications on the model is also described. Next, data flow of page read operation is explained with relevant diagrams from CoFluent and insight into how CoFluent reports performance and power numbers. Finally, analytical models that capture trend in performance and energy variation with different design parameters are included to validate the model.

3.2.1 SPU on CoFluent

Two major functions in Application model for SSD include SSD controller and channels of NAND Flash Memories. Each channel has a flash package that contains multiple

Table 3.1: Specifications

Parameters	Values
CPU Clock Frequency	**2 GHz**
CPU Gate Count per Core	200 M
CPU Dynamic/Leakage Power	**5.04 W/0.34 W**
DRAM Dynamic/Leakage Power	0.44 W/0.09 W
PCIe Interface Speed	24 Gb/s
SSD Controller Clock Frequency	1 GHz
SSD Controller Gate Count	20M
SSD Controller Dynamic/Leakage Power	156 mW/1.3 mW
Coprocessor Core Gate Count	1K - 1 M
Coprocessor Clock Frequency	**400 MHz**
Coprocessor Dynamic/Leakage Power	**3.12 uW/67 nW**
NAND ash dynamic/leakage power per die	40 mW/3 mW
Time for NAND ash page read to register	75 us
NAND Flash Bus Bandwidth	2.5 ns per Byte
Page Size	1KB, 2KB, 4KB, 8KB
Number of Channels	4, 8, 16, 32
Number of Dies	4 and 8

NAND dies as functions. Further, each die contains a function for pages of data. Other functions control data flow between SSD controller and actual pages. Coprocessor is also modeled as a separate function inside flash package.

Functions of SSD controller modeled for SPU are (i) handling page read/write and coprocessor requests from host system (ii) processing data from coprocessor before transferring to host. The controller receives a *command* from host that indicates operation to be performed. Command is defined as a *struct* that forms SSD's input parameters. They are: (i) type of operation which is one of read, write, erase or coprocessor (ii) data size - amount of data to read, write, erase or process (iii) Enable/Disable quad plane. Other set of design parameters used by SSD controller are page size, number of channels and dies.

Based on data size requested by host and the design parameters, SSD controller automatically initiates requests to NAND Flash arrays. All data sizes that are a multiple of page size are handled. Page allocation policy is according to [18], with parallelism priority modified as Channel-Pakage-Die-Plane, and modeled as shown in Algorithm1. As shown in Figure 3.1, SSD controller has send path and receive path. While send path caters to correct page assignment, receive path is responsible for transferring data back

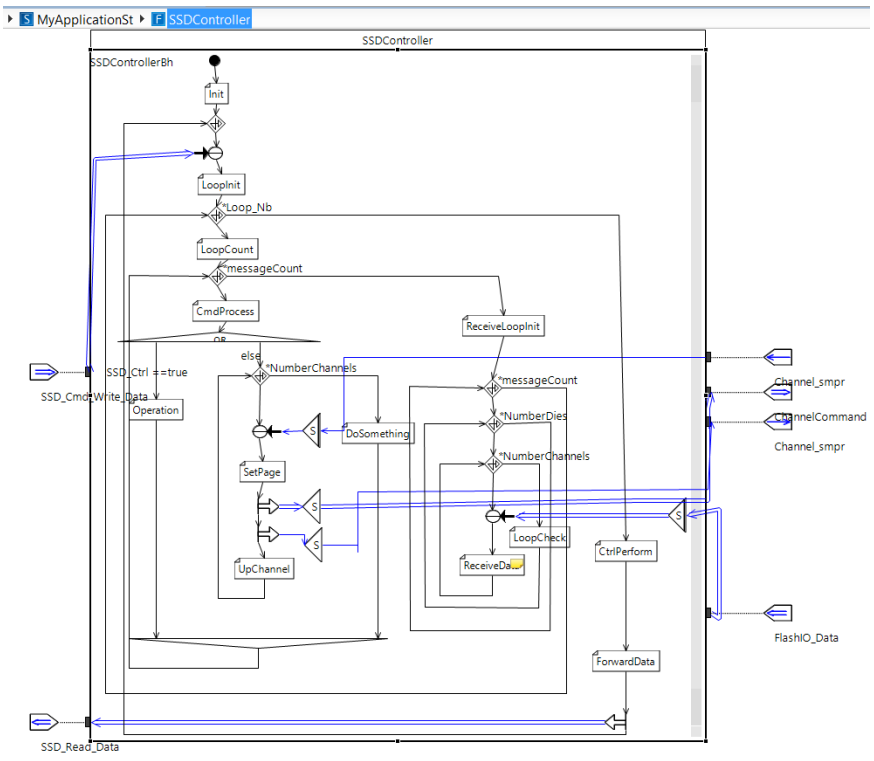

Figure 3.1: SSD Controller Function

to host. It also makes sure that correct amount of data is received from dies. When data size requested is larger than pages that can be read once from all dies and all channels, i.e., data size > (Nb_Channels × Nb_Dies × Page Size), multiple iterations of read from all the NAND dies are required. Thus, the function has to alternate between send and receive paths. Since CoFluent does not allow a function to send and receive simultaneously, the model is made such that SSD controller moves to receive path after every two requests. *CtrlPerform* operation models data processing in SSD controller. The location of this operation in Figure 3.1 indicates that data is processed and then forwarded to host. Time taken for the operation, dependent on the actual operation and data size, is set in *LoopInit* operation.

Commands from SSD controller is received by Flash Memory Controller inside NAND Flash package. The commands are then placed over NAND Flash Bus to be processed by Flash dies. Only one die can receive a command from Flash Memory

Controller at any instant for a given channel. When a die is busy processing the request, command is issued to next die and so on. This functionality is achieved using *Semaphore* operations in CoFluent. Each die receives the command after time to transfer commands to all previous dies has elapsed. Our model assumes a fixed order in issuing commands to dies. This time taken to transfer the command is modeled using NAND Flash bus bandwidth on message queues connecting the functions representing Flash Memory Controller and NAND Flash dies. If there are enough data to be fetched from all dies in a channel, data is read from flash memory to registers present in dies simultaneously. Please note that registers inside dies are different from the registers present in Flash memory controller where coprocessor resides. Since time to transfer command (hundreds of nanoseconds) is negligible compared to time taken to read (thousands of nanoseconds), dies in same flash package read in parallel. Please refer to Figure 2.2 for further details. Please note that time to transfer the command will represent transfer of data for write requests. NAND dies function route the received command to read, write and erase operations. Time taken to perform these operations are calculated dynamically based on data size and assigned to these operations. When this time has elapsed, NAND die function is ready to send data back for a read request. Now a semaphore is accessed to request for data transfer on the bus. When bus becomes available, a lock on semaphore is acquired and data from registers are transfered. After data transfer, lock is released. For a read operation, multiple dies contend for NAND Flash bus as data becomes available. In CoFluent model, multiple function instances of NAND dies attempt to acquire a lock (semaphore) on the shared object (NAND Flash bus). Transfer is complete when all dies in the channel have completed data transfer. This process proceeds in parallel on all channels. Depending on the command, pages read are transfered either to SSD controller or Coprocessor.

Algorithm 1 Pseudo-code for SSD controller Function Page Assignment

Input: Number of Channels, Number of Dies/Channel, Page Size, Data Size

Output: Commands to Flash Dies following Page Allocation Policy discussed in [18]

```
1: Pages_Iteration = (Number of Channels × Number of Dies × Page Size)
2: Total_Pages = Data Size / Pages_Iteration
3: if (Total_Pages > 1)
4:     Number_Iterations = Total_Pages → Number of iterations of read from all dies
5: if (Data Size % Pages_Iteration != 0) → Residual data read separately at the end
6:     Remainder = Data Size - (Pages_Iteration×Number_Iterations)
7: Dies_Count = Remainder / (Page Size×Number of Channels)
8: Dies_Remaining = Remainder % (Page Size×Number of Channels)
9: for (i = 1 to Number_Iterations)
10:    Send → Read Page Size × Number of Dies for all Channels
11:    Move to receive path and return after every alternate Send
12: while (Dies_Count > 0 ) Receive path has similar logic
13:    Send → Read Page Size × (Dies_Count + Dies_Remaining)
14:    Dies_Remaining = Dies_Remaining - 1
```

Coprocessor function models processing inside NAND Flash package. Data flow is such that a page transferred from die across NAND Flash bus is stored in a *buffer* space in Flash Memory Controller. Coprocessor reads a page from this buffer, processes it, transfers results to SSD controller and starts processing next page from buffer. Figure 3.2 shows Coprocessor receiving data from CoProcData message queue. The *Receive_Data* operation is analogous to buffer space; *ProcessData* and *Process* operations model data processing. Time taken to process data is calculated dynamically in Receive_Data based on page size, size of operands and operation being performed. When data is read from all dies, coprocessor will receive a page after a time lapseequal to time taken to transfer a page on NAND Flash bus. Thus, it is necessary for coprocessor function to return to Receive_Data to accept a new page so that NAND Flash bus and NAND die function can complete the transfer and return to initial states. If sum of time taken for process and send operation (which varies with application) is lesser than time to transfer a page, application flow moves to *else* path in 3.2. Data is processed,

sent to SSD controller and next page is received.

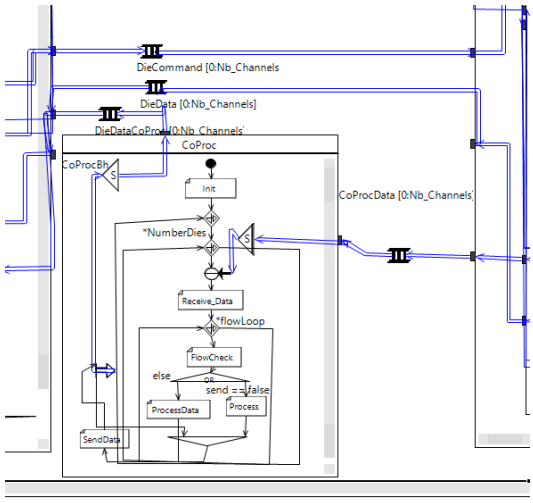

Figure 3.2: Coprocessor Function in TBM

On the other hand, if the sum is greater, it is necessary, in the CoFluent model, to interrupt processing current page to return to receive a new page and then continue processing current page. Depending on actual time values, one or multiple *Send==false* path is/are taken before moving to *else* path to transfer data. Time taken to process a page and amount of data sent from coprocessor are dependent on the application being modeled. These are set and varied manually for every application. Also, it is important to note that while NAND Dies and Channels are multiple instance functions, Coprocessor is a single instance function inside Channels.

3.2.2 ISA

Modeling applications to understand its characteristics requires a mechanism to represent its data flow and operations that can be executed on CoFluent model discussed in section 3.1. We developed a simple instruction set that encompasses primitive arithmetic and control instructions which are listed in Table 3.2. Syntax for these instructions is given below:

< command_name > loops < number > instructions < data size >

where *command_name* can be one of the commands listed in Table3.2, number is a place

Table 3.2: Instruction Set to Run Applications on SPU Model

Command	Description
ssd_nand_read	Reads data from SSD to Host
ssd_nand_write	Writes data from Host to SSD
ssd_nand_perform	Pseudo for other computations
host_dram_add	Add operation in Host
host_dram_mult	Multiply operation in Host
host_dram_perform	Pseudo for other computations
host_dram_write	Write data to DRAM
coproc_cmd_add	Coprocessor performs add operation
coproc_cmd_mult	Coprocessor performs mult operation
coproc_cmd_perform	Pseudo for other computations
ssd_ctrl_perform	SSD controller performs an operation
ssd_nand_loops	Denotes start of a loop
ssd_loops_end	Denotes end of loop
ssd_nand_end	Marks end of program

holder usually 1 and data size is the amount of data to be processed for this instruction. When one or several instructions need to be executed in a loop, following loop syntax is used,

ssd_nand_loops start < loop_count > < loop_identifier > < any no., usually 1 >

.....

.....

ssd_nand_loops end -1 < loop identifier > -1

By using multiple *loop identifiers*, nested loop structure can be formed. Applications are understood and reduced to primitve operations that can be represented using one of these instructions. A script is created capturing data flow and computations of an application. Host function in CoFluent model parses the script and executes instructions one by one.

