APPLICATION SUPPORT AND ADAPTATION FOR HIGH-THROUGHPUT ACCELERATOR ORCHESTRATED FINE-GRAIN STORAGE ACCESS

BY

VIKRAM SHARMA MAILTHODY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei Hwu, Chair
Professor Sanjay Patel
Professor Deming Chen
Dr. I-Hsin Chung, IBM Research

Abstract

Accelerators like Graphics Processing Units (GPUs) have become popular compute devices for HPC, cloud, and machine learning applications because of their compute capabilities and high memory bandwidth. However, GPUs and other accelerators still live within the confines of their modest memory capacity and rely on the inefficient software stack running on the CPUs to orchestrate access to data storage. This CPU-centric data orchestration is well-suited for GPU applications with parallel computation patterns that are flat and regular in nature, like dense neural network training. Unfortunately, many emerging workloads, such as graph and data analytics, recommender systems, or graph neural networks require fine-grain, data-dependent sparse access to storage. The CPU-centric data orchestration of storage accesses is unsuitable for these applications due to high CPU-GPU synchronization overhead, I/O traffic amplification, and excessive CPU software bottlenecks.

To overcome these limitations, this work analyzes and shows the feasibility of using GPUs to orchestrate high-throughput fine-grain direct access to storage for emerging workloads. We propose, implement, and evaluate the design of a cost-effective system architecture called BaM (Big Accelerator Memory). BaM capitalizes on the recent improvements in latency, throughput, cost, density, and endurance of solid-state storage devices and systems to realize another level of the accelerator memory hierarchy. BaM is an accelerator-centric approach where GPU threads can identify and orchestrate on-demand access to data where it is stored, be it in memory or storage, without the need to synchronize with the CPU. This significantly decreases the CPU-GPU synchronization overhead, avoids CPU software stack inefficiency, minimizes I/O amplification, and enables GPU programmers to treat storage as memory.

However, naively running applications on BaM does not result in performance and efficiency benefits. As BaM essentially extends the accelerator

memory hierarchy to the storage, favorable access patterns are needed for BaM to reach its full potential. This is because, on the one hand, BaM requires coalesced accesses for extracting high-throughput out of its cache, while on the other hand, the BaM I/O stack requires many concurrent I/O requests to hide the storage access latency. These conflicting requirements create a design dilemma, motivating the set of sophisticated optimization techniques and application adaptation strategies that allow applications to achieve peak performance on BaM.

The proposed techniques, cache-line aware parallel work assignment, and on-demand implicit tiling methods are generalizable across a wide range of data structures and emerging applications. Using these optimizations and application adaptations, we show that BaM is a viable, much less expensive alternative to the existing DRAM-only and other state-of-the-art CPU-centric solutions. Overall, this dissertation proposes a design of a system capable of performing GPU orchestrated storage access to extend the GPU's effective memory capacity and provides a set of generalizable application adaptations that enables application developers to maximize the performance, cost, I/O efficiency, capacity scalability, and simplified software development for emerging workloads, even without additional hardware support. With BaM, the user gets the teraflops of GPU compute capability and terabytes of GPU accessible memory at a low cost.

To my parents, and family, for their unwavering love and support.

Acknowledgments

I want to express my gratitude to my advisor, Professor Wen-mei Hwu, for bringing me to UIUC and for being an exemplary role model for me in every way. This includes his vision, curiosity, optimism, intellectual integrity as a researcher, patience, dedication, efficiency as a teacher and industry leader, and compassion as a person. Whatever the situation, Professor Hwu is always ready to listen and offer assistance. I'm here because of his persistence, wisdom, mentorship, trust, and undivided attention. I will forever be grateful for the advice, guidance, insights, and assistance I received as his student. I also want to thank Sabrina Hwu for looking after us like one big family along with Prof. Hwu!

I want to thank my committee for their insightful comments and interactions. I would also like to thank Professor Sanjay Patel for his advice and friendship, which immensely aided my professional growth as a researcher. I have never met anyone so busy yet always willing to listen and offer advice on any random topic. Additionally, I thank Professor Deming Chen for providing the thoughtful and responsive feedback and for going above and beyond the call of duty. I cannot express enough gratitude to Dr. I-Hsin Chung, a collaborator from IBM Research and a mentor who never stopped helping me when I was stuck. He shared critical industrial perspectives that significantly shaped this thesis that would have easily been missed in academia.

Next, I would like to thank my academic twin, Zaid Qureshi, for enduring me and supporting me in every possible way. He has been a friend and inspiration since day one of my Ph.D. and has profoundly influenced me professionally and personally. He is someone who can maneuver around most obstacles and helped me to rethink what is possible. And of course, not to forget our constant *noisy* interaction with jalapeno effects made me feel not alone during the Ph.D. journey. Our unique collaborative research style has immensely impacted both of our works.

I would also like to thank my collaborators, especially, Professor Jian Huang, Professor Andrew Miller, Professor Christopher Fletcher, Professor Jinjun Xiong, Professor Rakesh Nagi, Professor Nam Sung Kim, Professor Saugata Ghose, Professor Scott Mahlke, Professor Josep Torrellas, and many others from academia, Dr. William Dally, Dr. Micheal Garland, Dr. Isaac Gelado, Dr. CJ Newburn, Dr. Eiman Ebrahimi, Dr. Hubertus Franke, Dmitri Vainbrand, Brian Pan, and many others from the industry. Their views shaped my thinking and helped me make an impact on the real-world problems that matter at scale.

I want to thank the IMPACT Research Group and its members (past and present) for their feedback and companionship throughout the journey from day one. Talking to these folks, socializing, and making jokes were among the best experiences during my stint with this group! A special shout out to Professor Izzat El Hajj, Dr. Carl Pearson, Dr. Simon Garcia De Gonzalo, Dr. David Min, and Professor Sitao Huang for being personal role models and mentors during my Ph.D. I extend my gratitude to Te-Chia Kao, Marie-Pierre Lassiva-Moulin, and Andrew Schuh, for their positive energy and for making our life simple with less paperwork. I also extend my gratitude to the ECE, CS, and CSL staff and faculties for their help and assistance. Special thanks to Taryn Smith for proofreading this document.

I would like to thank my friends and mentors, including Umur Darbaz, Kartik Hegde and Sneha Bhat, Sidharth Gupta, Satwitk Tejaswi, Edward Richter, Karthik JVN, Ahmed Abulila, Varun Kelkar, Ranvir Rana, and many more who made every day special to live and cherish.

I am not who I am without my parents, my brother and his family, and my fiancé and her family. My parents, Chandrakala Delampady and Jayarama Sharma Mailthody, made me realize my potential whenever I was stuck or confused. I admire the patience, optimism, encouragement, support, and unlimited love they keep showering throughout my life.

Lastly, I thank my funding agencies and research grants that made this thesis possible, especially the IBM-C3SR research center and NVIDIA research. I would also like to acknowledge the valuable insight and donations received from several key stakeholders, including Intel, Samsung, Phison, H3 Platform, Broadcom, and the University of Illinois. Especially discussions with Brian Pan, Jean Chou, Jeff Chang, Yt Huang, Annie Foong, Andrzej Jakowski, Allison Goodman, Venkatram Radhakrishnan, Max Sim-

mons, Michael Reynolds, John Rinehimer, In Dong Kim, Young Paik, Jaesung Jung, Yang Seok Ki, Jeff Dodson, Raymond Chan, Hubertus Franke, Paul Crumley, and several others have been fundamental for this work to come to fruition. Vikram is partly supported by Joan and Bahl Fellowship and Dan Vivoli Endowed Fellowship during his Ph.D. This work is also partly supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network, and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDA).

Table of Contents

Chapter 1

Introduction

After more than a decade of phenomenal growth in the compute throughput and memory bandwidth [1, 2], GPUs have become popular compute devices for HPC and machine-learning applications and are proving to be effective general-purpose accelerators. Emerging high-value data-center workloads such as graph and data analytics [3–9], graph-neural networks (GNNs) [10, 11], and recommender systems [12–16] can potentially benefit from the compute throughput and memory bandwidth of GPUs. These applications access massive datasets organized into array data structures whose sizes typically range from tens of GBs to tens of TBs today and are expected to proliferate in the foreseeable future.

Storing these datasets as in-memory objects enables applications to naturally and efficiently process the data. However, GPUs and accelerators still primarily live within the confines of their modest memory capacity and are still controlled by the bulky software stack of the CPUs. The memory capacity of GPUs, despite a 53× increase from that of G80 to A100, is only 80GB, far smaller than the required capacity to accommodate the entire datasets of these workloads. Several alternatives have been introduced to address the GPU memory capacity problem.

First, some current state-of-the-art approaches rely on the CPU user/OS code to proactively partition the datasets into chunks and orchestrate the storage access and data transfers of these chunks into the GPU memory for application processing [14–24]. Second, some prior work relies on memory-mapped files and GPU page faults to reactively transfer data on demand whenever the application attempts to access data not present in GPU memory [25–28]. This thesis refers to both proposals as a *CPU-centric* approach.

The CPU-centric approach of computing works well for classical GPU applications with parallel computation patterns that are flat and regular in nature, like, dense neural network training or matrix multiplications. Un-

fortunately, for emerging workloads, where the data lookups are increasingly becoming on-demand, data-dependent, and sparse, neither of these CPU-centric approaches provide meaningful performance, efficiency, and cost benefits. Our in-depth analysis of these CPU-centric approaches shows that they cause excessive CPU software bottlenecks and CPU/GPU synchronization overheads, resulting in poor performance (See §4).

To overcome such inefficiencies in the CPU-centric approaches, some state-of-the-art solutions use the host memory [5], whose capacity typically ranges from 128GB to 2TB today, or pool together multiple GPUs' memories [16] to host the entire datasets of emerging workloads. We will refer to the use of host memory or pooling of multiple GPUs memory to extend the GPU memory capacity as the *DRAM-only* solution. While DRAM-only solution has become a common practice in industry and academia, extending the host memory or memory pooling from multiple GPUs to the level of tens of TBs is a costly proposition. For instance, an application requiring 10TB memory capacity would require 128 NVIDIA A100-80GB GPUs. Regardless of which memory, CPU DRAM expansion, or the pooled GPU is used to host these datasets, data must still be pre-loaded from storage into the extended memory, and not all data might be used due to conditions only known during run-time computation.

In this dissertation, we analyze and prove the feasibility of performing high-throughput fine-grain direct access from accelerators to storage for emerging workloads. We propose, implement, and evaluate the design of a cost-effective solution called BaM (Big Accelerator Memory). BaM capitalizes on the recent improvements in latency, throughput, cost, density, and endurance of solid-state storage devices and systems to realize another level of the accelerator memory hierarchy. BaM moves the needle in accelerator-initiated storage access significantly and comprises a storage driver stack to provision storage I/O queues and buffers in the GPU memory, a high-throughput cache for exploiting data reuse, and equips with a high-level abstraction to enable GPU self-orchestrated storage access.

BaM distributes and parallelizes the cache management across GPU threads and enables application threads in userspace to find data in parallel without relying on the traditional centralized authorities like operating system page-fault handlers. In BaM, GPU threads can identify and orchestrate on-demand accesses to data where it is stored, be it in memory or storage,

without the need to synchronize with the CPU, significantly minimizing I/O amplification and enabling GPU programmers to think of storage as memory.

However, naively running applications on BaM *does not* result in performance and efficiency benefits. As BaM essentially extends the accelerator memory hierarchy to the storage, favorable access patterns are required for BaM to reach its full potential. This requires embracing two fundamental yet conflicting requirements. Essentially, on the one hand, we need coalesced accesses to conserve the available bandwidth of the BaM cache, while on the other hand, the BaM I/O stack and storage requires many overlapping concurrent I/O requests to hide the storage access latency. However, having coalesced accesses in the BaM cache minimizes the number of I/O requests submitted to the storage and thus cannot hide the latency!

To address this design dilemma, a set of sophisticated optimization techniques and application adaptation strategies are proposed to achieve peak performance out of BaM. The ideal access pattern should make use of BaM resources perfectly and should adhere to the following set of guidelines: (1) it should generate a large number of overlapping concurrent I/O requests to tolerate the long latency of storage access, (2) it should maximize the achievable bandwidth from BaM cache and the available I/O bandwidth, (3) it should be general and applicable to a class of applications and yet not degrade the performance when the application data fit in the GPU memory and (4) it should ultimately achieve good application level performance.

To this end, we propose the cache-line aware work assignment, a simple generalizable work assignment-based optimization that meets the above goals for all studied emerging workloads. The key idea of cache-line aware work assignment is to map each warp (or a thread block) to work on cache-line data in a manner that minimizes the contention when accessing BaM metadata and also generates a sufficiently large number of concurrent I/O requests to saturate the BaM storage stack.

Although the cache-line aware work assignment provides a remarkable performance boost over naively BaM implementation, it may not be able to secure peak performance out of BaM. Thus, we discuss the limitations of this optimization in the context of graph analytics workload and then propose a novel application adaptation technique called "on-demand implicit tiling (ODT)" to perform efficient computation on Compressed Sparse Row (CSR) data-structure. In the ODT technique, each warp works on a tile of data in-

stead of the traditional vertex-centric approach such that each loaded tile is optimally reused while also exploiting the GPU's massive parallelism to generate many concurrent I/O requests to hide the memory access latency. We show that the *ODT* technique regularizes the irregular accesses pattern of CSR data structure and maximizes the performance achievable out of the BaM system.

We then apply these optimizations and application adaptations to several emerging workloads across multiple datasets, SSD types, and I/O sizes and show that BaM is a viable, much less expensive alternative to the existing DRAM-only and other state-of-the-art CPU-centric solutions. For example, for graph analytics workloads, known to be notoriously hard to achieve good performance due to random memory access, BaM, with these optimizations, provide on-par or better performance over the current state-of-the-art host-DRAM only solution and is only 4.27× slower than the GPU-HBM solution. We also show that the proposed optimizations are universal and improve or retain the same level of performance when the dataset fits within the GPU-HBM memory. And these performance benefits are extendable to other data-dependent workloads like the data-analytics where BaM can achieve up to 5.3× speed up over the host DRAM-only solution. Lastly and more importantly, BaM is the only system that can scale beyond 100TB capacity at a reasonable cost and yet provide simplistic abstractions for programmers to work with large datasets.

Overall, this dissertation proposes a design of a system capable of performing GPU orchestrated storage access to extend the GPU's effective memory capacity and provides a set of generalizable application adaptations that enables application developers to maximize the performance, cost, I/O efficiency, capacity scalability and simplified software development for emerging workloads, even without any additional hardware support. With BaM, the user gets the teraflops of GPU to compute capability and terabytes of GPU accessible memory at a low cost. We believe BaM is the significant step in building accelerator-initiated access to storage and opens up a pandora's box of research questions to answer, both in hardware and software. We also hope BaM enables ground-breaking research ideas and applications that cannot execute with current systems.

Chapter 2

Background

This chapter provides a comprehensive overview of the technologies and concepts needed to understand the thesis. This chapter covers a high-level overview of the GPU programming model and architecture, GPUDirect technology, and relevant details of NVMe specifications.

2.1 GPU Programming Model and Architecture

In this section, we briefly describe the relevant aspects of the GPU architecture and the NVIDIA CUDA programming model that are required to understand this thesis. Although this section discusses assuming the CUDA programming model, the fundamentals of GPU architecture and programming model remain consistent across different vendors. The GPU architecture consists of many streaming multiprocessors (SMs) connected to tens of gigabytes of high bandwidth global memory. Each SM supports up to 2048 threads, tens of kilobytes of fast registers, and tens of kilobytes of scratchpad memory. With CUDA, a GPU programmer can launch a compute kernel on the GPU with thousands to millions of threads organized into thread blocks. The GPU schedules thread blocks on the SMs as the resource constraints permit. Once a thread block is scheduled on an SM, it remains there until all threads in the thread block have finished their execution. Threads in a thread block can synchronize and share data through a fast scratchpad memory, referred to as shared memory.

An SM further divides a thread block into warps, consisting of 32 threads. The SM uses the threads in a warp as the unit of work to schedule on the cores of the SM. An SM can coalesce the memory accesses of a warp if its threads access the same cache-line. This enables the SM to generate larger and fewer memory requests, optimizing the memory bandwidth utilization of the GPU.

The SM hides the latency of memory and computes operations in warps by preempting them with other warps on the SM that are ready to execute. Threads executing on parallel SMs help hide the latencies experienced by anyone SM. As a result, GPUs have the hardware resources needed to provide high throughput for applications that exhibit a massive amount of fine-grain data parallelism.

2.2 GPUDirect Technology

GPUDirect is a set of features available in modern GPU architecture that allows peer-to-peer communication between two mapped device memory regions. NVIDIA introduced the GPUDirect RDMA technique in 2011 with Kepler architecture and CUDA 3.1 to accelerate data movement between the third-party PCIe devices and the GPU [29, 30]. With GPUDirect RDMA, the data transfer between a peer device and the GPU does not have to go through bounce buffers in the host memory, thus reducing latency and CPU overhead. Several prior works have used this technique to implement efficient data transfer from storage to the GPU [18–20, 22, 31]. The most recent attempt in this segment that is noteworthy is GPUDirect Storage [17] product from NVIDIA and similar efforts from AMD in the RADEON-SSG product lines [32].

However, the GPUDirect RDMA feature does not provide triggering control path modifications to the GPU. To address this, GPUDirect Async, a new feature introduced in CUDA 8.0 (2017), enables the GPU to trigger and synchronize with the peer devices by memory-mapping the third-party device BAR space within the CUDA address space. Transactions occurring with GPUDirect Async trigger memory-mapped I/O (MMIO) requests to the peer device enabling reduced involvement of the CPU in the critical path in application execution. Prior work has used this technique to interface with the network cards and communicates with remote GPU or storage [29, 30].

2.3 NVMe Queues

The NVMe [33] protocol is the latest standard defined by the industry to enable high-throughput access and to provide virtualization support for both server and consumer-grade SSDs. NVMe protocol allows up to 64K parallel submission (SQ) and completion (CQ) circular serial queues, each with 64K entries per device. In practice, today's NVMe SSDs support much fewer ($\sim$ 128) and smaller ($\sim 4K$ entries) queues. The NVMe device driver allocates a pool of buffers in the memory for use by the DMA engine of SSD devices for read and write requests. These queues and buffers traditionally reside in the system memory in the CPU-centric model.

An application requesting storage accesses causes the driver to allocate a buffer from the I/O buffer pool for the request and enqueue an NVMe I/O command at the tail of an SQ with a unique command identifier. The driver then writes the new tail value to the specific SQ's write-only register in the NVMe SSD's BAR space, i.e., it rings the queue's doorbell. For improved efficiency, a driver can ring the doorbell once after enqueueing multiple requests into an SQ.

For a read request, the SSD device controller accesses its media and delivers the data into the assigned buffer using its DMA engine. For a write request, the SSD device controller DMAs the data in the buffer into its media. Once a request is serviced, the SSD device controller inserts an entry into the CQ. When the host driver detects that the CQ entry for a command identifier is in place, it retires the request, frees up the space in the queue, and buffers for the request. The CQ entry also notifies the driver of how many entries in the SQ are consumed by the NVMe controller. The driver uses this information to free up space in the SQ. The driver then rings the CQ doorbell with the new CQ head to communicate forward progress. An SSD device can efficiently insert CQ entries for multiple requests in one transaction.

Chapter 3

Emerging Large Scale Applications

We discuss the key trends witnessed in the high-value data center applications and the system requirements for their efficient execution. This chapter covers three main types of workloads: graph analytics, GPU accelerated databases, and recommender systems. However, BaM can provide performance and cost benefits for several emerging applications such as graph neural network workloads but are not covered in this thesis.

3.1 Graph Analystics

Graph workloads are becoming increasingly common in various analytics applications such as social network analysis, recommender systems, financial modeling, bio-medical applications, graph database systems, web data, geographical maps, and many more [3, 7–9, 34–41]. A recent survey by the University of Waterloo [34] finds that many organizations use graphs for various applications that usually consist of billions of edges and consume hundreds of gigabytes of storage. Figure 3.1 shows the trends in memory required for executing graph analytics workload over time based on one dataset collection: Suite sparse dataset [35]. For efficient storage and access, graphs are stored in a compressed sparse row (CSR) data format as it has low memory overhead. In the CSR format, a graph is stored as the combination of a vertex list and an edge list, as shown in Figure 3.2. Even with CSR data format, significant graph datasets can far exceed the capacity of today's GPU memory.

Graph traversal algorithms are fundamental primitive operations performed on a graph to understand its properties. The most common graph traversal algorithms are bread-first-search (BFS), connected components (CC), page rank (PR), and single source shortest path (SSSP). These algorithms form

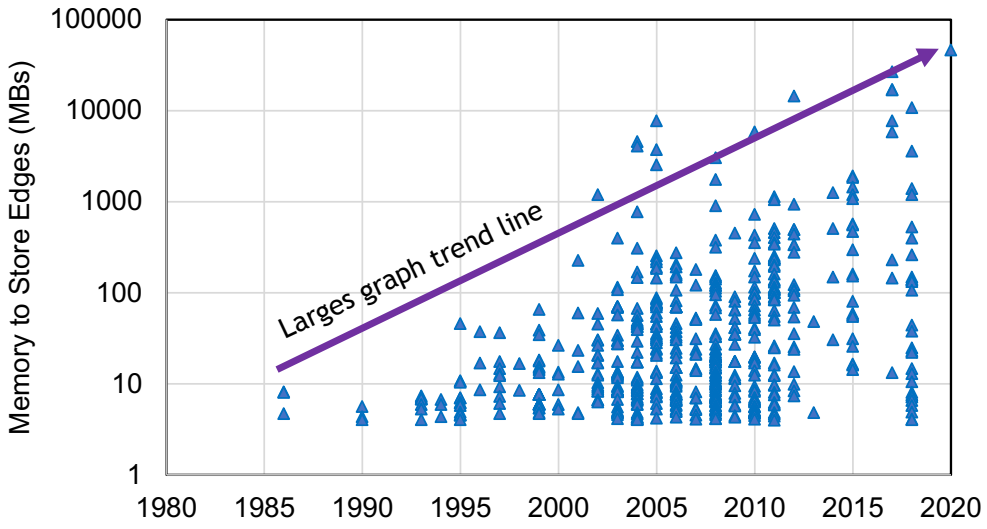


Figure 3.1: Memory required for executing graph analytics workload over time.

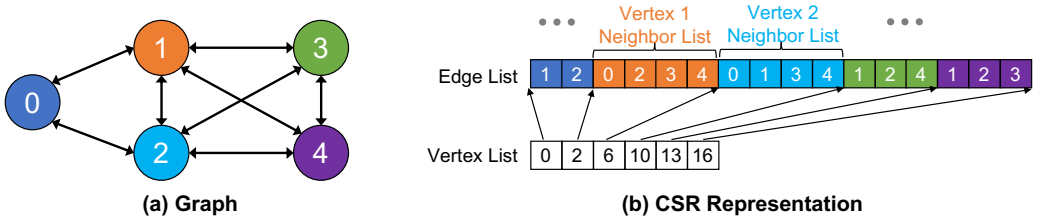


Figure 3.2: An example graph (a) and the graph CSR representation (b) in memory. As shown in (b) each vertex has variable size neighbor list.

the basis of many higher-level algorithms and libraries [5, 42–45]. Because of the massive parallelism available in graphs algorithms, these graph traversal algorithms are often accelerated using GPU [3, 5, 39, 40, 46–49].

In an optimized GPU implementation [5] of these algorithms for external I/Os, each GPU warp is assigned to a node being visited in the current iteration, where all threads in the warp collaboratively walk through the node's neighbor list. This is particularly true for applications such as BFS and SSSP. In the case of CC and page rank implementations, a similar assignment as BFS is followed, except that the application starts by examining all the nodes in the graph. Irrespective of the algorithm, each thread accesses a fine-grain data value (usually 4B or 8B, depending on the data type used to represent the graph) from the edge list for a given vertex. Since these accesses are data-dependent in nature, the access pattern becomes unpredictable.

3.2 Recommender Systems

Recommender systems are ubiquitous across online retailers and social media companies as they drive their sales [12,14,15,50]. Recommender systems aim to predict the user's interests, determine to rank and click through-rates and recommend product items that are likely interesting to the users.

Typical deep learning-based recommendation models (DLRM) comprise sets of fully connected layers and embedding tables. Figure 3.3 shows a high-level architecture of a DLRM system. The system takes two sets of inputs: dense and sparse feature vectors. Dense feature vectors represent continuous or numerical features extracted from the user-specific profile. Examples of dense feature vectors are embedding vectors extracted for user age, name, or gender. On the other hand, sparse feature vectors or categorical features are a collection of embedding tables describing user interaction with different entities in the platform. Some examples of these entities are a list of movie names, their genre, and production companies. Each entity has a unique embedding table, and each entry in the embedding table has a unique embedding vector extracted using natural language processing models.

Dense feature vectors are directly fed into various types of multi-layer perceptron (MLP) such as Fully Connected (FC) layers, CNNs, and RNNs, as shown in Figure 3.3. To handle categorical features, embedding is mapped to form a dense representation in abstract space (reduction operation). During this step, each embedding lookup can be either a one-hot vector or a weighted combination of multiple embedding vectors [12]. This is followed by a second-order interaction computation across different features by taking a dot product between all pairs of embedding vectors and processing dense feature vectors in a batch. The interaction layer captures the cross relationship between two distinct dense and sparse feature vectors in latent space, avoiding the cold start problem[1]. The output of the pairwise interaction layer is fed to a top-level MLP or fully connected layer to compute the likelihood of a click between the user and the item.

DLRM models are fundamentally limited by the memory capacity available in the GPUs [14, 16]. The total number of parameters in DLRM models

[1]Traditional matrix factorization techniques cannot capture unseen data points causing cold start problem. For example, a new movie that has not been rated by anyone yet, cannot be recommended to the user.

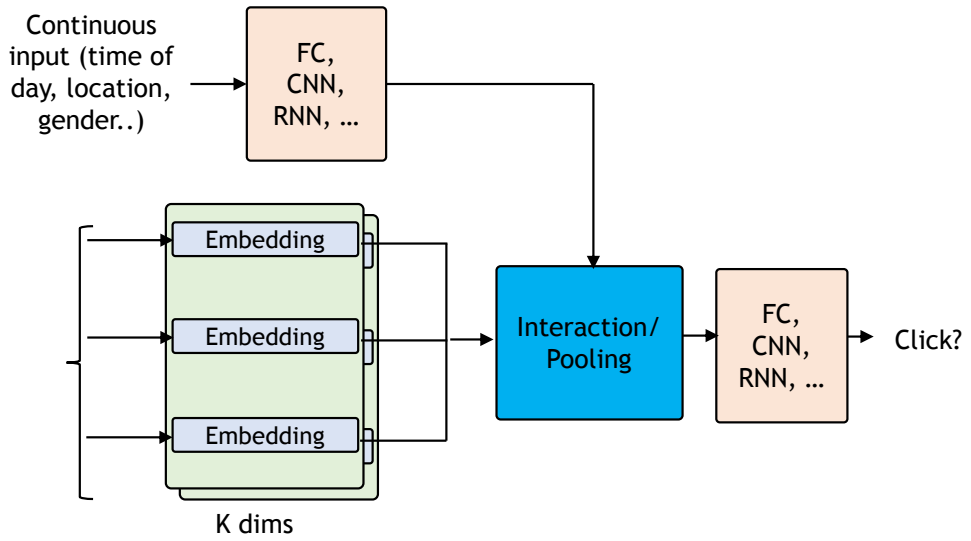


Figure 3.3: Model architecture of modern DLRM recommender systems.

tends to be very large, going up to trillions. This is because large embedding tables help disambiguate the effect of different entities, thus increasing the accuracy of prediction in recommendation models [51]. During the data-dependent lookup operation, visibility to entire embedding tables is needed as it is not possible to determine what data will be accessed when. Moreover, these models are trained frequently, usually every day, to capture the latest trending articles on the platform. This adds new entries in the embedding table over time, increasing the model capacity requirement. As new entities get created to capture new relationships between entities, additional sparse embedding tables are created and added to the model, further increasing the model capacity. Because of these, the number of parameters in the DLRM model is growing at an unprecedented rate as shown in Figure 3.4[2]. A trillion scale DLRM model constitutes up to 4TB of memory capacity requirement, assuming each parameter is a four-byte data type.

Besides memory capacity requirements, DLRM models exhibit sparse access patterns to the categorical embedding tables during the lookup operation. A *visited* bit vector is stored per entity to capture user interactions as the user may not have interacted with all the available entities on the platform. This bit vector is later used while performing sparse embedding lookup for the specific user in the DLRM model. Although the access pattern

[2]The analysis assumes each parameter consumes four bytes for data representation.

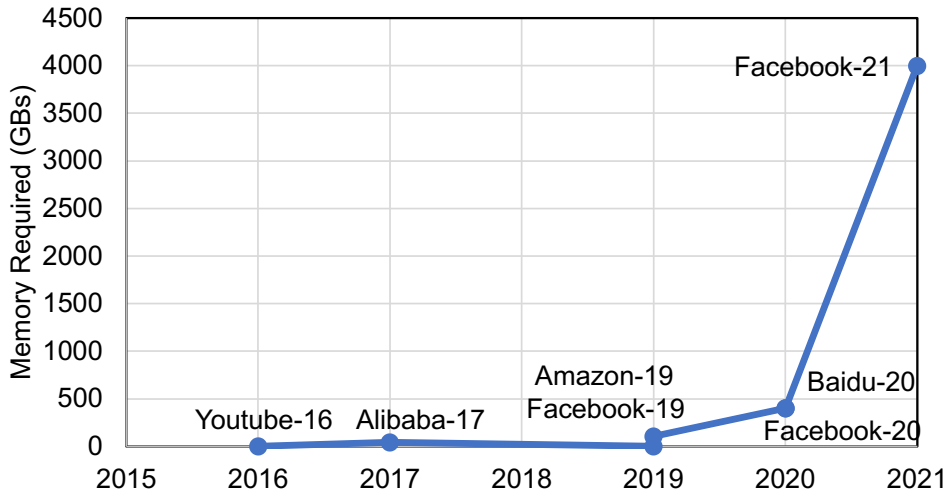


Figure 3.4: Trends in memory required for executing recommender systems workload across different companies.

is sparse, they exhibit power-law distribution when accessing the tables in memory [16]. This is because certain items are highly connected or occurring, resulting in frequent access. In addition, the access granularity of each request can vary depending on the DLRM model. The access granularity is equal to the embedding vector size of the given table and varies from 4B to up to 2KB in existing DLRM models [16, 52].

3.3 GPU Accelerated Data Analytics

Data analytics is the third most popular emerging workload of interest. The datasets used in data analytics typically consist of billions of records and tens or hundreds of metrics per record collected over a period of time. Data scientists analyze this massive amount of raw data to understand trends and metrics that would otherwise be lost in the mass of information. Data analytic workflows that rely on CPUs to load, filter, and manipulate data and perform analysis are slow and cannot exploit the massive data-level parallelism in these workflows. To address this, GPU accelerated analytic data pipelines such as NVIDIA RAPIDS framework [6] are used to execute queries on large datasets to generate meaningful insights from the data.

In RAPIDS, the GPU operates on DataFrames that provide a tabular view of the data, where there is a row per data record and columns for the various

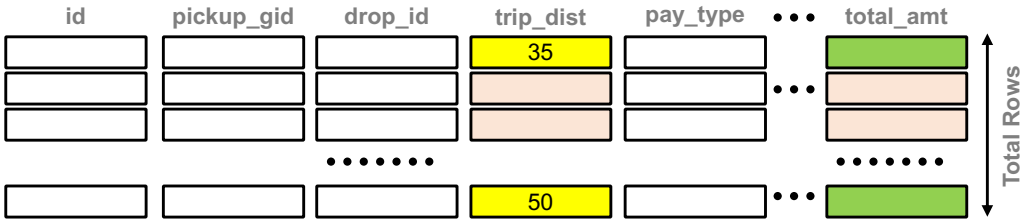


Figure 3.5: Data frame representation of the NYC taxi ride dataset with NVIDIA RAPIDS.

metrics per record. Figure 3.5 shows an example DataFrame representation of the NYC taxi ride dataset with RAPIDs [53]. The dataset consists of 200GB of encoded data organized as 1.7B rows and 49 columns stored in the Optimized Row Columnar (ORC) format [54] supported by RAPIDS [6]. The rows are trip records from 2009 to 2021, and the columns specify aspects of the trip such as pickup location, drop location, payment type, trip distance, taxi company, start and end times, etc. As Todd Schneider puts in his blog [55], "Taken as a whole, the detailed trip-level data is more than just a vast list of taxi pick up and drop off coordinates: it's a story of New York." This is because the dataset addresses several questions a data scientist might have with respect to trip data for NYC.

Suppose we ask the question "`Q1`: What is the average total cost per mile for trips with at least 30 miles?". As the dataset has almost two billion rows and exceeds the GPU memory capacity, the programmer must process the data in smaller row groups that fit in GPU memory by (1) finding and loading each row group (i.e., tile) into the GPU memory, (2) performing the query on the GPU over the row group, and (3) aggregating results across row groups. During query computation over the row group, the trip distance column of the row group is scanned, and for trips that meet the criteria with a trip distance of at least 30miles, the total amount of the corresponding total amount array element must be aggregated. This results in sparse data access to various locations in the dataset.

