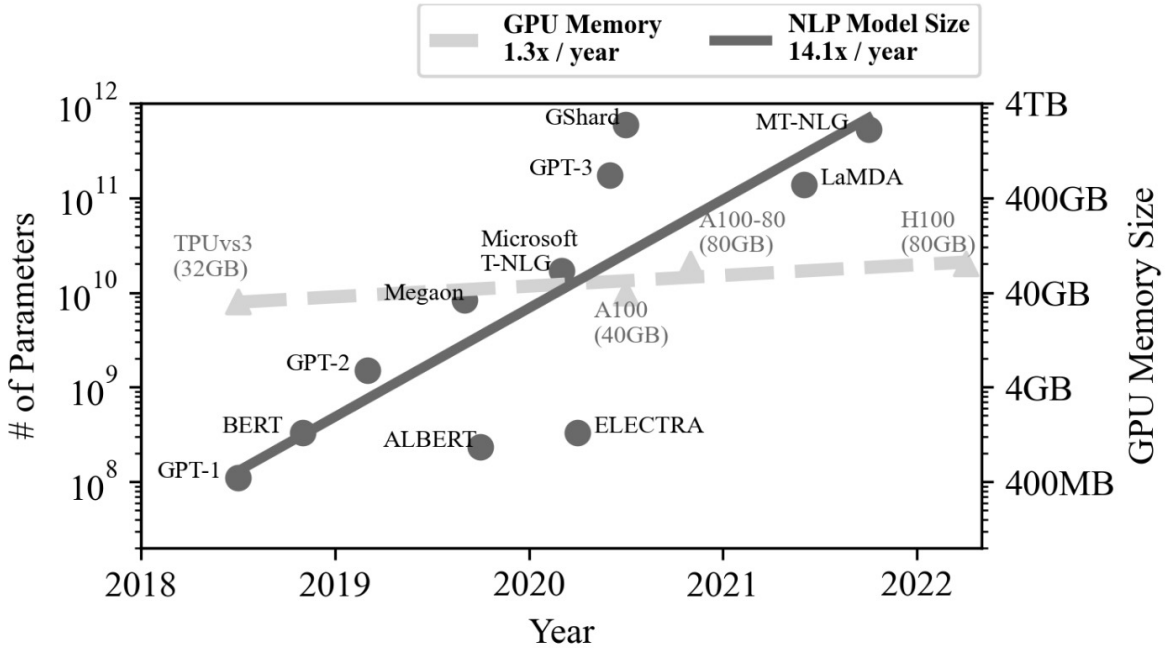
GPU-Initiated On-Demand High-Throughput Storage Access in the **BaM** System Architecture

Zaid Qureshi , Vikram Sharma Mailthody , Isaac Gelado , Seungwon Min , Amna Masood , Jeongmin Park , Jinjun Xiong , C. J. Newburn , Dmitri Vainbrand , I-Hsin Chung , Michael Garland , William Dally , **Wen-mei Hwu**

UIUC/NVIDIA/IBM/University at Buffalo/Standford

ASPLOS 2023

The Memory Wall



[*Shao-Peng + UNSENIX ATC'23*]

Memory wall

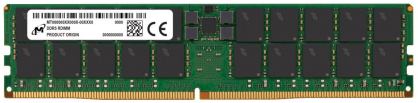
Growing imbalance between computing power and **memory capacity** requirement.

The Memory Wall

To better process data in GPU → Want to store **LARGE** datasets as in-memory object.

NVIDIA A100 : **80GB**
$19,500 from Amazon

DDR4 : **96GB**
$300 ~ $500 from Amazon



NAND Flash : **2TB**
$129 from Amazon

Solution: Let GPUs utilize DRAM/SSDs

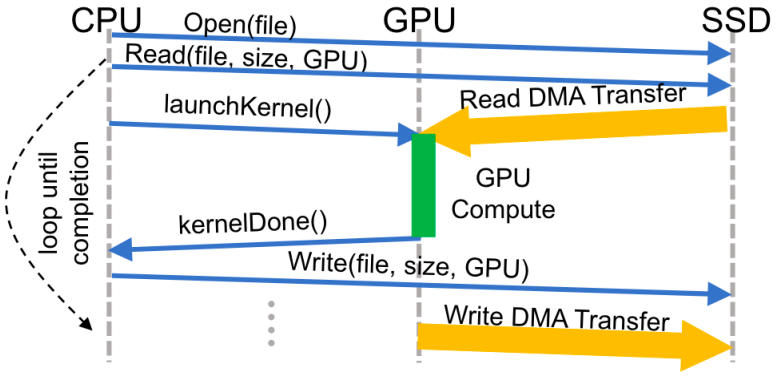
CPU-Centric `bottleneck`

- CPU/OS partition datasets into chunks and transfer data to GPU.
- GPU page fault to activate CPU page fault handler (through file map) to transfer data to GPU.