3.2.3 Data Flow and Analytical Validation

SPU model discussed in section 3.1 is validated using (i) technological parameters used in model, (ii) data flow and (iii) performance and energy trends on variation of parameters specific to SPU. Table3.1 already lists parameters used in the model. In this section, data flow is visualized using application models' timing diagrams and performance, energy numbers are evaluated against analytical expressions.

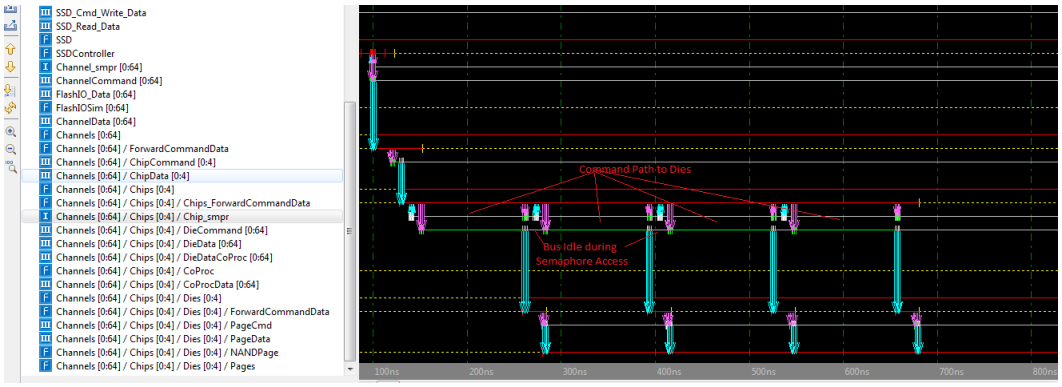

Figure 3.3: Command Flow to Dies in 4 channel, 4 Dies/channel SSD

Timing diagrams in application model helps visualize data flow between functions in the model. Command or page transfer to and from NAND flash dies and processing at coprocessor are highlighted in Figures 3.3, 3.4 and 3.5. Figure 3.3 shows interleaving operations within a channel and parallelism across channels during page read. Since this is a page read command, time to issue is very less compared to actual read time. It can be observed that all four dies read in parallel after four commands are transferred serially over NAND Flash bus. Figure 3.4 portrays data path as pages are transferred back to SSD controller. It highlights the instant when one die (one die in each channel) completes transfer and other die starts transfer of a page. It is important to note that multiple dies in one flash package are ready to transfer a page but wait to get access to bus. Time taken for each function and instruction are direct output of simulation of application model. Also indicated in figures is active/idle periods of the functions - red and green lines indicating active period, white and yellow lines denoting idle time. CoFluent stores these values and provides APIs to retrieve and post-process to compute energy consumption. After simulation is complete, these values are retrieved for each component and total energy is computed in Post-Simulation section of model. Figure 3.5 depicts data flow for a coprocessor command. One page of data is processed in every channel at any instant and when new pages arrive, coprocessor function returns to accept new page before continuing to process current page (3.1) . Active and idle times and power values of coprocessor is added to SSD subsystem's time and power numbers.

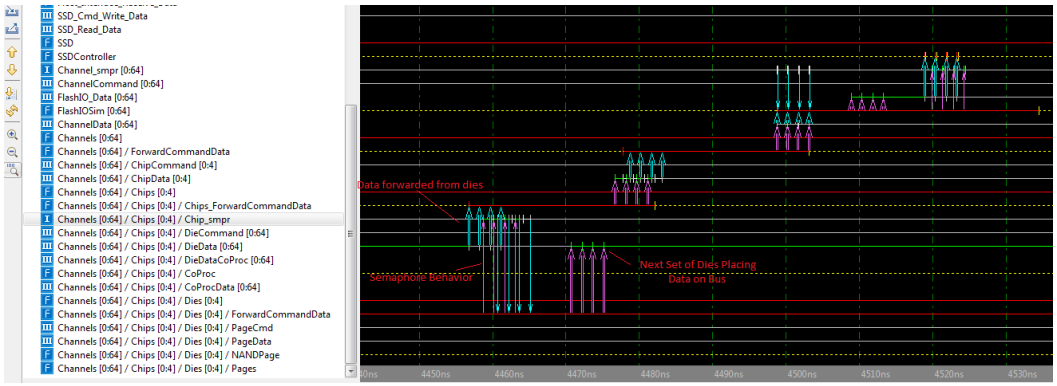

Figure 3.4: Data Flow from Dies in 4 channel, 4 Dies/Channel SSD

Total time to read data from die is used to arrive at analytical expressions to validate the model. Let us consider that total time is the sum of T_{ssd}, T_{pcie} (for transfer between SSD and host) and T_{cpu}. Without loss of generality, for all the cases, datasize is assumed to be an integral multiple of maximum data d that can be read when dies in all the channels read once in parallel and time taken for such a read is T_{ssd}. For example, in a configuration of 4 channels and 4 dies with a page size of 1KB, d equals 4 * 4 * 1024 = 16384 or 16KB. As d is doubled, number of pages read and thus T_{ssd} double. This approximation is true as various other operations of SSD controller such as wear leveling, garbage collection, ECC are not modeled in this architecture study. Since T_{cpu} is negligible for read from SSD, T_{total} can be given as,

$$T_{total} = (T_{ssd} + T_{pcie})\ \alpha \text{ datasize, without pipelining Or,}$$

$$T_{total}\ \alpha \max\{T_{ssd}\ ,\ T_{pcie}\}\ \alpha \text{ datasize, with pipelining}$$

where pipelining implies transfer over PCIe to host while next set of pages are read from SSD. Similarly, when number of channels is doubled, number of pages read simultaneously doubles reducing T_{ssd} by half. However T_{pcie} remains constant as datasize does not change resulting in,

$$T_{total} = T_{total} - 0.5 * T_{ssd}$$

when number of channels is 4, 8, 16 and 32. If number of flash dies in every flash package is doubled, number of parallel reads in every channnel doubles. However transfer of pages from (additional) dies is limited by serial transfer over NAND flash bus resulting in a saving of only the time to read.

$$T_{total}(T_{ssd}) = T_{total}(T_{ssd}) - T_{read}$$

where T_{read} is time to read a page to registers in dies and number of dies are 4, 8, 12 and 16. Now, it is important to note that T_{ssd} consists of two major components and can be given as,

$$T_{ssd}\ \alpha\ \max\ \{\ T_{nfsb}\ ,\ T_{read}\ \}$$

where T_{nfsb} is time to transfer a page on NAND flash bus. This property is exploited in incorporating and mapping applications on coprocessor which is explained in chapter 4. This is also useful in verifying trends on variation of page size and NAND flash bus bandwidth. Writing T_{ssd} in terms of pages and bus bandwidth as,

$$T_{ssd} = \frac{Number\ of\ pages\ read\ per\ channel}{x} * \frac{T_{nfsb}}{y}$$

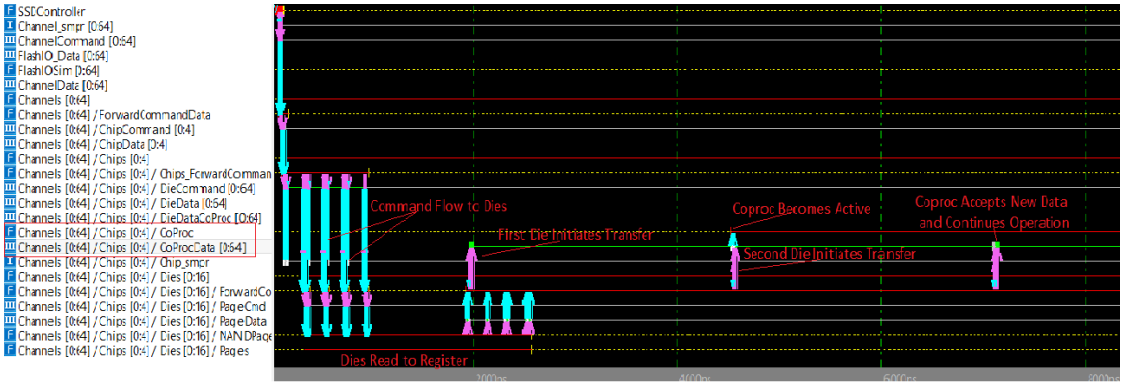

Figure 3.5: Data Flow through Coprocessor in 4 channel, 4 Dies/Channel SSD

When page size id doubled, for example 1KB, 2KB, 4KB, 8KB etc, x and y take values 2 and 1/2 respectively resulting in a constant T_{ssd}. On the other hand, when NAND flash bus bandwidth doubles or T_{nfsb} halves, x = 1 and y = 1/2, 1/4 etc translating to equivalent impact on T_{ssd} . It is very important to note that improving NAND flash bus bandwidth and increasing number of channels provide the best performance. However, adding more channels consumes more power and provide minimum energy savings. Even if doubling NAND flash bus bandwidth comes at a cost of doubling power consumption, increase in overall power for the operation is very less. This is because the contribution of bus power to overall SSD subsystem power is negligible compared to the contribution

of all the NAND flash channels. These trade-offs play an important part in optimizing applications on SPU and are discussed in detail in subsequent chapters.

3.3 Baseline Model

Baseline CoFluent model consists of three major components: Host processor, DRAM and PCIe bus. Transactions in baseline model reflect processing in a conventional architecture. An instruction is parsed, operands are fetched from memory, instruction is executed and results are written back to memory. Since this is an abstract model, memory hierarchies, pipelines etc are not explicitly modeled. Rather, their impact is quantified by varying CPI and using cycle counts appropriate to instructions taken from [19].

3.4 Applications

Five application kernels are modeled on SPU in this study: Sparse Matrix-Vector Multiplication, Sparse Matrix-Matrix Multiplication, Breadth First Search, K-Means Clustering and K-Nearest Neighbor. These applications belong to different domains and have different characteristics and limitations when mapped on a conventional architecture. They have also been chosen to understand and provide opportunity to exploit characteristics of SPU - for, not all applications in these domains can be mapped or can benefit from SPU. These kernels are also part of Rodinia Benchmark Suite [20] designed for implementation on heterogeneous computing infrastructures. Table 3.3 lists few properties of the applications.

Table 3.3: Appliations, Characteristics and Limitations

Applications	Dwarves	Domain	Limitations	Characteristics
Sparse BLAS	Sparse Linear Algebra	Linear Algebra	Intense and irregular memory access, Low Computational Resource Utilization	Memory Bound
Graph 500	Graph Traversal	Graph Algorithms	Communication and Synchronization, Auxiliary Data Structures	Memory Bound
K-Means	Dense Linear Algebra	Data Mining	High I/O time and memory footprint	Compute Bound
K-Nearest Neighbor	Dense Linear Algebra	Data Mining	Run time increases with large data sets	Compute Bound

Chapter 4

Mapping Applications on SPU

This chapter describes how different applications are mapped to run on SPU architecture. This includes analysis of identifying relevant kernels contributing to data movement, type of algorithm, data layout and memory requirements.

4.1 Introduction to Sparse BLAS

Sparse matrix computations are fundamental problems in several computation disciplines. From traditional algebraic operations like linear solvers, multi-grid interpolation to general purpose computing like image reconstruction, graph clustering and large scale modeling for weather and nuclear accelerators, sparse matrix computations are found in myriad of applications. In fact, large scale linear algebra problems constitute an estimated 70% of computing cycles in the HPC ecosystem [21]. Also, dealing with sparse matrix computations offer several challenges to parallel architecture community - Goumas*et.al* define four common problems: memory intensity, indirect memory references, irregular memory accesses for vector and short row lengths [22]. The ratio of computation to communication, inter-processor communication are other typical problems in conventional architectures. These features highlight our interest and motivation in mapping sparse BLAS kernels on SPU architecture.