Moreover, the query can be extended to answer more complex questions such as: "Q5: What is the average $/mile the driver makes for trips with at least 30 miles?". To answer this query, we need four more data-dependent metrics from the dataset, and for each metric added, we can create a new intermediate query. We have to add the surcharges (Q2), hail fee (Q3), tolls (Q4), and taxes (Q5) metrics to get the penultimate query. To generate the

final result, the query must perform selective column filtering with each new data-dependent metric.

Thus, the queries executing data analytics applications are data-dependent, requiring fine-grain access to the dataset. As the user dictates the sparsity (in this example, controlled by the condition criteria of 30 miles), different sparsity will result in a different result. Thus these requirements require data-dependent accesses, and what data requires to be fetched can only be determined at the query-execution time.

3.4 Summary

To summarize, emerging high-value data-center workloads such as graph and data analytics [3–6, 53], and recommendation systems [12–16] can potentially benefit from the compute throughput and memory bandwidth of GPUs. However, these workloads must work with massive data structures accessed in array format and with sizes ranging from tens of GBs to tens of TBs today and are expected to proliferate in the foreseeable future. However, the maximum memory capacity available in these accelerators, as of 2022, is only 128GB and remains significantly below the emerging workloads memory requirements. Moreover, in these workloads, data lookups are increasingly becoming on-demand, data-dependent and sparse, making it difficult to determine which part of the data is needed when. Intuitively, the key to efficiently analyzing a massive data set in these workloads is strategically touching as little data as possible for each application-level query. However, as we discuss in the next chapter, the traditional CPU-centric model where the CPU performs data orchestration severely limits the performance of existing systems.

Chapter 4

Issues with Current State-of-the-Art and Motivation

To address the memory capacity wall, application developers and system designers use fast NVMe solid-state-drives (SSDs) and rely on the application and operation system (OS) running on the CPU to orchestrate the data movement between the GPUs, CPU memory, and SSDs while supporting standard abstractions such as memory-mapped files [14–18,20,22–24,26,27,31]. In this chapter, we present three approaches, namely proactive tiling (§ 4.1), reactive page faults (§ 4.2) and leveraging pooled memory (§ 4.3), used by the state-of-the art CPU-centric systems to alleviate memory capacity limitation and also motivate the BaM design.

4.1 I/O Amplification Problem With Proactive Tiling

Proactive tiling is a CPU-centric solution that requires the programmer to explicitly decompose and partition the data into tiles that fit into the GPU memory. The CPU application code orchestrates data movement between the storage and the GPU memory to *proactively* preload tiles into GPU memory. It also launches compute kernels for each tile and combines the results from processing the individual tiles. Proactive tiling is effective for traditional GPU applications with predefined, regular, and dense access patterns, but it is difficult for emerging applications with dynamic, data-dependent, irregular access patterns, like data analytics. The developers are forced to use coarse-grain tiles due to the synchronization and CPU orchestration's execution time overhead, which exacerbates I/O amplification.

Consider running data analytics queries on the NYC taxi dataset described in §3.3. For executing `Q1`, the programmer must process the data in smaller row groups that fit in GPU memory as explained in §4.1 as the dataset has almost 1.7 billion rows and exceeds the GPU memory limit. The program-

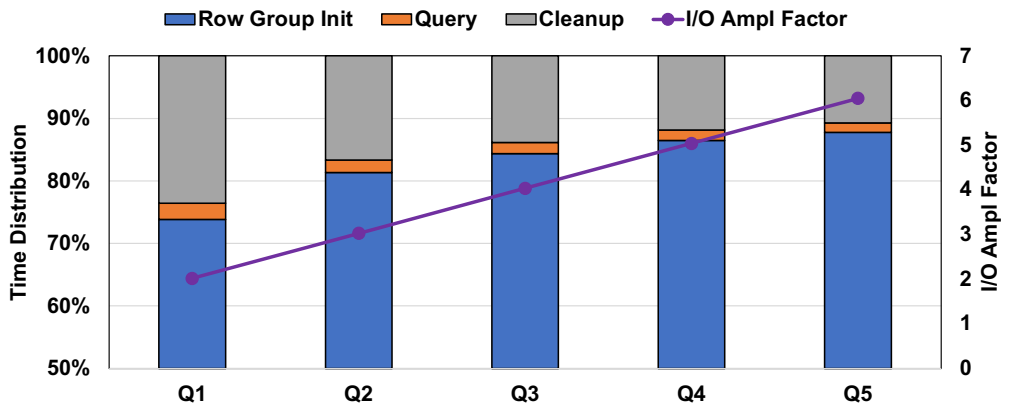


Figure 4.1: Execution time breakdown and I/O amplification in GPU accelerated data analytics application with the proactive tiling used in the state-of-the-art RAPIDS [6] system.

mer's responsibility can be described in three steps: (1) find and load each row group (i.e., tile) into the GPU memory, (2) perform the query on the GPU over the row group, and (3) aggregate results across row groups. During query computation over the row group, the trip distance column of the row group is scanned, and for trips that meet the criteria with a trip distance greater than 30miles, the amount value of the corresponding total amount array element must be aggregated. Assuming all data is present in the host OS page cache (or host DRAM-only), Figure 4.1 displays the profiling result for running the `Q1` query on the cutting-edge GPU-accelerated data analytics platform, NVIDIA RAPIDS framework. Even without storage access, the CPU code to initialize the row group, which entails finding, allocating memory for, and loading each row group of the columnar metric arrays into GPU memory, accounts for more than 73% and the CPU code to clean up the row group account for up to 23% of the end-to-end application time, reflecting the high driver and synchronization overhead.

Furthermore, since accesses to the total amount column are dependent on values in the trip distance column, the CPU cannot determine which total amount rows are required without explicit synchronization with the GPU. To address this, the state-of-the-art GPU accelerated data analytics framework, RAPIDS [6], trades off the cost of serialization with the storage bandwidth and fetches all rows of both columns from storage or host memory to GPU. As only 511K trips have their trip distance greater than 30miles, and thus

only 0.03% of the second column will be used[1]. As a result, the proactive tiling approach results in 2.02× I/O amplification for this query.

Figure 4.1 shows the I/O amplification in the RAPIDs framework for the rest of the queries described in §3.3. The I/O amplification is computed as the ratio of bytes RAPIDS reads into the GPU over the minimum bytes needed to execute each query. Furthermore, the CPU-centric approach results in gross I/O amplification with additional data-dependent metrics added on top of `Q1`. This is mainly because the CPU cannot determine which parts of each data-dependent column are required and ends up moving entire columns to the GPU memory. Hence, I/O amplification suffered by the CPU-centric model linearly scales to over 6× as the number of data-dependent metrics increased, as shown in Figure 4.1. The fine-grained, on-demand storage access capability in BaM mitigates such I/O amplification problems.

4.2 Software Overhead With Reactive Page-faults

Some applications, such as graph traversal, can not cleanly partition their datasets and thus prefer keeping whole data structures in the GPU's address space [5]. For example, assume that a graph is represented in the popular compressed sparse row (CSR) format, where the neighbor-lists of all nodes are concatenated into one large edge-list array. An array of offsets accompanies the edge-list, where the value at index `i` specifies the starting offset for the neighbor-list of node `i` in the large edge-list. Since any node, and its corresponding neighbor-list, can be visited while traversing a graph, traversal algorithms prefer to keep the entire edge-list in the GPU's address space [5].

Starting with the NVIDIA Pascal architecture, GPU drivers and programming model allow the GPU threads to implicitly access large virtual memory objects that may partly reside in the host memory using Unified Virtual Memory (UVM) abstraction [28]. Prior work shows that the UVM driver can be extended to interface with the file system layer to access a memory-mapped file [25]. This enables a GPU to generate a page fault for data, not in GPU memory which the UVM driver *reactively* services by making I/O request(s) for the requested pages on storage.

However, this the approach introduces significant software overhead in

[1]It is impossible to avoid transferring trip distance column.

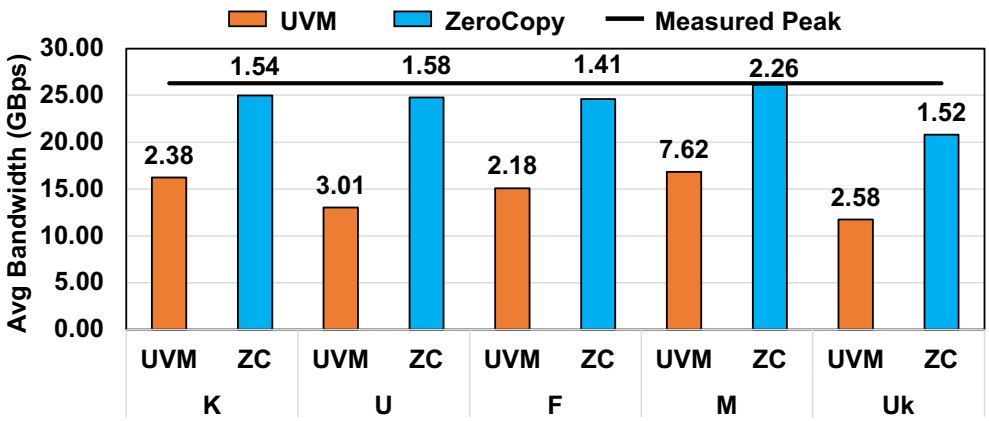


Figure 4.2: UVM page fault overhead for BFS graph traversal application across different datasets.

page fault handling mechanisms when the accessed data is missing from GPU memory. Figure 4.2 shows the measured host-memory-to-GPU-memory data transfer bandwidth for an NVIDIA A100 GPU in a PCIe Gen4 system executing BFS graph traversal on six different datasets (See Table 7.2) where the edgelists are in the UVM address space and initially in the host memory. In these experiments, there is no storage access, and measured data bandwidth is an upper bound of what a page-fault-based approach might potentially achieve.

From Figure 4.2, the average PCIe bandwidth achieved by the UVM page faulting mechanism is ∼14.52GBps which is only 55.2% of the measured peak PCIe Gen4 bandwidth, 26.3GBps. Profiling data shows that the UVM fault handler driver on the CPU is 100% utilized, and the maximum UVM page fault handling rate saturates at ∼500K IOPs. Such a rate is only half the peak throughput of a consumer-grade SSD like the Samsung 980 pro SSD, as noted in Table 5.3.

With these limitations, even if we integrate the file system layer into the UVM driver and assume no additional overhead, the UVM page fault handling mechanism cannot generate requests at a sufficiently high rate to fully utilize even one consumer-grade SSD for page sizes of 8KB or smaller. Thus the page-fault-based approach is a no-go for the targeted applications.

4.3 Cost Effectiveness With Pooled Memory

Applications can leverage CPU memory or even pool the memory of multiple GPUs to host large data structures. In the graph traversal example, the programmer allocates pinned (or zero-copy) buffers in these memories, loads the edge-list data from storage into the buffers, and maps the buffers into the address space of the GPU(s) executing the computation. The GPU threads can then access these buffers in cache-line granularity, and the GPU provides sufficient memory-level parallelism to saturate the interconnect between the GPU and these memories. Thus, these solutions can out-perform the UVM solution for critical graph traversal applications, as shown in Figure 4.2 and by previous works [5].

However, regardless of which memory, CPU or pooled GPU, is used to host these data structures, this approach suffers from two major pitfalls. First, *data must still be loaded from the storage to the memory before any GPU computation can start*. Often this initial data loading can be the main performance bottleneck as the dataset sizes are in GBs while the computation kernel requires only a few milliseconds to execute. Later in the evaluation (§ 7.1.1) we shall compare the proposed system with this Target (T) system). Second, *hosting the dataset in CPU or pooled GPU memory requires scaling the available memory*, by either increasing the CPU DRAM size or the number of GPUs in the system with the dataset size. Either of these two approaches is *prohibitively expensive in terms of dollar cost for massive datasets*. Moreover, there is a limit to the dataset sizes these systems can scale beyond which it is physically infeasible.

4.4 NVIDIA GPUDirect Storage And Challenges

NVIDIA GPUDirect Storage (GDS) [17] and several similar CPU-centric prior work [18, 20, 22, 23, 29, 31, 56–59] propose to enable peer-to-peer direct data transfer using GPUDirect RDMA. For storage access, the driver running on the CPU creates a direct data path for direct memory access (DMA) transfer between GPU and storage and avoids a bounce buffer through the CPU memory. The control plane lies squarely in the CPU in all these prior works. Thus, this is excellent if the application access pattern requires loading a

large chunk of data, like in the case of the "proactive tiling approach". But these systems pose several challenges for emerging applications that require fine-grain data-dependent access to small request granularities (512B-16KB).

Programming model problem: NVIDIA GDS and other proactive-tiling-approach-based systems require the CPU application code (basically, the burden is on the developer) to orchestrate the data movement between the storage and the GPU memory by proactively pre-loading tiles, launching computes kernels for each tile, and hoping it provides good performance. This programming model is excellent for workloads with heavy data re-use within the tile. However, as we studied in this chapter, proactive tiling exhibits significant drawbacks for emerging workloads, including I/O traffic amplification, frequent memory allocation and deallocation, constant interaction between GPU and CPU for synchronization, and, most importantly, a complex programming interface for developers.

Performance problem: Our microbenchmark study using NVIDIA GDS on an NVIDIA DGX system using single NVMe SSD, shows the NVIDIA GDS can saturate the storage bandwidth when the block size is above 64KB for random access read microbenchmark. However, for accesses at 4KB or lower, the random access read bandwidth is less than 500MBps, far lower than the storage capabilities. These bandwidth numbers are insufficient for providing performance benefits for the emerging workloads.

4.5 Little's Law: Leveraging Parallelism To Amortize Storage Access Overhead

Modern storage devices like SSDs provide very high performance, and their performance is growing yearly. Latest SSDs like Intel P5800X [60] can provide 10μs latency and maintain more than 1M IOPs for random 4KB accesses. As the storage device latency is reduced due to technological advancements like Optane or Z-NAND media, the software overhead is becoming a significant fraction of overall I/O access latency. Figure 4.3 shows the latency breakdown of an I/O request from a highly optimized CPU software stack, `io_uring`, to three NVMe SSDs: a high-end consumer grade SSD (Samsung 980 Pro) and two high-end ultra low latency data-center grade SSDs (Samsung DC 1735 and Intel Optane P5800X). *As device latency decreases, the software*

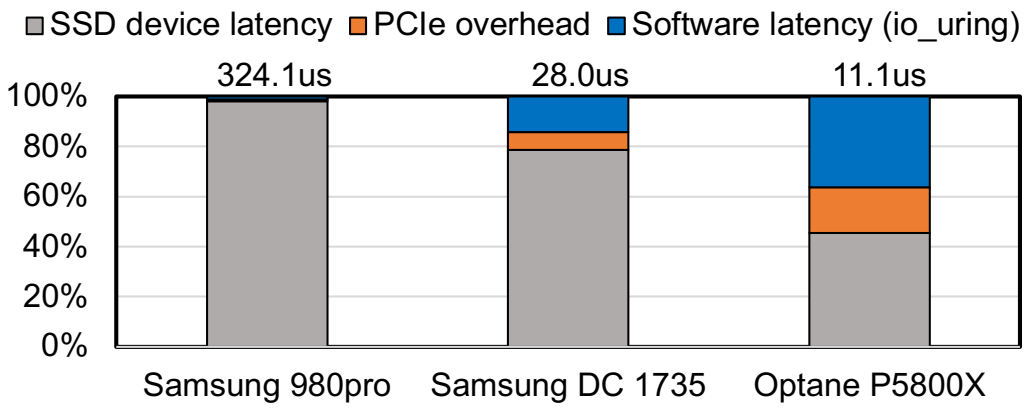


Figure 4.3: Latency breakdown of accessing the storage from CPU at 4KB using `io_uring`.

overhead becomes a significant fraction, up to 36.4%, of the total storage access latency.

To address this shift, emerging storage systems allow applications to make direct user-level I/O accesses to storage [61–69]. The storage system allocates user-level queue pairs, akin to NVMe I/O submission (SQ) and completion (CQ) queues, which the application threads can use to enqueue requests and poll for their completion. Using queues to communicate with storage systems forgoes the userspace to kernel crossing of traditional file system access system calls. Instead, isolation and other file system properties are provided through trusted services running as trusted user-level processes, kernel threads, or even storage system firmware running on the storage server/controller [61, 62, 66, 69].

In such systems, the parallelism required to tolerate access latency and achieve full throughput of the device is fundamentally governed by Little's Law: $T \times L = Q_d$, where T is the target throughput, L is the average latency, and Q_d is the minimal queue depth required at any given point in time to sustain the throughput. If we want to achieve the full potential of the critical resource, i.e., PCIe $\times$16 Gen4 connection providing $\sim$26GBps of bandwidth, then T is $26GBps/512B = 51M/sec$ and $26GBps/4KB = 6.35M/sec$ for 512B and 4KB access granularities, respectively. The average latency, L, depends on the SSD devices used, and it is 11μs and 324μs for the Intel Optane and Samsung 980pro SSDs, respectively. From Little's Law, to sustain a desired 51M accesses of 512B each, the system needs to

accommodate a queue depth of $51M/s \times 11\mu s = 561$ requests (70 requests for 4KB) for Optane SSDs. For the Samsung 980pro SSDs, the required Q_d for sustaining the same target throughput is $51M \times 324\mu s = 16,524$ (2057 for 4KB). Note that Q_d can be spread across multiple physical device queues. To sustain T over a computation phase, there needs to be a substantially higher number of concurrently serviceable access requests than Q_d over time.

GPUs provide not only the hardware features needed to hide such latencies, e.g., a massive number of threads and hardware scheduling but also the programming interface, e.g., CUDA, enabling applications to easily express massive parallelism and the high number of concurrently serviceable storage access requests. Thus, by exploiting a sufficient amount of parallelism, GPUs can hide the application storage access latency.

4.6 Summary

We discussed existing three approaches, namely proactive tiling (§ 4.1), reactive page faults (§ 4.2) and leveraging pooled memory (§ 4.3), used by the state-of-the-art CPU-centric systems to alleviate memory capacity limitation. We showed the limitations of current state-of-the-art systems like NVIDIA GDS [17] and discussed why enabling direct access from GPUs can amortize the storage access overhead. In the next chapter, we discuss how BaM addresses the memory capacity problem by enabling fine-grain high-throughput direct access to storage.

Chapter 5

BaM: Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage

5.1 BaM Vision And Design Goals

BaM addresses the insufficient GPU memory capacity problem by effectively increasing the memory capacity to terabyte-scale with the storage and uses high-bandwidth memory (HBM) as a software-managed cache. BaM proposes an accelerator-centric computing model in which threads in GPU can access data where it is stored, be it memory or storage, without relying on the CPU or the operating system (OS) to orchestrate the data movement. Using high-level abstractions like arrays, BaM enables accelerators to make on-demand, high-throughput access to storage. BaM uses storage as memory, and this enables BaM to scale to hundreds of TBs of memory capacity at a reasonable cost and yet be performant across applications.

BaM's userspace software-managed cache enables application threads to find data in parallel without relying on the traditional centralized authorities like the page-fault handlers in the operating system. BaM distributes and parallelizes the cache management across GPU threads and takes full advantage of the modern GPU's massive parallelism, memory bandwidth, and high atomic operation throughput. This makes BaM's cache scalable to thousands of threads and provides high throughput for applications when data resides in the GPU memory.

Unlike traditional memory mapped objects, BaM does not allocate a range of addresses within the virtual address space of a process for a BaM data object. Rather, BaM assigns an ID to each BaM data object and allows applications to access its elements using an offset which alleviates the problem of requiring a consecutive range of virtual addresses to host a memory-mapped storage object in a process' available virtual address space. For example, when two or more processes share a BaM data object, as long as they use

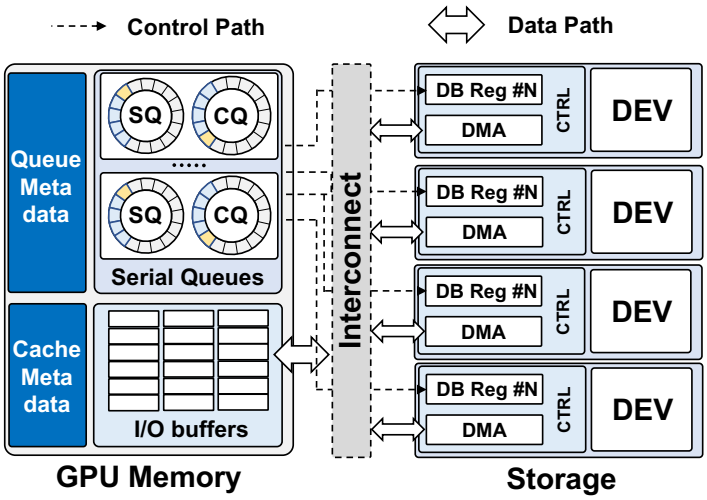


Figure 5.1: High level overview of BaM software stack.

the same BaM object ID, the BaM caches can manage the coherence between the BaM caches of these processes without the need to find a common range of locations in all the virtual address spaces of these processes[1].

While the CPU-centric storage access orchestration suffers from a low degree of CPU thread-level parallelism available to the OS page-fault handlers and device drivers, there is currently a lack of mechanisms for orchestrating storage accesses from GPUs. To address this issue, BaM provides a user-level library of highly concurrent submission/completion queues in GPU memory that enables GPU threads whose on-demand accesses miss from the software cache to make storage accesses in a high-throughput manner. This user-level approach removes the page fault handling bottleneck, incurs little software overhead for each storage access, and supports a high degree of thread-level parallelism.

Figure 5.1 summarizes the overall block diagram of BaM. BaM provisions storage I/O queues and buffers in the GPU memory and maps the storage doorbell registers to the CUDA address space (see §5.3). As storage devices have relatively low bandwidth and GPUs have limited memory capacity, BaM uses a high-throughput cache for optimal usage of these resources for applications (see §5.3.2). Also, since the existing GPU kernels generally do not expect to make storage accesses, BaM must provide high-level abstractions that hide BaM's complexity and make it easy for the programmer to inte-

[1]Supporting multiple data objects and sharing data objects across processes is under active development, goes beyond the scope of the thesis, and is part of our future work.

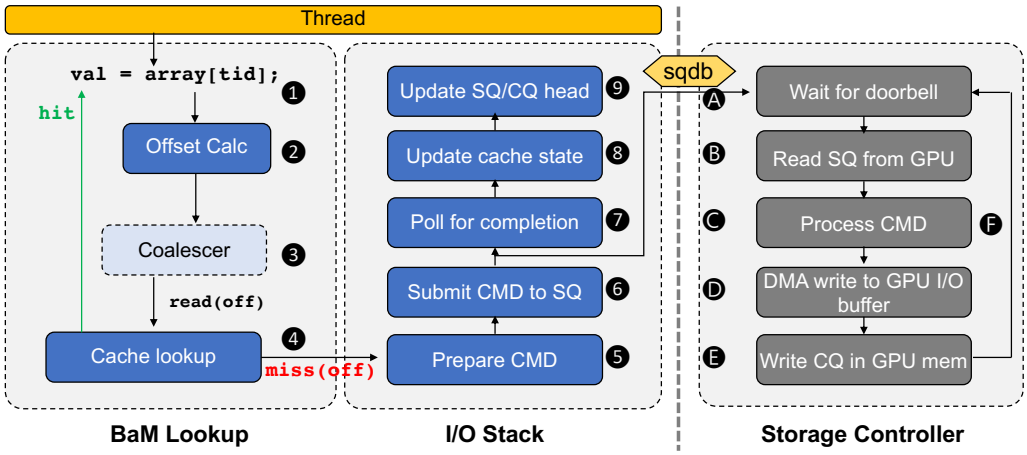


Figure 5.2: Life of a thread in BaM.

grate BaM into their GPU kernels (see §5.3.3). Lastly, emerging applications require high memory capacity for large data structures today. We show that it is possible to build a cost-effective prototype BaM system using existing off-the-shelf hardware components (see §5.3.4).

5.2 BaM System Overview

Figure 5.2 shows a programmer's view of the BaM system architecture. BaM presents `bam::array` high-level programming abstraction for easy integration with the existing application. An application can call BaM APIs to initialize the `bam::array` with data backed by storage, akin to `mmap`'ing a file.

Each GPU thread uses the abstraction to determine the cache-line offset (or virtual byte range) for the data being accessed ❶. The thread then uses the offset ❷ to probe the BaM software cache ❹ in GPU memory. The abstraction also has warp-level coalescer ❸ that can detect if multiple threads in warp access the same cache-line. If so, only one of these threads requires probing the cache on behalf of the rest and thus effectively increases the efficiency of accesses.

The thread can directly access the data in GPU memory if access hits the cache. If the access misses ❹, the thread must fetch data from the backing memory. The BaM software cache is designed to optimize the bandwidth utilization to the backing memory in two ways: (1) by eliminating redundant simultaneous requests to the backing memory and (2) by allowing users to

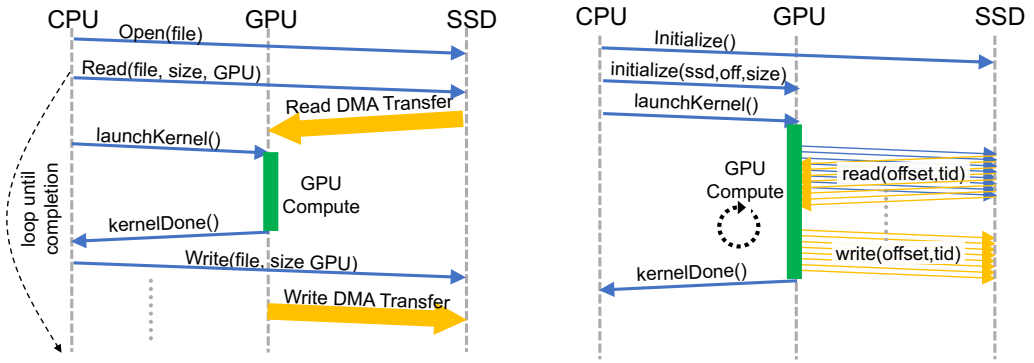
have fine-grain control of cache residency for their data.

If the storage system or device is backing the data, the GPU thread enters the BaM I/O stack to prepare a storage I/O request ❺, enqueues it to a submission queue ❻, and then waits for the storage controller to post the corresponding completion entry ❼. BaM exploits the massive thread-level parallelism offered by the GPU to amortize long latency accesses to the SSD and enables low-latency batching of multiple submission/completion queue entries to minimize the cost of expensive doorbell register updates in the storage protocol. On receiving the doorbell update Ⓐ, the storage controller fetches the corresponding submission queue entries Ⓑ and processes the command Ⓒ to transfer the data between SSD and the GPU memory Ⓓ. At the end of the transfer, the storage controller posts a completion entry in the CQ Ⓔ. After the completion entry is posted, the thread updates the cache state ❽ for the key, updates the SQ/CQ state ❾, and then can access the fetched data in GPU memory.

5.2.1 Comparison With The CPU-Centric Design

A comparison between the BaM approach and the traditional CPU-centric model, as shown in Figure 5.3a, highlights three main advantages of BaM. The first is proactive tiling; as the CPU manages the storage data transfer and GPU compute, it copies data between the storage and GPU memory, launching compute kernels multiple times to cover a large dataset. This is done with driver code with very limited thread-level parallelism and thus limited performance. Furthermore, each kernel launch and termination incurs costly synchronization between the CPU and the GPU. Since BaM allows GPU threads to compute and fetch data from storage, as shown in Figure 5.3b, the GPU doesn't need to synchronize with the CPU as frequently and can do more work in a single GPU kernel. Furthermore, the storage access latency of some threads can also be overlapped with the compute of other threads, thus improving the overall performance.

Second, because the compute is offloaded to the GPU and the data orchestration is managed by the CPU in the proactive tiling, it is difficult for the CPU to determine which parts of the data are needed and when they are needed; thus, it fetches many unneeded bytes. With BaM, a GPU thread



(a) CPU-Centric Model. (b) BaM Model.

Figure 5.3: Comparison between the traditional CPU-centric and BaM computation model is shown in (a) and (b). BaM enables GPU threads to directly access storage enabling compute and I/O overlap at fine-grain granularity.

fetches the specific data it needs only when required, reducing the I/O amplification overheads that the proactive tiling approach suffers.

Third, in the proactive tiling, programmers expend effort to partition the application's data and overlap compute with data transfers to hide storage access latency. BaM enables the programmer to naturally harness GPU thread parallelism across large datasets to hide the storage access latency.

5.3 The Design Of A BaM Prototype

We discussed the design philosophy and high-level working of BaM system. We use off-the-shelf hardware, including NVIDIA GPUs and arrays of NVMe SSDs, to construct a BaM prototype. We show the benefits of allowing GPUs to directly access storage with enough random access bandwidth to take full advantage of a GPU's PCIe Gen4 x16 link. *Once this level of data access bandwidth is achieved, a storage-based solution is as good as a host memory accessed through PCIe in terms of performance but significantly cheaper.* For simplicity, we describe the prototype assuming bare-metal, direct access to the NVMe SSDs.

5.3.1 Enable Direct NVMe Access From GPU Threads

For simplicity, we will use the NVMe SSD controllers as a simple example of storage controllers to explain the key features of the BaM. To enable GPU threads to access data on storage devices directly, we need to: 1) move the storage queues, i.e., NVMe queues and I/O buffers from CPU memory to GPU memory and 2) enable GPU threads to write to the queue doorbell registers in the NVMe SSD's BAR space. To this end, we create a custom Linux driver that creates a character device per NVMe SSD in the system, like the one by SmartIO [70]. Applications use BaM APIs to `open` the character device for each SSD they wish to use.

In the custom Linux driver, BaM leverages GPUDirect RDMA features discussed in § 2.2 to pin and map NVMe queues and I/O buffers in the GPU memory. BaM uses the `nvidia_p2p_get_pages` kernel API to pin the pages of NVMe queues, and I/O buffers pre-allocated in the GPU memory and then maps these pages for DMA access from another third-party PCIe device, like NVMe SSDs, using `nvidia_p2p_map_pages` kernel API. This enables the SSD to perform peer-to-peer data reads and writes to the GPU memory.

We leverage GPUDirect Async, discussed in § 2.2, to memory-map the NVMe SSD doorbells to the CUDA address space so that the GPU threads can ring the doorbells on demand. We use `cudaHostRegister` API with the `cudaHostRegisterIoMemory` flag to memory-map the SSD's BAR space into the application's address space. Using `cudaHostGetDevicePointer`, the application gets the virtual address that the GPU threads can use to ring the NVMe SSD doorbell registers. We note that other storage systems can be enabled similarly.

Next, we need to implement the storage I/O queues in the GPU memory and allow thousands of threads to synchronize and submit I/O requests to the device. BaM leverages the GPU's massive thread-level parallelism and fast hardware scheduling to maintain the queue depths needed to hide storage access latency and saturate the storage throughput. However, ringing doorbells, say after enqueueing commands or cleaning up SQ entries, in the existing storage I/O protocols require serialization. When a thread rings a doorbell, say to enqueue an I/O request, it must make sure that no other thread is writing to the same register and that the value it is writing is valid and is a newer value than any value written to that register before.

A naive solution for the serialization problem would be to acquire a lock before enqueueing a command to the submission queue and ring the doorbell. However, with the thousands of parallel threads on the GPU, such a design can lead to high latency and, more importantly, low throughput, as all I/O requests can get serialized. To address this, BaM exploits the CUDA memory consistency model (Volta architecture or later) and implements fine-grain memory synchronization enabling many threads to enqueue I/O requests in the SQ, poll the CQ, or mark queue entries for cleanup in parallel and without any locks. Threads only enter critical sections for moving the SQ's tail and CQ's head and ringing the doorbell. Since threads that enter a critical section move the head or tail as far as possible, most threads that make an I/O request never need to enter any critical section. This enables BaM to submit many concurrent I/O requests to the SSDs.

5.3.2 BaM's Software Cache

The BaM software cache is designed to enable optimal use of the limited GPU memory and off-GPU bandwidth. Traditional cache design used a centralized cache controller to manage cache entries and data buffers. Instead, BaM focuses on using each GPU thread as a software cache controller and uses atomic operations on the global state to manage cache metadata and data buffers. This enables BaM's cache to scale to thousands of threads and provides very high throughput for applications when data resides in the GPU memory.

Furthermore, traditional kernel-mode memory management (allocation and translation) implementations must support diverse, legacy application/hardware needs. As a result, they contain large critical sections that limit the effectiveness of multi-threaded implementations. BaM addresses this bottleneck by pre-allocating all the virtual and physical memory required for the software cache when starting each application. This approach allows the BaM software cache management to reduce critical sections, only requiring a lock when inserting or evicting a cache-line, which helps the BaM cache to support many more concurrent accesses.

BaM implements a reference counter based write-back cache. Each cache-line in the BaM cache has information about its mapping and it's current

cache state. Each cache-line can be in either (a) `Invalid (I)`, (b) `Valid (V)`, and (c) `Valid Dirty (D)` state. BaM manages the cache state of each cache-line using atomic operations. BaM encodes the state and reference counts in a 32-bit unsigned word where three most-significant bits represents the cache states and the least-significant 29 bits of the word represent the reference count. This encoding helps manage the state and reference count using one fused atomic operation. For instance, a thread can just execute `cur_state = state[i].fetch_add(1, acquire)` to increment both the cache-line reference count by one and read the cache-line's state.