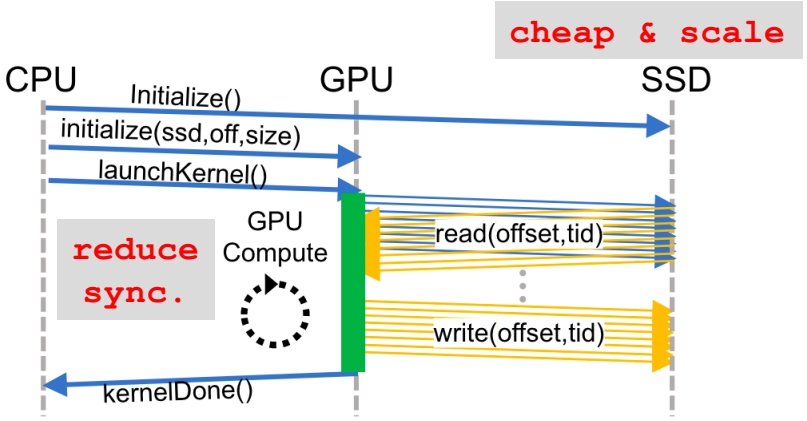
DRAM-Only `expensive`

- Use host memory || Pool multiple-GPUs' memory

What is BaM System?



(a) CPU-Centric Model.



(b) BaM Model.

CPU-Initiated Storage Access

- High CPU-GPU synchronization overheads
- I/O traffic amplification
- Long CPU processing latencies.

GPU-Initiated Storage Access

- GPU sends on-demand requests
- Thread request coalescer and software cache
- High throughput queue

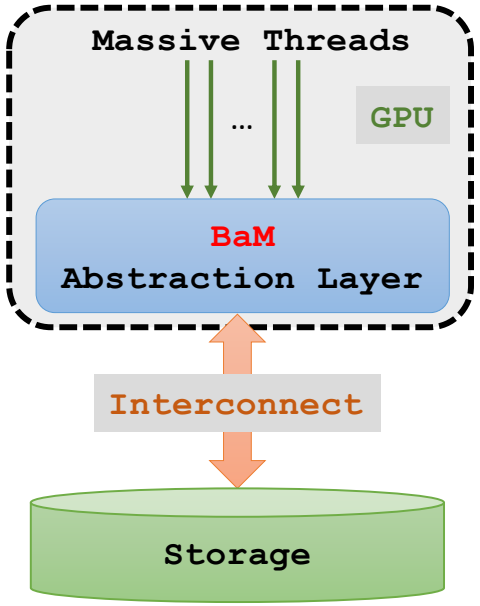
Compared to CPU-Centric

- Proactive Tiling
- On-Demand Access using the CPU (UVM/GPUfs)
- Abundant CPU Memory

https://youtu.be/qQVpaCpbCHY?t=73&si=q0I-UWuzxBbZu4pD

BaM in High Level

BaM (**B**ig **A**ccelerator **M**emory) Abstraction



Challenges for BaM

- Avoid CPU software bottleneck (page fault handler, etc.)
- Satisfy massive GPU threads high parallelism.
- Hide latency.
- Achieve high bandwidth.
- Software APIs.

Goal for BaM

Provide high-level abstractions
for **accelerators** to make
on-demand, **fine-grained**, **high-throughput** access
to storage.