Sparse BLAS defines a set of routines that operate on sparse matrices. We consider implementation of Level 2 and Level 3 operations which fall into following two computation kernels:

Matrix-Vector Multiply (spmv) : y = (A × x) + y and

Matrix-Matrix Multiply (spmm) : C = (A × B) + C

where A nd B are sparse matrices, x and y are dense vectors, and C is either a sparse or a dense matrix. This chapter covers basics of spmv and spmm kernels, delves into details of mapping on SPU and details performance and energy characteristics with various optimizations.

4.2 Sparse Matrix Vector Multiplication

4.2.1 Introduction and Baseline Model

First step in implementing spmv kernel involves choosing appropriate storage structure for a sparse matrix. Several storage strategies that take advantage of sparsity pattern of matrices have been developed and investigated previously [23],[24] . However, only row (CSR) and column (CSC) oriented storage formats are chosen for implementation on SPU for following reasons: *(i)* Our concern is not to achieve low storage requirements or memory footprints but to understand performance and energy characteristics on SPU at an abstract level. *(ii)* These two configurations provide different communication patterns creating opportunity for multiple computational capabilities and therefore multiple performance and energy trade-offs and *(iii)* Storing data in row and column formats directly map to current implementations at Flash Translation Layer (FTL) inside SSD controller. Matrix is therefore partitioned during *write* by default and available to coprocessors for parallel execution. Software level implementations like decoding storage format, handling array bounds etc are abstracted for both SPU and baseline implementations.

Pseudocode of row and column oriented spmv is shown in Figure 2 and Figure 3 respectively. In Row implementation, while multiplication and addition proceed in parallel between p processors, (Line 5), all processing elements need to access potentially all elements of vector x. A broadcast of *x[j]* is required if it is partitioned among processors. On the other hand, in the Column implementation, each column requires only one element of vector for multiplication. Whereas column vectors within one processing element can be added in parallel, column vectors of different processing elements need to be combined in a final step. These two communication pattern give rise to different

implementations on SPU architecture.

Algorithm 2 spmv CSR cofiuration

A n by n matrix distributed to p processors Each processor owns i rows such that p * i = n

```
1: procedure SpmV_Row_Algorithm
2:     start:
3:     for rows 0 to i in processor p
4:       for columns j = nnz(i)
5:         y[i] = y[i] + A[i][j] * x[j]
6:     end:
```

4.2.2 Mapping spmv on SPU

CSR and CSC Configurations

CSR implementation on SPU follows same data flow as in a conventional architecture. Elements of a row are multiplied by elements of vector and added together to get resultant vector. Matrix elements are read in the form of rows from dies. Number of rows present in a page depends on page size, number of elements and size of every element in the row. For example, two rows are fetched in a page read if page size is 1KB, number of non-zeros in row (nnz) is 128 and each non-zero takes 4 bytes of memory. Page size and nnz are inputs to CoFluent model that can be varied as desired. Storage of vector elements decides how spmv is executed. Since vector elements do not change between multiple read from dies, it is possible to *cache* the vector in coprocessor local memory, as done in conventional architectures. However size of local memory for coprocessor is expected to be limited and therefore caching the vector may not be possible in all cases. This leads to two possible implementations: *CSR-a*) Coprocessors multiply matrix elements with dense vector present in its local memory - vector elements are either read from DRAM in SSD controller which acts as global memory or already present in the local memory (from previous read). *CSR-b*) Matrix elements and corresponding dense vector elements for every row are combined within a page and written to flash package. Thus a page read will have both matrix and vector elements. While CSR-b increases storage requirements due to duplication of vector

elements, it eliminates repeated reads from DRAM as compared to CSR-a. In either case, elements of resultant vector is computed in coprocessors.

Algorithm 3 Algorithm-SpmV

A n by n matrix distributed to p processors Each processor owns j columns such that p * j = n

1: **procedure** SPMV_COLUMN_ALGORITHM
2: *start*:
3: for column 0 to j in processor p
4: for rows i = nnz(j)
5: y_p[i] = y[i] + A[i][j] * x[j]
6: $y = \sum_{x=0}^{p} y_p \rightarrow$ *Sum intermediate column vectors*
7: *end*:

CSC implementation on SPU offers a different challenge compared to CSR implementation. The matrix,stored in column layout, is read in columns from dies. One element of dense vector for every column is also read from dies similar to CSR-b. Resultant column vector from every coprocessor is an intermediate data structure with a size of upto *O(nnz)*. Now, these intermediate vectors are produced for every page read and need to be added together. Coprocessors forward the vectors to SSD controller for addition after every page read or perform addition by buffering one vector of size of upto O(n) in its buffer space. When additions happen in SSD controller, it is pipelined (explained below) with computations in coprocessor. This configuration is referred to as CSC-b. We specifically avoid costly writes (writing intermediate column vector) to dies as it is found to be energy inefficient and undesirable to flash devices. We also consider the case(CSC-a) when intermediate vectors are forwarded all the way to hosts for addition.

4.2.3 Performance and Energy Results

Before discussing performance and energy benefits of different configurations and also various other optimizations covered in this thesis, it is important to understand core

ideas that result in those benefits. A timing diagram with three components of interest to this study - NAND flash bus transfer time (T_{nfsb}), time taken by coprocessor to process one page (T_{coproc}) and time taken to complete processing, if any, in SSD controller (T_{ssd}) - is shown in Figure 4.1 . This diagram indicates that these operations are essentially *pipelined* In other words, transfer over NAND Flash bus, processing in coprocessor and processing in controller form three stages of a pipeline. Total time to compute is determined by the maximum of these three stages. Now, another inference is that $T_{nfsb} >> T_{coproc}$ or, time to compute in SPU is masked by the time to transfer data to coprocessor. This is true for all memory (or bandwidth) bound applications like spmv.

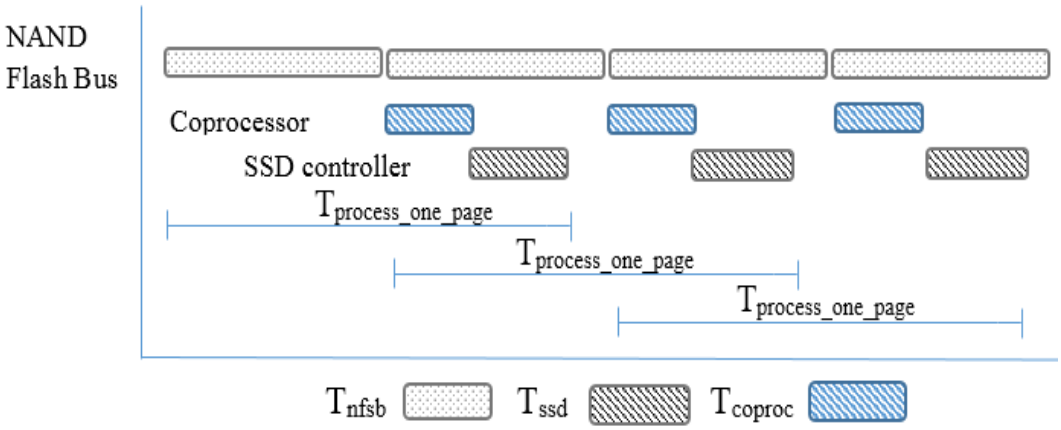

Figure 4.1: Pipelined Processing inside SSDs for Bandwidth Bound SPMV Kernel

Now, for baseline conventional architecture with p cores, for processing one row,

$$T_{\text{cpu}}\alpha((\frac{nnz}{p} \times T_{\text{mult\_cycles}}) + \frac{nnz}{p} + log(p)) \times T_{\text{cpuclk}}$$

where nnz is average nnz/row. This is nothing but total multiplication and add instructions (nnz each) parallelized with p cores. For SPU architecture, assuming a page size of kKB to store nnz and 4 page reads,

$$T_{nfsb} = 2.5 \times k \times 4 \text{ ns}$$

Four page reads are used to model processing from 4 dies in a channel. Also, calculated T_{nfsb} does not change as number of channels is increased. Now, speedup can be defined

as,

$$Speedup = \frac{T_{\text{cpu}} \times CPI\_Data\_Movement \times Number\ of\ Rows(Coprocessors)}{T_{\text{nfsb}}}$$

For example, without any data movement (CPI_Data_Movement = 1), when number of cores in host and number of coprocessors (channels) are both 8 with a page size of 4KB to store 250 non-zeros, conventional architecture achieves 15 times speedup over SPU. However, this case is ideal and SPU achieves better performance for spmv due to following reasons:

- Goumas *et.al*[22] show that spmv kernel contains irregular and indirect memory accesses that are difficult to optimize. Repeated and random memory access does not benefit from cache optimizations. Even with rows partitioned to cores, vector elements are shared and vector lengths are at least as large as last level cache in contemporary machines [25]. Thus CPI_Data_Movement = 1 is very difficult to be achieved for spmv. Performance of conventional architecture and SPU with 8 Channel SSD meet at a total CPI of 30 for spmv.

- Increasing number of channels in SSD increases number of rows processed simultaneously in coprocessors. As explained in [11] and also verified in this thesis, energy of SSD subsystem does not increase with additional channels (or coprocessors) in contrast to conventional system. Hence, for approximately same energy consumption, a 32 Channel SSD offers 4 times speedup compared to baseline at CPI=30.

- Spmv is a bandwidth bound kernel. While number of cycles for computing multiplication and addition is the same for host processor and coprocessor, from Table 3.1 $T_{\text{coprocclk}} = 5 \times T_{\text{cpuclk}}$. Assuming a single core coprocessor, time taken to process a page in coprocessor, $_{\text{coproc}}\ \alpha\ 2 \times \text{nnz} \times \text{CPI} \times T_{\text{coprocclk}}$ (nnz multipliations and nnz additions becomes 2 times nnz). Thus, $T_{\text{nfsb}} > T_{\text{coporc}}$ and changes in CPI of coprocessor does not impact overall performance. In addition, CPI of coprocessor cannot be directly compared with host as transfer of pages to coprocessor is explicitly accounted. Further, CSR-b and CSC configurations have vector elements embedded inside the pages. This isolates issues with irregular and indirect memory accesses.

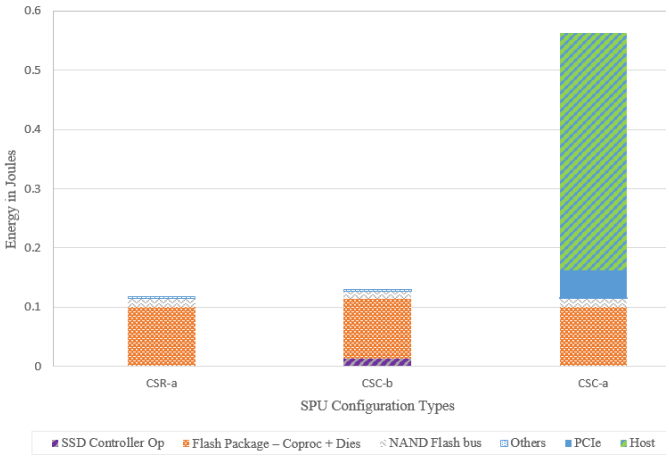

Figure 4.2: Energy Split of Components in SSD in SPU architecture for SPMV

Using non-zeros per row and matrix dimensions from various matrices in University of Florida Sparse Matrix Collection [26], energy consumption of different configurations is plotted in 4.3. Intuitively, since SPU uses low power processor and still achieves speedup over host processor, SPU is energy efficient than host system. While an ideal CPI=1 in host has comparable energy to SPU, any increase in CPI accounting for data movement results in direct energy savings for SPU. Figure 4.2 shows contribution of individual components in SSD. Within a flash package, computation in coprocessor alone takes 1000 times lesser energy compared to reading and moving data to coprocessor. In CSC-b configuration, addition in SSD controller takes 10 times lesser energy than reading and computing in coprocessor. Addition in hosts consumes maximum energy in CSC-a configuration taking it close to baseline energy consumption. From Table 4.1,it is clear that when processing happens entirely within SPU (CSR-a,b and CSC-a), speedup of 2-4x (confirm) is observed whereas CSC-b is barely achieves any speedup due to data transfers for addition operations in host processor. CSR-a achieves better energy savings than CSR-b as reductions in number of pages fetched outweigh reads from SSD controller's DRAM. CSC configuration is less energy efficient than CSR as addition operations happen in SSD controller/host. Energy associated with transfer of intermediate column vectors and their addition is higher than that for performing it in coprocessor. Amount of data transferred from SPU to SSD controller or host is equal