When a thread probes the cache with an offset, it directly checks the corresponding cache-line's state with the `fetch_add` atomic operation and increments the reference count. If the cache-line is invalid, the thread tries to set the cache state to `Busy` state to lock the cache-line using `fetch_or` operation. This locking prevents multiple requests to the backing memory for the same cache-line, exploiting spatial locality in the data and minimizing the number of requests to the backing memory. If the returned state from the `fetch_or` operation has the `Busy` bit unset, then the thread was successful in moving the state to `Busy`, and it can proceed with finding a victim to evict and requesting the data from the backing memory. If the returned state has the `Busy` bit set, then the thread must wait for the cache-line to become not `Busy`. After the cache-line is brought into the cache, the thread obtained the lock must move the cache state from `Busy` to `Valid`. This can be done using a `fetch_xor` operation. Once the cache state is in the `Valid` state, the thread can use the data in the cache-line. When the thread is done using the cache-line, its reference count is decremented.

When the thread probes the cache, it increments the reference count and checks if the state is in the `Valid` state. If the state is `Valid`, then the thread can directly use the data from the cache-line. Reference counting helps to determine if threads expect the data to be available in the cache and ensure no other thread evicts the cache-line until the thread has copied the data to its registers. This is important, especially working with concurrent cache accesses, as there is no guarantee in ordering from the hardware when the operations will get scheduled. Thus, it is crucial to ensure that when the thread accesses the data, it gets the correct data. The use of reference count provides such guarantees for the thread accessing the data by providing some notion of pinning of cache-line until the thread copies

the data into its registers. Reference count increment occurs when the higher level API such as `bam::array<T>` performs a probe operation using `cache->coalesced_acquire()` (see Listing 5.1 line 8) to read the data into its thread's register. The reference count is decremented when the thread successfully calls the `cache->coalesced_release()` operation ((see Listing 5.1 line 10). If the thread is writing to the cache-line, it will also set the cache-line's dirty bit.

The BaM cache uses a clock replacement algorithm proposed by F.J. Corbató [71] to enable concurrent evictions at different cache-lines and avoid contentions among concurrent evictions. The cache has a global counter that gets incremented when a thread needs to find a cache slot. The returned value of the counter tells the thread which cache slot to attempt to use. As BaM implements a concurrent cache, the cache state can have both the `Valid` and `Busy` bit set simultaneously with a non-zero reference count; it is essential to ensure the suitable victim is evicted out. For instance, when a thread probes the cache to find a slot to evict, the cache-line state can be `Valid` with a reference count equal to zero. When the thread tries to evict this cache-line, it needs to ensure the reference count does not change until the cache state transitions to the `Busy` state. This can happen as another thread in the GPU could increment the reference count observing the `Valid` bit in the state. If the evicting thread finds the victim to have a non-zero reference count or change in reference count or has the `Busy` bit set by another thread, the evicting thread will abort, resetting the `Busy` bit if it was the one that set it and find another victim by incrementing the counter and attempt to take the next cache slot.

5.3.3 BaM Abstractions And Software APIs

BaM's software stack provides the programmer an array-based high-level API (`bam::array<T>`), consistent with array interfaces defined in modern programming languages (e.g. C++, Python, or Rust). As GPU kernels operate on such arrays (see § 3.4), BaM's abstraction minimizes the programmer's effort to adapt their kernels. Listing 5.2 shows a random access GPU kernel code representative of graph analytics workload. To port the kernel to use BaM, only the data structures that require to be mapped with BaM (i.e.,

```
template <typename T>
struct array {
...
   T operator[](size_t i){
      size_t clid = get_cl_id(i);
      size_t cl_sub_idx = i % (cache->cl_size/sizeof(T));

      T* cl_addr = cache->coalesced_acquire(cl_id);
      T val = cl_addr[cl_sub_idx];
      cache->coalesced_release(cl_id);
      return val;
   }

   private:
        BamCache* cache;
        ...
}
```

Listing 5.1: High level overview of `bam::array<T>` abstraction design with warp coalescing.

```
__global__ void kernel(float *data,
 size_t n, float *out, unsigned* randidx) {
    size_t tid = blockIdx.x * blockDim.x + threadId.x;
    for(; tid < n; tid += (blockIdx.x * blockDim.x))
        *output += data[randidx[tid]];
};
```

Listing 5.2: Random access GPU kernel code representative of graph analytics application access pattern.

```
__global__ void kernel(bam::array<float> data,
 size_t n, bam::arrray<float> out, unsigned* randidx) {
    bam::array_ref ref(&data);
    size_t tid = blockIdx.x * blockDim.x + threadId.x;
    for(; tid < n; tid += (blockIdx.x * blockDim.x))
        *output += ref[randidx[tid]];
};
```

Listing 5.3: BaM GPU kernel code with `bam::array<T>` abstraction for random access. With BaM's `bam::array<T>` abstraction, base GPU code requires minimal changes to the baseline codebase to support accessing data from storage.

`data` and `out`) need to be changed to `bam::array<T>` type, and the rest of the kernel can remain intact, as shown in Listing 5.3. The `bam::array<T>`'s

overloaded subscript operator enables the accessing threads to coalesce their accesses, query the cache, makes I/O requests on misses, and return the appropriate element of type T to the calling function. In contrast, the proactive tiling CPU-centric model requires complete, non-trivial application rewrites to decompose the compute and data transfers into tiles that fit within the limited GPU memory.

BaM initialization requires allocating a few internal data structures that are reused during the application's lifetime. These data structures are used to instantiate storage devices, allocate memory for cache and I/O buffers, and map data ranges and layouts. Initialization can happen implicitly through a library construction if no customization is needed. Otherwise, the application specializes in the memory through template parameters to BaM 's initialization call, a standard practice in C++ libraries. BaM also provides four memory implementations for `bam::array<T>`: (1) storage with BaM's cache (default), (2) pinned CPU memory with BaM's cache, (3) pinned CPU, and (4) GPU memory. However, in most cases, specialization and fine-tuning are unnecessary, as shown later in § 7 where only BaM 's default parameters are used.

The `bam::array<T>` abstraction implements two additional optimizations: warp coalescing and cache-line reference reuse to improve the performance further when accessing the data from the cache. Warp coalescing improves the spatial reuse of cache-line metadata while the cache-line reference reuse exploits the temporal reuse of the cache-line metadata. More in depth design and implementation discussions on of these optimization can be found in [72].

Warp Coalescing: Even though BaM 's software cache minimizes the number of requests to the backing memory, it can add additional overhead on each access in the form of cache-line state management. Threads in a warp often contend themselves in accessing the BaM software cache, especially when consecutive threads try to access contiguous bytes in memory. This contention incurs significant overhead when the needed cache-line is already in the fast GPU memory. To overcome this, BaM 's cache implements warp coalescing in software using warp-level primitives. Warp coalescing helps reduce the number of accesses to the cache metadata by up to 32× (or the number of threads in a warp). In Listing 5.1, lines 8 and 9 implement the warp coalescing method within the `bam::array<T>` abstraction. When threads access the cache, the `__match_any_sync` warp primitive is used to

synchronize with other threads in the warp, and a mask is computed, letting each thread know which other threads in the warp are accessing the same offset. From that group, the threads decide on a leader, and only the leader manipulates the requested cache-line's state. The threads in the group synchronize using the `__shfl_sync` primitive, and the leader broadcasts the address of the requested offset in the GPU memory to the group.

Cache-line Reference Reuse: In addition to coalescing, another common access pattern is to traverse a contiguous chunk of data where each element in the chunk is accessed multiple times. With this access pattern, a single thread can probe the cache-line repeatedly, causing significant cache-line metadata traffic. Each cache probe requires calling `coalesce_acquire` and `coalesce_release` methods which can expose cache access latency to the application. To this end, BaM provides `bam::array_ref` reference abstraction that enables each thread to keep a local copy of the cache-line reference in its registers. Unlike the coalescing optimization, where the spatial reuse of cache-line metadata was exploited, this optimization exploits the temporal reuse of the cache-line metadata within a thread. As shown in Listing 5.3 line 3, a thread can construct `bam::array_ref` reference with `bam::array<T>` instance. Underneath the abstraction, the overloaded subscript operator checks if the new index falls within the current cache-line reference. If yes, the operator returns the data from the GPU memory using the cache-line reference address without accessing the cache. If not, a new cache-line reference is acquired, replacing the previous one. When the instance goes out of scope, the destructor releases the cache-line reference within the thread.

Usage of `bam::array_ref` is not strictly necessary if the GPU kernel does not allow temporal reuse of the cache-line metadata. This is because with cache-line reference reuse, the abstraction is forced to do additional checks to determine if the index being accessed is within the cache-line reference range. Moreover, the cache-line reference reuse optimization adds five registers per thread and can increase register spilling. However, if the application can exploit the temporal reuse of the cache-line metadata, this optimization can provide significant performance benefits.

Table 5.1: BaM prototype system specification

BaM Configuration	Specification
System	Supermicro AS-4124GS-TNR
CPUs	2× AMD EPYC 7702 64-Core Processors
DRAM	1TB Micron DDR4-3200
GPU	NVIDIA A100-80GB PCIe
PCIe Expansion	H3 Platform Falcon-4016
SSDs	Refer to Table 5.3
Software	Ubuntu 20.04 LTS, NVIDIA Driver 470.82, CUDA 11.4

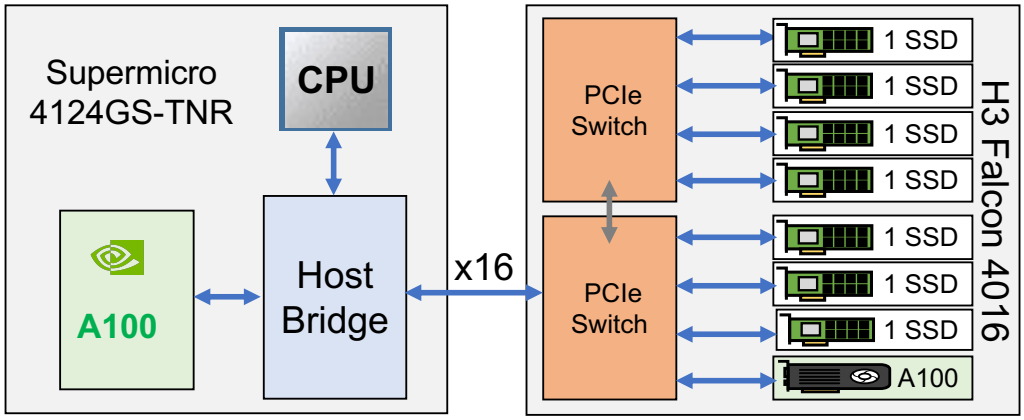


Figure 5.4: Logical view of prototype implementation of BaM using off-the-shelf components with seven SSDs.

5.3.4 BaM Hardware Design

As we initially set our design goals § 5.1, BaM needs to be built using the existing off-the-shelf hardware components. In this section, we discuss how we accomplish this goal. Scaling BaM using the PCIe slots available within a data-center grade 4U server comes with several challenges. The number of PCIe slots available in these machines is limited. For instance, the Supermicro AS-4124 system has five PCIe Gen4 ×16 slots per socket. If a GPU occupies a slot, it can only access four ×16 PCIe devices without crossing the inter-socket fabric. Furthermore, due to the chiplet design of modern multi-core CPUs, even when the five PCIe devices per socket access each other, they must cross the intra-CPU fabric. Crossing these different interconnects results in severe performance degradation as packets must be translated for each interconnect, increasing latency and limiting throughput. However, as no single NVMe SSD can provide the throughput to saturate a PCIe x16 Gen4 link, BaM hardware must scale to many NVMe devices to provide the necessary throughput to saturate the ×16 PCIe Gen4 GPU bandwidth.

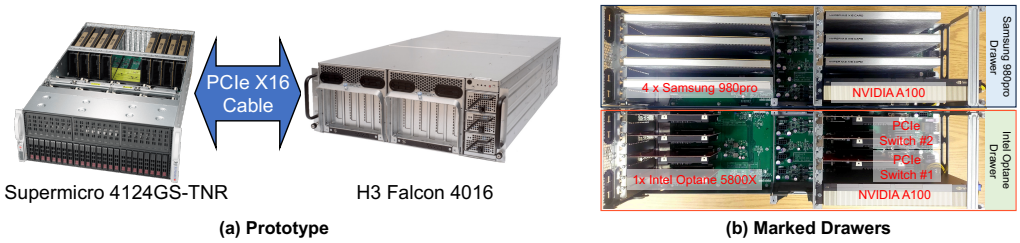


Figure 5.5: BaM Hardware (a) and drawers with marking (b)

Table 5.2: Comparison of different types of SSDs with DRAM DIMM

Technology	Sources	Product	$/GB
DRAM	multiple	DIMM (DDR4, 1 TB)	11.3
Optane [60]	single	Intel P5800X (1.6 TB)	2.54
Z-NAND [73]	single	Samsung P1735 (800 GB)	2.56
NAND Flash [74]	multiple	Samsung 980pro (1 TB)	0.51

To address this, we built a custom prototype machine for the BaM architecture using the off-the-shelf components as shown in Figure 5.4 and Figure 5.5. Table 5.1 provides the specification of the major components used for the prototype. BaM prototype uses a PCIe expansion chassis with a custom PCIe topology for scaling the number of SSDs. The PCIe switches support low-latency and high-throughput peer-to-peer access between PCIe devices. The expansion chassis has two identical drawers, which are currently independently connected to the host. Each drawer supports 8 $\times$16 PCIe slots (as shown in Figure 5.4). We use one $\times$16 slot in each drawer for an NVIDIA A100 GPU, and the rest of the slots are populated with different types of SSDs. Each drawer can only support 10 U.2 (Optane or Z-NAND) SSDs as the U.2 form factor takes up significant space. As the PCIe switches support PCIe bifurcation, a PCIe multi-SSD riser card enables more than 16 M.2 NAND Flash SSDs per drawer. Further increase in the number of SSD per GPU can be done by cascading drawers.

SSD Technology trade-offs: Table 5.2 and Table 5.3 lists the metrics that significantly impact the design, cost, and efficiency of BaM systems for three types of off-the-shelf SSDs. The `RD IOPS (512B, 4KB)` and `WR IOPS (512B, 4KB)` columns show the measured random read and write throughput of each type of SSD at 512B and 4K granularity respectively. The `$/GB` column presents the cost per GB for each SSD type, based on the current list price per device, the expansion chassis, and the risers needed to build the system. For DRAM-only system, the storage cost is excluded when com-

Table 5.3: Comparison of different types of SSDs with DRAM DIMM.

Technology	**Read IOPs (512B, 4KB)**	**Write IOPs (512B, 4KB)**	**Latency (μs)**	**DWPD**
DRAM	>10M	>10M	O(0.1)	>1000
Optane [60]	5.1M, 1.5M	1M, 1.5M	O(10)	100
Z-NAND [73]	1.1M, 1.6M	351K, 351K	O(25)	3
NAND Flash [74]	750K, 7500K	172K, 172K	O(300)	0.3

puting `$/GB` value. The `Latency` column shows the measured average device latency in μs. Comparing these metrics across SSD types shows that the consumer-grade NAND Flash SSDs are inexpensive with more challenging characteristics, while the low-latency drives such as Intel Optane SSD and Samsung Z-NAND are more expensive with more desirable characteristics. For example, for write-intensive applications using BaM, Intel Optane drives provide the best write IOPs and endurance.

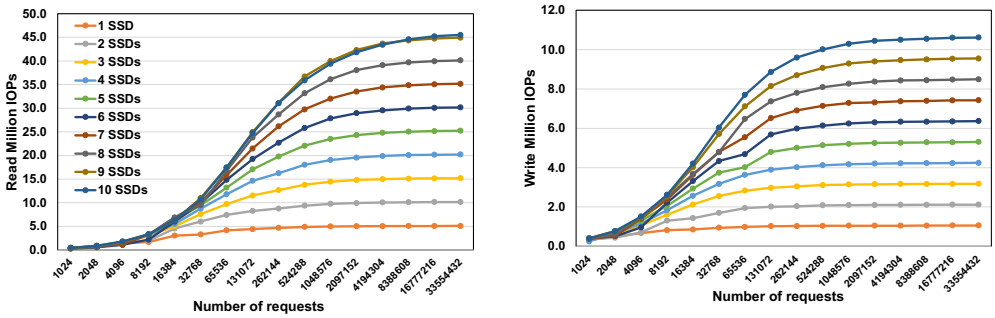
Irrespective of the underlying SSD technology, as shown in Table 5.2, BaM provides at least a 4.4-21.8× advantage in cost per GB, even with the expansion chassis and risers, over a DRAM-only solution. Furthermore, this advantage grows with additional capacity added per device, which makes BaM highly scalable as SSD capacity and application data size increase.

5.4 BaM I/O Stack Performance

This section will evaluate the throughput and bandwidth provided by the BaM I/O stack. We disable BaM cache for these experiments and map the entire SSD capacity to the GPU address space.

5.4.1 Raw Throughput Of BaM

We establish that BaM I/O stack can generate sufficient I/O requests to saturate the underlying storage system by measuring the raw throughput of BaM using microbenchmarks with Intel Optane SSDs and an NVIDIA A100 GPU. We allocate all the available SQ/CQ queue pairs into GPU memory with a queue depth of 1024 (max supported by the SSDs). We then launch a CUDA kernel with each thread requesting a random 512-byte block from the SSD via a designated queue. The intuition behind using a 512-byte block



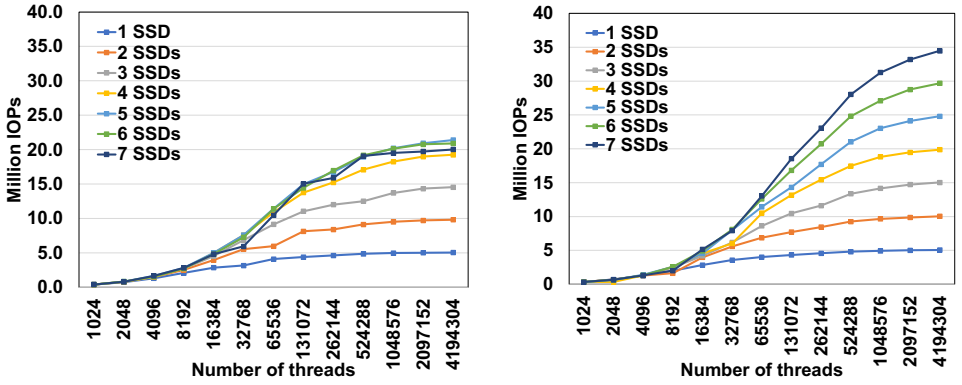
(a) Read (b) Write

Figure 5.6: 512B random read and write benchmark scaling with BaM on Intel Optane P5800X SSDs. BaM's I/O stack can reach peak IOPs per SSD and linearly scale for random read and write accesses.

size is to measure the capability of BaM I/O stack to issue many requests to saturate the storage device in the smallest access granularity. The requests are uniformly distributed across all queues with round-robin scheduling. We vary the number of threads and SSDs mapped to the GPU. For multiple SSDs, the requests are uniformly distributed across SSDs using round-robin scheduling. We measure I/O operations per second (IOPs) as a metric defined as the ratio of the number of requests submitted by the GPU and the kernel execution time.

Results: Figure 5.6 presents the measured IOPs for 512B random read and write benchmark. BaM can reach peak IOPs per SSD and linearly scale with additional SSDs for both reads and writes. With a single Optane SSD, BaM only requires about 16K-64K GPU threads to reach near peak IOPs (see Table 5.3). With ten Optane SSDs, BaM achieves 45.8M random read IOPs and 10.6M random write IOPs, the peak possible for 512B accesses to the Intel Optane SSDs. This is 22.9GBps and 5.3GBps of random read and write bandwidth, respectively. Further improvements in read bandwidth, which is about 90% of the measured peak bandwidth for Gen4 ×16 PCIe links, can be achieved by optimizing storage I/O queue implementations while scaling to more SSDs can improve the write bandwidth. Similar performance and scaling are observed with Samsung SSDs and not reported for brevity. *These results validate that BaM's infrastructure software can match the peak performance of the underlying storage system.*

Going over host: Figure 5.7 shows the measured IOPs for 512B random



(a) Using Host GPU

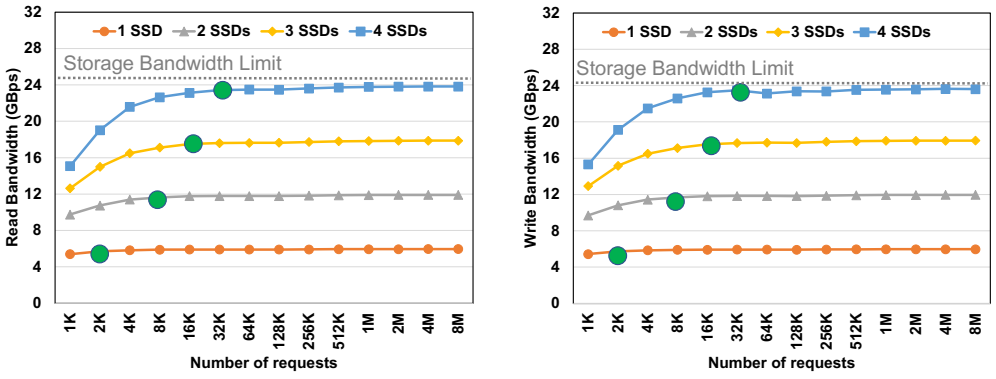
(b) Using Drawer GPU

Figure 5.7: 512B random read benchmark scaling with BaM on Intel Optane P5800X SSDs when the GPU is in host and drawer. Crossing the CPU I/O subsystem interconnects introduces severe performance scaling issues as shown in (a) and saturates at 20MIOPs. Moving the GPU to the drawer removes such scaling issues and continues to scale, saturating underlying storage.

read benchmark when GPU is in the host and SSDs are in a drawer without any changes to the topology. As we discussed in § 5.3.4, crossing the CPU I/O subsystem interconnects introduces extreme performance scaling issues as shown in Figure 5.4.1(a) and can only achieve up to 20MIOPs, even with seven Optane SSDs. Moving the GPU to the drawer removes such scaling issues and easily saturates underlying storage. A similar observation is made for write and larger access granularities and is not discussed for conciseness.

5.4.2 Bandwidth Achieved by BaM I/O Stack

Random I/O Access: We next evaluate the bandwidth achieved by the BaM I/O stack for random and sequential I/O access benchmarks using microbenchmarks. We first evaluate if the BaM I/O stack can be saturated at 4KB random read and write request. The experiment is run with four Intel Optane SSDs. We aim to understand if we can saturate the underlying storage stack bandwidth with random I/O operations. Figure 5.8 shows the bandwidth achieved by the BaM I/O stack for 4KB random read and write operations. The x-axis is the number of requests, and the y-axis shows the achieved bandwidth in GBps. Each line corresponds to how many SSDs are



(a) Read

(b) Write

Figure 5.8: 4KB random read and write bandwidth with BaM on four Intel Optane P5800X SSDs. BaM's I/O stack can saturate the underlying storage bandwidth with ease.

in the system. Each SSD can provide 6.2GBps of random read and write bandwidth. The dark green dots show the intercepting point when the BaM I/O stack saturates the SSDs. The BaM I/O stack can quickly saturate 1 SSD bandwidth of 6.2GBsp with 2K requests and then linearly saturate all SSDs and peak storage bandwidth in the system with additional I/O requests for both random read and write operations. Thus as long as requests are submitted, BaM I/O stack can achieve peak storage bandwidth. A similar observation is also made for other I/O granularities and Samsung SSDs but not reported.

Sequential I/O Access: Figure 5.9 shows the measured sequential I/O access bandwidth with different I/O granularities. Like the random I/O access benchmark, we launch a CUDA kernel with each thread requesting a consecutive block from the SSD via a designated queue. For the experiment, we keep the number of SSD at 4 (Intel Optane 5800X drives) and vary the number of threads and I/O granularities to measure the achieved bandwidth from the system. For 512B accesses, each SSD can provide 2.5GBps of bandwidth, and hence 4 Intel Optane drives can provide 10GBps peak storage bandwidth. Similarly, for 1KB, 2KB, 4KB, and 8KB, the peak achievable bandwidth by 4 Intel Optane drives are 15GBps, 20GBps, 24GBps, and 24GBps, respectively. From Figure 5.9 BaM I/O stack can achieve near peak bandwidth provided by the storage drives for each of the I/O granularities. A similar observation is found for write benchmark and for Samsung SSDs and not reported for

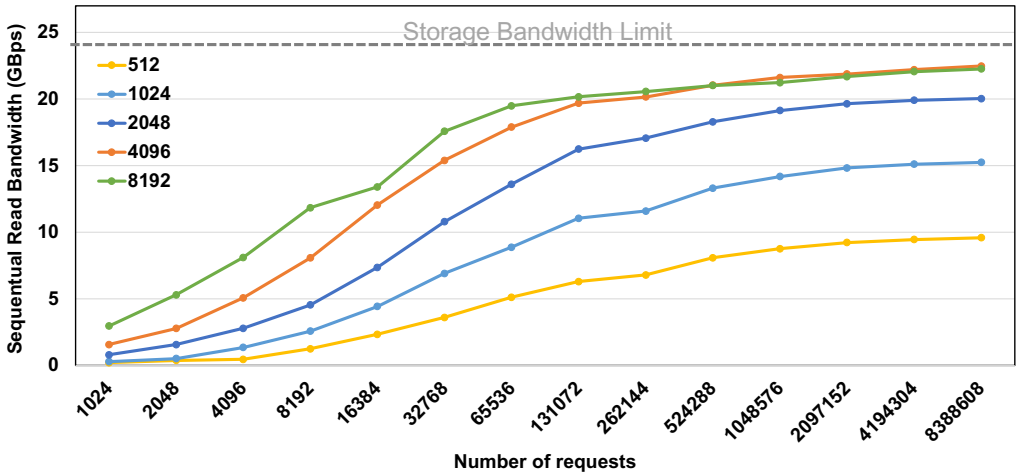


Figure 5.9: Sequential read bandwidth observed by BaM I/O stack for various I/O granularities.

brevity. These results further validate that BaM's infrastructure software can match peak performance even with sequential I/O access patterns.

5.5 BaM Cache Study Using Microbenchmark

In the previous section, we established that BaM I/O stack could quickly saturate the underlying storage bandwidth. However, the BaM cache was designed to amortize the relatively low storage bandwidth and to exploit the locality present in the application. In this section, we evaluate the performance of BaM cache. We measure several important metrics: (a) variation of miss latency depending on cache-line usage (see §5.5.1) (b) cache hit and miss throughput as a function of sequential and random access pattern (see §5.5.2).

5.5.1 Latency Of Accessing A Cache-line

First, we measure the access latency cost when accessing a cache-line in BaM as a function of cache-line usage. To measure this, we launch a CUDA kernel with one warp, and each thread in the warp reads 8B (`uint64_t`) element from the BaM cache-line. We then vary the total number of elements to be read from one element to all elements in the cache-line and report it as a

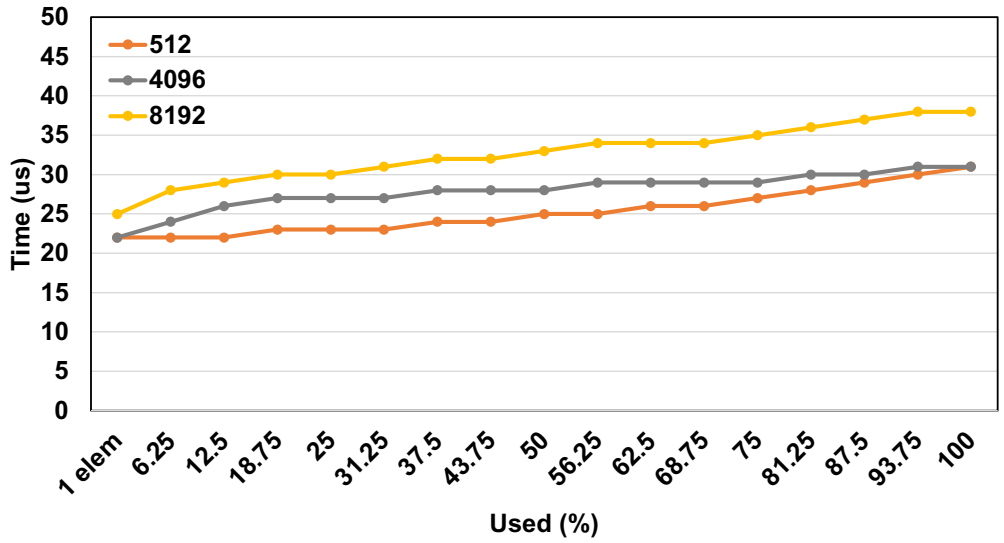
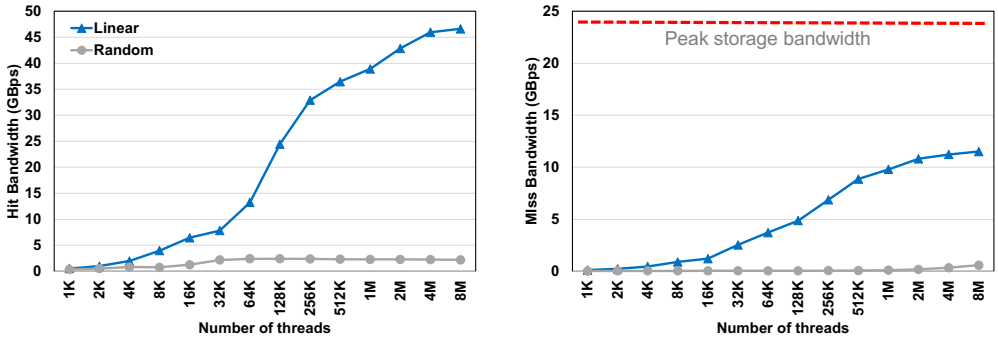


Figure 5.10: BaM cache access latency variation as a function of cache-line and its usage.

percentage used in Figure 5.10. Thread coarsening and loop unrolling are also performed when the number of elements exceeds the warp size. Lastly, as the access latency is significantly short, instead of measuring the kernel execution time, we measure time within the kernel using GPU SM clock instructions and averaged over one-hundred iterations. These experiments are conducted with single Intel Optane SSDs where a GPU kernel does a read operation through the BaM software stack while accessing `bam::array<T>`.

Figure 5.10 shows the measured access latency cost when accessing a cache-line in BaM as a function of the percentage used. To fetch the first element, BaM suffers a latency of 22-25μs. Recall that the device latency is around 7-10μs [60, 75] for 4KB access. Despite significantly lower single thread performance of GPU threads compared to CPU and with the complex BaM software stack, BaM latency is only about 2-2.5$\times$ the device latency for the fastest drive in the market. Massive parallelism in the GPU can easily hide this overhead in applications. We also note that the cache-line size of 4KB has the same miss latency as 512B. This is because, despite lower access I/O size in the case of 512B, the storage devices are optimized for 4KB access. After the first element is fetched, the rest of the access hits in the cache. Depending on the cache-line size and usage percentage, the effective cost of accessing a Byte from the BaM when all data resides in the GPU memory varies from 0ns to 140ns.



(a) Hit bandwidth

(b) Miss bandwidth

Figure 5.11: BaM cache hit and miss bandwidth for linear and random patterns. Naive usage of BaM provides meager hit performance.

5.5.2 BaM Cache Hit and Miss Throughput

We next evaluate BaM cache effective hit and miss throughput[2] for linear and random accesses benchmarks described next:

1. *Linear(**`Linear`**):* In this setup, consecutive thread reads consecutive eight byte (`uint64_t`) elements in the `bam::array<T>`. This results in coalesced accesses to the `bam::array<T>`.

2. *Random(**`Random`**):* Here, consecutive thread reads uniformly distributed random eight byte (`uint64_t`) elements in the `bam::array<T>`. This results in random accesses to the `bam::array<T>`.

For both benchmarks, we will use four Intel Optane drives with 4KB cache-line size as it provides sufficient storage throughput to saturate the storage bandwidth. We fix the cache capacity to 8GB and the dataset range to 1TB. Although we use 4KB cache-line size, a similar trend is observed for cache-line sizes from 512B to 32KBs. We then vary the number of threads and measure the cache's effective hit and miss bandwidth. Miss bandwidth is measured with the cold cache, while the hit bandwidth is measured after loading all the data to the cache and ensuring there are no misses. All experiments are executed several times, and the average bandwidth is reported.

Hit Bandwidth: Figure 5.11a shows the measured hit bandwidth of the BaM cache for linear and random access benchmarks. For linear access, the

[2]Effective throughput or bandwidth because it is as observed by the threads when accesses with `bam::array<T>` API.

system can only provide up to 46.62GBps of bandwidth, while for the random access case, the system barely crosses 2.20GBps of hit bandwidth. There are several reasons why the performance is so low. First, these two benchmarks severely limit the reuse of cache metadata and variables across warps and significantly increases the contention to manage these metadata using atomics resulting in poor performance. Second, as the hardware scheduler initially assigns consecutive thread blocks to consecutive SMs while executing, the total number of concurrent accesses is minimal and cannot hide the cache access overheads. In the next chapter, we will describe in detail the root cause of these issues and the steps to take to address them.