BaM System Overview

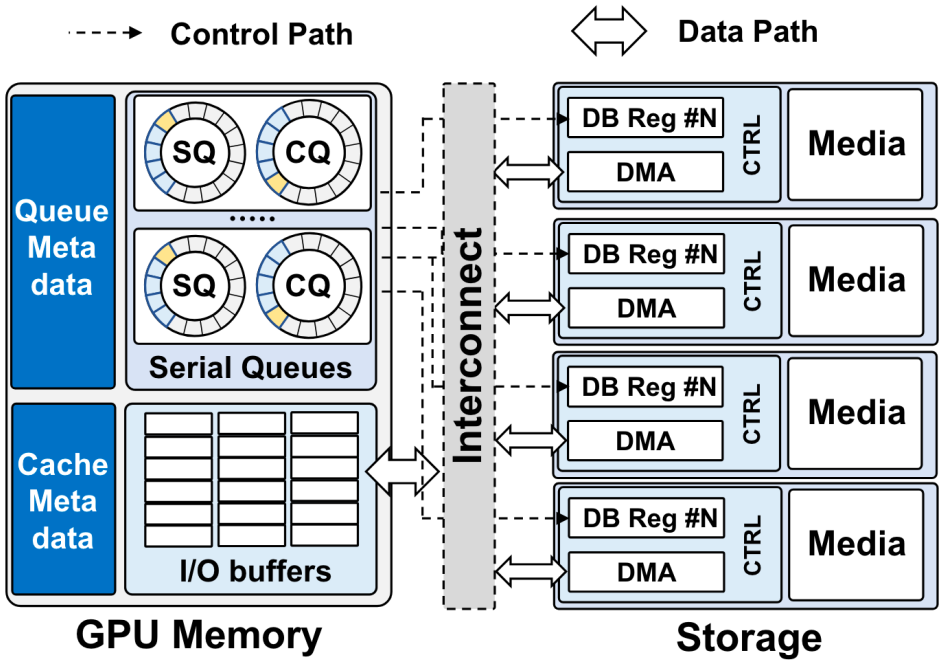


Figure 1: Logical view of BaM design.

Key Ideas

- NVMe high throughput queue design
- Map the needed storage address space
- In-GPU cache design