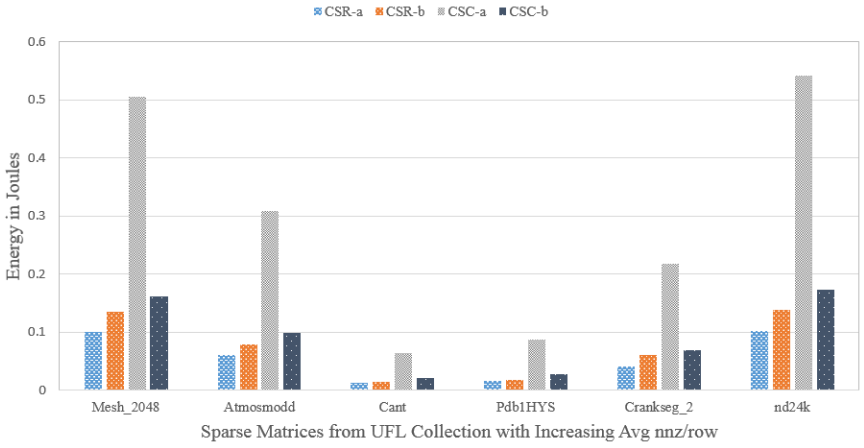

Figure 4.3: Energy Consumption for Different SPU Configurations

to the number of non zeros of the matrix. This is essentially same as in a conventional implementation. Thus one can note that the energy of CSC-b approaches the energy of baseline

Table 4.1: Energy Gain and Speedup for SPMV Crankseg_2 Data Set using in a 32 Channel 4 Dies SPU with Page Size = 4KB

Configuration	Energy Gain	Speedup
CSR-a	40.5	3.95
CSR-b	26.4	3
CSC-a	7	0.89
CSC-b	23.5	2.45

4.3 Sparse Matrix Matrix Multiplication

4.3.1 Introduction and Baseline Model

Matrix Multiplication is organized in several ways to improve time complexity of algorithm. Given matrices A ϵ $R^{m\times k}$ and B ϵ $R^{k\times n}$,each element in inner-product product C ϵ R $^{m\times n}$ usually serving as definition of matrix multiplication is given as,

$$C(i,j) = \sum_{n=1}^{k} A(i,n)B(n,j)$$

This algorithm involves O(mn) operations and is one of the naive implementations. Several optimizations are feasible by taking advantage of sparsity structure, storage characteristics and underlying architecture. However, for reasons established in 4.2.1, we consider inner-product and outer-product implementations on SPU. Since there aren't any restrictions on baseline implementation, sparse 2D SUMMA algorithm [27] is modeled on host processor for comparison. It is important to understand that this thesis tries to explain benefits of controlling data movement and does not necessarily make claims on identifying the best architecture. Outer-product algorithm proceeds by performing cross product of column of matrix A with rows of matrix B after the matrices are partitioned to p processors [28]. Each processor produces an intermediate matrix that is a sum of outputs of all the cross products. These intermediate matrices are further reduced to get resultant matrix.

4.3.2 Mapping spmm on SPU

We start with naive implementation assuming a row based storage for one of the input matrices, say A. The other matrix, B is streamed, in columns, across all coprocessors for every page (row) read. Storing a row/column pair in a single page results in similar implementation. Every row of matrix read from dies needs to be multiplied with every column of the matrix streamed across coprocessor from host/SSD controller. At each instant, the set of rows get multiplied with a set of columns. As only a small fraction of total number of rows can be processed from a page, pages need to be read several times again. Therefore this implementation is only feasible for matrices with small number of non-zeros per row (2-5) and lesser number of total rows/columns in the matrix as observed in Figure 4.4 . When nnz per row is very less, multiple rows are fetched in a page read. Also multiple such pages are processed in parallel in coprocessor reducing number of reads from flash for every column that is multiplied. Implementations of such algorithms inside coprocessor where a large input dataset needs to be read several times is therefore not scalable.

Considering a layout with a column of matrix A and row of matrix B stored in a single page results in outer product implementation. Hence, for every page read, coprocessor performs nnz×nnz multiplications in the worst case. This intermediate output matrix is forwarded to SSD controller which receives nnz×nnz×Number of Coprocessor amount

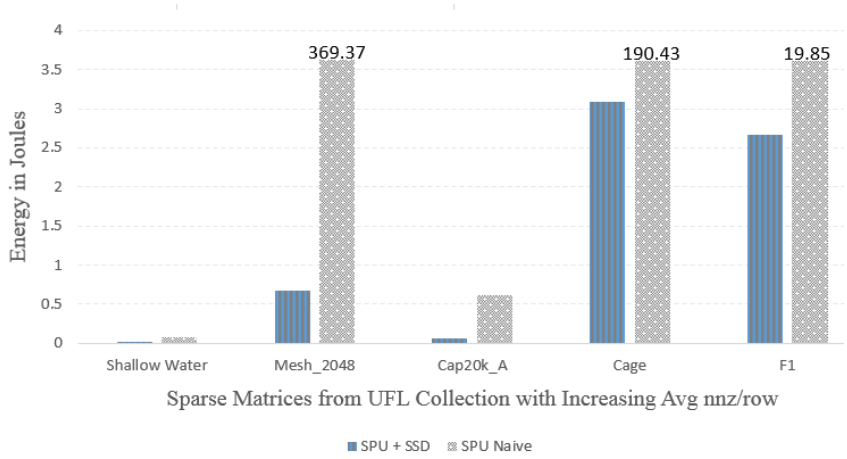

Figure 4.4: Energy Consumption of Naive and Outer Product SPMM on SPU

of data. SSD controller implements addition of these intermediate matrices. Worst case additions in SSD controller is when indices of matrix elements sent by all coprocessors match. As discussed in 4.2.2, these additions are pipelined with coprocessor.

4.3.3 Performance and Energy Results

Energy and performance profile of outer-product model is shown in Figure 4.5 . Important observations from the figure are discussed below:

- Performance of SPU varies with number of non-zeros. This is because T_{coproc} is dependent on nnz×nnz which can become greater than T_{nfsb} as opposed to spmv kernel. For example, assuming a page size of 1KB and average of 40 nnz/row which represents F1 dataset in Univ of Florida Collection [26], T_{nfsb} is 2.5×1024 = 2560ns and

  $$T_{coproc}\ \alpha\ (40\times40)\ \times\ T_{coprocclk} > T_{nfsb}$$

  Similarly, for nnz=2, $T_{coproc} << T_{nfsb}$. Therefore, performance of spmv kernel has a lower bound determined by T_{nfsb} and upper bound determined by T_{coproc} depending on number of non-zeros present in a page.

- Performance of host processor is a combination of computation cost and communication cost both dependent on nnz×n/$\sqrt{p}$. A CPI=30 as taken in spmv kernel is

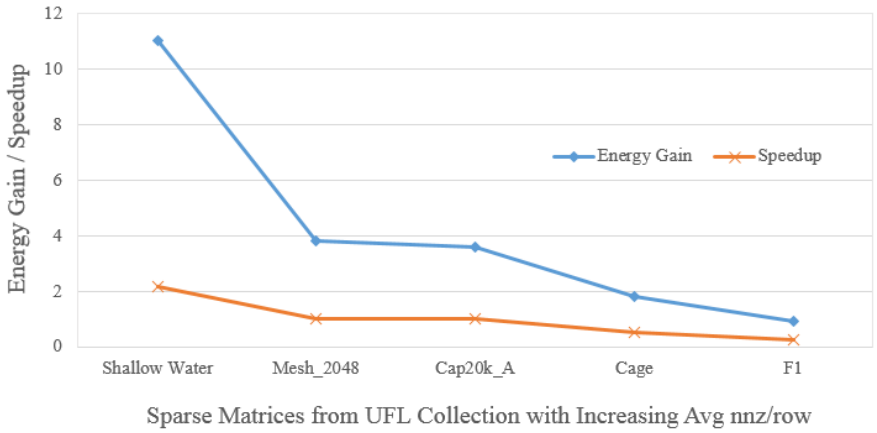

Figure 4.5: Energy Gain and Speedup of Outer Product SPMM on SPU

assumed to account for communication cost. Spmm kernel has low surface volume ratio [28] validating the use of this CPI.

- Speedup obtained decreases as number of non-zeros increases. Assuming a 32 Channel SSD, speedup of a maximum of 2x is obtained for less nnz/row whereas higher nnz/row does not result in any speedup. To improve performance in such cases, a many core coprocessor can be incorporated. We scale performance and power of coprocessors accordingly and estimate maximum speedup attainable. Upto 2x improvement in performance can be obtained by distributing nnz×nnz operations to two coprocessor cores in every channel. This improvement is pronounced when average nnz/row is such that $T_{coproc} > T_{nfsb}$. Performance and speedup plateaus as speedup is determined by max (T_{nfsb},T_{coproc}/Number of Coprocessor cores) for all nnz/row . Increasing number of coprocessor cores results in kernel becoming bandwidth bound.

- Since additions of intermediate matrices from coprocessors are *reduced* in SSD controller, number of SSD cores are varied to ensure T_{ssd} is below max(T_{coproc},T_{nfsb}) to maintain pipelining. As number of channels (or coprocessors) are increased for performance, number of additions in SSD controller scales also increases proportionally. It also varies with the number of non-zeros/row and row/col pairs

fetched in a page. Increasing SSD cores beyond the minimum number of cores to maintain does not provide much benefit. This is expected as either T_{nfsb} or T_{coproc} determines overall performance.

Table 4.2: Energy Split of Different Components in SPU 32Channel 4 Dies and Page Size 1KB for F1 Data Set

Component	Energy Consumption in mJ
SSD Controller Operation	101.4
Flash Package - Coproc + Dies	228.91
NAND Flash Bus	0.0528
Others	3.580

- While spmv kernel obtained upto 40x energy gains, spmm kernel obtains a maximum of 12x gain. As shown in Table 4.2, additional operations in SSD controller consumes half of energy consumption at coprocessor. Also, number of computations in coprocessor for same page size is proportional to nnz×nnz compared nnz computations for spmv. This case indicates that utilizing coprocessor for maximum number of computations if feasible helps achieve higher energy gains. As data from coprocessor keep flowing across compute hierarchy to various processing elements, energy consumption increases.

4.4 Graph500 on SPU

Graphs are used to represent and process many real world phenomena that belong to diverse fields because of its ability to abstractly model entities and their interactions. Graph traversal algorithms are inherently memory and compute intensive and successful parallel implementations of such algorithms involve several challenges [29]. These challenges are only compounded by the recent deluge of data [30]. Graph500 [31] is a benchmark suite proposed to use graph algorithms as a measure of evaluating parallel and distributed architectures. Green Graph500 [32] lists top 500 energy efficient architectures stressing the need for more energy efficient implementations of graph algorithms. Breadth First Search (BFS) of undirected graphs is one of the evaluated kernels in the Graph500 benchmark. There are two kernels in Graph500: first kernel generates a graph defined by its input parameters *scale* and *edgefactor* using R-MAT graph

generation algorithm [33]. Graph contains 2^{scale} vertices and each vertex has average number of edges equal to edgefactor. Adjacency matrix representing the graph is stored for processing by the BFS kernel. We evaluate performance and energy efficiency of SPU architecture running BFS by modeling it on CoFluent.