Miss Bandwidth: Figure 5.11b shows the measured miss bandwidth of the BaM cache for linear and random access benchmarks. Like the hit bandwidth case, both linear and random saturate at 11.2GBps and 0.58GBps of bandwidth. Intuitively, the random access benchmark should have generated more I/O requests, but as the cache-line sizes are significant, it only generates an I/O request. This is because BaM cache is designed to maximize the reuse within the cache-line and only submit an I/O request if no other threads have already submitted it. This conflicts with the goal of generating parallel I/O requests to the storage to hide the storage access latency and achieve high throughput from the BaM I/O stack. Despite the storage stack being capable of providing high throughput for storage access, because of the limitations mentioned above, these two benchmarks cannot generate sufficient I/O requests to saturate BaM I/O stack and thus ends up exposing the long storage access latency. This results in a long execution time and hence poor effective bandwidth observed by these benchmarks. A similar performance trend is observed with Samsung SSDs and at the various cache-line size.

5.6 Summary

In this chapter, we discussed the design of BaM and compared it with the state-of-the-art CPU-centric model. We discussed the software and hardware design choices needed to make BaM performent. Using microbenchmarks, we showed BaM I/O stack can saturate the underlying storage devices with ease and linearly scale with the addition of more storage drives. Using microbenchmark, we showed BaM provides the least latency when the cache-

line size is less than 4KB. Using linear and random access benchmarks, we showed BaM cache introduces significant performance degradation if naively used in the applications. To address this fundamental problem, in the next chapter, we discuss in depth the root cause of the problem and provide simple optimization techniques to incorporate in the application to enable BaM to provide on-par or better performance compared to the state-of-the-art systems.

Chapter 6

BaM's Design Dilemma: Access Coalescing vs. Latency Tolerance

The premise behind the design of BaM is to extend the GPU memory capacity to that of the storage to accommodate the programmatic access to massive datasets by emerging applications. With BaM, we essentially add a slow memory hierarchy to the system. As in any new memory hierarchy design, access patterns for the system need to be optimized for best performance. It becomes even more interesting in systems like BaM as one needs to optimize the application access pattern with two fundamental yet conflicting requirements. On the one hand, we need to coalesce accesses to the BaM cache to minimize the consumption of its limited access bandwidth. On the other hand, the BaM I/O stack and storage require many overlapping I/O requests to hide the latency. Unfortunately, coalescing accesses to the BaM cache implies the reduction in the number of concurrent I/O requests submitted to the storage and thus exposes storage access latency! This is essentially a battle between the number of concurrent I/O requests versus the BaM cache throughput for exploiting the performance of each access with re-use. Hence the title of the chapter is BaM's design dilemma, where we discuss the steps and methods a developer can use to optimize the applications to circumvent this adversity and achieve the balance required for high throughput from the BaM system.

This chapter briefly describes BaM's design dilemma using a linear access benchmark workload. The linear access benchmark is vital for BaM because the data access patterns of BaM's target applications can be approximated with piece-wise linear accesses. We propose a cache-line aware optimization, a generalizable optimization that addresses the problem associated with the baseline linear access pattern and shows how simple changes can make a huge performance difference for the studied microbenchmarks. Although cache-line aware optimization provides significant performance improvement, it may not be able to reach the peak performance for BaM for all

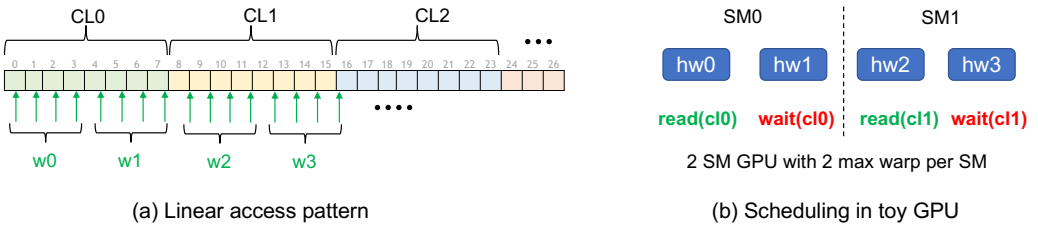


Figure 6.1: Work assignment and scheduling of linear access benchmark in modern GPUs when using BaM. The example assumes the cache-line size has 8 elements and describes using a toy GPU with two SMs with two warp schedulers each with four threads.

studied workloads. Thus we discuss the limitations in the cache-line aware optimization in the context of graph analytics workload and then propose a novel application adaptation technique called "on-demand implicit tiling" (*ODT*) to perform efficient computation on CSR graph data structures in the BaM system. We show that *ODT* technique regularizes irregular accesses patterns and maximizes the performance achievable out of BaM system.

6.1 BaM's Design Dilemma: The Problem

We will illustrate BaM's design dilemma using linear access pattern where consecutive threads read consecutive elements in the `bam::array<T>`. We showed in Chapter §5.5.2 that this sort of access pattern is not performant for BaM systems. To understand this in-depth, it is essential to recall the modern GPU scheduling strategies discussed in § 2.1.

A GPU programmer launches a compute kernel on the GPU with thousands to millions of threads organized into thread blocks. The GPU schedules thread blocks on the SMs as the resource constraints permit. The GPU hardware scheduler prefers to schedule the blocks in sequential order based on their blockIDs which is observable by microbenchmarks [2]. An SM further divides a thread block into warps, each consisting of 32 threads. The SM uses the threads in a warp as the unit of work to schedule on the cores of the SM. Inside an SM, initially, SMs assign each warp to a warp scheduler in a round-robin fashion. The SM then hides the latency of memory and compute operations in warps by preempting them with other warps on the SM that are ready to execute.

```
__global__ void kernel(bam::array<uint64_t> data,
 size_t n, uint64_t *out) {
    size_t tid = blockIdx.x * blockDim.x + threadId.x;
    bam::array_ref ref(&data);
    ...
    for(auto i=tid; i < n; tid += blockDim.x*gridDim.x)
        val += ref[i];
    ...
    out[0] = val; //dummy
};
```

Listing 6.1: Linear access pattern in BaM kernel code.

Listing 6.1 and Figure 6.1 shows an example of the linear access benchmark. Figure 6.1b describes the associated scheduling with BaM system. The example assumes a cache-line size of 8 elements and uses a toy GPU with two SMs with two warp schedulers in it, and each warp has four threads. As the linear access benchmark requires consecutive threads to read consecutive elements, only one thread in the even warps will submit an I/O request to the BaM I/O stack, while the rest of the threads will wait for the previous I/O request to complete as shown in the Figure 6.1b. This is because from the BaM cache, there can only be one request pending for a given cache-line.

Let us assume the cost of generating a read request from BaM software cache is 10μs, storage time is 30μs, and no time is spent for computation, as shown in Figure 6.4. To better understand the problem associated with the linear access benchmark, let us assume warps in the toy GPU work in lockstep when accessing the data from storage, and in each wave, each warp has to read a cache-line, wait for the storage data and then compute on the elements of the cache-line.

So, to compute four cache-lines worth of data, as shown in Figure 6.4a, the linear access pattern would require two waves where in each wave, four warps will look up the cache to generate a read request and wait for the data from the storage and then compute on the read elements. Thus, the linear access pattern will consume 80μs of total execution time for this toy example. Furthermore, only 50% of the eligible warps submitted I/O requests in this toy GPU when using the linear access pattern. However, when we expand this to real-world GPU like NVIDIA A100, the linear access pattern results in 93.5% of executable warps waiting for the data from storage.

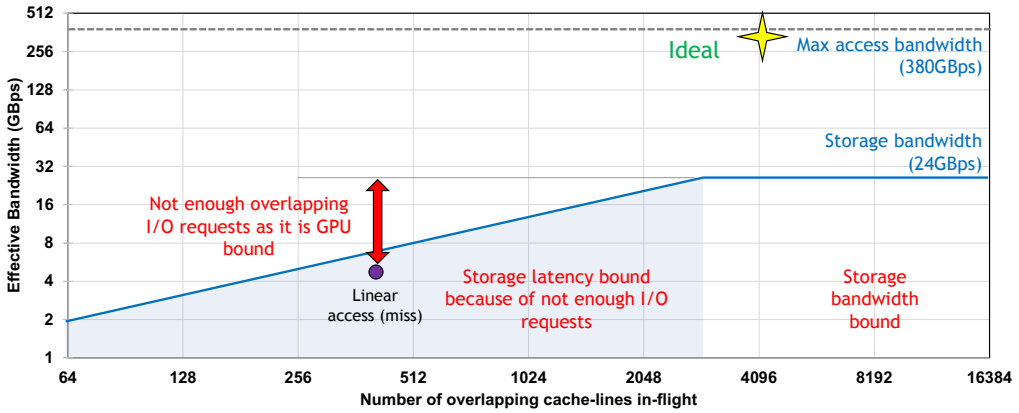


Figure 6.2: First principle model for the linear access pattern when running with BaM using four Intel Optane P5800X SSD at 4KB cache-line size.

6.1.1 First-Principle Modeling For Linear Access Pattern

We built a first-principle model to understand further the limitation associated with the linear access pattern when used in BaM. The model follows a similar methodology as the widely used roofline modeling technique [76] but is fine-tuned for BaM system. The model assumes the system has four Intel Optane SSDs, and the cache-line size is 4KB. The model also assumes the usage of NVIDIA A100 GPU. Figure 6.2 shows the first principle model for BaM with linear access pattern. The x-axis shows the number of overlapping cache-lines in flight to the storage stack, while the y-axis is the effective bandwidth observed by the computational threads. The ridge point for storage bandwidth is determined empirically using microbenchmarks.

The model greatly helps to understand the performance expectations from the BaM system for linear access patterns with no reuse. We note that without enough overlapping cache-line requests to the storage, the entire system becomes bounded by the storage access latency. This implies that the application and the BaM software stack are not submitting enough I/O requests to saturate storage. Limited I/O requests from the application and software stack could result from poor cache-line reference reuse in the application, contention on the cache metadata as multiple neighboring warps content on the same cache-line, or even GPU LLC port contention or all the above.

If the access pattern ensures the number of overlapping I/O requests is above the ridge point, from the model, we know that the application is purely bounded by the storage bandwidth. Thus, how many active cache-line are

in-flight in each GPU wave determines whether the application is storage access latency or bandwidth bound. An ideal application would maximize the bandwidth from both storage and cache and lie near the max access bandwidth line, as shown in Figure 6.2.

Next, let's see where the linear access benchmark lies in this model. With the linear access benchmark, the BaM cache filters out many of the concurrent I/O requests to maximize the throughput to the cache while trading off storage access latencies. Thus for this benchmark, it is quantitatively possible to calculate the maximum number of concurrent I/O requests (Q_d) that can be outbound from the BaM cache as:

$$Q_d = \frac{S * W * WS * sizeof(element)}{sizeof(CL)} \tag{6.1}$$

where S is the number of SMs in the GPU, W is the maximum number of warps that can be resident in an SM, WS is the number of threads in the warp, and CL is a cache-line size in bytes.

For NVIDIA A100 GPUs, the number of SMs is 108, each SM can have up to 64 warps, and each warp can have up to 32 threads in it [2]. Thus the total number of active warps at any given time is 6912, while the total active threads are 221K. Assuming each element is of size 8B and cache-line size is 4096B, the maximum number of concurrent I/O requests that can outbound is only 432 requests. This assumes the BaM stack and the entire system do not add any overhead, and the GPU does perfect scheduling of work with no divergence, be it thread or memory. In reality, it is far from the truth, and with limited concurrent overlapping I/O requests, BaM cannot amortize the overhead from the software stack and exposes entire storage access latency to the application.

Of course, the next question is why the hit bandwidth in the linear access benchmark can only reach up to 47GBps. The rationale behind low throughput is quite simple: with the linear access benchmark, the work assignment leads to contention for a small number of cache-lines resulting in performance degradation, results in severe GPU LLC port contention, and the overhead of accessing the cache-line reference for the data in the GPU memory is not amortized as there is never reuse of it as each warp loads the reference and discards it after use. Because of these reasons, the achieved instructions per cycle (IPC) in the GPU thread is low, and profiling results show that in-

structions get issued only every 8.4 cycles in each warp scheduler with the linear access benchmark.

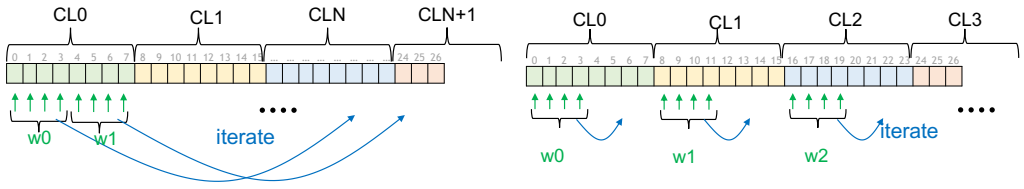
6.1.2 Summary

Thus far, we have shown that we need to increase the number of overlapping I/O requests to get the best out of the BaM system. However, this is not an easy task, especially for a system like BaM where the access pattern needs to be optimized with two conflicting requirements. We need coalesced access to extract high performance from the cache, while the storage requires large overlapping I/O requests to hide the latency. Having coalesced access reduces the number of concurrent overlapping I/O requests submitted to the storage and thus cannot hide the latency! So this creates the BaM's design dilemma. Thus, naively porting applications does not result in performance and efficiency benefits from BaM. To address this, in the next section, we propose a set of application adaptation and optimization techniques that try to maximize achievable performance out of BaM.

6.2 Cache-line Aware Optimization: The Solution

The ideal access pattern obtained by optimizations and application adaptation should perfectly make use of BaM resources and adhere to the following set of guidelines. It should quickly generate many overlapping concurrent I/O requests to tolerate the long storage access latency. It should maximize the achievable bandwidth from BaM cache by increasing the reuse of the data and metadata. The adaptation and optimization should be generalizable, implying it should be applicable with or without BaM and should work for a class of workloads and not be dataset-specific. And lastly, it should provide good end-to-end application-level performance benefits.

If the goal was to generate many overlapping I/O requests to the storage, then assigning each thread to work on a cache-line of data would have been perfect. However, this results in poor performance due to significant thread and memory divergence. Moreover, this sort of access pattern can further amplify thread and memory divergence in BaM software stack due to excessive GPU hardware cache misses as each thread touch too much BaM cache



(a) Far-away warp-coarsening (b) Cache-line aware warp-coarsening

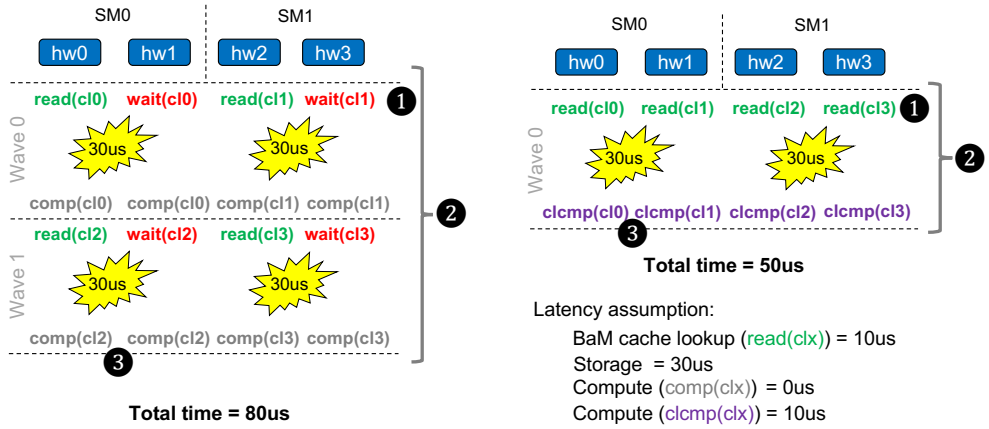
Figure 6.3: Different warp-level coarsening strategies that can be implemented on linear access pattern. Far-away coarsening does not make any changes to the baseline linear access pattern and fails to provide performance benefit while the cache-line aware warp-coarsening makes best use of BaM resources.

metadata and thrash the GPU Last-level Cache (LLC).

Thus, it is required for the application to access the `bam::array<T>` in a (piece-wise) linear manner to extract high performance from the BaM cache and minimize the divergence but also generate just enough I/O requests to the BaM I/O stack and storage to amortize the high cost of accessing storage. To this end, we propose a set of application adaptation and optimizations that provide favorable access patterns to reach BaM's full potential.

First, cache-line aware optimization is a generalizable optimization that works for all studied workloads. Recall to get the best performance out of BaM, we need to amortize the cost of accessing the software-managed BaM cache and generate the necessary I/O requests concurrently and as early as possible. An intuitive way to do that is to perform warp-level coarsening. However, applying warp coarsening can be done in many ways. First is the far-away coarsening technique, as shown in Figure 6.3a. In the far-away coarsening technique, each warp loops over grid dimensions or a large stride and is trivial to implement in any application. However, far-away coarsening does not increase the number of I/O requests generated or assist with amortizing the cache accesses overhead as it does not change anything compared to the baseline linear access pattern.

Instead, if the coarsening is done on consecutive warp-elements in the cache-line, as shown in the Figure 6.3b, it amortizes the BaM cache access overhead and also widens the window of data being computed at any given time. In addition, we must also constrain the coarsening factor as a function of cache-line size to get the best performance out of BaM. Doing so enables the cache-line aware linear access pattern to generate more I/O requests to



(a) Baseline (b) Cache-line aware

Figure 6.4: An example illustrating the computation of four cache-line worth of data in BaM using the baseline and cache-line aware linear access pattern. The example assumes a toy GPU with 2 SMs and 2 warps per SM and considers no other component in the system adds additional latency apart from the latencies stated.

hide the storage access latency.

Listing 6.2 provides an example of a GPU kernel with cache-line aware optimization for the linear access pattern. The user prepares the contents of the `assignment` array. The contents of the `assignment` array determine if the cache-lines are accessed in linear, strided, random, and any particular permutation order determined by the user. The `assignment` array is only used for generating various microbenchmark access patterns, and the application would not need it. In this listing, the cache-line aware optimization fixes the worker size to warp-size, i.e., a single warp is responsible for traversing entire cache-line data. The specific implementation of cache-line awareness is shown in lines 9-12 and lines 18-25.

Like the baseline linear access pattern, let us assume the exact cost for reading the request from BaM software cache is 10μs, storage time is 30μs but 10μs of additional time is spent for computation due to additional work added by the cache-line aware optimization. As each warp works on a unique cache-line in the cache-line aware optimization, each warp can submit a read request to the BaM I/O stack concurrently, as shown in Figure 6.4b and does not have to stall. This significantly increases the number of overlapping cache-line accesses in-flight. Thus, to compute four cache-line worth data,

```
#define WARP_SIZE 32

__global__ void kernel(bam::array<uint64_t> data,
  uint64_t *assignment, size_t n, size_t n_warps,
  uint64_t *out) {
    size_t tid = get_thread_id();
    bam::array_ref ref(&data);

    //Cache-line aware: Generate warp mapping
    size_t oldwarpid = tid / WARP_SIZE;
    size_t laneid = tid % WARP_SIZE;
    size_t n_elems_per_cl = CL_SIZE/sizeof(uint64_t);
    ...

    if(warpid < n_warps){ //boundary conditions
      //Cache-line aware determine start location
      auto start_cl  = assignment[warpid];
      auto start_idx = start_cl*n_elems_per_cl + laneid;

      //Cache-line aware warp coarsening
      for(auto j=0; i < n_elems_per_cl; j += WARP_SIZE){
         auto idx = start_idx + j;
         if(idx < n_elems)
           val+= ref[idx];
        }
    }
    ...
    out[0] = val; //dummy
};
```

Listing 6.2: Cache-line aware optimization example.

as shown in the Figure 6.4b, a cache-line aware linear access pattern would compute in a single GPU wave and consume 50μs. Compared to the baseline linear access pattern, for this toy example, this results in 1.6× speedup.

Besides the faster execution, cache-line aware optimization for linear access patterns provides multiple additional benefits. First, unlike the baseline access pattern (❶ in Figure 6.4), there are no warps in the GPU that are stalled, waiting for the data from another warp. With the cache-line aware optimization, each warp is eligible to submit a read request to a unique cache-line. This is critical enablement in the cache-line aware optimization as this enables the generation of many more overlapping I/O requests to move the application from storage latency bound to storage bandwidth bound.

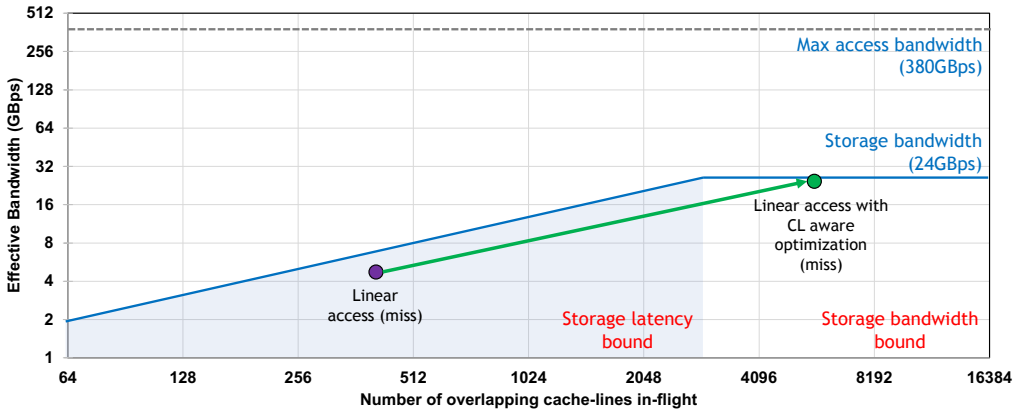


Figure 6.5: First principle model with the cache-line aware optimization for linear access pattern when running with BaM using four Intel Optane P5800X SSD at 4KB cache-line size. For NVIDIA A100 GPU, the cache-line aware implementation generates 6912 number of overlapping requests and becomes bounded by the storage bandwidth.

Second, as each warp works on an entire cache-line, each warp reads an entire cache-line worth of data, and as there are many concurrent warps, a large working set is loaded in each GPU wave. This enables CL-aware optimization to require significantly fewer GPU waves to process the same amount of data compared to the baseline linear access implementation (❷ in Figure 6.4). Although more work implies an additional computational burden on each warp, as the computation is done on the data already in the GPU memory, the increase in computational overhead is insignificant compared to the exposure to the storage access latency.

Third, as each warp works on a unique cache-line, cache-line aware optimization increases the cache-line reference and metadata reuse within the warp and reduces the contention between warps (❸ in Figure 6.4). With this optimization, each warp using the `bam::array<T>` abstraction requires probing the BaM cache once and reusing the obtained cache metadata to read data from the GPU memory. This is because, underneath the `bam::array<T>` abstraction, a pointer to the cache-line in the GPU memory is captured and stored in registers. This helps to amortize the cache lookup overheads when accessing the data in the BaM cache.

6.2.1 First-Principle Modeling For Cache-line Aware Linear Access Pattern

Figure 6.5 shows the updated first-principle model with the cache-line aware linear access pattern. The baseline linear access pattern was bounded by the storage access latency as the number of overlapping requests in-flight was limited to 432 requests. After applying the cache-line aware optimization, the linear access pattern becomes limited by the storage bandwidth, as shown. We can also measure the maximum number of concurrent I/O request outbound in the cache-line aware linear access pattern and is equal to the maximum number of resident warps in the GPU as shown in Equation 6.2

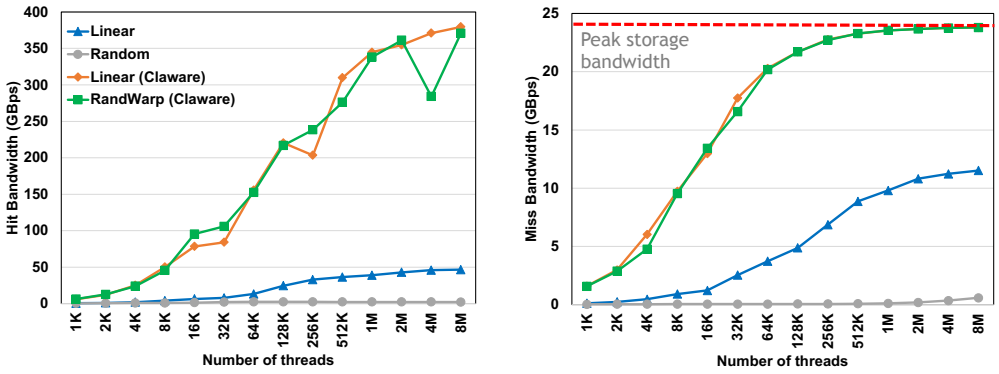
$$Q_d = S * W \tag{6.2}$$

Where S is the number of SMs in the GPU, and W is the maximum number of warps that can be resident in an SM.

Taking NVIDIA A100 GPU as an example, it has 108 SMs, and each SM can have up to 64 warps. Thus the maximum number of concurrent cache-line I/O requests outbound at any time is 6912. Compared to the baseline access pattern, the proposed access pattern generates 16× more cache-line requests to the BaM storage stack. This significant increase in concurrent cache-line I/O requests allows the application to tolerate the storage access latency better. Next, let us measure the hit and miss bandwidth using this optimized linear access pattern and see if we can indeed saturate storage.

6.2.2 BaM Cache Hit And Miss Throughput With Cache-line Aware Optimization

Figure 6.6 shows the hit (or access) and miss throughput obtained by the cache-line aware linear access pattern. The x-axis in this figure shows the number of threads that are launched. One can compute the number of I/O requests generated as the total number of threads launched over warp size (32 threads). The proposed optimization can achieve up to 379.54GBps of hit bandwidth when all data reside in the GPU memory. Compared to the linear access benchmark implementation, the hit bandwidth is improved by up to 8.14×, thanks to the reduction in the contention, improved coalescing, and increased reuse of the references. Profiling shows that compared to



(a) Hit bandwidth

(b) Miss bandwidth

Figure 6.6: BaM cache hit (or access) and miss bandwidth with cache-line aware optimization for linear access pattern. With the optimization, BaM can easily reach near peak storage bandwidth while the BaM cache hit bandwidth is an order of magnitude more than the storage bandwidth enabling high throughput for the data that is reused in the cache.

the baseline linear access pattern, the cache-line aware optimization reduces the contention on the atomic hardware unit by 50%, increases instruction executed per cycle by 1.8×, reduces the stall by 52.9%, hence improves the memory bandwidth throughput by 8.5×.

However, BaM access bandwidth is 4× lower when compared with the native raw GPU memory bandwidth (1.5TBps). This degradation is mainly due to the additional memory look-up to determine if it's a hit or miss. Despite these overheads, ∼380GBps observed application level bandwidth is decent. BaM implements software cache-controller in each GPU thread, enabling the building more scalable, robust high throughput cache design. Furthermore, BaM cache hit bandwidth is at least 15.9× more than the storage bandwidth over PCIe. Thus, if the application exhibits reuse, BaM cache can provide significant performance benefits by exploiting GPU memory bandwidth efficiently.

Figure 6.6b shows the updated miss bandwidth observed by the linear access pattern with the cache-line aware optimization. The proposed optimization quickly saturates the storage access bandwidth and achieves 23.8GBps with just a 512K number of threads. As we show later, such a level of parallelism is easily achievable in emerging applications. Similar performance trend changes are also observed in the case of Samsung SSDs at various cache-

line sizes and are not shown in brevity. This makes the proposed cache-line aware optimization agnostic of SSD types.

6.2.3 Extensions To Cache-line Aware Optimization

The cache-line aware optimization discussed previously assumes each warp works on all elements of the cache-line. However, this is strictly not necessary. If the application is storage latency bound after assigning each warp to a cache-line, the application can be fine-tuned to support sub-warp optimization. This enables exploiting even more parallelism offered by the modern GPUs, hence submitting more I/O requests to the storage. A sub-warp can be a half-warp, a quarter-warp, or even smaller but must have at least two consecutive threads. Sub-warp optimization extends the cache-line aware optimization to a group of threads, i.e., each sub-warp maps to work on a cache-line or group of cache-line data, and consecutive sub-warps work on consecutive cache-lines.

Sub-warp optimization increases the maximum number of concurrent I/O requests out-bound from the GPU as follows:

$$Q_d = S * W * \frac{WARPSIZE}{P} \tag{6.3}$$

where S is the number of SMs in the GPU, W is the maximum number of warps that can be resident in an SM, and P is the number of cache-lines mapped to a single warp resulting in the generation of concurrent I/O requests. For example, for half-warp implementation on NVIDIA A100 GPU results in the generation of maximum concurrent I/O requests to be $Q_d = 108 * 64 * \frac{32}{2} = 13824$.

The only reason why the developer has to go down the route of sub-warp optimization is to exploit the I/O parallelism further to hide the storage access latency in their application. Otherwise, it is recommended to avoid this optimization as this requires complex changes to the program. As the GPU streaming multiprocessor schedules a minimum warp size of 32 threads concurrently for non-divergent coalesced access, the sub-warp access results in both thread and memory divergence if not carefully handled in the program. However, sub-warp implementation such as half-warp can still provide performance benefits for long latency drives or smaller cache-line sizes where

we require more overlapping I/O requests to hide the storage access latency.

Instead of mapping each warp or sub-warp working on a cache-line, we can have a group of warps or a thread block to work on cache-line elements. This increases the parallelism when working on the data that is read from the storage. Alternatively, we can assign multiple cache-line to a single warp to increase the cache-line reference reuse and improve the cache access bandwidth. Cache-line coarsening enables efficient prefetching of the following cache-line metadata to the GPU L1 cache from LLC. This enables the optimization to enhance the instructions executed per cycle and thus helps increase the cache access bandwidth. Cache-line coarsening does not reduce the number of I/O requests generated to the I/O stack and retains the rest of the properties. However, having multiple cache-lines assigned to a single warp can limit the occupancy when using with BaM and hamper performance. This is because, with multiple cache-lines assigned to a single warp, we also increase the kernel resource usage, like registers. As the BaM software stack already adds much pressure on the kernel resources, multiple cache-lines can result in an aggressive spill of registers to the cache, reducing occupancy, and polluting the hardware LLC and hence hampering the overall execution time significantly.

6.3 Regularizing Irregular Workloads In BaM

Modern analytics and recommender systems place great emphasis on analyzing and extracting relational information from large datasets [5]. This relational information is often stored in a graph structure, a sparse format, as it provides a compact representation of the data. However, practical graphs come in huge sizes and offer massive parallelism that can be used to perform analytics. Because of the massive parallelism in these graph analytics algorithms, GPUs are widely used to perform the computation to extract meaningful insights. However, the ability to process large graph datasets in GPU is currently severely hampered by the limited accelerator memory capacity (see §3.1). To address this, we can potentially use the BaM system and host large sparse graph datasets in the storage and enable GPU threads to perform on-demand access to the storage during computation. However, accessing large graph datasets stored in storage poses an important problem:

sparse graph dataset accesses often result in highly irregular memory access patterns and can severely degrade the performance achievable by BaM.

Several alternative techniques have been introduced in the literature [77–80] to address this problem, where they either: (1) partition the graph and transfer a large chunk of data from storage to the GPU memory and hope to exploit spatial locality or (2) propose various new sparse data format representations for reducing the irregular access either by padding or tiling and improving the memory access efficiency [81, 82] or (3) pre-processing the graph to create active sub-graphs to compute at run-time [79]. Unfortunately, all these techniques cause significant overhead and require constant interactions with the CPU for computation which goes against the design philosophy of BaM.

To this end, we discuss how to port graph traversal algorithms to BaM with the goal to provide competitive performance against the host-memory-based DRAM-only graph analytics solution on large graphs. We start by evaluating the current state-of-the-art graph traversal framework EMOGI [5] that allows efficient graph traversal computation by performing optimized zero-copy access to the host-memory to BaM system. Next, we examine the limitations present in a naive port of two graph traversal algorithms to use BaM for accessing storage and discuss how applying various application adaptation techniques can improve the overall performance. We then propose a novel "on-demand implicit tiling" technique to perform graph computation on CSR data structure. We show "on-demand implicit tiling" technique when used in BaM can regularize irregular access patterns and hence help in maximizing the bandwidth to the BaM cache while exploiting the GPU's massive parallelism to generate many overlapping concurrent I/O requests to hide the long storage access latency.