BaM System And Architecture

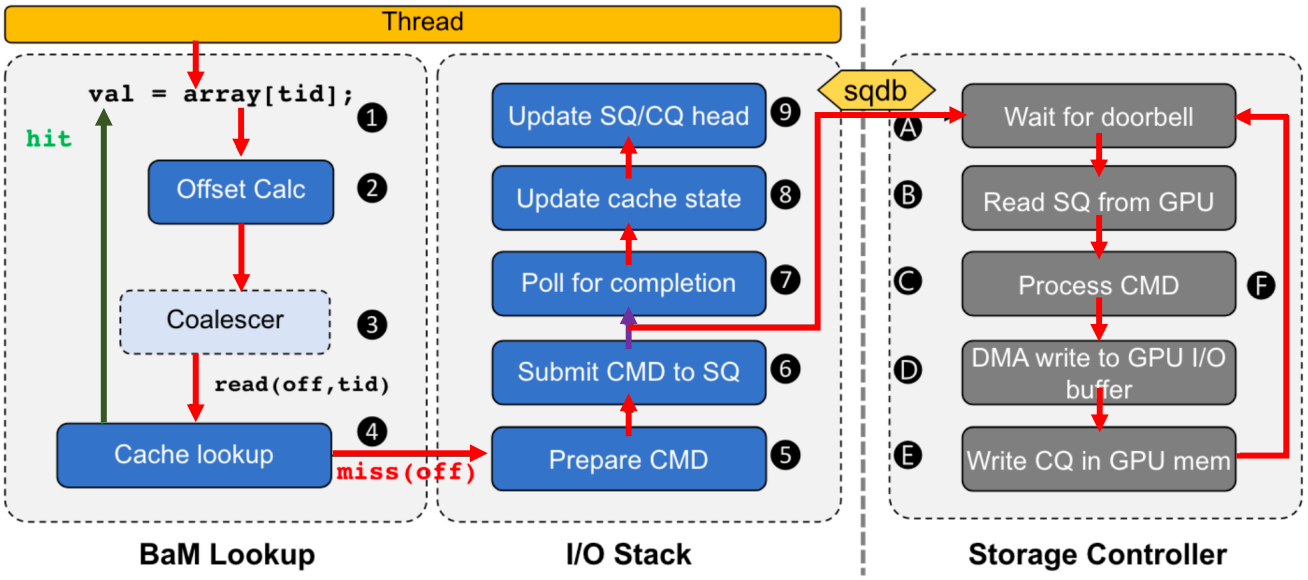


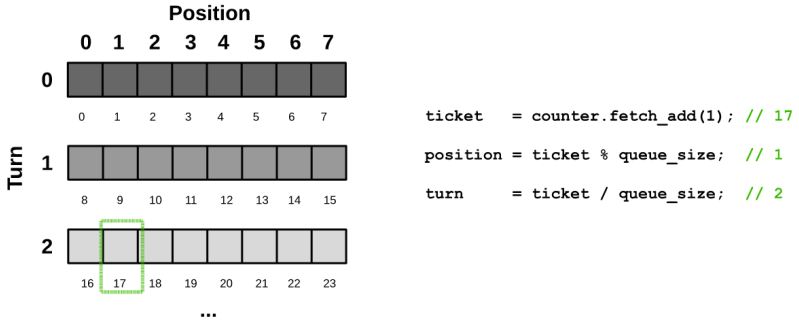
Figure 1: Life of a GPU thread in BaM

High Throughput Queue Design

- Hide latency
- Achieve high throughput

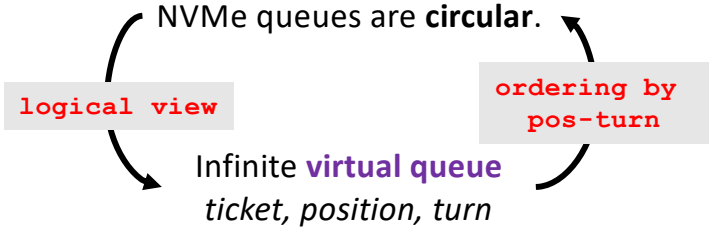
Enqueuing a command and ring the doorbell requires **critical section** involved.

(a) Virtual infinite queue for a physical queue of size 8




```
ticket   = counter.fetch_add(1); // 17
position = ticket % queue_size;  // 1
turn     = ticket / queue_size;  // 2
```