4.4.1 Introduction: Breadth First Search

One of the widely followed parallel implementation of BFS kernel is Level Synchronous BFS [34]. At the outline, algorithm uses ability to process nodes in the same level in a graph in parallel to improve efficiency over serial version. Level refers to distance of nodes from source vertex. The algorithm starts with a source vertex and proceeds by exploring its neighboring vertices - adding them to list of vertices to be processed in next level and becoming their parent vertex. In subsequent levels, vertices become parents of those neighbors that were not visited before i.e., any vertex can have only one parent determined by order in which graph is processed. The algorithm outputs parent list of vertices and their minimum distance from source vertex. Common characteristics of algorithm is to maintain data structures (queues, bitmap etc [35],[36],[30],[37]) to store list of vertices to be processed in current and next levels, list of visited vertices, and process adjacency list of vertices in the current level in parallel. We abstract actual implementation of the algorithm and focus on the data flow. While it is not difficult to expose the nature of parallelism in BFS, the need to implement synchronized operations to ensure fidelity either through atomic updates or point-to-point communications with other processing elements impacts performance and energy particularly with increasing graph sizes.

4.4.2 Mapping graph500 on SPU

The basic idea of executing BFS kernel on SPU architecture stems from the opportunity to map adjacency list of vertices at a given level in the graph to coprocessors enabling parallel computation. It is assumed that adjacency matrix corresponding to the graph is written to NAND Flash dies by kernel 1 utilizing mechanisms described in 2.1 that exploit parallelism available within SSD. Adjacency list is therefore partitioned across channels and dies and readily available for coprocessors without additional preprocessing

steps for redistribution or partitioning. With this data layout, coprocessors implement a level synchronous BFS by processing edges of vertices in a given level in parallel. Next step in BFS is the synchronization. SSD controller aids the coprocessor in this step. Before delving into further details of BFS model, it is necessary to understand memory requirements of the operation as real world graphs are typically very large and several data structures need to be managed by SPU.

4.4.3 Memory Requirements

List of vertices to be processed in current and next level by each coprocessor, parent list and visited list will be referred using the term Intermediate Data Structures. For levels with more than millions of vertices processed in each level, memory required to store this data structure is at least in the order of few GBs. Parent and visited lists are cumulated set of nodes visited in each level and are several times larger than current or next list of vertices. In addition to intermediate data structures, BFS processing element needs memory to read adjacency list. The understanding that coprocessor private memory is limited is very important and determines data flow in implementing BFS in SPU.The SPU architecture enables different implementations of level synchronous BFS kernel in the way of (i) implementing synchronization, (ii) storing and accessing intermediate data structures. The proposed implementations are approached with a primary motive of reducing data transfer between coprocessors and also between coprocessors and SSD controller whenever feasible. Two cases, considered for comparison with each other and baseline BFS, their associated memory, compute and modeling requirements are discussed below:

Case 1:

Intermediate Data Structures - Storage and Data Flow: In this case, intermediate data structure is read/written only from/to private memory of coprocessor without having to use the dies. SSD controller's main memory holds all the data structures and coprocessor holds only those needed to process the adjacency list fetched from one page of data. This is true because, at any instant coprocessor works only with one page and the process repeats until all the nodes are processed. For example, a 4KB page read from a die could correspond to adjacency list of 128 vertices assuming an edgefactor of

8 and 4 bytes/edge. Assuming 4 bytes/vertex, it takes 512 bytes to store current level vertex list. Storing next level vertices follows a similar procedure with actual memory dependent on the new edges identified. The same discussion however cannot apply to visited list of vertices because of its size.Therefore, coprocessor reads value for a vertex in visited list for every edge from global memory.

Synchronization: SSD controller aids coprocessor in implementing synchronization at each level. It performs an union of next level and parent lists received from the coprocessors at each level. Since it is possible that one vertex is visited by multiple coprocessors in parallel or by the same coprocessor during a different page read, SSD controller removes redundant update and keeps one parent for every vertex. The Graph 500 benchmark allows this relaxation of choosing the parent from a set of parents belonging to the same level to allow maximum parallelism. It is important to note that SSD controller implements this operation pipelined with coprocessor as discussed in spmv and spmm kernels.

Case 2:

Intermediate Data Structures - Storage and Data Flow: In this case, entire visited list is written to dies at the beginning of each level. This offloads repeated read/writes to global memory albeit through costly writes to Flash. It is important to note that directly writing complete visited list inherently distributes it to coprocessors. To make sure all the coprocessors receive the entire visited list, it needs to be replicated across the coprocessors thereby increasing the total data written. The coprocessor reads adjacency and visited lists for corresponding vertices and writes the vertex list for the next level back to the dies. Note that all the intermediate data structures are processed from the dies in this case.

Synchronization: SSD controller implements same operations as in case 1, but only once for every level unless other optimizations result in writing to the global memory in which case operations are similar to case 1. Since this case creates huge amount of writes by modifying pages every time vertex list is updated, we do not consider this case for study of energy efficiency. Increasing amount of writes to flash leads to write amplification [38] and fastens wear out of SSD.

4.4.4 Performance and Energy Results

Multiple parameters are varied to model different aspects of the kernel. CPI of coprocessor is varied to account for number of cycles taken to update parent list of every edge. Values chosen are: 1,5, 10 and 15. A page of adjacency list and visited list are explicitly transferred to coprocessor and thus are not accounted in CPI. Since this transfer is pipelined, it does not affect total performance but account for marginal increase in power. Number of edges in a page discovered is also varied from none to all edges in steps of one-quarter of total number of edges. Number of cycles for edges that are discovered and that which were discovered previously is differentiated. Additionally, SPU and SSD cores are varied similar to spmv and spmm kernels.

Table 4.3: Energy Split of Different Levels of Graph in SPU and Baseline

Number of Vertices	Energy Consumption in SPU in mJ	Energy Consumption in Baseline
7103690	120.96	5363.7
9088766	154.7	6862.57
130298	1.86	98.3
109239	2.22	82.4

- Similar to spmv kernel, performance of BFS kernel is bounded by T_{nfsb}. This is because number of cycles to update parent vertex of an edge for all edges in a page is very less compared to T_{nfsb}. This is true for different number of cycles/edge modeled.

- Performing synchronization operation is not a bottleneck as in conventional implementation. While host processors synchronize through costly broadcasts (or communication) of frontier vertices, SPU uses SSD controller. Discovery of edges and formation of new vertices for subsequent reads from pages are done in parallel. Also, even with many number of edges in one level, synchronization is limited to number of edges discovered in one page $\times$Number of Coprocessors.

- Processing in host processor involves update of edges for vertices in current level and communication/gather of vertices for next level. With multiple reads/writes and broadcasts and increasing sizes of number of vertices in each level becoming larger than size of memory, energy due to data movement is high.

- Table 4.3 shows energy consumption in conventional and SPU systems across different levels in a real world graph taken from [36]. Also, Figure 4.6 plots energy with coprocessor and SSD core variation. Only marginal difference in energy is observed as coprocessor cores are increased. This is expected as BFS is bandwidth bound. Increasing SSD cores beyond what is required to achieve pipelining leads to diminishing returns. This is also expected as overall energy is determined by transfer of pages on NAND flash bus.

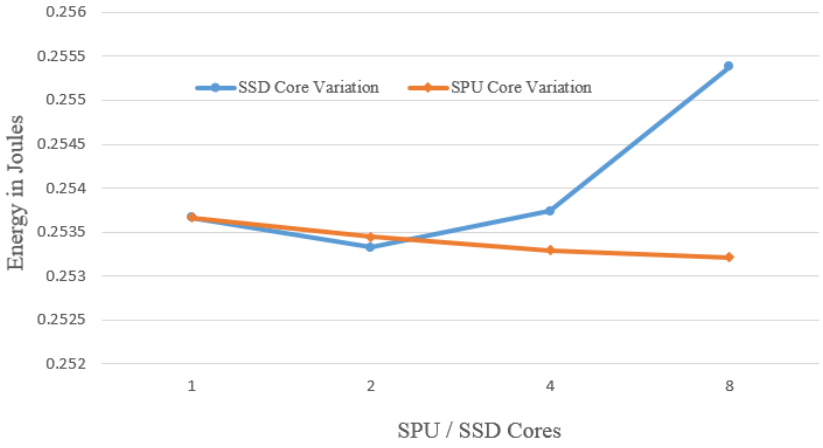

Figure 4.6: Energy variation with Coprocessor and SSD cores for BFS

4.5 Data Mining Applications on SPU

In this section, modeling of two data mining kernels - K-means clustering and K-Nearest Neighbor (k-NN) are discussed. Motivation to model applications from data mining stems from the problem of big data. Data centers and cloud services now work with peta and exa scale data sets. With more and more devices connected to Internet - smart phones to sensors to medical instruments - data in cloud is growing leaps and bounds. Energy efficient processing of such data sets is an open research problem and solutions and ideas from entire system stack is necessary. Moreover, as applications using these data sets have varied characteristics, different *accelerators* aid general purpose processor to achieve desired performance and power budgets. We envision use of SPU as one such

device enabling energy efficient compute of big data applications. Intelligent SSDs for query processing and other common operations such as sort, aggregate have already been studied (refer Chapter6). Applications considered in this chapter have computations and data flow that are different from these and also from sparse computations and graph traversals already considered in this thesis.

Following sections cover basics of k-means and k-NN, mapping of these algorithms on SPU, associated optimizations and their performance and energy characteristics. A brief account of other relevant applications considered for modeling on SPU is presented at end of the chapter.

4.5.1 K-Means Clustering

Clustering is a procedure to group objects into clusters. Each cluster has a meaning specific to the type of objects and their characteristics. An observation of radius of n circles can be grouped into clusters of circles with similar radius ; an observation of heights of students in a class can be grouped into clusters of similar heights. Algorithm takes N observations and K cluster values as input and outputs K centroids. Centroids are cluster centers that minimize sum of squared distances of each observation to it *closest* cluster center. In this study, we model K-Means algorithm discussed in [39]. Briefly, the objective is to minimize squared error function i.e., square of distance between observations and cluster centroids. This function is given as,

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} ||x_{\mathrm{n}} - c_{\mathrm{k}}|| \times ||x_{\mathrm{n}} - c_{\mathrm{k}}||$$

where x_{n} is observation, c_{k} is cluster center. For a cluster k, μ_{k} is defined as the mean of observations that belong to that cluster and given as,

$$\mu_{\mathrm{k}} = \frac{1}{N_{\mathrm{k}}} \sum_{n \epsilon c_{\mathrm{k}}} x_{\mathrm{n}}$$

Algorithm proceeds by assigning observations to closest cluster center. After all observations are assigned, cluster centers are reevaluated. The process repeats until a threshold is met. Latest cluster centers are the centroids.

4.5.2 Mapping K-Means on SPU

Observations are written to flash exploiting existing parallelism in SSD to different channels. They are therefore assumed ready for processing by SPU. We do not use actual values of clusters and observations, but rather model data flow and computations. We assume that all coprocessors are initialized with same cluster centers. Computations involved are given by lines 3 to 8 in pseudo-code. Coprocessor reads a page of observations from dies, computes J, determines closest cluster center and calculates sum and number of observations belonging to each cluster. This happens in all coprocessors. Now, to calculate μ_k, coprocessor maintains a data structure of size of O(K) to store sum of observations. Similarly, another data structure of O(K) is maintained to store count of number of observations in each cluster. Both these data structures are updated every time an observation identifies its closest cluster center. Coprocessor also stores updated values of centroids. Since value of K is very less (64, 128 are typical in large data sets), storing O(K) elements in coprocessor memory is assumed to be feasible. After every observation is read and cluster center is computed, each coprocessor is left with its own cluster centroids. It forwards sum, count and J values from its memory to SSD controller. SSD controller implements a merge of these individual centroids to get single set of cluster centers that are sent back to coprocessors for next iteration. SSD controller also determines if threshold (J'- J) is met.