6.3.1 Baseline

We study two popular graph analytics algorithms: Breadth-first-search (BFS) and Connected Components (CC), and port them from EMOGI [5] framework to the BaM. Doing so requires minimal code changes as shown in Listing 6.3. EMOGI adopts a vertex-centric graph traversal algorithm and implements graph traversal in a manner to maximize PCIe bandwidth efficiently.

```
#define WARP_SIZE 32

template<typename T>
__global__
void baseline_assignment(bam::array<T> edgeList,
                         T *vertexList, ...) {
    bam::array_ref edgeListRef(&edgeList);
    thread_id = get_thread_id();
    // Group by warp
    lane_id = thread_id % WARP_SIZE;
    warp_id = thread_id / WARP_SIZE;
    ...
    start = vertexList[warp_id] & ~0xF; //memory aligned
    end   = vertexList[warp_id + 1];
    ...
    // Every thread in a warp traverses the edgelist
    for (i = start+lane_id; i < end; i += WARP_SIZE) {
      if(i>=start){//prevent underflow
         edgeDst = edgeListRef[i];
         ...
      }
    } ...
}
```

Listing 6.3: Baseline work assignment in graph traversal with BaM

For the BFS algorithm, EMOGI maps each GPU warp to work on a node that is being visited in the current iteration, and all threads in the warp collaboratively walk through the node's neighbor list as shown in Figure 6.7a. For the CC implementation, EMOGI follows a similar work assignment as BFS except that the application starts by examining all the nodes in the graph, thus exhibiting a more bursty access pattern than BFS.

Let us consider the case where we must process all the graph vertices. This can occur in many graph analytics algorithms, like the first few iterations in the CC workload. Figure 6.7a shows the baseline work assignment, and Figure 6.7b shows the associated scheduling with the BaM system. The example assumes the cache-line size of six elements and uses the same toy GPU previously described. GPU hardware scheduler schedules the blocks in order of their blockIds. Because of this, all warp on GPU accesses a contiguous subset of edgelist stored in the storage. This work assignment was excellent for the baseline implementation as it enables perfectly coalesced

accesses, but for BaM, a single cache-line can have multiple neighbor-list of neighboring vertices. When the GPU scheduler schedules the warps, $warp_0$ and $warp_2$ will submit a read request to the storage while the $warp_1$ and $warp_2$ will wait for the previously submitted request to finish, akin to the baseline linear access benchmark previously studied in §6. Thus, in this work assignment, the storage access latency gets exposed and hence cannot achieve full throughput of the BaM system.

As the access pattern in graph analytics workload depends on the input graph, unlike the linear access benchmark, it is not easy to determine the maximum number of concurrent I/O requests outbound from BaM at any time. However, it is possible to speculate the best and worst case scenarios. The best case occurs when the size of the neighbor list for every vertex is equal to the BaM cache-line size. This enables each warp to submit one cache-line I/O request to the storage and then traverse over the cache-line, closely resembling the cache-line aware optimization. Unfortunately, while such an input graph could exist in theory, its probability of existence is practically zero in reality.

The worst-case scenario occurs when the neighbor list size is tiny, say one element. As most social graphs follow a power-law distribution, the neighbor list size of many vertices inside a given graph is likely to be tiny. Let us assume each edge is of eight bytes to accommodate graphs larger than four billion vertices. The worst case occurs when every neighbor list has only one edge. Thus, a single cache-line contains 512 vertices' neighborlist assuming 4KB cache-line size. As each warp works on a vertex id with the baseline assignment, 512 warps would contend on a cache-line in BaM, resulting in severe performance regression. Thus the worst case overlapping I/O request using NVIDIA A100 GPU would be $\frac{\#MaxResidentWarps}{\#WarpsPerCL} = \frac{108*64}{512} = 14$ requests, far lower than the number of requests required to hide the storage access latency.

6.3.2 Transposed Work Assignment Adaptation

We can address the limited number of I/O requests generated in the baseline work assignment with a straightforward two-line change as shown in Listing 6.4 (Line 13 and 14 are new lines added). Instead of each warp working

```
#define WARP_SIZE 32

template<typename T>
__global__
void transposed_assignment(bam::array<T> edgeList,
        T *vertexList, stride, vertex_count, ...) {
    thread_id = get_thread_id();
    bam::array_ref edgeListRef(&edgeList);
    // Group by warp
    lane_id = thread_id % WARP_SIZE;
    oldwarpid = thread_id / WARP_SIZE;
    ...
    // transposed work assignment
    pivot = ceil(vertex_count/stride);
    warp_id = (oldwarpid/pivot)
                + ((oldwarpid % pivot)*stride);
    ...
    start = vertexList[warp_id] & ~0xF; //memory aligned
    end   = vertexList[warp_id + 1];
    ....
    // Every thread in a warp traverses the edgelist
    for (i = start+lane_id; i < end; i += WARP_SIZE) {
      if(i>=start){//prevent underflow
         edgeDst = edgeListRef[i];
         ...
      }
    } ...
}
```

Listing 6.4: Transposed work assignment in graph traversal.

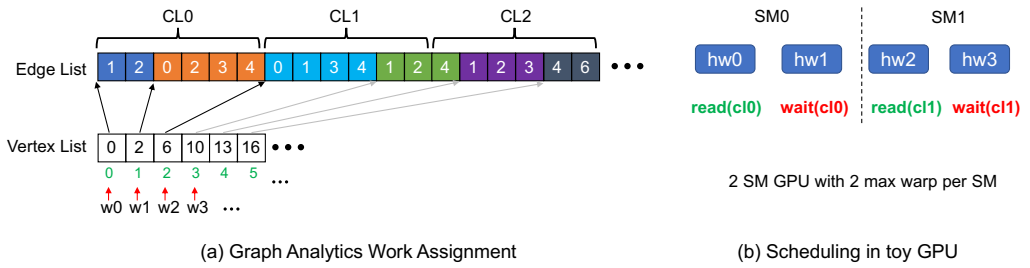


Figure 6.7: Example illustration of work assignment in state-of-the-art graph traversal algorithms.

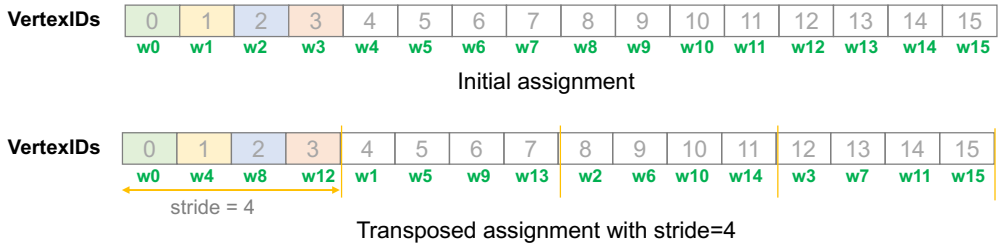


Figure 6.8: Example illustration of transposed work assignment using vertex-centric graph analytics approach.

on a vertex that has the same id as the warp, we can assign each warp to work on certain stride-away vertex ID. Picking the proper stride enables each warp to work on a unique cache-line edgelist. Say, the stride size is 4 as shown in the Figure 6.8, then $warp_0$ works on $vertex_0$, $warp_4$ works on $vertex_1$, $warp_8$ works on $vertex_2$. The assignment circles back after four vertices and start with $warp1$ working on $vertex_4$, and so on and so forth. This work assignment is akin to the corner-turning or transposition technique widely used in matrix multiplication computation [83]. The transposed work assignment addresses the limited I/O access issue by widening the window of data being accessed, which allows the cache-lines to be touched earlier than the original work assignment.

However, there are several challenges associated with this transposed work assignment optimization. First, the best stride value is input graph dependent, and the programmer must sweep the parameter *manually* to determine the optimal stride manually. Second, optimal stride size is crucial for achieving good performance. This is because if the stride size is too tiny, multiple warps can still contend on the same cache-line resulting in poor performance. If the stride size is too large, then we might bring in a lot more data than

needed to the BaM cache and result in BaM cache thrashing. Although thrashing is a problem even observed by the baseline implementation, it is more aggravated with the transposed work assignment. As the BaM cache implements a clockwork replacement algorithm, a new cache-line that was recently brought in can be evicted to make space for another cache-line requested by another warp. This leads to thrashing in the BaM cache and cannot be avoided with the transposed work assignment. Third, the transposed work assignment does not maximize the cache-line reference and metadata reuse. Thus, this optimization may not provide the best performance despite increasing the number of in-flight requests to the storage.

6.3.3 Applying Cache-line Aware Warp Coarsening

Alternative to transposed work assignment adaptation is to perform cache-line aware warp-level coarsening where a single warp works on two or more vertex ids. This reduces the amount of contention that occurs in the given cache-line. Moreover, as each warp works on more than one vertex id, depending on the neighborlist size of the input graph and coarsening factor, the number of I/O requests submitted to storage can significantly increase and hence help to amortize the storage access latency. The coarsening factor should be a function of cache-line size and the average degree distribution of the input graph. Of course, like the transposed work assignment, the best coarsen factor is input graph dependent and can only be determined at runtime by manually fine-tuning. However, warp-level coarsening can require substantial changes to the kernel code to support and may not be easy to implement in all cases. Moreover, the contention issue is unresolved as the neighbor lists are likely of varying sizes, and it is impossible to enforce full cache-line awareness.

6.3.4 On-Demand Implicit Tiling Technique

The key to gaining optimal throughput out of the BaM design is to have the warps touch as many required cache-lines as early as possible to saturate the interconnect bandwidth and maximize the reuse of each cache-line with the most negligible overhead. This implies that we need to generate sufficient

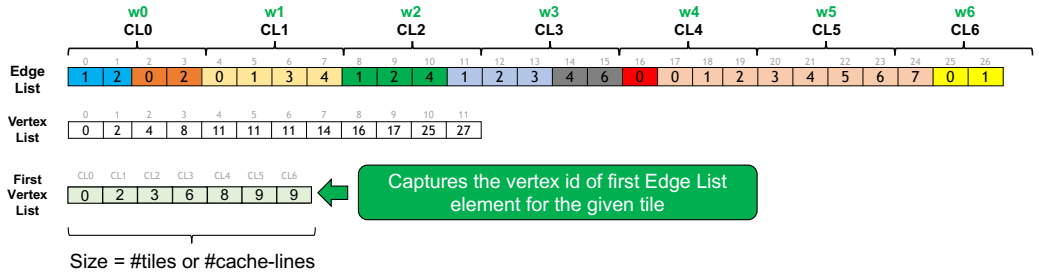


Figure 6.9: Example illustration of on-demand implicit tiling technique for graph algorithms where the graph is represented with a CSR data-structure. The example assumes the tile size is equal the cache-line size to keep the illustration simple.

concurrent overlapping I/O requests to saturate the interconnect bandwidth, and each cache-line should only be brought in once from the storage, and all computation for that cache-line must be completed as early as possible. This application modification would essentially exploit the full throughput of the BaM cache and leverage the full throughput of the BaM I/O stack simultaneously.

However, this requires us to move away from the traditional vertex-centric approach of graph traversal and propose a novel access pattern called "*On-demand Implicit Tiling (ODT)*" designed for CSR-like data-structures stored in BaM. The *ODT* technique is designed with the same spirit of cache-line aware optimization but with a critical constraint that it should only bring each cache-line or tile once from the storage, and all computation for that tile must be completed. Thus, each warp in the *ODT* technique reads a tile of data from the backing memory and computes ~~it~~ on each element of the tile irrespective of the number of vertices or neighborlists mapped to it.

The *ODT* technique can be illustrated with the example graph shown in Figure 6.9. The graph contains 11 vertices and 27 edges and is represented in a CSR data structure (edgelist and vertexlist) as shown in the Figure 6.9[1]. For simplicity, let us assume the tile size equals the cache-line size of 4 elements. In BaM system, the edgelist of the graph is mapped to the `bam::array<T>` abstraction. This essentially implies that the edgelist spans six cache-lines stored in storage, as shown in Figure 6.9.

[1]The above example CSR data structure is synthetically created to test various components of *ODT* technique and should not be confused as a real-world graph. The specific example sufficiently explains the *ODT* technique discussed in this thesis.

In the vertex-centric implementation, each warp works on a vertex and traverses the neighborlist. Each warp indexes the vertexlist to determine the start and end points, and then uses them to index the edgelist to do the traversal. If the neighborlist size is small, multiple warps contend on the same cache-line reducing throughput. This results in inefficient use of BaM resources.

Instead, the *ODT* technique takes an entirely different approach and enables each warp to work on a cache-line as shown in Figure 6.9. That is, $warp_0$, $warp_1$, $warp_2$, and so on works on CL_0, CL_1, CL_2 and so on respectively. This enables *ODT* technique to minimize the lookup cost, maximizes the reuse of data once loaded from the storage to the GPU memory, and generates many overlapping I/O requests to the storage. The *ODT* technique captures the maximum possible reuse of a cache-line in the CSR data structure.

Enabling *ODT* technique for any graph algorithm requires some additional data structure support. Each warp will read the entire cache-line and compute all elements in it. This requires each warp to know the corresponding mapping of vertexids for each cache-line. If we know the first vertex that maps to the given cache-line, then we can compute on elements of the cache-line very efficiently. To do this, we need to generate a temporary metadata structure called *firstvertexlist (FVL)* and use this FVL to traverse the CSR data structure. *Firstvertexlist array captures the vertex id of first edgelist element for a given cache-line.* For example, the first edgelist element's vertex id of CL_0, CL_1, CL_2 and so on are 0, 2, 3 and so on respectively becomes the first few elements in *FVL* array as shown in Figure 6.9. The firstvertexlist array's size equals the total number of cache-lines in the edgelist.

Listing 6.5 shows the pseudo-code for performing graph traversal using the proposed *ODT* technique. Using the graph example shown in Figure 6.9 and the *ODT* algorithm shown in Figure 6.10, we will explain how to perform graph traversal with *ODT* technique. Initially, as each warp works on the cache-line, we need to precompute the cache-line start and cache-line end index values of the given cache-line. This can easily be computed by knowing the number of elements in the cache-line (line 5-7 in Listing 6.5). For example, $warp_0$ works on CL_0 and hence the `clstart` and `clend` are 0 and 4 respectively. Another example, $warp_3$ works on CL_3 and hence the `clstart` and `clend` are 12 and 16 respectively.

```
void odt_technique(bam::array<T> edgeList,
        T *vertexList, T *fvl, num_elems_per_cl, ...) {
    warpIdx = get_warp_id();
    bam::array_ref edgeListRef(&edgeList);
    ...
    // calculate cache-line start & end indexes
    clstart = warpIdx*num_elems_per_cl;
    clend   = (warpIdx+1)*num_elems_per_cl;
    ...
    // look up the fvl array to get current vertexid;
    cur_vertexid = fvl[warpIdx];
    // start is always clstart in ODT technique
    start = clstart;
    // initialize the end
    end   = vertexList[cur_vertexid + 1];
    stop = false;
    ...
    while(!stop) {
        // clend is reached, terminate
        if(end >= clend){
            end  = clend;
            stop = true;
        }
        ...
        // do warp-wide compute of the algorithm
        for(i=start+laneIdx; i< end; i+=WARPSIZE)
                do_compute();
        ...
        // determine if more vertices are
        // part of the cache-line.
        // if yes, update start and end.
        if(end < clend) {
            cur_vertexid = cur_vertexid + 1;
            if(cur_vertexid<vertex_count){
                start= vertexList[cur_vertexid];
                end  = vertexList[cur_vertexid+1];
            }
            else {
                stop = true;
            }
        }
    }
 ...
}
```

Listing 6.5: Pseudo-code for performing graph traversal using the proposed *ODT* technique in each warp. Some of the boundary conditions are skipped for space.

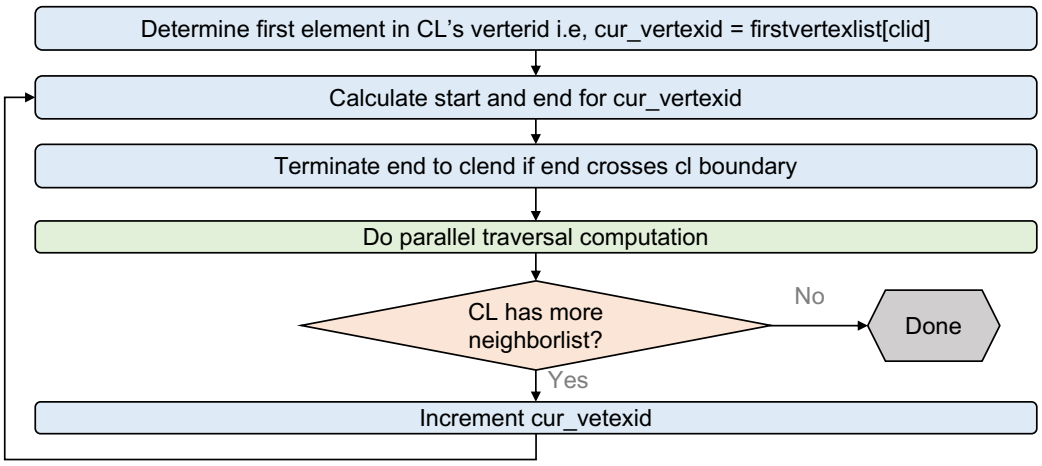


Figure 6.10: ODT algorithm for performing graph traversal using CSR data structure.

Now, the traversal operation requires two necessary operands: start and end indices of the vertexlist for which the warp is performing the computation. The start of the traversal in *ODT* technique is quite simple to calculate as it is equal to the `clstart`. If the neighborlist of a given vertex is shared between two cache-lines, as in the case of $vertex_6$ whose neighborlist spans CL_2 and CL_3, the start value is still equal to `clstart`. This is because, in the *ODT* technique, a warp is only allowed to work on the elements of the given cache-line, and no other cache-line data can be fetched or computed.

However, computing `end` of the traversal requires us to determine the vertexid of the first edgelist in the given cache-line. This is because each cache-line can map to neighborlists of multiple vertices, or a single neighborlist can span multiple cache-lines. Thus, the end of traversal for the given vertex can be either before the `clend` or the `clend` itself.

One can quickly determine if the cache-line maps to one or more neighborlist of vertices by looking up the firstverterlist array, as shown in line 14. The returned value, `cur_vertexid` tells the current vertexid of the first element of the cache-line. Thus, to determine if multiple vertices are mapped to the cache-line, one can look up the vertexlist with `cur_vertexid+1` value. If the returned value, `end` is equal to or greater than the `clend`, then there is only one or partial neighborlist of the `cur_vertexid` (line 19-22). If `end` is less than the `clend`, this implies more than one vertices are part of the given cache-line.

If multiple vertices are mapped to the given cache-line, at run-time, we

need to perform the traversal for the given neighborlist and then update the `cur_vertexid`, `start` and `end` variable to compute the traversal of the next vertex that is mapped to the cache-line to do the traversal (line 29-37). This requires us to add a loop that stops after processing all the elements in the cache-line (lines 15, 17, 21, 39). Listing 6.5 shows the implementation of *ODT* algorithm described in Figure 6.10 including the computation at line 24-27. Furthermore, if the neighborlist of a vertex spans multiple cache-lines, like in the case of $vertex_9$, there will be more than one warp working on behalf of that vertex. Thus, there may need to be atomic operations to avoid race conditions between the warps that work on the same vertex's neighborlist.

6.3.5 Generation of FirstVertexList

Now that we have figured out how to perform the computation using the *ODT* technique, the next important question to answer is how to generate *firstvertexlist* or `fvl` for any graph. As previously mentioned, `fvl` is an array that captures the vertexid of the first edgelist element for a given cache-line, and the size of the `fvl` is equal to the number of cache-lines in the dataset. The `fvl` is generated by pre-processing the vertexlist and does not come in a critical path. Although there are many ways one can compute the `fvl`, in this thesis, we propose an offline creation using an embarrassingly parallel implementation.

The `fvl` is generated in two steps as shown in Figure 6.11. The first step takes `vertexList` as input and generates a temporary `winnerList` whose size is equal to the number of cache-lines in the dataset. The `winnerList` contains a speculative vertexid (or winner vertex) of the first edgelist element for the given cache-line. Initially the output `winnerList` is set to maximum unsigned integer value (`max`). Then, `winnerList` generation kernel is launched where each thread first computes the cache-line index (`clid`) corresponding to each element in the vertexList. As the vertexList contains the start and end points of a neighborlist, the `clid` computation provides a speculative estimate on which cache-line index this neighborlist may belong to. Using the `clid` as an index, the `winnerList` generation kernel then inserts the minimum threadIdx as a value into the `winnerList` as shown in Figure 6.11. As this operation can result in a data race, an atomicMin op-

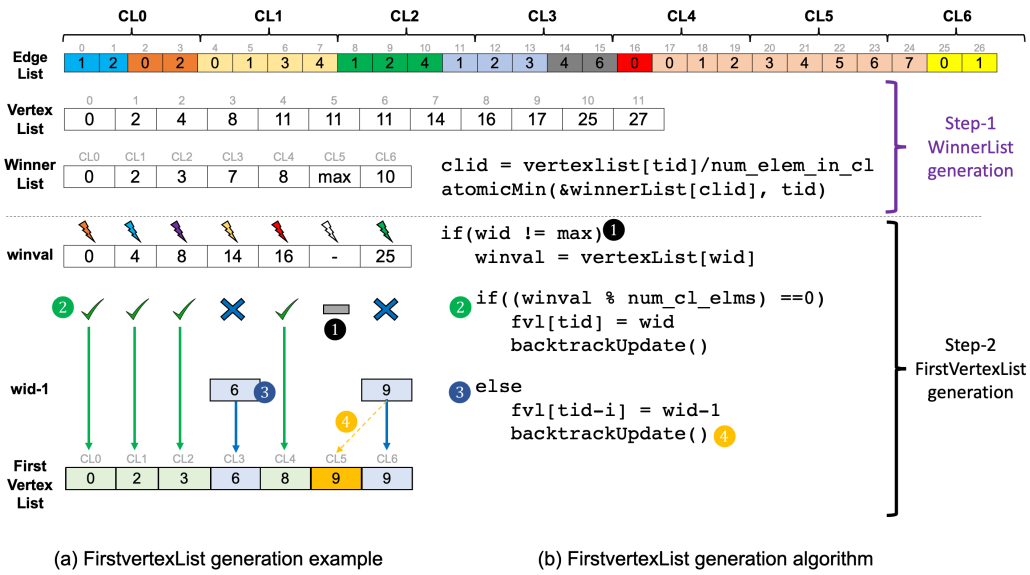


Figure 6.11: Firstvertexlist (`fvl`) generation algorithm with the example illustration.

eration is used. This step creates a speculative reverse map of the vertexids corresponding to the first edgelist element for the cache-line.

The top part of the Figure 6.11 shows an example of creating a `winnerList`. After the `winnerList` is created, a device synchronization is called before moving to the second step in generating `fvl`. A device-wide synchronization is needed to ensure a winner vertex is picked for all cache-line in the dataset. The second step takes the created `winnerList` and generates the `fvl`. Initially the `fvl` is set to zero and then `fvl` generation kernel is launched where each thread works on a valid winnerList entry to generate one or more entries in the `fvl` array.

The generation of `fvl` using the `winnerList` can be described using four different cases as shown in the second half of the Figure 6.11. First, if the thread in the `fvl` generation kernel determines the value of winner is equal to `max` by looking up the `winnerList`, then the thread terminates without adding any element to the `fvl` (❶). Some other thread in the `fvl` generation kernel is responsible for inserting the value for this entry. This is because an entry in the `winnerList` with `max` value essentially implies that the cache-line is part of a large neighborlist spanning more than one cache-line and hence does not contain any start or endpoints. CL_5 is an excellent example of this situation for the example graph.

Second, if the thread in the `fvl` generation kernel determines the value of winner is less than the `max` value, then using the winner value (`wid`), the thread will index the `vertexList` to obtain `winval`. If the `winval` is a perfect multiple of the number of elements in the cache-lines, then the vertex's neighborlist starts precisely at the cache-line boundary; hence the `wid` is the vertex id of the first edgelist element for the given cache-line. Thus, the thread goes ahead and inserts `wid` into the `fvl` as shown by green lines (❷) in the Figure 6.11 (CL_0,CL_1, CL_2 and CL_4 are excellent examples of this). After inserting the `wid` to `fvl`, the thread checks if the previous entry in the `winnerList` was `max` and if yes, then it inserts `wid-1` to the `fvl` until it finds a valid entry in the `winnerList`. This step is represented by the function `backtrackUpdate())` in the Figure 6.11.

Third, if the `winval` is not a perfect multiple of the number of elements in the cache-lines, then the cache-line has a partial neighborlist. CL_3 and CL_6 in the Figure 6.11 are excellent examples of this case. Here, the winner vertex must be decremented by one so that the `fvl` has the correct value of the vertex id for the first edgelist element. Then the thread goes ahead and inserts the decremented value to the `fvl` as shown by ❸ in Figure 6.11.

Fourth, after inserting the correct value to the `fvl`, the thread requires to check if the previous entry in the `winnerList` was `max` and if yes, then it inserts the decremented value to the previous entry in the `fvl` as shown by the ❹ in Figure 6.11. This step is represented by the function `backtrackUpdate()` in the figure. Once all the elements in the `fvl` is filled, the `fvl` generation process is complete and `winnerList` can be freed.

The generation of `fvl` can be done in an embarrassingly parallel manner, and thus, it does not add significant overhead during *ODT* technique-based graph analytics. Although the current implementation has not been optimized and has uncoalesced memory accesses and thread divergence, the *ODT* library API can generate `fvl` list in less than 10ms for a 134M node graph on an NVIDIA A100 GPU. Moreover, the `fvl` can be reused multiple times after generating it once, further amortizing the pre-processing overhead.

6.3.6 Discussions and Limitations

The best part of the *ODT* technique is that it *regularizes the irregular access at run-time*. Before the *ODT* technique, one needed to look up the vertex list in the CSR data structure to determine the index in the edgelist and then issue a read operation to the edgelist. This introduced an irregular access pattern. However, after the *ODT* technique, each cache-line is implicitly read in a perfectly coalesced manner, maximizing the cache-line usage and then fetching the next cache-line on-demand. Another way to look at this is that each cache-line we are bringing in is equivalent to a tile of data in a sparse matrix! This enables a very powerful abstraction for computing with sparse data structures, which traditionally were impossible.

Furthermore, most prior work in literature [81,84] working on sparse data-structures preprocess the large graph to create tiles and optimize access patterns for the given loaded tile. These prior works propose to optimize the access to the sparse matrix by modifying the underlying CSR data structure with an optimized zero-padding technique or representing the data in a memory-efficient manner for coalesced memory accesses. However, they incur significant overhead requiring additional metadata or expensive preprocessing stage consuming already constrained hardware resources.

Compared to these, the *ODT* technique requires a metadata array whose size is equal to number of tiles in the dataset and the metadata array is generated dynamically at run-time in an embarrassingly parallel manner. This enables *ODT* technique to generate tiles on-demand and loaded when needed. This reduces the programmer burden significantly and enables the freeing up of compute resources for other tasks. Moreover, *ODT* technique maps each warp to work on a tile of edgelist, enabling coalesced access to the memory without the need for explicit optimizations. Although in the previous section, we kept the tile size as the cache-line size. However, for all practical purposes, this does not require to be the case. The ODT technique works perfectly for any tile size: smaller or more significant than the cache-line. These make *ODT* technique an up-and-coming candidate to study in the future to apply to algorithms such as sparse-matrix vector, sparse matrix-multiplication, or any other algorithm that work on CSR data structures.

The current implementation of the *ODT* technique has much room for optimization. For instance, we can enable shared memory to keep the elements

from firstvertexlist to reduce the lookup cost overhead or even load the tile of data from the backing memory and store it in the shared memory. We plan on supporting shared memory optimization as part of future work. If we are storage latency bound, we can further enable sub-warp optimization techniques to increase the overlapping I/O requests to the storage. We could also optimize the BaM cache exclusively for the *ODT* technique and obtain even more performance and space efficiency. Third, it is entirely possible to create firstvertexlist dynamically. However, supporting such modifications and abstraction using the *ODT* technique is left as future work.

The *ODT* technique has a critical limitation at present. The *ODT* technique assumes concurrency within the traversal is handled by lock-free algorithms or by atomic operations, but if there is a critical section, then a complex locking scheme might require implementation, which can severely degrade performance. However, we have yet to come across a workload that requires such complex concurrency and is left as future work.

6.4 Summary

Naively enabling BaM may not provide any performance benefit out of BaM for many emerging workloads. This is because to utilize BaM best, one needs to perform coalesced accesses to the BaM cache and yet generate a sufficiently large number of I/O requests to BaM I/O stack. To this end, in this chapter, we proposed a set of BaM optimization techniques to achieve peak performance of BaM. We proposed a novel first principle-based modeling technique for the root cause of the performance limitation. We then proposed cache-line aware optimization, a simple generalizable optimization, capable of maximizing the bandwidth achieved from the BaM cache and storage. We discussed the limitations in the cache-line aware optimization when applied to the graph analytics workload. We then proposed a novel *ODT* technique, a CSR data-structure specific optimization whose ultimate goal is to maximize the reuse of the cache-line once loaded from the storage while exploiting the GPU parallelism to generate many concurrent overlapping I/O requests to hide the storage access latency. In the next chapter, we will evaluate how these techniques performed on respective emerging workloads.

Chapter 7

End-to-end Application Case Studies

Now that we have a better understanding of the BaM system and its characteristics for various access patterns, we can talk about how to use BaM effectively for end-to-end workloads. We start by studying the applications that have data-dependent accesses patterns. Data-dependent access patterns are extensively used in popular applications, e.g., graph analytics, recommender systems, and in frameworks, e.g., RAPIDS for accelerating analytical queries. As shown in Chapter 3 and Chapter 4, many of these applications have data structures that cannot fit within the available memory, and using current state-of-the-art techniques provide poor performance due to I/O amplifications, complex tiling strategies, and overheads from memory and I/O management.

This chapter shows that BaM excels on applications with data-dependent execution. We study two emerging workloads in depth: graph analystics (see § 7.1) and data analytics (see § 7.2) with variety of large datasets. We apply the optimization discussed in §6 to these workloads and study the contribution of each component of BaM for overall execution. We also show that the proposed *ODT* technique can reach peak performance out of BaM for graph analytics workload while the transposed work assignment and cache-line aware optimization reach near optimal performance with simple modifications to the existing implementations. We then extend the cache-line aware work assignment optimization to data analytics workload and show that it can provide up to 5× performance boost over the baseline implementation.

Lastly, we also evaluate BaM on two regular workloads: vectorAdd and reduction algorithms (see § 7.3) and show that BaM provides on-par or slightly lower performance compared to the complex tiling strategy used in the state-of-the-art implementations. Although BaM does not excel on regular workloads, it greatly simplifies the programmer's burden when accessing the data stored in the storage.

Overall this application level end-to-end evaluation shows that

- proposed optimization techniques greatly help with maximizing the performance of each application and achieving peak performance out of BaM system.
- BaM's performance is either on-par with or outperforms the state-of-the-art solutions for all types of studied workloads.
- BaM's design is agnostic to the type of storage used, enabling application-specific cost-effective solutions.
- BaM reduces I/O amplification and CPU orchestration overhead significantly for data-analytics workloads (see § 7.2).

7.1 Performance Benefit Of BaM For Graph Analytics

In this section, we evaluate the performance benefit of BaM for graph analytics workload. Graph analytics is a complex emerging workload and notoriously hard to get good performance. Because of this, graph analytics also make a good test vechicle to evaluate and understand various design implications of BaM. Table 7.1 summarizes the key description and insights obtained from various experiments performed with BaM on graph analytics workload.

7.1.1 General Setup

The goal of the BaM is to provide competitive performance against the host-memory-based DRAM-only graph analytics solution. To this end, a highly optimistic target baseline `T` allows the GPU threads to directly perform coalesced fine-grain access to the data stored in the host-memory during graph analytics execution [5][1]. We use a CPU with a large DRAM capacity to host the input graphs and make a direct comparison of the performance between BaM and `T`.

[1]This is a highly optimistic baseline as the data is assumed to be preloaded to the host-memory and no time is spent on accessing the data from storage.

Table 7.1: List of experiments and key insight obtained from BaM on graph analytics workload.

No	Experiment	Description (D) and Insight (I)
1	Comparison with the Target (T) system, scaling and cost benefit (§7.1.2)	**D:** Comparison of BaM performance with the state-of-the-art baseline target system. The baseline system assumes an ideal storage bandwidth and peak throughput out of PCIe interface after loading the data to the host memory. BaM performance is compared with two configuration: with single and four Intel SSD at 4KB cache-line size. Four SSD configuration provides ISO bandwidth between the Target system and BaM.
		I: BaM can saturate one Intel SSD with ease and then linearly scale to four Intel SSD. Overall BaM achieves 1.00× and 1.49× performance benefit over an highly optimistic target system. Also, BaM offers upto 4.45× and 6.63× perf/$ cost benefit over the target baseline system using expensive four Intel Optane SSDs. Compared to the all in GPU-HBM memory solution, BaM is upto 4.27× slower.
2	BaM performance breakdown (§7.1.3)	**D:** Slices up the overall execution time of the BaM to understand the breakdown of time from various components
		I: With BaM four SSD configuration, compute time constitute to 10-44%, BaM cache-access API constitutes to 4-45% and storage access constitutes to 18-80% of overall execution time based on type of workload and datasets
3	Impact of SSD type (§ 7.1.4)	**D:** BaM evaluation with Samsung datacenter grade DC1735 and consumer grade Samsung 980 pro SSDs
		I: For Samsung DC 1735 SSDs, BaM provides similar performance as the Intel Optane SSDs as both have similar SSD performance characteristics. For consumer grade Samsung 980pro SSDs, compared to the Intel Optane SSDs (datacenter grade), BaM is only upto 3.21× slower.
4	Impact of cache-line size (§ 7.1.5)	**D:** Evaluate the impact of cache-line on BaM performance on various graph workloads with different datasets.
		I: 4KB cache-line size provides the best performance for all applications tested. At 512B cache-line size, the application is only 1.86× and 2.36× slower compared to the 4KB cache-line size.
5	Impact of various optimizations (§ 7.1.6)	**D:** Captures and discusses several trade-offs associated with various optimizations and how they affect performance in the context of the BFS and CC workloads.
		I: *ODT* technique provides the best performance across most workloads and dataset. However for 8GB BaM cache, transposed work assignment and cache-line aware coarsening provides slightly lower performance and can be enabled with few lines of modifications in the GPU kernel code.
6	Impact of cache size (§7.1.7)	**D:** Measures the impact of cache-size on the BaM performance
		I: *ODT* technique performance is not dependent on cache size and can work with cache size as small as 16-32MB for all workloads providing peak performance out of BaM.
7	Impact of number of queues and queue depth(§ 7.1.8)	**D:** Sensitivity study to evaluate the impact of number of queues and queue depth on BaM system
		I: *ODT* technique requires less than 512 CQ/SQ entries to reach peak performance with 4 Intel Optane SSDs at 4KB access granularity.