(b) Mapping the virtual queue entry in the physical queue



Parallelism for Enqueuing Commands

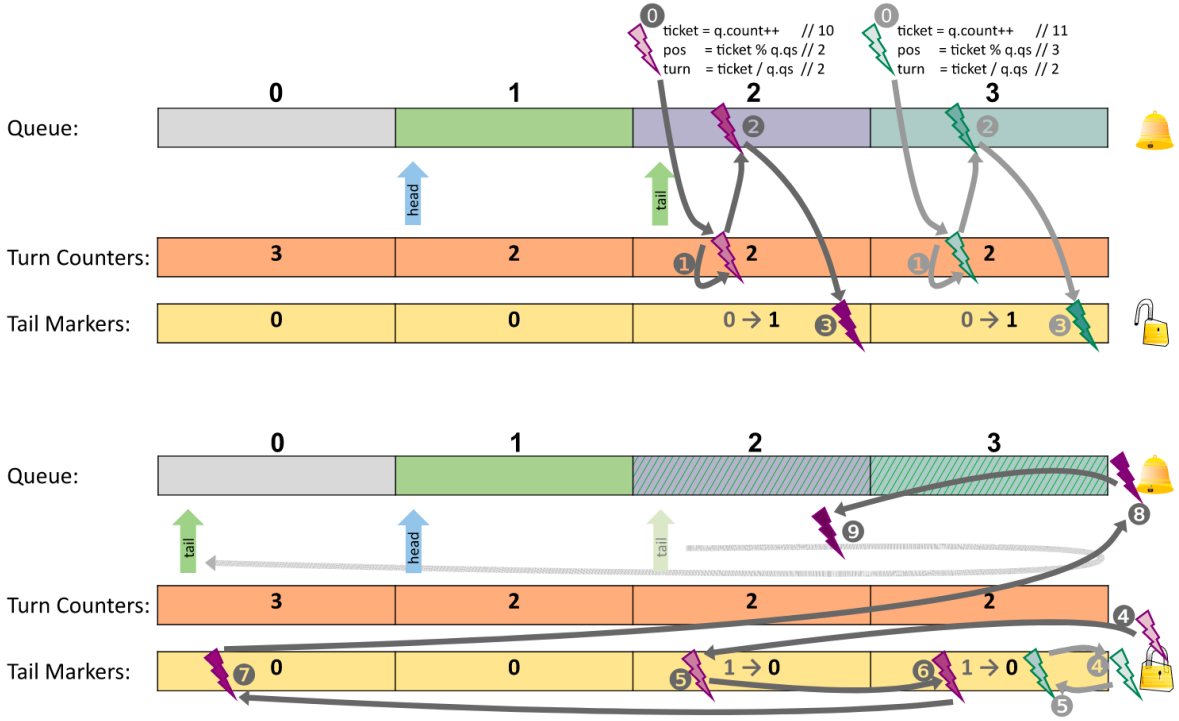


Figure 1: Example of Enqueuing Commands

Software API Design

```
__global__
void kernel(unsigned* data, size_t elems_per_thread, unsigned* ret)
{
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t wid = tid / 32;
  size_t lane = tid % 32;
  size_t start = wid * elems_per_thread * 32 + lane;
  size_t end = (wid+1) * elems_per_thread * 32;
  for (; start < end; start+=32)
      *ret += data[start];
}
```

(a) GPU linear access benchmark kernel code

- Pure **software** implementation
- Lightly modification to existing codes.
- Transparent to programmers.

```
__global__
void kernel(bam::array<unsigned> data, size_t elems_per_thread, unsigned* ret)
{
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t wid = tid / 32;
  size_t lane = tid % 32;
  size_t start = wid * elems_per_thread * 32 + lane;
  size_t end = (wid+1) * elems_per_thread * 32;
  for (; start < end; start+=32)
      *ret += data[start];
}
```

(b) GPU linear access benchmark kernel code with BaM abstraction

Enable Direct NVMe Access From GPU Threads

To enable GPU threads to **directly access data** on NVMe SSDs we need to:

- Move the NVMe queues and I/O buffers from the host CPU memory to the **GPU memory.**
- Enable GPU threads to write to the queue doorbell registers in the NVMe SSD's **BAR space**.

NVIDIA P2P

For NVMe SSD controller

Other storage systems can be enabled similarly.

- A custom Linux driver that creates a character device per NVMe SSD in the system like **SmartIO**.

  Applications use BaM APIs to open the character device for each SSD

- **GPUDirect RDMA APIs** to **pin and map** NVMe queues and I/O buffers in the GPU memory.

  SSD to perform peer-to-peer data reads and writes to the GPU memory

- **GPUDirect Async** to **map** the NVMe SSD doorbells to the CUDA address space.

  GPU threads can ring the doorbells on demand

The BaM Hardware Prototype

Independent Drawers

Not Multi-GPUs

Figure 1: Physical BaM prototype

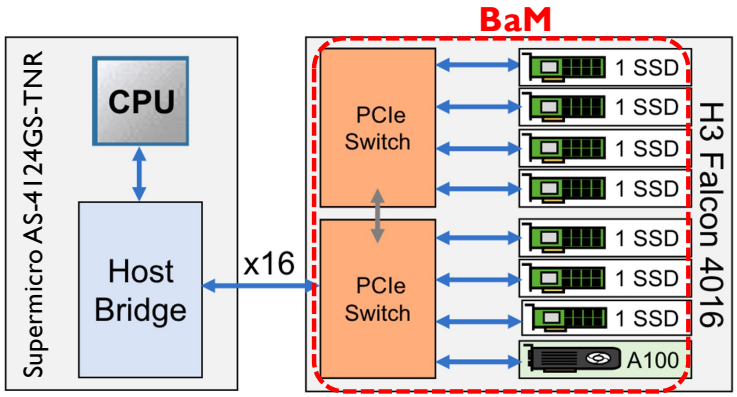
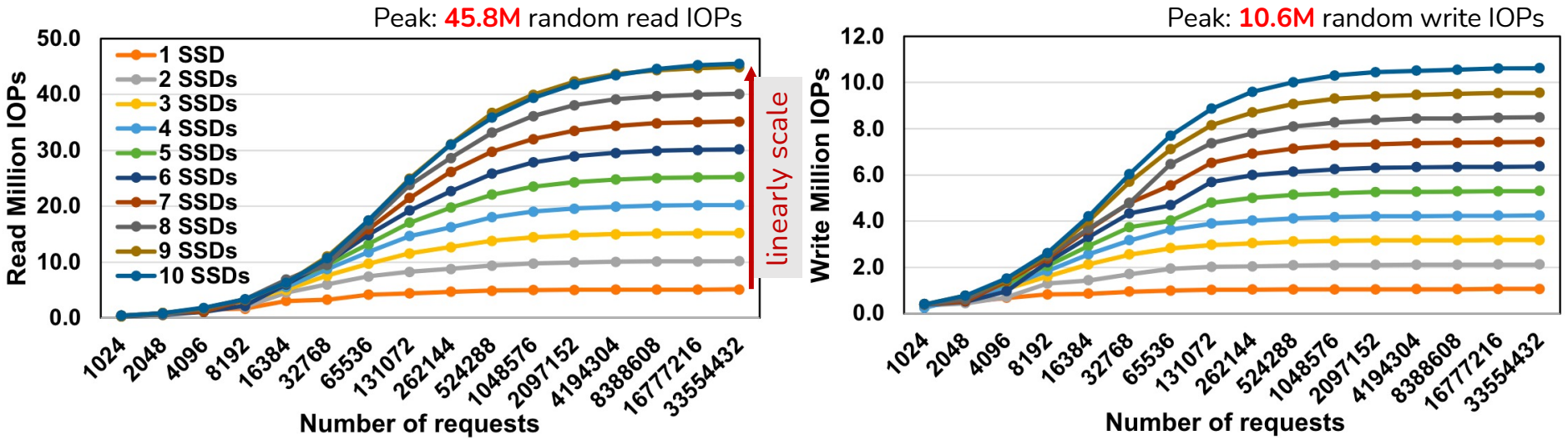