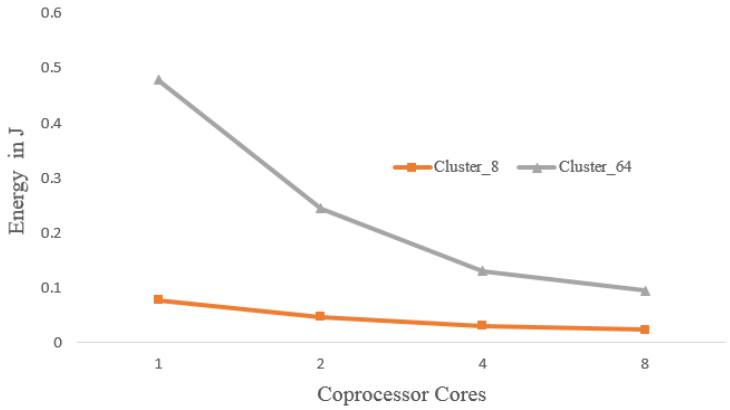

Figure 4.7: Energy variation with CCoprocessor Cores for K-means

Assuming observations of 100000 and 8,64 cluster centers and 36 iterations as used in [40], performance and energy consumption with varying number of coprocessor cores is plotted in Figure 4.7. With configurations of baseline and SSD same as discussed in previous chapters, speedup of a maximum of x and energy gains of up to x is obtained. Decomposing various components contributing to total time will facilitate understanding reasons behind performance gain. We find that to perform computations to calculate J and μ_k , we see that $T_{coproc} >> T_{nfsb}$. This is intuitive as calculating J for one observation alone involves k multiplications. For example, with a page size of 1KB containing 128 observations, 8192 multiplication operations are done with number of clusters being 64. Also, T_{ssd} for compute is also pipelined when merging cluster centroids. Now, given a set of input observations, coprocessor and host processor take equal number of cycles (nk multiplications and other operations) to compute a cluster center. T_{coproc} can be expressed as,

$$T_{\text{coproc}} = T_{\text{cpu}} \times Host\_CPI \times \frac{Number\ of\ host\ processors}{Number\ of\ Coprocessors} \times \frac{Tcpuclk}{Tcoprocclk}$$

This can be interpreted as: while number of host and coprocessor cycles are same, number of observations processed by coprocessor is scaled down by number of channels in SSD at 5x slower clock. It can be noted than speedup increases with host CPI which accounts for data movement and number of channels in SSD. Since T_{nfsb} and T_{ssd} are pipelined with T_{coproc}, total performance is dependent only on T_{coproc}. This case clearly portrays that if number of computations that can be performed in coprocessor given a limited page size is much higher NAND flash bus bandwidth (compute bound) and comparable to that of host, then significant performance gains can be achieved without increase in energy of the system. This characteristic of k-means makes it different from other kernels modeled before in this study. Above discussion also leads to reasoning of energy gains of k-means - incorporating multiple low processor coprocessors inside SSDs that provide enough parallelism to offset its low frequency of operation. Doubling coprocessor cores directly impacts T_{coproc} and hence overall performance and energy of SPU model. As discussed in spmm kernel in 4.3.2, performance and energy plateaus as T_{coproc} becomes less than T_{nfsb} .

4.5.3 k-Nearest Neighbor

K-NN is one of the most fundamental classification algorithm which does not require prior knowledge of distribution of data. The training phase of the algorithm involves storing the data points and its labels [41]. Every data point in the training set contains a set of vectors and a label associated with it. Distance between input and every point in the dataset is computed. The 'k' smallest distance measures corresponds to elements that influence classification of unknown sample. Selecting a suitable neighborhood value 'k' and distance metric applied determines classification performance. K-NN might appear to be advantageous when training data is large as there is minimal cost for training phase. However, it might suffer from costs in terms of computation and run time if not optimized suitably. We use euclidean distance in 2 dimensional space as measure of closeness between input data and training set. However, the model can be extended to include different distance measure and higher number of dimensions.In summary, we are trying to map distance computation and sorting of resultant values to find k parameters on SPU architecture. Input dataset for is taken from OpenStreetMap project [40]. The 2-dimensional dataset contains road network for 50 states and occupies 6.6 GB.

4.5.4 Mapping K-NN on SPU

In a naive implementation of knn algorithm on SPU, distance computation is carried out by coprocessors and sorting by SSD controller cores. Each coprocessor has input data point in its private memory and training data points are read from flash. We assume 8 bytes for each coordinate and 4 bytes for ID totaling to 20 bytes. Therefore, for a page size of 1KB, each coprocessor computes approximately 50 distance measures. These values are forwarded to SSD controller for sorting. It repeats sorting of distance measures received from coprocessor after processing of every page. We assume merge sorting with a time complexity of (N/P)log(N/P) + Nlogp. (N/P)log(N/P) term refers to sort distance values for each coprocessor, N logp refers the time to merge k×Number of Coprocessor values to k values. As modeled in other kernels, we increase SSD cores such that T_{ssd} is within max (T_{coproc}, T_{nfsb}). As SSD controller starts sorting of distance measures, we model it to choose 'k' closest parameters for every sort. These 'k' values are

sent back to coprocessors. Now, when each coprocessor computes a distance measure, it compares calculated value with the maximum of k values received from SSD controller. This way amount of data forwarded from coprocessor is reduced as values larger than the maximum need not be forwarded for sorting. We expect that in such an implementation while most of the distance values are forwarded from coprocessors initially, it reduces gradually as algorithm starts to converge. Even in conventional implementations several heuristics are applied to reduce distance computation and sorting for large data sets.

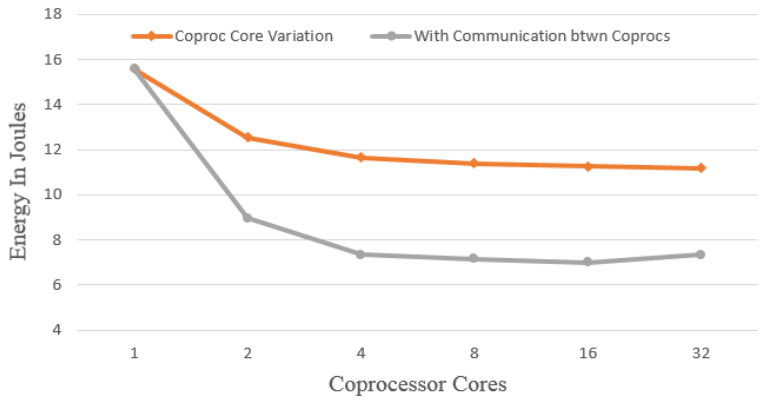

Figure 4.8: Energy variation with Coprocessor Cores and Communication for K-NN

Similar to k-means, reducing amount of data processed by coprocessor with increasing number of channels when compared to host processor results in higher speedups at constant energy benefits. Assuming a host CPI of 30 gives speedup of x and energy gains of x with a 32 coprocessor SSD. Since k-NN is compute bound, increasing coprocessor cores parallelizes distance computations achieving increased speedup and energy gains until T_{coproc} becomes less than T_{nfsb}. While this benefit appears due to data points having only two dimensions as lot of distance measures can be computed from single page, it can apply to other data sets as well. This is because increasing dimensionality increases time to compute distance. Figure 4.8 shows energy savings with increasing number of SPU cores. Another optimization considered is to move sorting to coprocessors. The main idea is to use coprocessor to compute distance measures and then sort them to obtain k closest values. These k values are sent to neighboring coprocessor to

sort 2k elements. This process is repeated till all the elements are sorted to get only k elements. At any instant each coprocessor sorts a maximum of 2k elements and it takes log(Number of Coprocessors) sort steps. From Figure 4.8 we can see that, this optimization yields better energy savings compared to sorting in SSD controller. The coprocessors can share data among each other in two ways: through Flash Translation Layer (FTL) of SSD controller or through a light-weight network between coprocessors. We assume the latter case. This network should be used for light data trafc and not lead to additional writes to flash. The detailed analysis of a light-weight network for coprocessors is a part of future work.

Chapter 5

Optimizations

In previous chapters, several applications were mapped on SPU architecture, different optimizations were identified and analyzed for impact on energy and performance. In this chapter, these parameters are considered in unison and automated exploration using CoFluent is presented. Characteristics of SPU that help decide type of application that can be mapped and nature of mapping are also discussed. A summary of optimizations is given below:

- Varying coprocessor cores to expose parallelism available within data in a page is one of the important optimization that resulted in impressive results. K-means and K-NN kernels being compute bound offer maximum scope with additional cores while spmv and BFS kernels are limited by bandwidth of NAND flash bus. This is an effective optimization because additional gates maximize use of available silicon providing energy savings as coprocessor operates at low power.

- As number of coprocessors increase, it is possible for SSD controller to become a bottleneck in the system. This is true especially in cases like spmm where coprocessor's output data is larger than a page. Therefore, it is important to set the right number of cores such that operations are pipelined with coprocessor. It is also noticed that increasing SSD cores beyond this number results in diminishing energy benefits.

- To maximize computation within flash package or limit data transfer, coprocessors can communicate data between each other. As shown in K-NN algorithm, moving

sorting operations from SSD controller to coprocessors increased energy benefits. However, it is important to make sure that such communication and subsequent computation is completed without affecting pipelined operations.

- Increasing number of channels (or coprocessors), directly results in proportionate performance benefits without any increase in energy consumption. This is in complete contrast to a conventional system where increasing number of cores attempt to increase performance sacrificing energy consumption.

- Mapping certain kernels also involve modifying data layout and storage to make use of processing within flash. Need for such modifications is largely due to limited private memory for coprocessor, avoiding repeated read requests to global memory in SSD controller and writes to flash. While row and column oriented formats in spmv offered different communication patterns, CSR-b configuration modified storage of vector elements. Significance of understanding memory requirements and managing different data structures was shown with BFS kernel as an example.

5.1 Automated Design Space Exploration

Traditionally, problem of optimization has been formulated based on minimization of single objective. In most cases, performance is considered as the objective [42]. However, as computer architectures evolved, it is imperative to simultaneously optimize for wide variety of objectives such as energy consumption, resilience, cost, code complexity apart from execution time [43]. These objectives are also possibly competing and hence compounding optimization problem. For example, increasing processor frequency increases temperature affecting resiliency, ineffective mapping of application to processor affects energy, distributing an application between different accelerators impacts code complexity and so on. Thus it is necessary to consider all the factors in unison to evaluate system configurations. This is referred to as Multi-Objective Optimization. When objectives are considered together, often there may be a set of optimal solutions possible. These solutions are said to be on a Pareto front i.e., when two or more objectives are simultaneously minimized, it may not be possible to improve one without degrading other objective. Execution time and power on a conventional processor is a

classic example. Formal definition of Pareto-front from [43] is given below for reference: *If $F_1(x)$, $F_2(x)$,...$F_p(x)$ are p objective functions, F(x) is function to be minimized, we say that F(x)≤F(y) if Fi(x) ≤ Fi(y) for all i = 1,...,p, and F(x) ≠ F(y); in this case y is dominated by x. We say that a point x ∈ X is Pareto optimal or non-dominated, if there is no y ∈ X with F(y) ≤ F(x). Set of Pareto-optimal points is denoted by $X^* \subseteq X$. The set of objective function values of all Pareto-optimal points, $F^* = F(x) : x \in X^*$, is called the Pareto front.* Analysis of Pareto front provides better insights than single objective formulation. It helps in identifying different optimal configurations within our region of interest.

5.1.1 Simulator Setup

To study optimization on SPU, it is first necessary to create a framework for automated design space exploration.In this study, design space is explored on CoFluent SPU model by automated variation of different user defined parameters summarized above. For all possible parameter values, performance and energy numbers are captured. These values are also used to find the nature of Pareto-front for SPMV with performance and energy as objectives. First, an exhaustive search is initiated by varying Number of Channels, coprocessor cores and data layout for SPMV. Varying data layout refers to simulating CSR and CSC configurations in the SPU model. In CoFluent, a design parameter is created to point to these configurations. Now, for each of these configurations, time to compute in coprocessor, SSD and host (if needed) changes.
Also, amount of data transfered from coprocessor to SSD controller or host changes. Depending on the type of configuration, appropriate expressions to calculate these time and data values are included in the Timed Behavioral Model.