Because the access patterns and performance characteristics of graph traversal applications can vary significantly across different types of graphs, the experiments should cover a diverse set of graphs. We use the graphs listed in

Table 7.2: Graph Analytics Datasets.

Graph	Num. Nodes	Num. Edges	Size (GB)
GAP-kron (K) [38]	134.2M	4.22B	31.5
GAP-urand (U) [38]	134.2M	4.29B	32.0
Friendster (F) [3]	65.6M	3.61B	26.9
MOLIERE_2016 (M) [4]	30.2M	6.67B	49.7
uk-2007-05 (Uk) [39, 40]	105.9M	3.74B	27.8

Table 7.2 for the evaluation. K, U, F, and M are the four largest graphs from the SuiteSparse Matrix collection [35] while the Uk is taken from LAW [36]. These graph datasets cover diverse domains, including social networks, web crawls, bio-medicine, and synthetic graphs.

We run two graph analytics algorithms, Breadth-first-search (BFS) and Connected Components (CC), with the target system and BaM with different SSDs listed in the Table 5.3. For both systems initially, the edgelist is stored in the storage. For target system T, we load the entire edgelist to the CPU memory and pin it. We calculate the storage I/O time for the target system T with the assumption that the CPU and the operating system do not add any overhead and can achieve peak storage bandwidth when loading the edgelist to the CPU memory. Thus, to calculate storage access time, we divide the edgelist size in bytes over the peak achievable theoretical storage bandwidth in gigabytes per second. Although this is a theoretical and highly optimistic assumption, we want to compare BaM to the best possible baseline out there.

We apply the optimization discussed in §6 for both target baseline system and various BaM configurations. For BFS, we report the average run time after running at least 32 source nodes with more than two neighbors. We do not execute CC on the Uk dataset since CC operates only on undirected graphs. We also use the best performing optimization for BaM unless explicitly specified (in most cases it is the proposed *ODT* technique; see § 7.1.6 for more details). By default, unless explicitly stated, we run experiments in the target (T) baseline system and BaM with four Intel Optane SSD and one NVIDIA A100 GPU. Also, unless explicitly stated, the cache size for BaM is set to 8GB, and the cache-line size is set to 4KB.

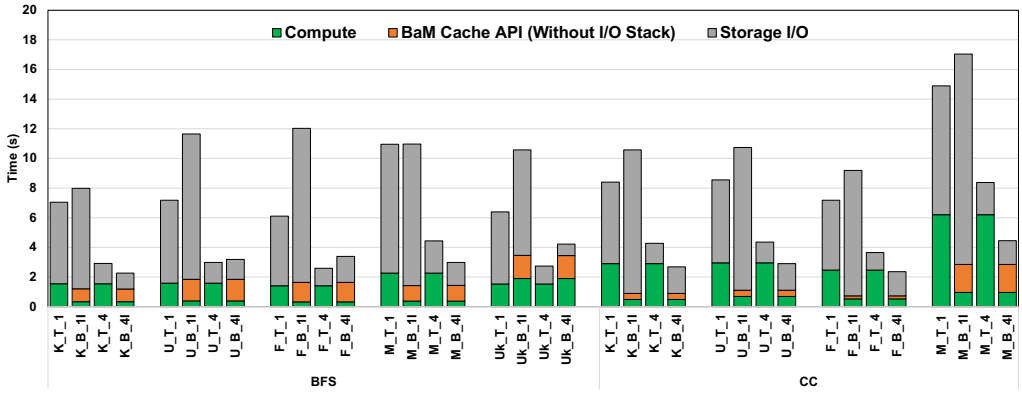


Figure 7.1: Graph analytics performance of BaM with cache-line of 4KB and the Target (T) system using Intel Optane SSD. End-to-end execution time is reported in the Y-axis (lower is better). On average, BaM's is 1.00× and 1.49× faster than the Target system with 4 SSD configuration for BFS and CC workload respectively. Even with single Optane SSD, BaM is only slow by 1.4× and 1.27× (Execution time lower is better).

7.1.2 Overall Performance With Intel SSDs

Figure 7.1 shows the performance of both the target system (`T`) and BaM with single (`B_1I`) and four (`B_4I`) Intel Optane SSDs. The single SSD BaM configuration (`B_I`) is upto 8.54× slower than the target `T` system without considering the initial file loading time for the `T` system. However, if we include initial file loading time for the `T` system, BaM is on average slow by 1.43× and 1.27× for BFS and CC workload respectively. This is because BaM's `B_I` is limited by the throughput of the single SSD's ×4 Gen4 PCIe interface while the target system `T` greatly benefits from full ×16 Gen4 PCIe bandwidth between the host and GPU.

We now scale the number of SSDs to four (`P_4I`) and replicate data across SSDs to increase the BaM's aggregate bandwidth. With four SSDs and 4KB cache-line size, the underlying BaM infrastructure provides similar bandwidth as ×16 Gen4 PCIe interface as in the target system. Compared with the target system `T` with file-loading time, *BaM with four Intel Optane SSDs provides on average 1.0× and 1.49× speedup on BFS and CC applications respectively.* This is mainly because in the case of BaM, the storage and compute operations are fully overlapped, can saturate PCIe Gen4 ×16 bandwidth, and observes much less I/O amplification. In contrast, the target system `T` needs to wait until the file is loaded into host memory before it can offload the compute to GPU. The superior host-memory bandwidth of the

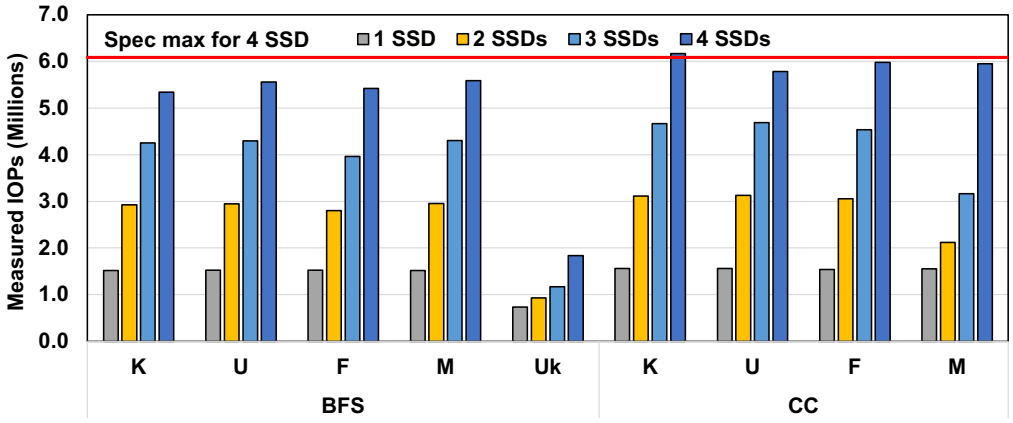


Figure 7.2: Measured IOPs with different number of SSDs with the BaM system. Each SSD is capable of providing ∼1.5MIOPs. With 4 SSDs, the peak IOPs as per specification is ∼6 MIOPs. BaM is able to saturate across most of the workloads with ease, except BFS with Uk graph. For most datasets and workload, measured IOPs with BaM is with-in 0-9.2% of the peak storage IOPs (except for BFS on Uk graph). (IOPs higher is better)

`T` system cannot overcome the initial file loading latency. As a result, BaM achieves either on-par or higher end-to-end performance.

Scaling and Measured IOPs: Figure 7.1 also shows the speed up obtained by the BaM with the additional number of Intel Optane SSD compared to the single SSD configuration. As expected, both the target `T` system and BaM gain scaling benefit with additional number of SSDs. The target `T` system sees 2.4× and 1.96× theoretical[2] scaling speed up for BFS and CC workload when the number of SSDs is increased from one to four. On the other hand, BaM observes a scaling speed up of 3.48× and 4× (geomean) for BFS and CC workload, respectively. The main reason why the BFS workload scaling is worse than CC in the case of BaM is due to the BFS algorithm's poor performance with the `Uk` dataset. To understand what is happening, let us measure the storage IOPs obtained by these workloads on BaM and see if it saturates the storage stack.

Figure 7.2 shows the measured IOPs from BaM I/O stack across various datasets and graph analytics workloads. From Table 5.3, each Intel Optane SSD is capable of providing ∼1.5M IOPs and four Intel Optane SSD configuration should provide ∼6 MIOPs. BaM can nearly saturate all SSDs

[2]We used this term for target `T` system mainly because we assume RAID0 [85] configuration can achieve a linear performance scaling and can reduce the storage I/O time by 4× when the number of SSDs is increased from one to four.

across most of the workloads with ease, except with the BFS algorithm on the `Uk` graph. This is because `Uk` graph has a varying degrees of depth across random sample iterations, and many nodes have a tiny neighborlist. With tiny neighborlist and deep depths (>100+), the total number of overlapping concurrent I/O requests that are indeed submitted in each iteration of BFS to the BaM I/O stack is deficient. Out of 100+ BFS kernel executions, less than 10 have enough I/O requests to saturate the storage, while the rest accesses a few 10s of cache-lines. This exposes cache access overhead to the application, and further addition of SSDs to the system does not provide additional performance benefits.

Compared to the `T` baseline system, for BFS on `Uk` graph, BaM with four SSD configuration is 1.54× slower. One way to address this issue is to use a frontier-based BFS algorithm. With a naive BFS frontier-based approach [83], BaM achieves 1.54× the speedup over the target (`T`) baseline system with 4 SSD configuration and is able to hit upto 3.9M IOPs. Further optimization in the BFS frontier-based implementation can provide additional speedup but is left as future work.

With this, we have shown that the BaM performance for these applications (except `Uk` graph as it is still bounded by the concurrent I/O requests) is bounded by storage access bandwidth.

Cost benefit: What is more exciting is the cost benefit provided by BaM system when compared to the baseline target system. To do this, instead of taking absolute prices, we will use $/GB metric used in Table 5.2 to discuss perf/$ achieved by BaM when compared to the host-DRAM only solution assuming constant memory capacity. Note that this is a system-cost comparison and not a media cost comparison, and SSD cost reduces with the increased capacity. BaM with Intel Optane SSD is at least 4.45× cost-effective when compared to the DRAM-only solution for the same memory capacity. Thus, with the single Intel Optane SSD, BaM offers a minimum of 0.69× and 1.13× perf/$ (geomean) compared to the host-DRAM only solution without the initial file loading time for BFS and CC workloads respectively. With the initial file-loading time, BaM offers a minimum of 3.1× and 3.48× (geomean) perf/$ compared to the target baseline system for BFS and CC workloads, respectively, despite using the most expensive SSD out there.

With four Intel Optane SSD, the BaM perf/$ further improves. BaM can provide 2.32× and 4.48× (geomean) perf/$ compared to the host-DRAM

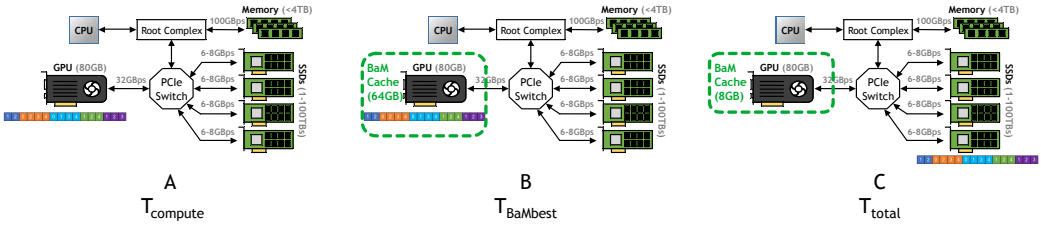


Figure 7.3: Setup description for BaM performance breakdown.

only solution without the initial file loading time for BFS and CC workloads, respectively. And with the file reading time included, BaM perf/$ offers a minimum of 4.45× and 6.63× (geomean) perf/$ compared to the target baseline system for BFS and CC workloads, respectively. And this perf/$ benefit will undoubtedly increase with the additional storage capacity added to the system or by changing the storage to a less expensive drive. Further cost benefits from BaM solution are available such as reduced total cost-of-ownership (due to lower power consumption) and are not considered in these measurements.

Comparison with the GPU-HBM solution: We now compare the performance of BaM when all the datasets fit in the GPU-HBM memory. We assume the edgelist is preloaded into the GPU-HBM memory for this evaluation. In the case of all GPU-HBM solutions, the entire HBM bandwidth of 1.55TBps is at the application's disposal. On the other hand, BaM with four Intel Optane SSDs and only 24GBps of external bandwidth, BaM see only 3.21× and 4.27× slowdown compared to the all-in GPU-HBM memory. This is exciting because even for a complex workload like graph analytics, BaM with the cache-line aware optimization can maximize the access bandwidth when accessing the data stored in the storage stack. Moreover, BaM provides a much faster, economical, and more practical method to scale GPU's effective memory capacity when compared to the fixed small size HBM memory capacity increase seen in each generation.

7.1.3 BaM Performance Breakdown

We now slice up the overall execution time of the BaM to understand how each component in BaM contributes to the overall execution time.

Setup: Figure 7.3 shows the overall setup of the experiment. Initially, we

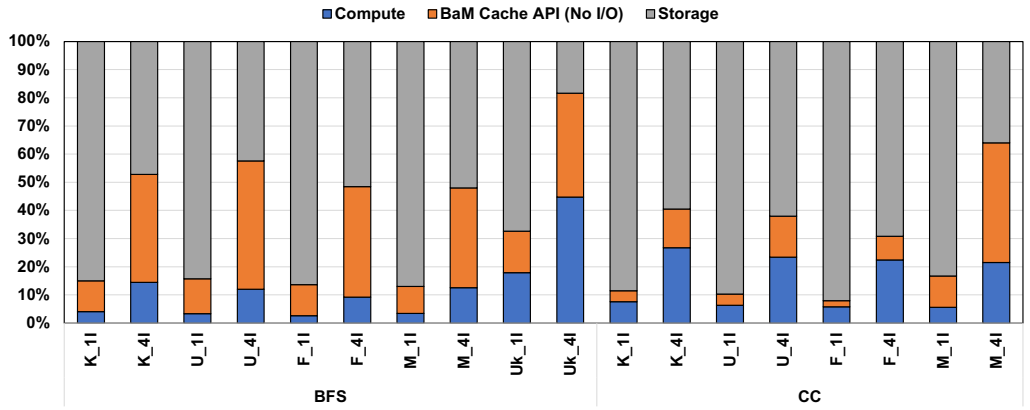


Figure 7.4: BaM performance breakdown for graph analytics workload with the optimized implementation. 1I stands for single Optane SSD while 4I stands for four Intel Optane SSD in the system.

load the entire dataset in the GPU-HBM memory and measure the execution time. This captures the best possible performance that one can achieve if all data is the GPU-HBM memory and is represented as Tcompute (or *Compute* in blue color). Next, we measured the total execution time when the data is in the GPU-HBM memory, but each application access must go through the BaM cache API. This is the best case performance one can achieve if all accesses were hit in the BaM cache and is represented as TBaMbest. In the figures, this is represented as *BaM Cache API* in orange. Next, we constrain the BaM cache size to 8GB and measure the total execution time which is shown as Ttotal. Using these three execution time measurements, we can compute the overhead introduced by the BaM cache very quickly. We can subtract the compute time from TBaMbest to capture the cache API overhead. Similarly, we can measure the time spent for storage access (*Storage I/O*) by subtracting the cache overhead and compute time from the total execution time. We acknowledge this methodology is not perfect and does not consider overlapping accesses but is sufficient to understand the average distribution of time spent in each component of the BaM system.

Figure 7.1 shows the absolute execution time breakdown across different layers in BaM while Figure 7.4 shows the relative distribution of time across three key components in the BaM system. First, these two figures show an overhead when accessing the data via BaM cache. For a single SSD, the cache overhead is about 2-15% of overall execution time. For four SSD, the cache overhead varies from 4-45% of overall execution time. With a single SSD, the

application was bounded by the storage throughput, and adding additional SSD helped improve the bandwidth, thus minimizing the storage access overhead significantly. The cache overhead mainly comes from contention when managing the metadata, load imbalance, limited hardware cache capacity resulting in GPU LLC thrashing, long latency atomic operation, and warp scheduling in the face of polling threads. Some of this cache-overhead can be resolved with the modern GPUs using cache-hints to pin specific commonly accessed metadata in the GPU-LLC. Modifying the BaM cache implementation to replicate the cache state can reduce the cache contention overheads and provide a performance benefit. We leave this optimization as future work.

Second, an upper limit exists on scaling the number of SSDs to saturate the application performance irrespective of type and number of SSDs. This upper limit is bounded by the rate at which the GPU threads can submit I/O requests and how efficiently the applications utilize the cache. For most workloads, with the *ODT* technique, current BaM implementation is near its peak performance.

Third, with the optimal access pattern, cache design, and sufficient number of SSDs, it is possible to hide the storage access latency in application execution by exploiting the massive parallelism offered by the GPU. Currently, from Figure 7.2, BaM is bounded by the storage throughput from the storage system, and as long as the storage system can provide the same level of performance, it is possible to achieve peak performance out of BaM.

7.1.4 Impact Of SSD Type

We now evaluate BaM with different types of SSDs: datacenter grade Samsung DC 1735 and consumer grade Samsung 980pro. Figure 7.5 shows the slowdown observed by the BaM prototype configured with four Samsung DC 1735 and Samsung 980pro SSDs compared with four Intel Optane SSDs. Samsung DC 1735 and the Intel Optane SSD have similar performance for almost all workloads. This is because these SSDs provide similar random read IOPs for 4KB transfers. Despite the latency of Samsung DC 1735 being more than 2× that of Intel Optane, the BaM performance does not change much. Samsung 980pro SSDs have more than 30× more latency and 2 ×

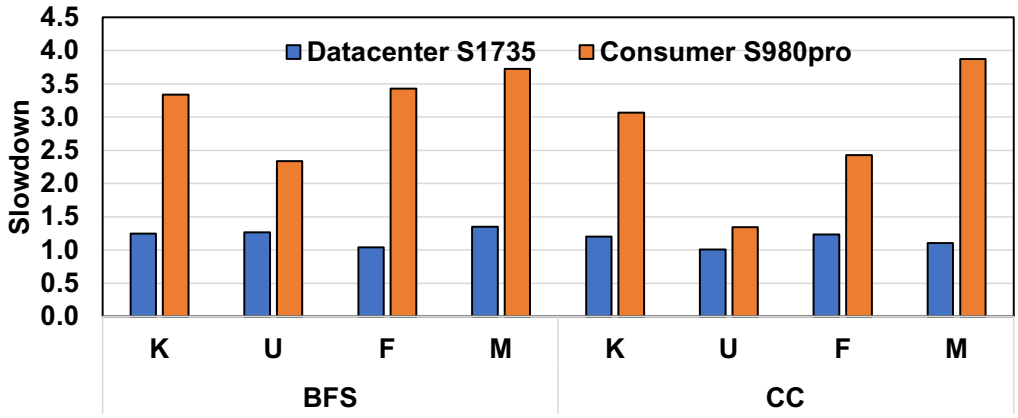


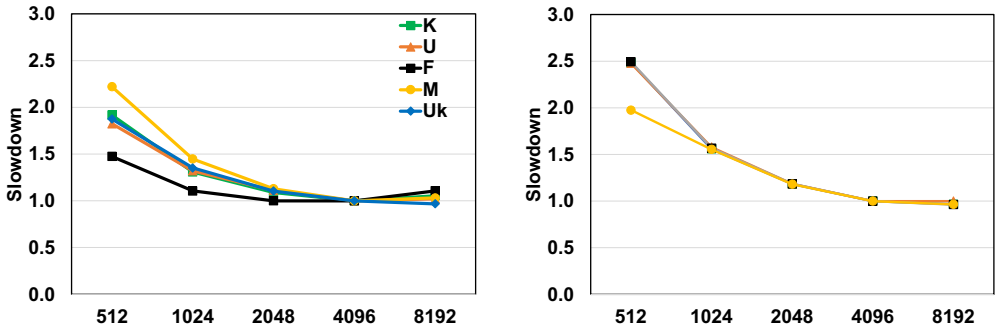
Figure 7.5: The slowdown observed by the BaM configuration with four Samsung DC 1735 and Samsung 980pro SSDs when compared with four Intel Optane SSDs. (Slowdown lower is better)

lower throughput per SSD when compared to the Intel Optane SSDs, and yet BaM prototype with the Samsung 980pro SSD is on average 3.21× and 2.68× slower for BFS and CC workload compared to the BaM with the same number of Intel SSDs. Increasing the number of consumer-grade SSDs in the system and fine-tuning further can reduce this gap. Despite that, these are very encouraging results for consumer-grade SSDs as they provide the best value among all the SSD technologies. One can also afford to increase the number of consumer SSDs used and keep the cost much lower than data center-grade SSDs as long as the workload is read-intensive.

7.1.5 Impact Of Cache-line Size

We vary the cache-line size of the BaM software cache from 512B to 8KB to understand the impact of access granularity on graph analytics workloads. Recall that the BaM cache-line size determines the granularity of access to the storage. The evaluation is done using sta four Intel Optane SSDs shown in Figure 7.6.

From Figure 7.6a and Figure 7.6b, as we decrease the cache-line size from 4KB to 512B, BFS and CC workloads on average slow down by 1.86× and 2.361× respectively. This is because graph workloads exhibit spatial locality within their neighbor lists and can benefit from larger accesses. The 4K granularity takes advantage of the spatial locality of large neighbor lists in



(a) BFS (b) CC

Figure 7.6: Impact of cache-line size on BaM performance for BFS and CC workload with four Intel Optane SSD compared to 4KB cache-line size. 4KB cache-line size provides best performance. (Slowdown lower is better.)

some of the graphs, and the extra bytes transferred for smaller neighbor lists don't deteriorate performance as the PCIe bandwidth is not over-saturated. Increasing the number of SSDs in the case of 512B cache-line size access does not help further improve the performance (not shown). This is because at 512B cache-line sizes, the BaM is bounded by the throughput of the cache and its ability to generate concurrent I/O requests to the BaM I/O stack. Improvements in BaM cache design can address this problem and be left as future work.

Furthermore, BaM's fine-grain access reduces I/O amplification, thereby improving the effective bandwidth. Otherwise, the application would have had up to an 8× slowdown upon reducing the cache-line size from 4KB to 512B. Increasing the cache-line size from 4KB to 8KB barely impacts the overall performance. This is because at 4KB with four Intel Optane drives, the application is near its peak performance and saturating the storage. Increasing the cache-line size further does not improve the already saturated storage.

7.1.6 Impact Of Various Optimization

We next evaluate the contribution of various optimization on the overall execution time of graph workload when running with BaM. Description of each of these optimizations are described next:

1. **Initial**: This captures the baseline implementation discussed in $6.3.1.

Here, the initial graph algorithms are ported directly to BaM, and the edgelist is moved to the storage and accessed via `bam::array<T>` API. *Initial* implementation uses a vertex-centric approach where each warp works on the vertex that is being visited in the current iteration, and all threads in the warp collaboratively walk through the neighborlist.

2. **Trans+optstride**: Transposed with optimal stride follows a vertex-centric approach where each warp works on vertexid and consecutive warps work on optimal stride away. More details on the design is discussed in §6.3.2. A parameter sweep on stride is performed to obtain an optimal stride giving the best performance. For most graphs, the optimal stride is 128 except for `M` graph, which has the optimal stride of 512.

3. **Coarsen**: This implements cache-line aware warp coarsening technique using vertex-centric approach discussed in §6.3.3. Here each warp work on eight consecutive neighboring vertices' neighborlist. The coarsening factor of 8 gives the sum of neighborlist size nearest to the cache-line size in these workloads.

4. **ODT**: This implementation follows the design principles laid out in §6.3.4. Instead of a vertex-centric approach, each warp works on a tile of data. The *ODT* technique tries to maximize each cache-line reuse and generate as many concurrent I/O requests as possible from the given system. The tile size is equal to the cache-line size.

We evaluate the above designs on NVIDIA A100 GPU with four Intel Optane SSD. The cache-line used is 4KB, and the cache capacity is 8GB. For all experiments, we will be picking the max NVMe CQ/SQ queue depth and the max number of queues supported by the device. We report total execution time, the breakdown of each component, and its variation with respect to various optimizations. The breakdown is calculated in the same manner as the BaM performance breakdown discussed in § 7.1.3.

Figure 7.7 and Figure 7.8 shows the overall execution time with breakdown of each component across various optimizations when running with BaM on BFS and CC workload with different datasets respectively. For both algorithms, the *initial* baseline only achieves up to 1.43 MIOPs (geomean) and 0.92 MIOPs (geomean) for BFS and CC workload, respectively. This

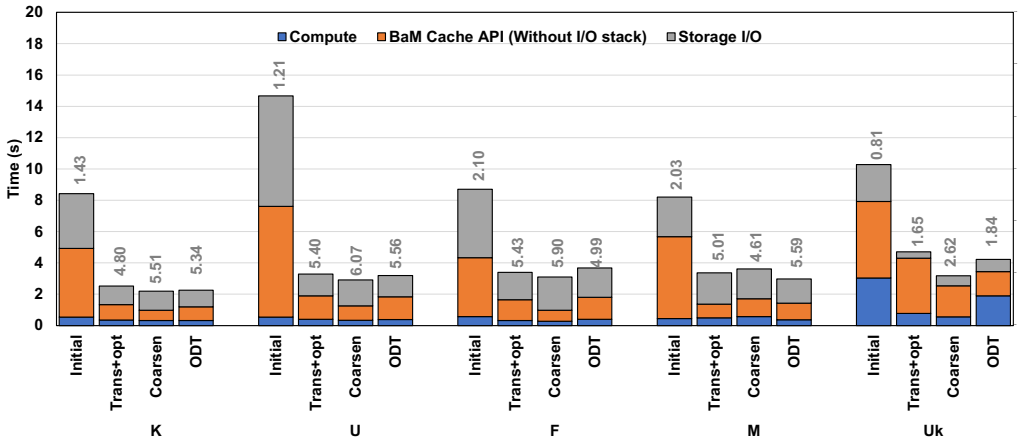


Figure 7.7: Overall execution time breakdown for BFS algorithm with various optimization in BaM using one NVIDIA A100 GPU and four Intel Optane SSDs with 8GB BaM cache. The grey color numbers above each bar graph represent the measured IOPs in millions. (Time lower is better, IOPs higher is better.)

is far lower than the achievable ∼6MIOPs of BaM. This is because, across both algorithms, the *initial* implementation provides deficient performance as multiple warps contend on the same cache-lines increasing the cache-access overhead and reducing the number of concurrent overlapping I/O requests submitted to the storage. The cache-overhead constitutes 53% and 59% of the overall execution time of BFS and CC workloads, respectively. Thus the initial baseline implementation cannot saturate the BaM I/O stack.

The *Transposed* work assignment, although improving the performance by 1.28× and 1.40× for BFS and CC workload, respectively, is still far from saturating the storage (not reported in the graph). With the optimal stride, the transposed work assignment (*Trans+optstride*) when compared to the *initial* implementation provides 2.90× and 5..68× performance benefits. Significant reduction in the cost of accessing cache-access API (3.45× and 7.7× for BFS and CC workload compared to the *initial*) is helping the *Trans+optstride* optimization to achieve such performance. The transposed work assignment can generate significantly higher IOPs but falls short of saturating the storage. This is because this optimization still suffers from the contention for metadata in the cache and suffers cache thrashing. Yet transposed work assignment sees 4.10MIOPs and 4.79MIOPs for BFS and CC workloads, respectively.

The *Coarsening* optimization provides a similar performance benefit as the

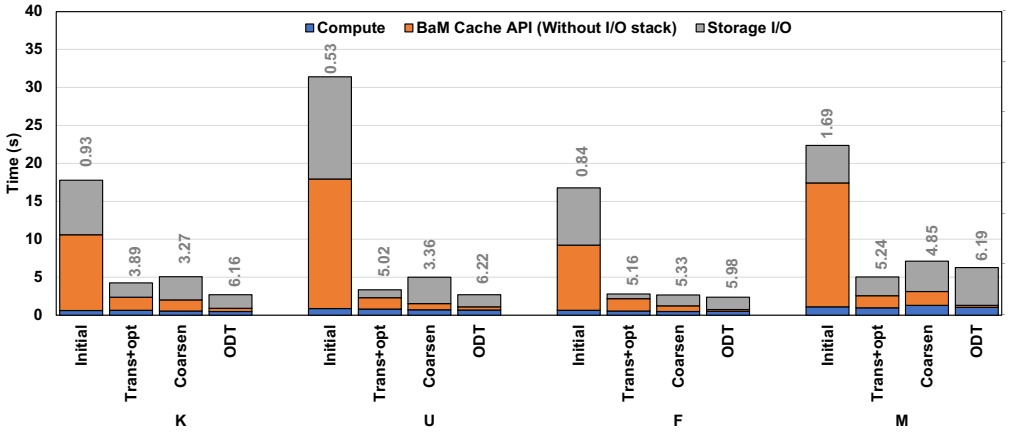


Figure 7.8: Overall execution time breakdown for CC algorithm with various optimization in BaM using one NVIDIA A100 GPU and four Intel Optane SSDs with 8GB BaM cache. The grey color numbers above each bar graph represent the measured IOPs in millions. (Time lower is better, IOPs higher is better.)

Trans+optstride work assignment. Compared to the *initial* implementation, this optimization is 3.31× and 4.58× better for BFS and CC workload respectively. This optimization tries to maximize the cache-line reference reuse within a cache-line and generates many concurrent overlapping I/O requests to the storage. Because of this, *Coarsening* optimization provides up to 4.73 MIOPs and 4.10 MIOPs for BFS and CC workload, respectively. Although it does not saturate like the *Trans+optstride* work assignment, it shows the fact that cache-line aware optimization is equally applicable to these workloads with minimal code changes.

Lastly, across both workloads, as expected, *ODT* technique on-average (geo-mean) provides speed up of 3.07× and 6.65× compared to the *initial* implementation. This is because *ODT* tries to maximize the reuse of the cache-line once brought into the cache, thus maximizing the use of BaM cache bandwidth and launching a sufficiently large number of warps to hide the storage access latency. Across both workloads, *ODT* technique has the least cache-access API overhead (constituting 40% and 10% of overall execution time). *ODT* technique can achieve peak throughput out of the storage, reaching up to 6.22M IOPs for some workloads, with geo-mean being 5.34M IOPs and 6.18M IOPs for BFS and CC workloads, respectively.

The average (geomean) IOPs is slightly lower for BFS workload with *ODT* technique is mainly because of `Uk` graph. As we previously stated in § 7.1.2,

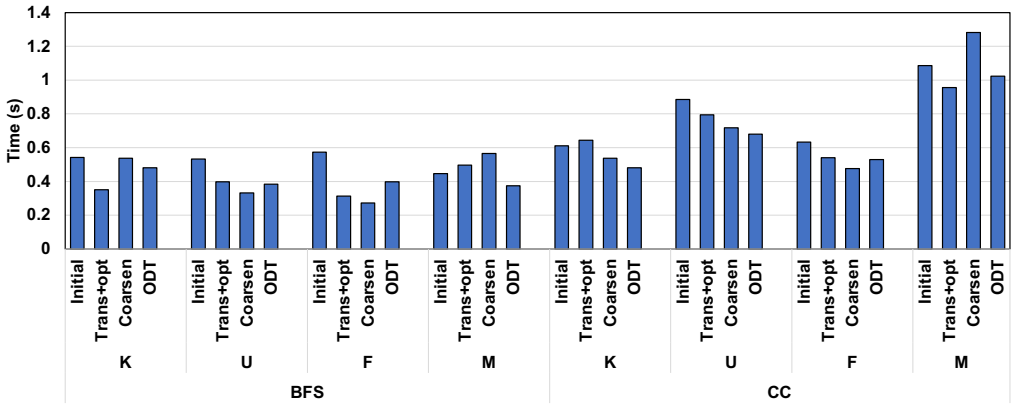
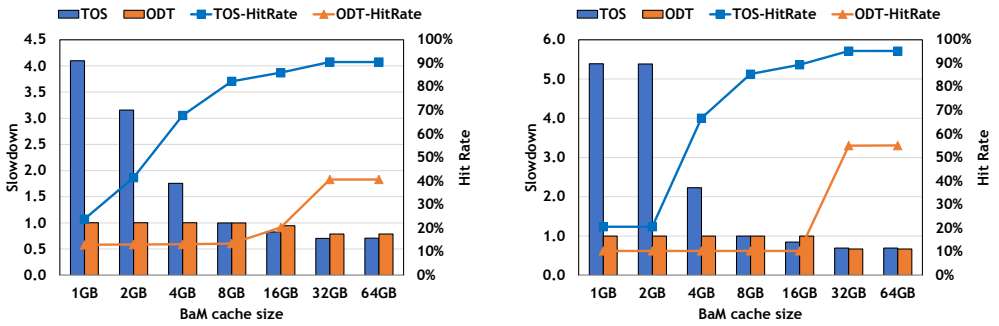


Figure 7.9: Overall execution time for BFS and CC algorithm with various optimization when the entire dataset fits in the GPU-HBM memory. The optimization maintains on-par or provides slightly better performance when the dataset is in the GPU-HBM memory. (Time lower is better.)