Figure 2: Logical BaM prototype

BaM Throughput Scaling

Intel Optane P5800X SSDs; Queue Depth:1024; 512B request/thread (RR distributed)
Random R/W

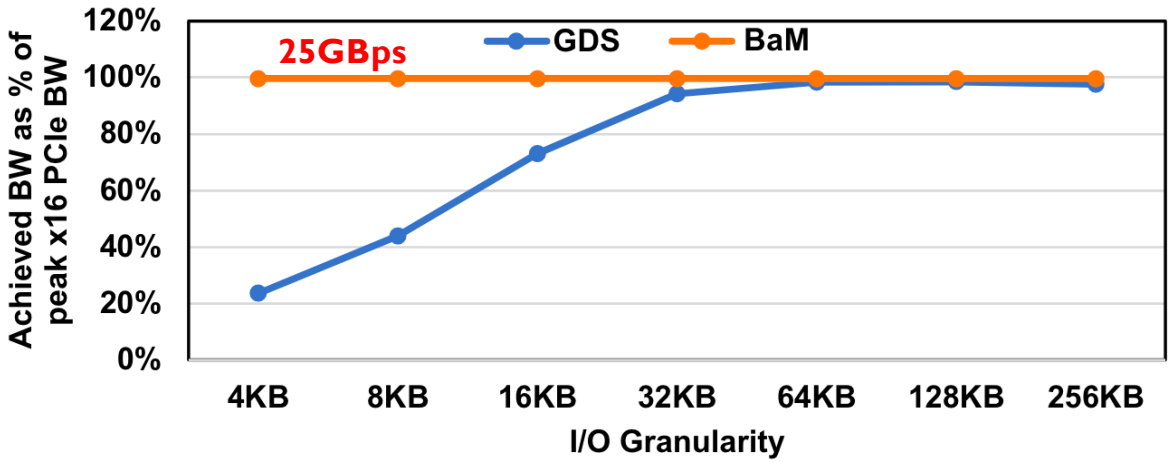


Random Read : **22.9GBps** (90% of the measured peak bandwidth for Gen4 ×16 PCIe links)
Random Write: **5.3GBps**