5.1.2 SPMV Pareto Front

Figure 5.2 shows Pareto Front of performance and energy obtained for SPMV kernel from an exhaustive search with Number of Channels (4, 8, 16 and 32), Number of Coprocessor Cores (1, 2, 4, 8, 16) and configurations (CSR-a, CSR-b and CSC-a). It is clear from the figure that the set of optimal solutions are from CSR data layout. CSR-a configuration completely *dominates* other solutions. Further, Pareto Front is formed

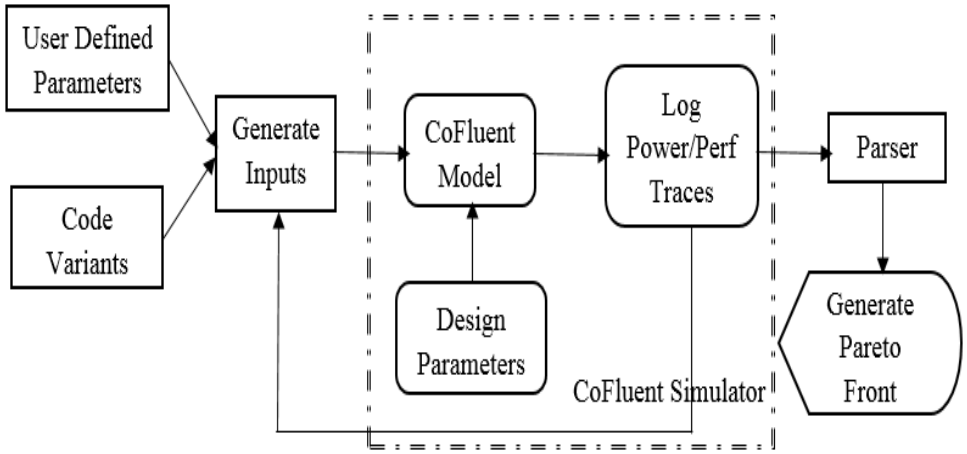

Figure 5.1: Automated Design Space Exploration using CoFluent

with energy and performance values obtained by varying number of channels of SSD for CSR configuration. With equally distributed workload to coprocessors assuming no movement of pages between flash packages(or channels), increasing number of channels improves performance at minimum increase in energy. This trend is observed across different number of average non-zeros per row.

5.2 SPU Architecture Characteristics

- **Coprocessor memory**: Coprocessor is designed to function within the flash memory controllers and available additional silicon is ought to be used for implementing processing functions. Thus *local* memory of coprocessor is expected to be limited to few pages. This memory is a buffer to data moving across the controller. Flash arrays act as *persistent* memories to coprocessor. While data in pages are read from flash arrays to coprocessor, writing to flash memory is avoided. Also, it is better if pages are not moved between flash packages to maintain workload balance and minimize communication costs. DRAM in SSD controller acts as *global* memory. This memory model can be compared to other architectures like host general purpose system or a GPU.

- **Data layout and writes to Flash memory**: If a matrix is divided into pages by

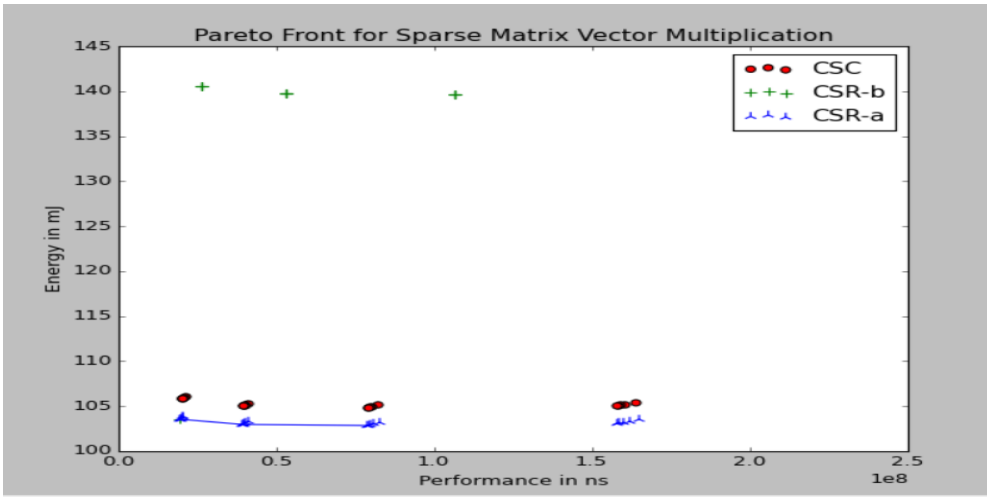

Figure 5.2: SPMV Exploration-Varying number of channels for CSR-a configuration forms Pareto front

rows and an applications accesses rows and/or columns simultaneously to perform an operation, it becomes inefficient to be implemented inside SSD. Further, whenever feasible, it is necessary to store data in pages such that maximum parallelism can be extracted within data in a single page. Also, if multiple reads of the same data set is needed for implementation in coprocessor as opposed to single read in host system, then such reads can lead to inefficiency. This is because the slower NAND Flash interface and coprocessor speed are amortized with die interleaving and channel parallelism. Repeated reads discount that advantage. Absence of in-place updates in flash, long write times and write amplification restrict use of coprocessor to applications that lead to constant update to input/auxiliary data sets.

- **Synchronizations in SSD Controller**: SSD controller can aid *loosely connected* coprocessors by synchronizing their outputs and/or producing auxiliary outputs used by coprocessor for processing subsequent pages. Further as this synchronization can potentially be pipelined with other coprocessor functions, impact of communication overhead on performance can be greatly reduced.

- **Pipelined Processing**: Transfer of pages from NAND dies, compute in coprocessor and SSD controller are pipelined. Hence maximum efficiency can be obtained by minimizing the difference, T_{nfsb} - T_{coproc} and ensuring that SSD controller operations are lightweight and does not affect pipelined processing. When host is involved in processing output from SSD, transfer through PCIe and processing and host are again pipelined as in conventional architecture. In such, it is even possible to use a low power processor when transfer through PCIe determines overall performance to maximize energy efficiency.

Chapter 6

Related Work

This chapter covers research work relevant to this thesis i.e., active disks, active flash and intelligent SSDs and other near data processing schemes such as Processor-In-Memory. Active disks section discusses work on processing capabilities in hard disks, its favorable and limiting features . Following active disks, next section presents an extensive account of active flash and intelligent SSDs, compares and distinguishes it with concepts and ideas explored in this thesis. Final section throws light on other near data processing efforts.

6.1 Active Disks

Active storage concepts that move computation towards dates back to Gamma databases in early 1980s [44]. But research on shifting computation to storage controllers inside the device were first shown by IDISK [45] and Active disk [46]. IDISK for databases offloaded data processing from desktop processors to lower-power processors to improve cost-performance. Active Disk combined on-drive processing and large memory to allow disks to execute application level functions in the device. Most work exploited excess computing cycles of the hard disk drive controllers embedded processor for useful data processing. Acharya et al. focused on the programming model and algorithms rather than device architecture [47]. They proposed a streaming programming model for application development, model for secure task execution and operating system support at host and device. SmartSTOR [48] consisted of a processing unit coupled to one or more

disks and semantically smart disks [49] were capable of various high-level functionalities. Mueller et al. introduced an external module (e. g., FPGA) attached to a disk to implement system intelligence [50]. Teradatas Extreme Performance Appliance [51] and Oracles Exadata [52] added complex processing into their storage servers. The demonstrated benefits from active disks were primarily from parallelized data-local execution for many database related computing activities. These benefits have been realized by spreading stored data across commodity servers and partitioning data processing across those same servers. While active disk approach is not a failure nor is obsolete, it didn't take off due to limited storage interface, overheads of changing database software, high manufacturing cost (of adding additional hardware for processing) and increasing real time demands.

6.2 Active Flash and Intelligent SSDs

Due to the disadvantages listed above and to exploit characteristics of Solid State Storage devices, recent research developments have targeted processing using NAND Flash based SSDs. Modern SSDs are efficient in concurrent random writes, and have powerful processors, memory, and multiple I/O channels to ash memory, enabling in-storage processing with minimum hardware changes [53]. In addition, offloading I/O tasks allows a host system to fully utilize devices' internal parallelism without knowing the details of their hardware configurations. Active Flash, Active SSD, Intelligent or Smart SSD, In-Storage Computing, In-Situ Processing using SSDs etc are common terms referring to the change in computing paradigm. Within this idea, there are two approaches: first approach uses computational capabilities attached to an SSD in a server farm or a cluster or a high performance computing infrastructure. For example, Ridel et al. added application programmability on disk resident CPUs [54], John et al. used parallel file system storage nodes [55] and Andersen et al. developed a FAWN architecture [56] with low power wimpy nodes connected to Flash drives in a ring. Further Ding [57] implemented a smart storage system in a high performance cluster that allow programmers to write Map-Reduce like code which can automatically offload data-intensive computation multi-core processors in the storage node. However these systems had could not meet performance expectations and also suffered from scalability problems [58]. Need

for distributed protocols and lack of portability were some of the issues with such an architecture. This approach different from this thesis as processing inside SSDs, rather than using compute nodes attached to permanent storage, to reduce data movement is the main concern.

Second approach involves processing inside SSDs which follows same contours as this thesis. Tiwari et al. [59] studied processing in SSD controller in high performance computing (HPC) environment on large-scale supercomputers. They analyze energy and performance models of active SSDs and utilization of multiple SSDs on supercomputers. They also study policies for scheduling computations on SSDs such as offline, during GC, after GC etc. and consider cases where data output rates impose real-time constraints on active computation. In contrast this thesis models a single ash device with FMC and SSD controller acting as compute nodes and also studies a single SSD connected to host rather than an entire HPC workow. Do et al. [60] ported query processing (selection, selection with aggregation and TPC-H queries)components into SSDs. They provided simple session based APIs to manage commands from the host, threads for query operation, memory inside the SSD, and data in the DRAM. They showed 18.7X and 2.9X energy gains for the entire system with the aggregate query. They also highlighted issues such as buffer pool caching and coordination with DBMS transaction manager. Unlike this study, their performance is limited because of low frequency embedded CPUs in SSD controller. Kang et al. [53] constructed a prototype implementing smart SSD model on a real SATA-based SSD. They used an object-based protocol for low-level communication with the host, and extended Hadoop Map-Reduce framework to support smart SSD. Their experiments revealed a reduction of 50% in energy consumption. They also found that current SSD architecture is not sufficient to support complex tasks. In contrast this thesis is not restricted to map-reduce type of workloads and showed that processing inside flash memory controllers can help achieve greater benefits. Lee et al. [61] focused on improving the external sorting algorithm which is used extensively in data-intensive computing. While this approach doesn't limit data transfer inherent in the application, it reduces additional read and writes caused in conventional Map-Reduce based external sorting. Since all the above approaches are limited by capabilities of SSD core quickly becoming a bottleneck for large data sets

and slower DRAMs, FPGA and GPU based SSD controller implementations were studied [62], [63]. While this implementation is impressive addition of new hardware and considerable modifications to SSD internals to enable such an architecture combined with its niche application use case make it less affordable. Finally, the work of Bae et al. [64] and Cho et al. [65] are very relevant to this thesis. In fact, they both exploit SSD controller and Flash Memory Controller for processing. While Bae et al. work only focuses on data mining applications, Cho et al. examine various data intensive kernels such as kmeans, naive bayesian, word count, linear regression etc and show 24 performance increases and 527 energy efficiency gains. However, this thesis is unique because of the much deeper design space exploration of the application and architectural space. Further several optimizations have been proposed to better fit benchmarks on SPU architecture. Use of APIs to communicate with active SSD/Flash included in approaches discussed above is orthogonal to this study and thus not explored.

6.3 Other Near Data Processing Techniques

In contrast to processing inside SSDs, attempts for processing-in-memory (PIM) were demonstrated more than a decade ago. Simple coprocessors were added near DRAM and embedded DRAM arrays and potential for improved performances were indicated. Rajeev et al. [66] point out that widespread commercial adoption of PIM remained elusive notably due to costly logic process for integration in memory leading to increase in cost per bit and lack of feasible programming models. They further state that a resurgence in PIM is in progress motivated by 3D stacked memories, high degrees of parallelism in big data workloads and the increasing joule per operation in host. As an example of evidence, Kumar et al. [67] integrated hardware accelerator in last-level-cache (LLC) . Named SQRL, it consists of a data structure specific LLC refill engine with a light weight compute array to execute kernels. They showed performance benefits in the range 13x to 121x for datacentric kernels such as recommender systems, data cubing etc. Guo et al. [68] presented overall architecture of 3D-stacked Memory Side Accelerator, which relies on a configurable array of domain-specific accelerators and demonstrated using a prototype up to 179x and 96x better energy efficiency than Intel Haswell processor for FFT and matrix transposition algorithms respectively. Loh et al.