`Uk` graph has varying depth across random sample iterations, and many of the nodes have a tiny neighborlist. With tiny neighborlist and deep depths (>100+), the total number of overlapping I/O requests that are indeed submitted in each iteration of BFS to the BaM I/O stack is significantly less. With a low number of concurrent I/O requests, the BaM stack cannot amortize the cache and storage access latency and hence sees lower performance.

One way to address this issue is to use a frontier-based BFS algorithm. With a naive BFS frontier-based approach [83], BaM achieves 2× the speedup over the *ODT* technique for this specific dataset. This tells us that the work efficiency, essential merit of the frontier-based BFS algorithm, plays an important role when the graph is deep and has a tiny neighborlist. The *ODT* technique can be enabled with work efficiency in mind and is discussed in detail on how to enable it in the future work chapter.

Overall, *ODT* technique provides the best performance across most workloads and datasets. However, for an 8GB BaM cache, transposed work assignment and cache-line aware coarsening technique provides slightly lower performance (about 7-15% lower) and can be enabled with few lines of modifications in the GPU kernel code. Similar performance variations are also observed with Samsung SSDs and are not reported. Even more interesting is that the proposed optimization in most cases also helps improve the performance when the dataset fits in the GPU memory, as shown in Figure 7.9. This implies the optimizations done for BaM are generalizable across memory



(a) BFS (b) CC

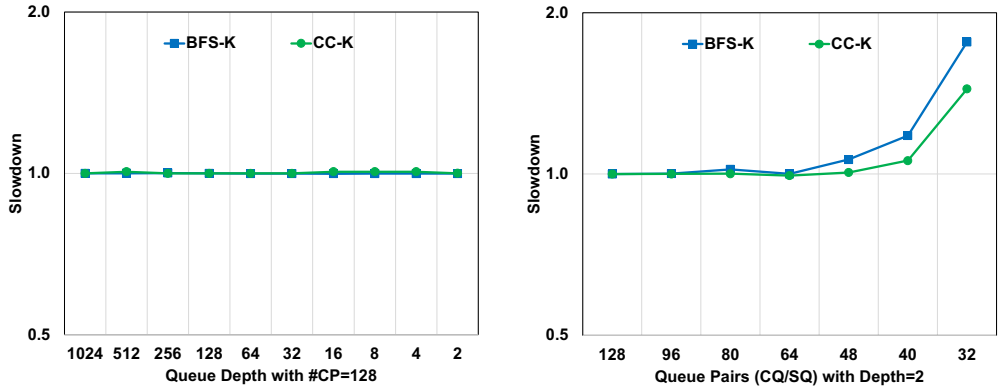
Figure 7.10: Impact of cache capacity on two optimization with four Intel Optane SSD and cache-line size of 4KB for K dataset on BFS and CC workload respectively. *ODT* technique requires a very little BaM cache capacity, infact only 16-32MB to provide same performance as the 8GB cache capacity required by transposed work assignment implementation (Slowdown lower is better, Hit rate higher is better)

hierarchies and do not degrade baseline performance. This meets one of the key goals when designing the BaM adaptation and optimization techniques.

7.1.7 Impact Of Cache Capacity

Next, we measure the sensitivity of cache capacity on the two best-performing optimizations: transposed with optimal stride size and *ODT* technique. For this evaluation, we use K dataset and run both BFS and CC workload with four Intel Optane SSDs at 4KB cache-line size. We vary the cache capacity from 1GB to 64GB and measure the slowdown when compared to 8GB BaM cache.

Increasing the cache as expected provides performance benefits for both optimizations and is nothing surprising. This is because large cache sizes can support loading the entire dataset to the GPU memory and can exploit higher cache bandwidth. However, for all practical purposes, requesting a cache capacity above 8GB is untenable as GPU memory is a very scarce resource, and most often, the application requires the local GPU memory for hosting workload-specific data structures. Decreasing the cache capacity reduces transposed work assignments with the optimal stride drastically. This is because the cache hit rate reduces significantly, as shown in Figure 7.10.



(a) Queue depth

(b) Number of queue pairs

Figure 7.11: Impact of NVMe CQ/SQ queue depth on overall execution of BFS and CC workload with `K` dataset. BaM requires a minimum of 48 queue pairs with two entries to provide unwavering performance for the applications. (Slowdown lower is better.)

On the other hand, *ODT* technique does not see any performance degradation with the reduction in cache capacity. For the *ODT* technique to work, it only requires 32MB of BaM cache-capacity at 4KB cache-line size and still provides the best performance. This is because *ODT* technique reads a cache-line from storage and computes all elements in the cache-line. This maximizes the reuse of the cache-line once brought in, and hence, hit rates remain constant. Thus, for a given GPU, with *ODT* technique, it is possible to determine the minimum cache capacity needed to get the best performance and can be computed as $cachesize = SM * W * CL_{size}$ where SM is the number of SMs in the GPU, W is the number of warps per SM, CL_{size} is cache-line size. For NVIDIA A100 GPU with BaM cache-line of 4KB, this comes to $108 * 64 * 4KB = 27MB$.

Thus, if the application developer is highly constrained by the available memory for executing graph workloads, *ODT* technique provides the best performance with the minor memory requirement for cache, enabling exposure to substantial GPU memory capacity using storage. If not, the transposed implementation provides near optimal performance with straightforward changes to the codebase.

7.1.8 Impact Of Number Of Queues and Queue Depth

In §4.5, we mentioned based on Little's law that as long as the application running on BaM can generate sufficient concurrent requests, GPUs can hide the storage access latency. For BaM with 4KB cache-line size, the Little's law provides $T \times L = Q_d$, where T is the throughput of the storage system, L is the average latency of accessing a 4KB cache-line using BaM stack and Q_d is the minimal queue depth required at any given point in time to sustain the throughput. With four Intel Optane SSDs at 4KB access size, T is $24GBps/4KB = 6MIOPs$. From §5.5.1, the average latency of accessing a 4KB cache-line using BaM is 31μs for Intel Optane SSDs through BaM stack. Thus, from Little's law, to sustain a desired 6M IOPs at 4KB, the BaM system needs to accommodate a minimum number of queue entries of $6MIOPs * 31\mu s = 186$ requests. However, for all practical purposes, this number should be more than 186 entries.

In this evaluation, we want to determine the minimum number of queue entries needed to provide the same performance as the nearly unlimited queue entries. To do this, we evaluate the impact of the number of queues and their depth using the BFS and CC graph traversal workload on the `K` dataset. We use the ODT technique optimization with 32MB cache-size and 4KB cache-line size with four Intel Optane SSDs. Each Intel Optane P5800X SSDs supports up to 135 SQ/CQ pairs, and each queue can be 1024 deep. With four Intel Optane SSDs, the BaM system has $135*1024*4 = 540K$ available entries and significantly exceeds the required queue entries for efficient execution in BaM. We use this configuration as a baseline and compute the slowdown when we reduce the queue depth and the number of queues, as shown in Figure 7.11.

Figure 7.11a shows the impact of queue depth, keeping the SQ/CQ pairs (CP) equal to 128. As the queue depth is reduced from 1024 entries to two entries (minimum is two) for each SSDs, the application observes no difference in the overall execution time. Keeping the queue depth constant at two and varying the queue pairs from 128 to 32, we see a drastic slowdown in the application as shown in Figure 7.11b. The slowdown starts when the number of queue pair is equal to 64, with each with two entries. This results in effective entries of $64 * 4 * 2 = 512$ entries. With only 512 entries, BaM provides similar performance as the nearly unlimited queue entries. Similar

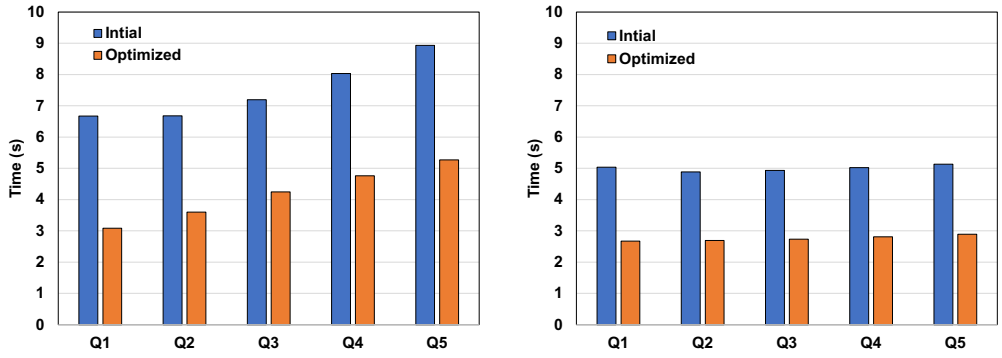
results are also measured in the case of other datasets but not reported.

7.2 I/O Amplification Benefit Of BaM For Data Analytics

We assess the performance advantage of the BaM prototype for enterprise data analytics workloads in addition to graph analytics. These new data analytics workloads are frequently employed to decipher, identify, or suggest significant patterns in data that has been gathered over time or from unstructured data lakes. The workload specifics can be found in §3.3. The purpose of this case study is to demonstrate the advantages of the BaM approach for lowering I/O amplification and software overhead when working with large structured datasets (see § 4.1).

7.2.1 General Setup

We discussed the I/O amplification issue in § 4.1 using the NYC taxi ride dataset [53] as an example. The dataset consists of 200GB of encoded data structured into 1.7B rows and 49 columns per trip using the Optimized Row Columnar (ORC) format [54]. We run five data-dependent queries within a user-specified bound described in § 3.3 to compare the performance of BaM against state-of-the-art GPU accelerated RAPIDS framework [6]. We use RAPIDS v21.12 version in the evaluation. All accessed metrics during the query execution, including `trip_distance`, are stored as 8-byte double-precision floating-point values. The user can change the bound to vary the sparsity of the queries that determine how many data-dependent accesses occur over the entire dataset. We evaluate the queries with two bounds, one where the `trip_distance` is at least 20 miles and the other where the `trip_distance` is at least 30 miles. Changing the bound from 20 miles to 30 miles changes the sparsity of data from 0.47% to 0.03%. Furthermore, we pin the entire ORC file in the Linux CPU page cache, enabling RAPIDS to read the file contents directly from the CPU DRAM without issuing an I/O request to the storage. When all data is stored in the host DRAM alone, one can consider this to be the maximum performance modern systems can achieve. Lastly, for evaluating BaM, we replicate the data across SSDs,



(a) 20 miles (Sparsity=0.47%)

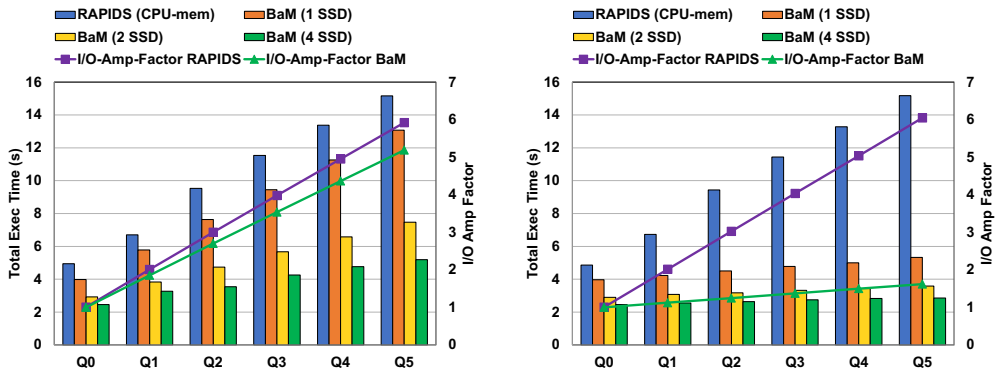
(b) 30 miles (Sparsity=0.03%)

Figure 7.12: Performance of BaM with and without cache-line aware optimization. The cache-line aware optimization provides at least 1.7× performance benefit over the initial implementation.

using up to four Intel Optane P5800X SSDs with 4KB cache-line size and 8GB cache capacity.

7.2.2 Applying Cache-line Aware Optimization

Naively implementing data analytics kernels of RAPIDS in BaM results in poor performance as it follows classical linear access pattern. Each thread in the GPU kernel is assigned to a row in the naive implementation, and it determines whether the value in the `trip_distance` column satisfies the requirement of being at least 20 or 30 miles depending on the input sparsity. If it is, the thread then accesses the other needed metrics, performs local computations and accumulates the results into a global shared variable using CUDA atomics. As each thread works on consecutive row elements, this access pattern closely resembles the linear access pattern discussed in §6.1. Because of this, only a limited number of overlapping concurrent I/O requests are issued to the storage resulting in exposure to the storage access latency to the application. To address this, we can apply the proposed cache-line aware optimization where each warp works on 512 consecutive rows in the data. In these queries, each value is of eight byte data type and mapping each warp to work with 512 rows of data enables generation of requests in cache-line granularity for the `trip_distance` column. Thus, the cache-line aware optimization loads a much larger working set and is able to issue a lot



(a) 20 miles (Sparsity=0.47%) (b) 30 miles (Sparsity=0.03%)

Figure 7.13: Performance of BaM and RAPIDS for data analytics queries on NYC Taxi dataset. BaM is up to 2.92× and 5.30× speed up over the CPU-centric RAPIDS framework for 0.47% and 0.03% sparsity.

more number of overlapping I/O requests to the storage.

Figure 7.12 shows the performance benefit provided by the cache-line aware optimization with two different sparsity with four Intel Optane SSD. The x-axis shows the different queries described in §3.3. The y-axis shows the end-to-end application execution time observed by the initial and optimized implementation. Irrespective of the sparsity in the input query, the cache-line aware optimization provides at least 1.7× speed up over the initial implementation. This improvement is mainly coming from better usage of the storage bandwidth during the query computation.

7.2.3 Performance Over RAPIDS

The performance benefit of BaM compared to RAPIDS with two different input query sparsities is shown in Figure 7.13. The x-axis shows the various queries described in §3.3. The left y-axis shows the total execution time for baseline RAPIDS and BaM systems in seconds. The right y-axis shows the I/O amplification factor observed by baseline RAPIDS and BaM systems. The I/O amplification factor is computed as the ratio of bytes the application reads into the GPU over the minimum bytes needed to execute each query.

Figure 7.13 includes a baseline query, "`Q0`: What is the average trip distance for trips with at least 20 (or 30) miles ?". This query only requires loading `trip_distance` column and then generating the result based on the given sparsity criteria. No additional data-dependent columns need to be fetched

for this query. Thus, this query helps in isolating the overhead of the data-dependent accesses. Since both systems must read the entire `trip_distance` column to the GPU memory before executing the query, neither RAPIDS nor BaM experience any I/O amplification overhead for `Q0`. Yet, BaM with the four Intel Optane SSD configuration sees a speed up by 2× over the state-of-the-art RAPIDS implementation. This is because, despite having the entire dataset preloaded into the CPU page cache, RAPIDS experiences software overheads on the CPU to find and move data and manage GPU memory.

Additionally, as shown in Figure 7.13, the baseline RAPIDS execution time linearly increases with the addition of a data-dependent metric over the initial query. Furthermore, the RAPIDS execution time is independent of the input query's sparsity. This is primarily because the query takes up less than 3.6 percent of the total execution time in RAPIDS, with the remaining time being used for row group initialization and clean-up. With additional data-dependent metrics, the baseline experiences even more I/O amplification and software and operating system overheads on the CPU.

In contrast, BaM outperforms RAPIDS in all configurations even with a single Intel Optane SSD, as shown in Figure 7.13. With the sparsity of 0.47%, BaM is mainly limited by the storage bandwidth with the single Intel Optane drive. BaM sees a speedup up to 2.52× when the number of SSDs is increased from one to four. The speedup is mainly from reduced memory management overhead enabled by fixed BaM cache size. Even then, BaM scaling gets limited by the set-up and clean-up overhead incurred by allocating and pinning storage queues, allocating cache memory, and cleaning up memory after query execution. With four Intel Optane drives, these overheads constitute up to 75% of the total execution time. Note that these overheads scale with the number of SSDs in the BaM system. This is because the I/O buffers and queues must be mapped for DMA with each SSD to allow any SSD to perform DMA to any I/O buffers in the GPU memory.

Increasing the sparsity to 0.03% and with four Intel Optane drives, BaM provides up to 5.30× speed up compared to the baseline RAPIDS implementation. The performance gain is attributed to BaM's reduced I/O amplification due to on-demand data fetching while the baseline must transfer entire columns to the GPU memory. BaM's ability to make on-demand access to data and overlap compute, cache management, and many I/O requests help it handle multiple data-dependent columns nearly as efficiently as the single

data-dependent column. This is clearly visible when the sparsity is varied from 0.47% to 0.03% in Figure 7.13. However, scaling the SSDs from one to four with the reduced sparsity queries, BaM's overall execution time, including the overheads, improves up to 1.76×. This is because the set-up and clean-up overhead cost increases and determines the achievable speedup when scaling the number of SSDs in the BaM system. Similar performance variations are also observed with Samsung SSDs and are not reported.

7.3 Programming Simplicity Of BaM For Regular Workloads

We evaluated the performance benefit of BaM and its optimization for workloads with self-referencing data structures and/or data-dependent access patterns. Now in this section, we evaluate two regular workloads: vectorAdd and segmented reduction kernels on large arrays and show the simplicity of programming these applications in BaM. We will also apply the cache-line aware optimization to these regular workloads and discuss the achieved performance benefits.

Baseline: VectorAdd application takes two input vectors, A and B, and generates an output vector C, where each element in the C vector is an element-wise sum from the vectors A and B. A simple GPU implementation of vectorAdd kernels maps each element computation to a GPU thread and assumes the input/output vector to be present in the GPU memory. However, if the length of the vectors is very large, the proactive tiling approach is used to perform vectorAdd computation. Here, a tile of the vector A and vector B elements are loaded from storage to the GPU memory, and a kernel is launched to partially compute the vectorAdd on the elements on input tiles to generate a partial result in the C vector in the GPU memory, and then transferred back to storage after computation. The process is then repeated with the next set of data tiles until all vector computation is completed. To hide the cost of access storage, overlapping reading of the next tile with writing results of previous tile computation can be performed.

A similar problem exists for the segmented reduction algorithm when the input array is large. Reduction is an algorithm that reduces the number of elements in a vector or array into one element. It is an important category of

collective operations widely used in parallel computing [83]. In a segmented reduction, an array of elements is split into multiple sub-arrays, and a reducing operator is performed on the inner arrays to get a vector of reduced elements [83]. The segment size usually governs the size of the segment. An excellent example of a segmented reduction kernel is computing an average of columns in a matrix to get a vector of average column values.

In a parallel implementation, each thread block works on a segment to generate a partial result. The segment size is equal to twice the thread block size, and each thread block loads two segments, the current working segment and a strided away segment of the elements, into shared memory before performing a parallel reduction to generate a partial result. The final result can be performed by launching another reduction kernel or in the CPU [83]. If the input array is massive and cannot fit in the GPU memory like vectorAdd, a proactive tiling approach is used to perform the segmented reduction. However, it is essential to note that when the input is enormous, even for a simple task like vectorAdd and reduction operation, it requires a ninja programmer to make it efficient by carefully enabling the CPU to orchestrate the operations hiding the storage access latency.

BaM Implementation: BaM's simplistic abstraction `bam::array<T>` eases the life of programmers when working with very large arrays. The `bam::array<T>` API eliminates the complex tiling operation required by the baseline implementation when accessing the extensive input data. As long as the input data is accessed by `bam::array<T>` API within the kernel, the threads can access the elements as if they were in the global memory of GPU without worrying about data transfer between the storage and GPU memory.

To port vectorAdd and reduction algorithm to BaM, one needs to make only minimal changes to the codebase as described in § 5.3.3. If required, we can apply cache-line aware optimization discussed in § 6.2 to gain additional performance out of BaM. We next evaluate the performance of BaM against the baseline system.

Setup: For this evaluation, we use two regular workloads, vectorAdd and reduction algorithms, described above, to work with four billion elements with each element of size eight bytes. For the baseline implementation, we will use a proactive tiling approach and split the four billion elements into four tiles, each consisting of one billion. For BaM implementation, the GPU kernel works on an entire dataset (no tiling) and uses cache-line aware im-

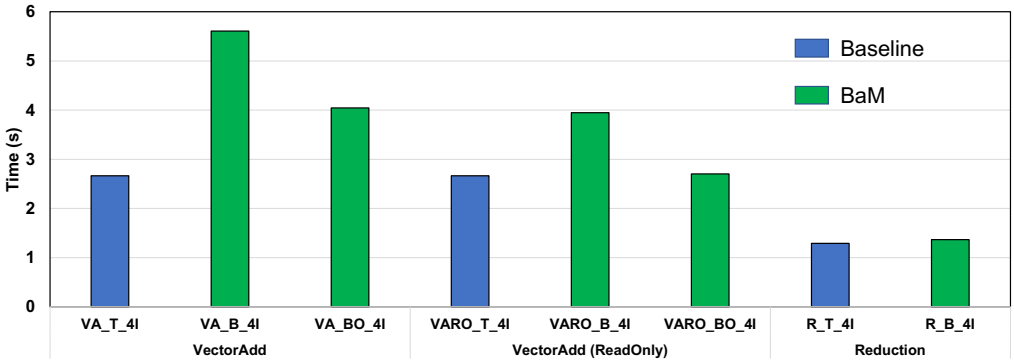


Figure 7.14: Performance of BaM on vectorAdd and reduction kernel. Two version of vectorAdd is shown: one with the output in the GPU memory (read-only) and other output written to the storage using BaM.

plementation for vectorAdd algorithm while segmented reduction algorithm from [83] without any optimizations. For the baseline, we will assume the data is stored in the SSD and loaded proactively to the GPU. For vectorAdd workload, we assume the baseline can perfectly overlap write operation with reading (although, in practice, this is very hard to achieve). We will also assume the number of SSDs in the system is set to four Intel Optane SSD. We will assume the cache size for BaM is set to 8GB, and the cache-line size is set to 4KB. The compute time for both implementations is negligible and is completely hidden.

Figure 7.14 shows the performance of BaM on vectorAdd algorithm when compared with the baseline implementation. `*_T_4I` represents the baseline with four Intel Optane SSD (4I), `*_B_4I` represents the BaM without cache-line aware optimization, `*_BO_4I` represents the BaM with the cache-line aware optimization. From Figure 7.14, we note that the naive vectorAdd implementation in BaM is upto 2× slower compared to the baseline implementation. Although the cache-line aware implementation improves the BaM performance by 1.39 ×, it is still slow by 1.5× when compared to the baseline.

To better understand what is causing the performance regression in the case of BaM, we implemented the second version of BaM where the output is written to the GPU memory (ReadOnly or RO). In the case of the baseline, the writes to the output in the storage can be completely hidden with the reads of the next tile operation; while the BaM despite the overlap between the reads and writes, the write latency is exposed to the application execution as shown in the Figure 7.14. One way to address this issue is to enable

asynchronous writes in the BaM system, which we leave as future work.

In the case of the reduction algorithm, BaM performance is 1.06× slower than the baseline even without any optimization. The segmented reduction kernel [83] is a well-optimized kernel to maximize the memory bandwidth utilization avoiding divergence, and hence continues to perform well even in the case of BaM. This tells us that not all workloads may require unique optimization, and it is entirely possible for regular workloads that have well-optimized memory access patterns may work just nicely out of the box with the BaM. This makes BaM an easy framework to enable programmers to work on massive arrays even for regular workloads.

7.4 Summary

To summarize, we showed BaM's performance is either on-par with or outperforms the state-of-the-art solutions for all types of studied workloads. We discussed in-depth the performance breakdown from each component in BaM system and how they contribute to overall execution. We then discussed the impact of various optimizations and how they greatly help with maximizing the performance of each application and achieving peak performance out of the BaM system. We also showed that the optimizations proposed for BaM improve or retains the same level of performance when the dataset fits within the GPU-HBM, making these optimizations universal. Lastly, we showed that using these optimizations, BaM performance for existing applications like graph analytics, which are notoriously hard to achieve good performance because of random access behavior, BaM is on-par or better than the current state-of-the-art host-DRAM only solution and is up to 4.27× slower than the GPU-HBM solution. And this performance benefit is extendable to other data-dependent workloads like data analytics, where we can achieve even better performance with four SSDs. Lastly and more importantly, BaM is the only system that can scale beyond 100TB capacity at a reasonable cost and yet provide a simplistic abstraction for the programmer to work with large datasets. Using end-to-end application evaluation, we concluded that overall BaM is the only system that can offer performance, cost, capacity scaling, and simplification benefits for emerging workloads.

Chapter 8

Future Work And Expected Impact

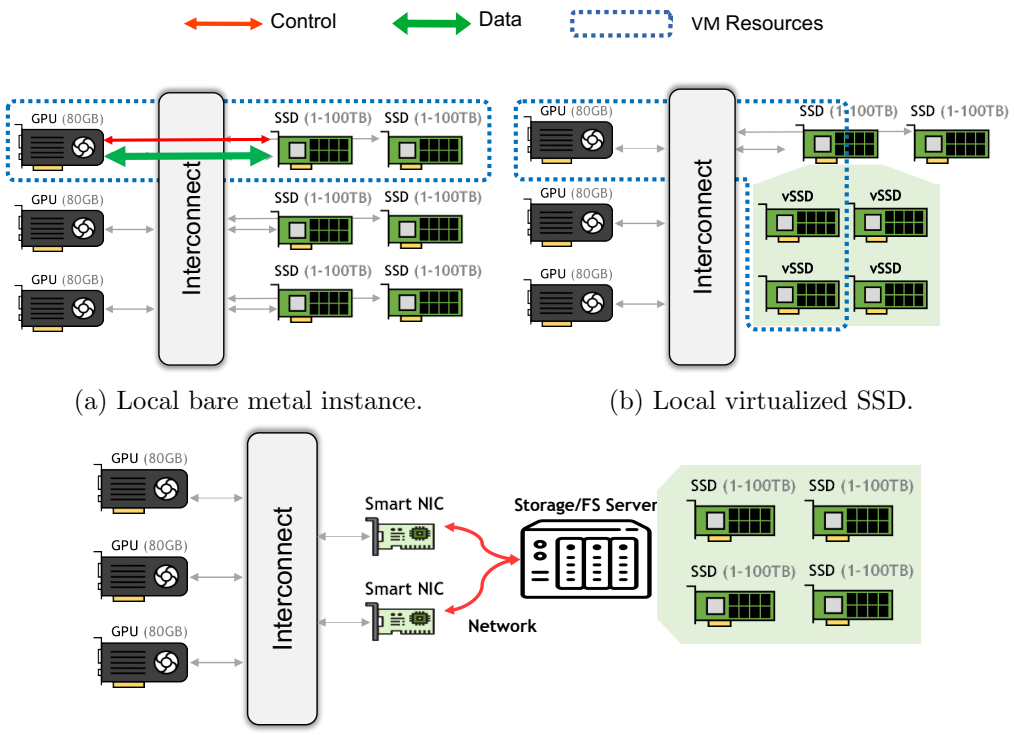
We discussed the design philosophy of BaM and showed that BaM could provide performance and cost benefits for various emerging applications. Next, we cover some of the design considerations to take when enabling BaM on a cloud environment and discuss immediate future work that can propel BaM in the right direction for long-lasting impact.

8.1 BaM Virtualization And Sharing For The Cloud

BaM addresses the limited memory capacity of accelerators by enabling accelerator-orchestrated access to the storage. However, until now, we assumed the storage devices are bare metal and available in entirety. This may not be true in the case of a cloud computing setup where the storage devices are typically virtualized. Storage virtualization enables sharing of the storage devices across multiple virtual machines (VM) and users. Thus, to this end, the next step in the BaM design is to enable isolation and virtualization capabilities to support hybrid-cloud infrastructures.

There are various models of storage virtualization in the cloud. These models differ in performance, bandwidth utilization, quality of service, security, resource flexibility, and many more aspects critical for enhancing efficiency. Here, we cover three widely used models to enable BaM in a hybrid cloud.

Local raw SSDs: In this model, the raw locally attached NVMe SSDs are exposed as bare-metal SSDs over PCIe. The entire SSD is exposed to the VM and assumed to be private to the VM. Figure 8.1a shows the bare-metal instance with raw NVMe SSDs connected to the GPU using BaM. The blue dotted line represent the resources (GPU and SSDs) dedicated to a specific virtual machine or an application as determined by the administration or hypervisor. Making BaM work on a bare-metal instance is straight-forward



(a) Local bare metal instance.

(b) Local virtualized SSD.

(c) Remote Storage Services.

Figure 8.1: Design of BaM in hybrid cloud setup. Depending on the provider, one can enable BaM with any or all of these combinations.

without current implementation. We only require to bring up BaM software stack on a VM connected with bare-metal SSDs. With this, the instance should achieve peak performance out of storage when the GPU threads access the data stored in the SSD.

Local virtualized SSDs: In this model, there are two ways to enable sharing for a single SSD: hardware-assisted virtualization or using namespaces. In the hardware-assisted virtualization, an SR-IOV (Single-Root Input/Output Virtualization) capable SSD is used. SR-IOV allows sharing devices across different guest VMs, as shown in Figure 8.1b. SR-IOV-enabled devices provide multiple virtual functions (VFs) that can be mapped to different VMs by the trusted service provider. Enabling BaM in this model follows a similar procedure as mapping the raw SSD, where instead of physical functions, virtual functions of each SR-IOV capable SSD get mapped to the applications, and the I/O queues and buffers are DMA-mapped to the virtual SSD (vSSD). This model also requires the enablement of IOMMU in the host to provide memory protection and to mitigate potential DMA

attacks. From the application point of view, there should be no change as only the lower-layer in the BaM stack requires a change with this setup. In terms of performance, BaM with hardware virtualization should provide similar performance as the bare-metal model as the hardware virtualization has proven to provide negligible performance overhead (assuming no loss in network bandwidth).

Alternatively, we can also enable locally shared SSDs with the help of `namespaces`, a technique provided by the NVMe specification [33]. An NVMe `namespace` is part of an NVMe device that consists of a group of logical blocks. The controller in the NVMe devices manages these `namespaces` and the NVMe protocol provides the required APIs to create, delete, and provide access controls. A trusted service provider uses these APIs to map each `namespaces` to each VMs or application allowing sharing of storage devices. BaM can be enabled even in this setup by DMA-mapping I/O queues and buffers to each `namespace` to specific virtual functions. Like SR-IOV, IOMMU must be enabled to avoid potential DMA attacks and can use the same GPU stack to issue I/O requests to the virtualized SSDs. However, in terms of performance, BaM with `namespaces` based sharing should see performance degradation as management of `namespaces` can come in the critical path (assuming no loss in network bandwidth).

Remote private/shared SSDs: In this model, the storage devices are accessed via a network fabric and are not locally present. A remote storage/file system server like Lustre or S3 manages the low-level storage devices and can provide file system services such as journaling, logging, and metadata tracking. Each compute node has an accelerator device like GPU has to interact with a SmartNIC like Bluefield DPU or FPGA that runs a trusted service exposing a communication channel between the node and the storage server as shown in the Figure 8.1c. The channel is a network fabric that can be either Infiniband, ethernet, or similar protocols based on the provider.

To enable BaM in such a system, the SmartNIC should reserve some of the BAR space for the application to ring doorbells for the storage I/O queues. This BAR space is then mapped to the application. Then, the I/O queues and buffers in the GPU memory can be DMA-mapped to the SmartNIC service, allowing it to perform Remote Direct Access Memory (RDMA) operations. This enables each application using BaM to communicate with the trusted service in the SmartNIC using the storage I/O queues to sub-

mit I/O commands. As the BaM applications are written with the help of `bam::array<T>` abstractions, the application does not require any changes, and they can use the same GPU software stack and allow each GPU thread to make I/O requests to the SmartNIC trusted service. These requests submitted to the SmartNIC are then processed to check authentication and other required operations before communicating with the remote storage server to grab the data and write it to the accelerator memory. The performance of an application using BaM with a remote storage server depends entirely on the implementation choices and the latency and throughput of the SmartNIC and remote storage server and cannot be speculated without an actual implementation which we leave as future work.