Compared to NVIDIA GPUDirect Storage

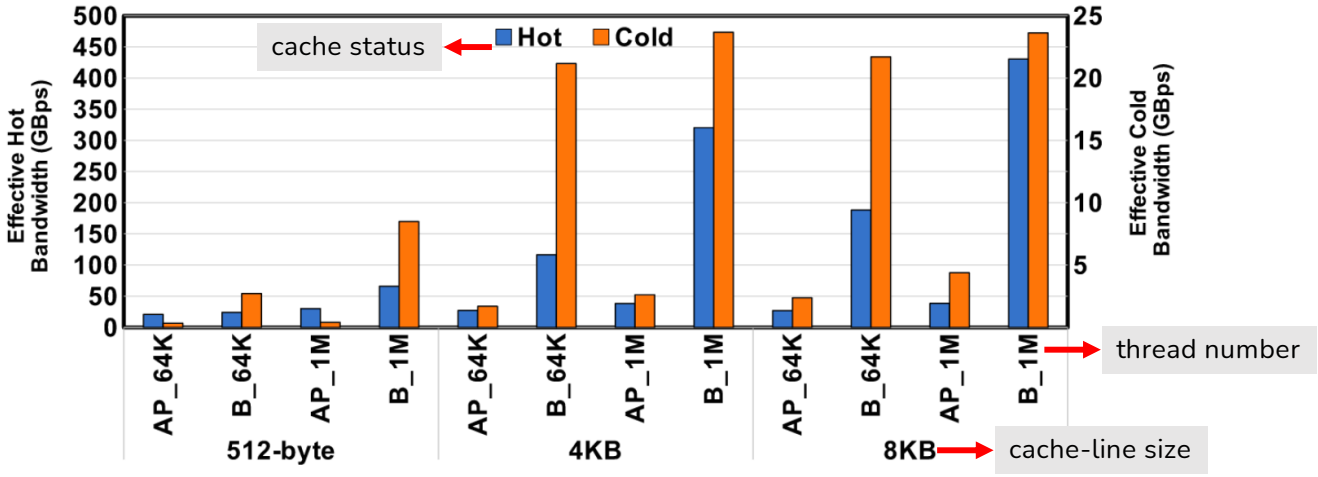
`fio`: Transfer 128GB of data from 4 SSDs to GPU

GDS: With 16 CPU threads.



Compared to ActivePointers

A warp is assigned to read 1024 contiguous 8-byte elements in a file where threads access the elements in a coalesced manner.



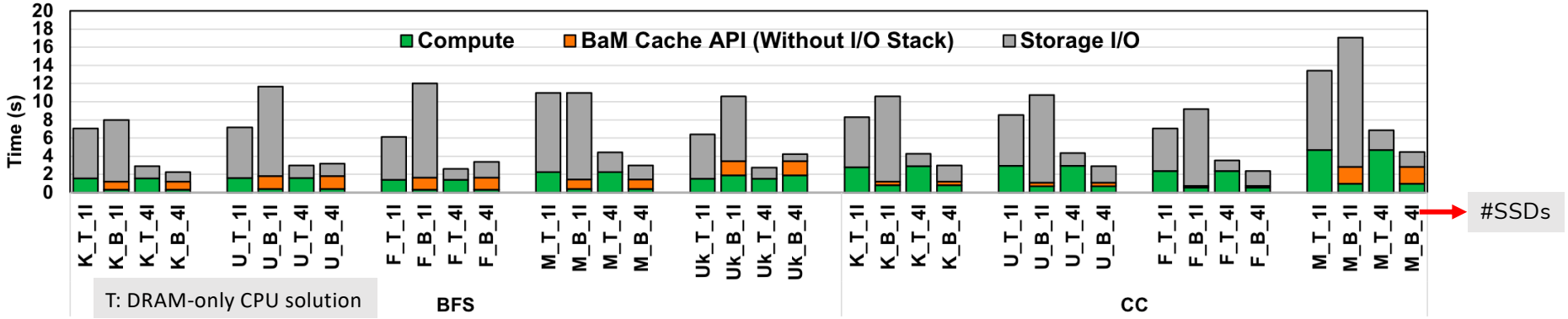
ActivePointers (AP): file is **pinned** in the Linux page cache in CPU memory.
BaM (B): file is kept on four SSDs.

[ActivePointers:]

Evaluation on Graph Analytics

Graph	Num. Nodes	Num. Edges	Size (GB)
GAP-kron (K) [7]	134.2M	4.22B	31.5
GAP-urand (U) [7]	134.2M	4.29B	32.0
Friendster (F) [68]	65.6M	3.61B	26.9
MOLIERE_2016 (M) [59]	30.2M	6.67B	49.7
uk-2007-05 (Uk) [10, 11]	105.9M	3.74B	27.8