[69] presented a case for fixed function computing inside memories. On the software side, Tseng et al. and Chu et al. [70],[71] created programming models enabling PIM. These research efforts are parallel to the motives of this thesis and together indicate favorable changes in the realm of computing.

Chapter 7

Summary and Future Work

Modern computer architectures have reached a phase where doing computations happen at low power and at frequencies that can keep energy values *low*. Constant movement of data to perform those computations - either inherent in applications or the need for better performance or both - consumes relatively much higher energy. Long latency and limited bandwidth of DRAM, large caches and larger data sets, broadcasts and synchronizations between processors and accelerators are few dominant reasons for such higher energy. Storage Processing Unit attempts to mitigate such data movement by using processing elements from within a NAND Flash package to host processor. By moving computation closer to data, this thesis presented energy and performance results from modeling applications on such an architecture. Important properties of NAND flash SSDs resulted in energy savings - (i) Rich parallelism and high I/O bandwidth within SSD - Interleaving requests to flash dies within a package and multiple flash packages operating in parallel offsets slower read and write times compared to SRAMs and DRAMs in host processor, (ii) Ability to use low power compute device as reduction in frequency (to enable low operating power) does not affect performance, (iii) Constant coprocessor to memory ratio with increasing number of channels(or coprocessors) and (iv) SSD controller synchronizing outputs of coprocessors inside multiple flash packages.

While SPU architecture provided impressive features, not all applications can directly benefit from it. Specifically, lack of cache like private memories for coprocessor, write-amplification [38] in flash, one-dimensional data access in the form of pages are main reasons that create a need to identify suitable applications. Hence, Sparse BLAS,

BFS, K-Means and K-NN were applications selected from a large array of other applications. Using these kernels, performance and energy characteristics of SPU was explored with various design parameters, code variants and mapping strategies. These kernels were further classified into bandwidth and compute bound from SPU perspective. Such differences in application characteristics resulted in exploring multiple optimizations tailored for each of them. Optimizing data layout, increasing coprocessor cores, communication between coprocessors were useful optimizations that resulted in increased performance and energy benefits.

Performance and energy optimizations cannot be considered in isolation as several solutions often exist when they are considered together. An automated framework to perform Pareto analysis for different design parameters and code variants of sparse BLAS was created. Varying number of channels inside SSDs for CSR configuration formed the Pareto front.

- As part of future work, CoFluent model of simulation can be compared with other similar and ISA/interval/cycle accurate simulators and/or validated with results from real systems. Open source simulators that can be explored in this regard include SSD Simulator [72], NAND Flash Simulator [73] and Eagle Tree [74]. Further, since these simulators are widely accepted in research domain, feasibility to add coprocessors to existing simulator framework is also an option.

- The CoFluent model itself can be modified to include other accelerators both as a conventional aid to host and as coprocessor inside NAND flash package. This will provide a richer design space to optimize applications for energy efficiency. Accelerators can be programmable general purpose/domain specific or fixed function operations. In such a case, current framework for automated design space exploration can also be extended and different optimization criteria can be evaluated.

- Another avenue for extension is to consider SPU in a distributed/cloud/data center environment. Specifically application of SPU for big data analytics using existing database engines such as SciDB [75] or D4M [76] is an open research problem. This can include identification of queries, suitable data structures and extension of the database engines to support processing in SSDs. Array data model of SciDB and

sparse matrix operations of D4M are characteristics that can potentially benefit from SPU.

- FTL in a SSD performs several functions such as page allocation, garbage collection, wear leveling and compression. Current CoFluent set up models static page allocation part of FTL. The impact of other functions on processing inside SSDs has not been explored in detail. Since SPU model is an abstract representation, a probabilistic model for example representing movement of pages (or workload distribution to coprocessor) can add value to the research.

References

[1] Ulya R. Karpuzcu, Brian Greskamp, and Josep Torrellas. The BubbleWrap Many-core: Popping Cores for Sequential Acceleration. *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458, 2009.

[2] Mueen Uddin and Azizah Abdul Rahman. Server consolidation: An approach to make data centers energy efficient and green. *arXiv preprint arXiv:1010.5037*, 2010.

[3] Gartner. Sustainable IT. *A Gartner Briefing*, 2008.

[4] Gartner Kumar, R. Media Relations. available at:www.gartner.com/it/. 2008.

[5] Ivan Rodero and Manish Parashar. Energy efficiency in HPC systems. *Energy-Efficient Distributed Computing Systems*, pages 81–108, 2012.

[6] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25, 2011.

[7] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The Terasys massively parallel PIM array. *Computer*, 28(4):23–31, 1995.

[8] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *Proceedings of 24th Conference on Very Large Databases*, pages 62–73, 1998.

[9] D Narayanan, E Thereska, A Donnelly, S Elnikety, and A Rowstron. Migrating enterprise storage to SSDs: Analysis of tradeoff. 2008.

[10] Cagdas Dirik and Bruce Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. *ACM SIGARCH Computer Architecture News*, 37(3):279–289, 2009.

[11] Peng Li, Kevin Gomez, and David J Lilja. Exploiting free silicon for energy-efficient computing directly in NAND flash-based solid-state storage systems. *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, 2013.

[12] Dawon Kahng and Simon M Sze. A floating gate and its application to memory devices. *Bell System Technical Journal*, 46(6):1288–1295, 1967.

[13] Gyu Sang Choi and Mankyu Sung. Investigating page sizes in nand flash memory. *Science and Technology*, 2011:0008686, 2011.

[14] Process Integration, Devices, and Structures(PIDS). *The International Technology Roadmap for Semiconductors*, 2012.

[15] ARM Ltd., Cortex A9 processor. *http://www.arm.com/products/processors/cortex-a/cortex-a9.php*.

[16] OCZ RevoDrive PCI-Express SSD Specications. *http://ocz.com/consumer/revodrive-3-pcie-ssd/specifications*.

[17] Intel System Modeling and Simulation. *http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html*.

[18] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *Computers, IEEE Transactions on*, 62(6):1141–1155, 2013.

[19] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 2009.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[21] Juan Gonzalez and Rafael C Núñez. Extreme-Speed Scalable Direct Sparse Solvers for Heterogeneous Supercomputing–An Enhancement to the LAPACKrc Library.

[22] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. pages 283–292, 2008.

[23] Richard Vuduc. Automatic Performance Tuning of Sparse Matrix Kernels. 2003.

[24] Yousef Saad. Iterative methods for sparse linear systems. 2003.

[25] John D Davis and Eric S Chung. SpMV: A memory-bound application on the GPU stuck between a rock and a hard place. *Microsoft Research Silicon Valley, Technical Report14 September*, 2012, 2012.

[26] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

[27] Aydin Buluc and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing*, 34(4):C170–C191, 2012.

[28] Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 222–231, 2013.

[29] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[30] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. page 14, 2012.

[31] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[32] Green Graph500. *http://htor.inf.ethz.ch/publications/img/hoefler-gg500.pdf.*

[33] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. 4:442–446, 2004.

[34] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25, 2005.

[35] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: enabling bottom-up search. pages 1618–1627, 2013.

[36] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. pages 78–88. IEEE, 2011.

[37] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. Scalable graph exploration on multicore processors. pages 1–11. IEEE Computer Society, 2010.

[38] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. page 10. ACM, 2009.

[39] Jing Zhang, Gongqing Wu, Xuegang Hu, Shiying Li, and Shuilong Hao. A parallel k-means clustering algorithm with mpi. pages 60–64. IEEE, 2011.

[40] Open Street Map. *http://www.openstreetmap. org.*

[41] Sadegh Baf, Eh Im, and Mohammad Bol. OPEN ACCESS Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background.

[42] Prasanna Balaprakash, Stefan M Wild, and Paul D Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136–2145, 2011.

[43] Prasanna Balaprakash, Ananta Tiwari, and Stefan M Wild. Multi objective optimization of hpc kernels for performance, power, and energy. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 239–260. Springer, 2014.

[44] David J DeWitt and Paula B Hawthorn. A performance evaluation of data base machine architectures. *Proceedings of the seventh international conference on Very Large Data Bases-Volume 7*, pages 199–214, 1981.

[45] Kimberly Keeton, D Patterson, Joseph Hellerstein, John Kubiatowicz, and Katherine Yelick. The intelligent disk (idisk): A revolutionary approach to database computing infrastructure. *Database*, 9(6 S 5), 1998.

[46] Erik Riedel, Garth Gibson, and Christos 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.

[47] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. volume 32, pages 81–91. ACM, 1998.

[48] Windsor W Hsu, Alan Jay Smith, and Honesty C Young. Projecting the performance of decision support workloads on systems with smart storage (smartstor). In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 417–425. IEEE, 2000.

[49] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I Popovici, Timothy E Denehy, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Semantically-smart disk systems. In *FAST*, volume 3, pages 73–88, 2003.

[50] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[51] Teradata Extreme Performance Appliance. *https://www.ndm.net/datawarehouse/pdf/EB6227.pdf*.

[52] Oracle TimesTen in-memory database. *http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html*.

[53] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12. IEEE, 2013.

[54] Erik Riedel, Christos Faloutsos, Garth A Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[55] Tina Miriam John, Anuradharthi Thiruvenkata Ramani, and John A Chandy. Active storage using object-based devices. In *Cluster Computing, 2008 IEEE International Conference on*, pages 472–478. IEEE, 2008.

[56] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[57] Zhiyang Ding. *An Active and Hybrid Storage System for Data-intensive Applications*. PhD thesis, Auburn University, 2011.

[58] Simona Boboila, Youngjae Kim, Sudharshan S Vazhkudai, Peter Desnoyers, and Galen M Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[59] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In *FAST*, pages 119–132, 2013.

[60] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David 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.

[61] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating external sorting via on-the-fly data merge in active ssds.

In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, pages 14–14. USENIX Association, 2014.

[62] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd. 2013.

[63] Jian Ouyang, Shiding Lin, Zhenyu Hou, Peng Wang, Yong Wang, and Guangyu Sun. Active ssd design for energy-efficiency improvement of web-scale data analysis. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 286–291. IEEE Press, 2013.

[64] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent ssd: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1573–1576. ACM, 2013.

[65] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102. ACM, 2013.

[66] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *Micro, IEEE*, 34(4):36–42, 2014.

[67] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. Sqrl: hardware accelerator for collecting software data structures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 475–476. ACM, 2014.

[68] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design.

[69] G Loh, N Jayasena, M Oskin, et al. A processing in memory taxonomy and a case for studying fixed-function pim. In *Near-Data Processing Workshop*, 2013.

[70] Hung-Wei Tseng and Dean M Tullsen. Data-triggered multithreading for near-data processing.

[71] Michael L Chu Nuwan Jayasena Dong and Ping Zhang Mike Ignatowski. High-level programming model abstractions for processing in memory.

[72] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pages 125–131. IEEE, 2009.

[73] Myoungsoo Jung, Ellis Herbert Wilson, David Donofrio, John Shalf, and Mahmut T Kandemir. Nandflashsim: Intrinsic latency variation aware nand flash memory system modeling and simulation at microarchitecture level. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[74] Niv Dayan, Martin Kjær Svendsen, Matias Bjørling, Philippe Bonnet, and Luc Bouganim. Eagletree: exploring the design space of ssd-based algorithms. *Proceedings of the VLDB Endowment*, 6(12):1290–1293, 2013.

[75] M Balazinska, J Becla, D Heath, D Maier, M Stonebraker, and S Zdonik. A demonstration of scidb: A science-oriented dbms. *Cell*, 1:a2, 2009.

[76] Jeremy Kepner, Christian Anderson, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Matthew Hubbell, Peter Michaleas, Julie Mullen, David O'Gwynn, et al. D4m 2.0 schema: A general purpose high performance schema for the accumulo database. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.