Other considerations: BaM is designed to enable access to storage services over a communication interface. However, in a cloud environment, this network interface is typically shared; hence, the available bandwidth per VM can directly impact the overall performance of BaM. For example, we have two graph analytics VMs working on two separate datasets but mapped to the same storage device via remote SSDs. If the workloads are time-shared, both workloads get full network bandwidths out of the system, but their execution latency still can increase. If the workloads are spatially shared, the available bandwidth per VM depends on the isolation guarantees governed by the SmartNIC and hypervisor. Assuming both workloads get equal distribution of the bandwidth, then, if the workload is bandwidth bound, then likely the performance of BaM can be degraded by at least 2×. One way to address this issue is to increase the number of SmartNICs per VM and effectively increase the available bandwidth.

8.2 Work Efficiency Optimization for ODT technique

The current implementation of *ODT* technique optimization does not address work efficiency, which can cause performance regression, especially with a dataset with deep depths. As the ODT technique works on edge-list directly, it is unaware of whether the cache-line requires to be processed or not. Because of this, a work-efficient kernel like the frontier-based designs can provide additional performance when the dataset fits in the GPU memory. Fundamentally, *ODT* technique lacks additional information that

a work-efficient kernel knows. For example, in the case of BFS workload with *ODT* technique, each iteration launches a kernel with the same number of warps, i.e., the number of cache-lines in the dataset. Inside the kernel, the current *ODT* technique implementation determines if the launched warp should perform any compute or terminate. The assumption here is that the cost of checking if the warp has work to do will be negligible compared to storage access. However, this creates an implementation that is not work efficient and can become a performance inhibitor if many small kernels are working on tiny neighbor lists like in the case of BFS on `Uk` dataset.

To address this problem, we need to add a temporary list called "ActiveCachelineID" that captures only the unique cache-line ids for the next iteration that requires to be active in an iterative algorithm like BFS. Whenever a node is added to the frontier list, unique cache-lines corresponding to the node's neighborlist must be added to the "ActiveCachelineID", and an atomic unique cache-line counter needs to be incremented. It is imperative to ensure the "ActiveCachelineID" list contains only the unique cache-line ids. This can be ensured in multiple ways: a) before inserting a cache-line ID to the list, check if its already present, b) use sorting and remove duplicates, or c) have a bit array to determine if the given cache-line ID is already part of the "ActiveCachelineID" list or not.

Using the "ActiveCachelineID", the workload like BFS can become work efficient. For example, in the next iteration, one needs to launch the number of warps equal to the total number of active unique cache-lines in the "ActiveCachelineID"" list, and each warp should read only the data using this "ActiveCachelineID"" and perform the required traversal computation as it would have done with regular *ODT* technique. This would address the work efficiency problem and provide similar performance as the frontier-based approach. Theoretically, this technique should provide the same work efficiency asymptotically as the frontier-based approach but with the capability of a piece-wise cache-line aware linear access pattern that provides the best performance for the system like BaM. Although, it is unclear at the moment to the degree to which this will help when all data fits in the GPU-HBM. When compared to the frontier design, the "ActiveCacheLineID" approach may achieve lower performance because of additional comparison inside the tile. However, for the BaM, it should also help reduce the launch overhead and provide performance gains for a dataset with a tiny neighborlist. The

implementation of work-efficient *ODT* technique will be added in the next version of BaM.

8.3 Future Of BaM System Software Stack

GPU-Accelerated Storage Server: We focused on the design of BaM as a consumer of data from the storage. The GPU threads pull the data from the storage and perform computation here. For this use case, we showed BaM software stack can quickly scale to achieve peak 45M random read IOPs with nine Intel Optane P5800X SSDs at 512B access granularity without contending on precious CPU-only resources. Alternatively, it is possible to make BaM a producer of the data from the storage. In this model, when a request arrives, the GPU threads pull the data from the storage with very high random read/write throughput from storage and then aggregate the results and push the data chunk back to the requesting process or node. This would enable BaM to become a GPU-accelerated storage server providing extremely high random IOPs even at very small access granularities. This would address many performance limitations present in current storage server designs in modern data centers where they cannot keep up with the request rates from compute nodes and become a critical performance limiter. This opens up a new system design methodology where accelerators are used to accelerate and optimize system software and can change how future heterogeneous systems are designed.

Byte-addressable SSD and Persistent Memory Support: Although the design of BaM discussed with the assumption of using SSD as memory, the BaM can be extended to support emerging byte-addressable SSDs and persistent memory technology. To enable persistent memory, the bottom layer of the BaM stack that interfaces with the storage must be updated to support the persistent memory interface, and the rest of the component can remain the same. Supporting these emerging memory technologies are left as future work.

Caches and abstractions: Compared to the traditional cache design with a centralized cache controller, BaM implements a more robust high throughput cache design by enabling each GPU thread to act as a software cache controller. This creates a more scalable solution, especially with the

emerging memory technologies like high-bandwidth memories (HBM), whose bandwidth is approaching 3-5TBps [86]. Furthermore, the current cache design can be exported as a standalone library and generalized as a go-to library for accessing data from the remote memory or storage over RDMA. However, the design of BaM cache is an unresolved research problem. For example, the current BaM cache design trades-off metadata size for fast lookup operation. However, with optimization techniques like cache-line aware optimization, it is not clear which cache design and algorithms would provide optimal performance for GPUs. This is because a prior study has not evaluated the various trade-offs associated with the design of software-only cache in the GPU memory. We envision future works would consider exploring these aspects and discuss the impact of different associativity, search algorithms, and replacement policies for software-only cache on GPUs.

The goal of the BaM abstractions was to minimize the programmer and application developer effort when integrating BaM into their applications. The current abstraction `bam::array<T>` supports a general array-like abstraction. However, one can add more applications specific, even higher-level abstractions like data columns in tables for data analytics applications to gain additional performance and flexibility. This would enable leveraging application-specific information to lessen the cost of accessing the data from the cache and storage.

Multi-GPU: The current BaM implementation and evaluation only consider using SSDs to expand the effective memory capacity of a single GPU device. However, modern data center systems have multi-GPUs, and many applications use them. Although technically, it is possible to enable BaM and share the same set of SSDs across multiple GPUs, it opens up a new set of challenges that are not addressed in this thesis. First, designing a performant high-throughput software-only cache across multi-GPU to avoid going to storage is a problem unstudied. Fundamentally, it is required to determine how to efficiently share cache states between GPUs using system-wide atomics to maintain coherence. However, this can also result in excessive coherence traffic between GPUs and reduce the overall performance. Thus, how we enable coherence across GPU fast so that the cache access cost does not become a bottleneck when accessing the data from the storage needs to be studied. We believe ideas such as directory-based cache-coherence similar to the IEEE scalable cache coherence protocol [87–89] can be used as a baseline

in this context, but designing directory-based cache-coherence with massive parallelism of GPU is uncharted territory. Second, with sharing SSDs across multiple GPUs, we could contend on the same SSD and PCIe interconnect, which can result in performance regressions. Thus, it is required to study cache-line placement and data layout and provide application-level guidelines for optimal performance in this multi-GPU system design with shared cache and SSDs. This requires further in-depth study before it can be incorporated in a future version of BaM.

8.4 Hardware Changes For BaM

Specifying and defining an interface between the hardware and software is critical to system and computer architecture. As much as this interface depends on the underlying hardware, it also depends on how the software uses it. In this work, we showed a novel usage model of an architectural storage interface, i.e., NVMe, to make such devices accessible to compute accelerators GPU using BaM. We discussed how to enable a single GPU to execute complex applications over massive datasets efficiently. However, we can further enhance the performance of BaM with several hardware modifications.

Before BaM, hardware architects only considered the design and implementation of NVMe and storage protocols for CPUs, where software stack overhead and limited application parallelism reduce the pressure on the storage protocols. However, with BaM, the speed and efficiency of the storage protocol are critical for the application performance. For instance, the widely used storage interface NVMe queues are inherently sequential and can hinder BaM's performance, especially for long latency devices. After requests are enqueued into an NVMe SQ, the queue's doorbell must be rung with the updated queue tail to notify the storage controller of the new request(s). As these doorbell registers are write-only in the current NVMe specification [33], when a thread rings a doorbell, it must make sure that no other thread is writing to the same register and that the value it is writing is valid and is newer than any value written to that register before. Implementing queue insertion as a critical section, while simple, imposes significant serialization latency, which might be fine for the CPU's limited parallelism but can cause substantial bottleneck for thousands of GPU threads. To address this, the

hardware architects can consider implementing efficient protocols exclusively to exploit massive GPU parallelism and avoid this serialization cost.

We can further improve the performance of BaM by adding hardware support to the BaM cache. This would reduce the cache API overhead significantly, helping improve the overall performance. With the hardware-enhanced cache, register usage within a thread for managing cache state can be avoided, enabling the GPU threads save more of the resources for application's work. Moreover, a hardware-enhanced cache can avoid the overhead associated with additional memory lookups for managing cache metadata as they can occur implicitly without software intervention. Coupling with the cache-line aware optimization, a hardware-enabled cache would provide performance benefits even for fine-grain access granularity like 512B or lower.

Although the hardware-enhanced BaM cache is complex to implement and might require a lot of hardware resources and changes, we can enable a near-term benefit by increasing the size and banking of L1 and LLC caches in modern GPUs. This is because the modern GPUs like NVIDIA A100 have only 40MB of LLC cache, which is significantly tiny to hold the metadata for a software cache and often exhibits port contention and thrashing on the cache metadata. Run-time modification using a learned technique to determine which memory regions should be pinned to avoid LLC cache thrashing can alleviate this problem to a certain extent. Improvement in IPC, increasing the pool of schedulable threads per SM, polling aware warp scheduling, increasing the size of register file per SM, and new ways to perform fast warp-level or thread block-level communication can significantly help in improving the overall performance of BaM system.

Performance of BaM has a more direct impact on the available interconnect bandwidth. This is understandable given the limited interface bandwidth currently BaM lives in. Exploring the benefit of newer interconnects such as CXL, ethernet, and NVLink can help address the limited interface bandwidth problem. Even increasing the PCIe interconnect bandwidth by 2× can significantly improve the overall BaM performance and provide even more cost-benefit for the systems.

Another lesser studied topic is the data placement and topology design between the GPU and SSDs and their impact on the overall application performance. We already saw significant performance implications based on

where the GPU is placed in the system topology when building the BaM prototype using the off-the-shelf components. However, as the system becomes more disaggregated and heterogeneous, mapping data computation and placement can become crucial for overall end-to-end performance efficiency. For instance, to reduce the I/O amplification when reading the data from storage, we could perform part of scatter-gather computation using the in-storage computing [90] and reduce the overall data transfer to the GPU. This would significantly benefit workloads like data analytics and recommendation system and is currently left as future work to study.

We can provide an even more immediate impact by increasing the throughput of each SSDs in the system. Currently, each SSD *only* provides 1.5M IOPs at 4KB read/write access which is very low compared to the available PCIe Gen4 bandwidth. This is meager compared to what systems enabled with BaM require. Doubling or quadrupling this would greatly help with the performance. Moreover, because of low IOPs per SSD, each GPU requires a minimum of four SSDs to saturate the ingress bandwidth. As each of these SSDs has a storage capacity-oriented form factor, four SSDs consume four PCIe $\times$16 slots in systems that are designed for performance. Thus changing the form factor or enabling $\times$16 SSDs capable of providing 4$\times$ the IOPs per SSD can greatly help BaM design.

Moreover, while prototyping with the actual hardware, we discovered that the SSDs offer different throughput at various I/O granularities. For instance, Samsung DC1735 SSD provides 1.6M IOPs for 4KB while only 1.1M IOPs for 512B random read I/O access sizes. This results in 6.4GBps and 0.55GBps of bandwidth from one SSD for 4KB and 512B granularities, respectively. An ideal storage device should provide the same throughput irrespective of I/O access granularities, which will significantly help simplify the software stack design.

Apart from these above-listed changes, system level changes and enhancements to virtualization capabilities both in SmartNIC, CPU, and accelerators can greatly benefit BaM. Furthermore, BaM also opens up exciting changes to the software ecosystems like moving storage services to trusted storage devices or enhancing the throughput of storage services in the era of accelerator-initiated access to storage. It is fair to say that the BaM design philosophy opens up a new research frontier for designing hardware accelerators, SSD, and system design and software altogether.

8.5 Additional Applications

BaM provides exceptional performance for applications with data-dependent accesses and/or self-referencing data structures. Self-referencing data structures are data structures where one needs to be able to access the entire data structure for computation. Data dependent access patterns and self-referencing data structures are present in many emerging applications. In this §7 section, we discussed the performance benefit of BaM for graph and data analytics workloads in-depth. However, while building BaM, two additional workloads: graph neural networks [10, 50] and recommender systems [14, 16] were extensively studied but not reported as part of the evaluation. This is because the publicly available datasets create tiny models, and mapping these small models to the storage stack does not make much sense as these workloads exhibit a Zipfian access pattern. This is in sharp contrast to the industry where the embedding tables for these workloads are gigantic [10, 14, 16, 50]. To address this gap, we are actively working on creating new large-scale datasets and models for these workloads and will be made public in the future.

8.6 Expected Impact

There has been a phenomenal growth in the accelerator compute throughput in the recent decade, and their easy accessibility has drastically impacted the progress of fundamental scientific discoveries and applications, like deep neural network training and inference. In fact, in many emerging applications demanding high compute throughput, the computation has shifted from CPU to accelerators like GPUs. Yet, the accelerators are treated as second-class citizens where the CPU off-loads operations to them to perform computation and manages the data movement. This CPU-centric model where the CPU manages the data orchestration works well for classical GPU applications like training dense deep neural network models. However, with the development of complex applications requiring data-dependent accesses, rapid increase in the size of the datasets, and the slow scaling in GPU memory capacity and GPU external bandwidth, the traditional CPU-centric model has failed to live up to the application expectations.

BaM tackles the problems with the CPU-centric model head-on by enabling an accelerator-centric computing model where the accelerator-like GPUs can orchestrate high-throughput, fine-grain accesses into the NVMe SSDs. With high-level abstractions like `bam::array<T>`, BaM brings in a new layer to the accelerator's memory hierarchy and directly addresses the memory capacity problem. By building a hardware prototype using off-the-shelf hardware components and then optimizing the access pattern to access `bam::array<T>`, we showed it is possible to achieve peak performance out of the system even on graph analytics workloads that are deemed to be notoriously hard to achieve good performance because of random access behavior. We showed BaM 's performance is on-par or better than the current state-of-the-art host-DRAM only solution and is only 4.27× slower than the GPU-HBM solution. Among the three, only BaM can scale beyond 100TB capacity at a reasonable cost, opening up new frontiers in building cost-effective large memory systems for accelerators.

Although BaM uses GPUs as prototype hardware, any accelerator manufacturer with capabilities like GPUDirect can build BaM and integrate it into their ecosystems. We believe BaM opens up a Pandora's box of research questions to answer, design, and develop in the face of building large-scale memory systems to address the memory wall problem. Despite being in an early stage of design, it is already showing promising results in addressing the memory wall problem. With BaM, the user gets the teraflops of GPU to compute capability and terabytes of GPU accessible storage in a box. As the number of cores and threads in CPUs increases, we expect that the CPU storage accesses will also benefit from adopting the BaM software stack. We believe that such capabilities will foster discussion on how to best implement, use, and optimize BaM and enable new ground-breaking research, ideas, and applications that are infeasible or impractical today.

Chapter 9

Related Work

9.1 Optimized CPU-centric Model

Most GPU programming models and applications were designed with the assumption that the working dataset is smaller than the GPU memory. If the dataset does not fit within the GPU memory, application-specific techniques like tiling or data partitioning is employed to enable the processing of large data on GPUs [17, 18, 20, 22, 23, 29, 31, 56–59].

SPIN [18] and NVMMU [19] propose to enable peer-to-peer (P2P) direct memory access using GPUDirect RDMA from SSD to GPU and exclude the CPU from the data path. SPIN integrates the P2P into the standard OS file stack and enables page cache and read-ahead schemes for sequential reads. GAIA [20] further extends SPIN's page cache from CPU to GPU memory. Gullfoss [31] provides a high-level interface that helps in setting up and using GPUDirect APIs efficiently. Hippogriffdb [22] provides P2P data transfer capabilities to the OLAP database system. GPUDirect Storage [17] is the most recent product that migrates the data paths from CPU to GPU in the NVIDIA CUDA software stack using GPUDirect RDMA technology. Similar efforts from AMD are seen in RADEON-SSG product lines [32]. As we discussed in detail in §4.4, all of these works still employ a CPU-centric design where the CPU is responsible for data orchestration. In the proposed work, we migrate both control and data paths to the GPU, allowing any threads in GPUs to initiate, read and write data in a high-throughput fine-grain manner directly to SSD. This fundamentally allows application developers to rationalize SSD as an extended memory hierarchy and enables GPU-centric design on modern systems.

9.2 Prior Attempts Of The Accelerator-centric Model

We acknowledge that ActivePointers [27], GPUfs [26], GPUNet [29] and Syscalls for GPU [21] have pioneered the efforts to enable accelerator-centric model for data orchestration. GPUfs [26] and Syscalls for GPU [21] first allowed GPUs to request file data from the host CPU. GPUfs provides a file abstraction for accessing the data from the storage from the GPU threads. ActivePointers [27] adds a memory-map-like abstraction on top of GPUfs to allow GPU threads to access file data like an array. Both these prior work uses the CPU as a proxy for executing storage I/O requests on behalf of the GPU threads, often translating the file access to POSIX system calls. As files can handle inter-process communication in UNIX, this enables GPU to access contents from the storage and other system services. However, in doing so, this method incurs significant overheads when accessing data from storage. For example, it must keep the data and OS page cache consistent for the abstraction to work correctly. Moreover, GPUfs and other prior work on this segment fall short on several other aspects and cannot provide the required performance for the emerging workloads.

First, the publicly available implementation of GPUfs [26] (and ActivePointers [27]) is prone to dead-locking, specifically the buffer cache update. The copy operation design implemented is not guaranteed to make forward progress resulting in dead-locks. Second, the performance offered by these prior works [26, 27] is too low to saturate PCIe bandwidth despite using large pages, prefetching, and a pre-filling Linux page cache to keep the data and practically eliminate any real I/O access to storage. This is because the abstraction proposed in these prior work imposes strict restrictions on synchronization barriers limiting their applicability to particular application access patterns.

Third, despite using host memory to cache all data, GPUfs [26,27] requires much larger I/O sizes to provide meager bandwidth out of the system. For sequential access benchmark, GPUfs [26] provides the best performance at 64KB I/O sizes. However, for the emerging real-world applications discussed in §3, 64KB access granularity is too big, and at such large granularities, one may as well use the NVIDIA GDS [17] in a CPU-orchestrated solution rather than enabling accelerator-initiated storage access.

Dragon [25] is the most recent work that enables accelerator-initiated

storage access. Dragon [25] proposed to incorporate storage access to the UVM [28] page faulting mechanism, and when a requested data is missed in the CPU host page cache, a read operation is performed to bring the data from storage to the CPU and then is copied to the GPU address space. However, as discussed in §4.2, this reactive page fault-based mechanism relies significantly on less-parallel GPU to handle the data demands of massively parallel GPUs and cannot provide a performance benefit. Moreover, all these prior works focused only on functional enablement and thus ended up with gross under-utilization of hardware resources and poor overall performance.

In contrast, BaM redesigns accelerator-initiated storage access from scratch with the ultimate goal of achieving peak performance of the system even at small granularities by innovating the cache and queue designs and optimizing the application access patterns. BaM provides a memory-like abstraction to enable any GPU threads to read or write any part of the data from BaM. BaM memory abstraction is consistent with the current memory model to access data from the GPU memory. Because of this, integrating BaM to GPU kernels becomes trivial. BaM implements a highly scalable cache design where all accessing threads also contribute to the lookup and miss handling efforts, and all of BaM performance numbers and benefits are measured with a cold BaM cache. Using microbenchmarks and application case studies, we show BaM achieves peak storage bandwidth, on-par with a much more expensive state-of-the-art DRAM-only solution widely used in industry and academia. The design of BaM is open sourced [91], and an early version of the BaM is released to public [92] along with BaM vision [93].

9.3 Hardware Extensions

Extending the support of non-volatile memories for GPUs has been proposed by directly replacing the global memory with flash memory or closely integrating it with the GPU memory system [94–98]. DCS [99] proposed enabling direct access between storage, network, and accelerators with the help of a dedicated hardware unit like an FPGA providing the required translation for coarse-grain data transfers. Enabling persistence within the GPU has recently been proposed [100]. We acknowledge these efforts and further validate the need to enable large memory capacity for emerging workloads. More

importantly, some of these ideas can be potentially used further to enhance future versions of BaM.

9.4 Using SSDs As CPU Memory

Several prior works have proposed to leverage memory-mapped interface and swapping mechanism of operating systems to extend the CPU memory with SSDs [101–107]. Most of these work, treat the SSD as a block device and rely on a paging mechanism to migrate data from the SSD to the CPU memory. The paging mechanism is in-efficient if the application exhibit data-dependent fine-grain data access to the dataset. To address this limitation, FlatFlash [108] proposed to exploit the byte-accessibility of SSD and bypass the storage software stack, focusing on fine-grain I/O operation to reduce I/O amplification. Our proposal is similar in spirit to of FlatFlash approach, and we extend the concepts of fine-grain accesses to GPUs. Compared to the CPUs, GPUs offer massive parallelism that can be leveraged to hide the long latency SSD accesses.

Chapter 10

Conclusion

Emerging workloads like graph and data analytics demand large memory capacity. However, accelerators like GPUs have limited memory far lower than the required memory capacity to accommodate emerging workloads. Thus, the state-of-the-art systems use CPU-orchestrated data movement strategies between the accelerator and storage to execute emerging workloads and struggle to perform well.

To this end, we examined the requirement of a system capable of supporting modern high-value emerging workloads and discussed the issues associated with the current CPU-centric state-of-the-art implementations. Based on these insights, we made a case for enabling accelerators like GPUs to orchestrate high-throughput, fine-grain accesses into storage devices like NVMe solid state drives. We proposed, implemented, and evaluated a design of a cost-effective system architecture called BaM, capable of supporting the ever-growing need for large memory capacity in accelerators for these workloads. In BaM, GPU threads can read/write data on-demand without the need to synchronize with the CPU, significantly reducing software stack overhead, minimizing I/O amplification, and enabling GPU programmers to think of storage as memory. BaM's array abstraction is extremely powerful, and enables easy application porting to support BaM with a few lines of changes.

However, naively running applications on BaM does not provide performance and efficiency benefits. As BaM essentially extends the GPU memory hierarchy to the storage, favorable access patterns are needed for BaM to reach its full potential. Nevertheless, this requires embracing two fundamental yet conflicting requirements. BaM requires coalesced accesses for extracting high-throughput out of its cache, while the BaM I/O stack and storage requires large overlapping I/O requests to hide the long latency of storage access. These conflicting requirements create a design dilemma, motivating the set of sophisticated optimization techniques and application adaptations

to achieve peak performance on BaM.

The proposed techniques in the thesis are: cache-line aware optimization and on-demand implicit tiling. The cache-line aware optimization is a simple generalizable technique that addresses the above problem for all studied workloads. The key idea of this optimization is to map each warp to work on cache-line data that minimizes the contention when accessing BaM metadata and generates a sufficiently large number of concurrent I/O requests to saturate the BaM storage stack. And the second optimization is a CSR data-layout specific method called the on-demand implicit tiling technique (ODT). The ODT technique maps each warp to work on a tile of data instead of the traditional vertex-centric approach and maximizes reuse on the loaded tile while also exploiting GPU's massive parallelism to generate many overlapping I/O requests to hide the storage access latency.

We applied these optimization techniques to several emerging workloads and showed on multiple datasets, SSD types, and access granularities that BaM with optimizations is a viable alternative to the DRAM-only and other state-of-the-art CPU-centric solutions. For example, for graph analytics workload, notoriously known to be hard to achieve good performance due to random access behavior, BaM is on-par or better than the current state-of-the-art host-DRAM only solution and is only 4.27× slower than the GPU-HBM solution. We also show that the proposed optimizations are universal and improve or retain the same level of performance when the dataset fits in the GPU-HBM memory. And these performance benefits are extendable to other data-dependent workloads like the data-analytics, where BaM can achieve up to 5.3× speed up over a host-DRAM only solution. Lastly and more importantly, BaM is the only system that can scale beyond 100TB capacity at a reasonable cost and provide simplistic software abstractions for programmers to work with large datasets.

Overall, this dissertation proposes a design of a system capable of performing GPU orchestrated storage access to extend the GPU's effective memory capacity and provides a set of generalizable application adaptations that enables application developers to maximize the performance, cost, I/O efficiency, capacity scalability, and simplified software development for emerging workloads without additional hardware support. With BaM, the user gets the teraflops of GPU to compute capability and terabytes of GPU accessible memory at a low cost.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, p. 39–55, Mar. 2008.

[2] "NVIDIA Tesla A100 Tensor Core GPU Architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[3] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities based on Ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[4] J. Sybrandt, M. Shtutman, and I. Safro, "MOLIERE: Automatic Biomedical Hypothesis Generation System," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1633–1642.

[5] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. mei W. Hwu, "EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs," *Proceedings of VLDB Endowment*, vol. 14, pp. 114–127, 2020.

[6] "CUDA RAPIDS: GPU-Accelerated Data Analytics and Machine Learning," https://developer.nvidia.com/rapids.

[7] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (CPU+ GPU) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC'18)*, Boston, USA, 2018.

[8] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss Decomposition on GPU," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, Boston, USA, 2019.

[9] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on Triangle Counting on GPU," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, Boston, USA, 2019.

[10] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations (ICLR'17)*, 2017.

[11] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec, "OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs," *arXiv preprint arXiv:2103.09430*, 2021.

[12] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *CoRR*, vol. abs/1906.00091, 2019.

[13] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[14] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models," 2021.

[15] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems," in *Third Conference on Machine Learning and Systems*, 2020.

[16] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2021, pp. 802–814.

[17] "GPUDirect Storage: A Direct Path Between Storage and GPU Memory," https://developer.nvidia.com/blog/gpudirect-storage/.

[18] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless operating system integration of peer-to-peer DMA between ssds and gpus," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 167–179.

[19] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 13–24.

[20] T. Brokhman, P. Lifshits, and M. Silberstein, "GAIA: An OS page cache for heterogeneous systems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 661–674.

[21] J. Veselý, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic System Calls for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018, pp. 843–856.

[22] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 14, p. 1647–1658, Oct. 2016.

[23] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[24] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti, "Bandana: Using non-volatile memory for storing deep learning models," *CoRR*, 2018.

[25] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[26] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 485–498.

[27] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A Case for Software Address Translation on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 596–608.

[28] "Unified Memory for CUDA Beginners," https://developer.nvidia.com/blog/unified-memory-cuda-beginners.

[29] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 201–216.

[30] E. Agostini, D. Rossetti, and S. Potluri, "Offloading communication control logic in gpu accelerated applications," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. IEEE Press, 2017, p. 248–257.

[31] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Gullfoss : Accelerating and simplifying data movement among heterogeneous computing and storage resources," in *Technical Report, Department of Computer Science and Engineering, University of California, San Diego*, 2015.

[32] AMD, "RADEON-SSG API Manual," https://www.amd.com/system/files/documents/ssg-api-user-manual.pdf.

[33] "NVMe 1.3 Specification ," http://nvmexpress.org/resources/specifications/.

[34] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, p. 420–431, Dec. 2017.

[35] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[36] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: a scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[37] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[38] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[39] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[40] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[41] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[42] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[43] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 456–471.

[44] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016.

[45] "nvgraph," https://developer.nvidia.com/nvgraph.

[46] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 197–208.

[47] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with gpus and cuda," *Parallel Comput.*, vol. 36, no. 12, pp. 655–678, Dec. 2010.

[48] S. Che, "Gascl: A vertex-centric graph model for gpus," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[49] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 267–276.

[50] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph Convolutional Neural Networks for Web-Scale Recommender Systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 974–983.

[51] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. M. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018.

[52] Criteo 1TB Click Logs dataset, https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[53] The City of New York, "TLC Trip Record Data," https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[54] The Apache Software Foundation, "Apache ORC: The smallest, fastest columnar storage for Hadoop workloads," https://orc.apache.org/.

[55] Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance, https://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.

[56] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically Managed Data for CPU-GPU Architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 165–174.

[57] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010, p. 347–358.

[58] H. Seo, J. Kim, and M.-S. Kim, "GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 253–254.

[59] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 53–65.

[60] "Intel® Optane™ Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[61] Weka.io, "WekaFS Architecture Whitepaper," https://www.weka.io/wp-content/uploads/files/2017/12/Architectural_WhitePaper-W02R6WP201812-1.pdf, 2021.

[62] Y. Ren, C. Min, and S. Kannan, "CrossFS: A cross-layered Direct-Access file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ren pp. 137–154.

[63] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Scale and performance in a filesystem semi-microkernel," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 819–835.

[64] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. Association for Computing Machinery, 2019, p. 478–493.

[65] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.

[66] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a True Direct-Access File System with DevFS," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA, 2018.

[67] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: A proposal for an exascale storage system," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 585–596.

[68] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China, 2017.

[69] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[70] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz, "Smartio: Zero-overhead device sharing through pcie networking," *ACM Transactions on Computing System*, vol. 38, no. 1–2, jul 2021.

[71] F. J. Corbato, "A Paging Experiment With The Multics System," *Technical Report, Massachusetts Institute of Technology, Cambridge, Project MAC*, 1968.

[72] Z. Qureshi, "Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2022.

[73] "Samsung Z-NAND Technology Brief," https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[74] "Samsung 980 PRO SSD," https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.

[75] "Techinsights publishes preliminary analysis of 3d-xpoint memory," https://www.anandtech.com/show/11454/techinsights-publishes-preliminary-analysis-of-3d-xpoint-memory.

[76] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, apr 2009.

[77] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 31–46.

[78] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, p. 1119–1133, Mar. 2020.

[79] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[80] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[81] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 300–314.

[82] M. Hidayetoglu, T. Bicer, S. G. de Gonzalo, B. Ren, D. Gursoy, R. Kettimuthu, I. T. Foster, and W.-M. W. Hwu, "Memxct: Design, optimization, scaling, and reproducibility of x-ray tomography imaging," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2014–2031, 2022.

[83] D. B. Kirk and W. mei W. Hwu, in *Programming Massively Parallel Processors (Third Edition)*. Morgan Kaufmann, 2016.

[84] M. Hidayetoglu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W. mei W. Hwu, "At-scale sparse deep neural network inference with efficient gpu implementation," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2020.

[85] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 109–116.

[86] "NVIDIA Tesla H100 GPU Architecture," https://resources.nvidia.com/en-us-tensor-core, 2022.

[87] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 148–159.

[88] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: Association for Computing Machinery, 1991, p. 224–234.

[89] D. Gustavson, "The scalable coherent interface and related standards projects," *IEEE Micro*, vol. 12, no. 1, pp. 10–22, 1992.

[90] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "DeepStore: In-Storage Acceleration for Intelligent Queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52, 2019, p. 224–238.

[91] IMPACT Research Group, "BaM source code," https://github.com/ZaidQureshi/gpudirect-nvme, 2022.

[92] Z. Qureshi, V. S. Mailthody, I. Gelado, S. W. Min, A. Masood, J. Park, J. Xiong, C. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W.-m. Hwu, "BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage," 2022. [Online]. Available: https://arxiv.org/abs/2203.04910

[93] W. W. Hwu, I. El Hajj, S. Garcia de Gonzalo, C. Pearson, N. S. Kim, D. Chen, J. Xiong, and Z. Sura, "Rebooting the Data Access Hierarchy of Computing Systems," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 2017, pp. 1–4.

[94] J. Zhang, M. Kwon, H. Kim, H. Kim, and M. Jung, "FlashGPU: Placing New Flash Next to GPU Cores," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[95] J. Zhang and M. Jung, "ZnG: Architecting GPU Multi-Processors with New Flash for Scalable Data Analysis," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 1064–1075.

[96] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *Computing in Science Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[97] J. Zhang and M. Jung, "Ohm-GPU: Integrating New Optical Network and Heterogeneous Memory into GPU Multi-Processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 695–708.

[98] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for gpu energy efficiency through software-hardware co-design," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 93–102.

[99] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim, "DCS: A fast and scalable device-centric server architecture," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 559–571.

[100] A. W. Baskara Yudha, K. Kimura, H. Zhou, and Y. Solihin, "Scalable and Fast Lazy Persistency on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 252–263.

[101] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-mapped SSDs with FlashMap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, OR, 2015.

[102] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, Boston, MA, 2011.

[103] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, 2009.

[104] M. Saxena and M. M. Swift, "FlashVM: Virtual Memory Management on Flash," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, Boston, MA, 2010.

[105] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient Memory-Mapped I/O on Fast Storage Device," *Trans. Storage*, vol. 12, no. 4, May 2016.

[106] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "Improving SSD Lifetime with Byte-addressable Metadata," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, VA, 2017.

[107] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2b-ssd: The case for dual, byte- and block-addressable solid-state drives," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 425–438.

[108] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "FlatFlash: Exploiting the Byte-Accessibility of SSDs Within a Unified Memory-Storage Hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.