(a) Dataset used in graph analytics



(b) Performance breakdown

1 SSD : 1.43x **slowdown**
4 SSDs : 1.00x **speedup**

1 SSD : 1.27x **slowdown**
4 SSDs : 1.49x **speedup**

Evaluation on Graph Analytics

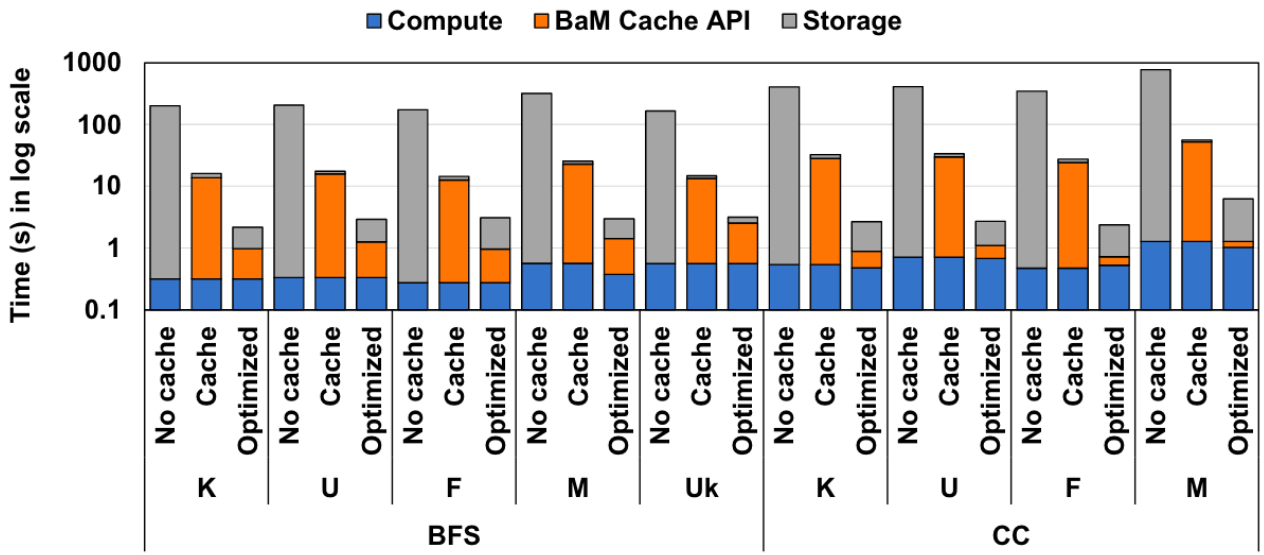


Figure 1: Source of performance improvement in BaM

Naive Cache. : **11.9x** (BFS) and **12.6x** (CC) speedup
Optimized Cache: **additional 6.07x** (BFS) and **11.24x** (CC) speedup

Evaluation on Data Analytics

NYC taxi ride dataset (over 50GB in size) and six queries

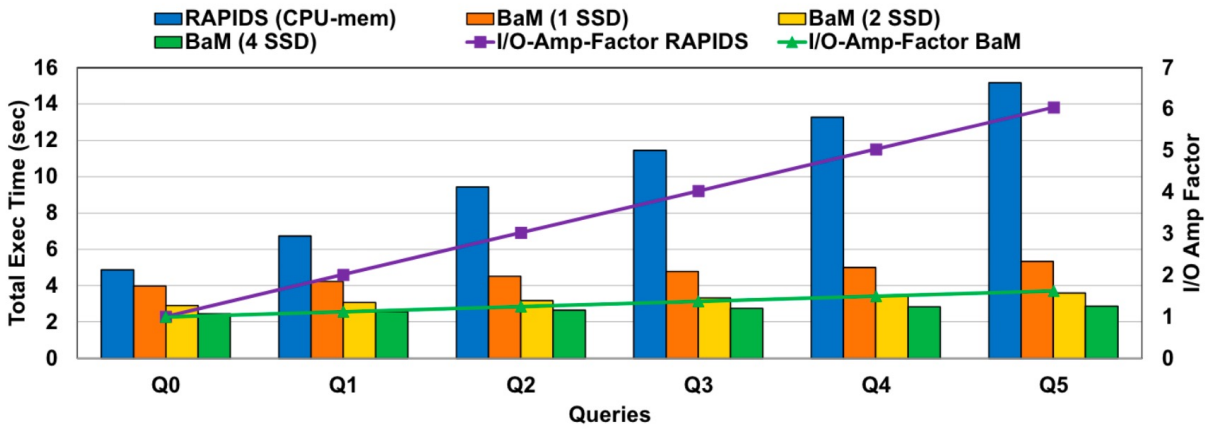


Figure 1: Performance of BaM and NVIDIA RAPIDS.

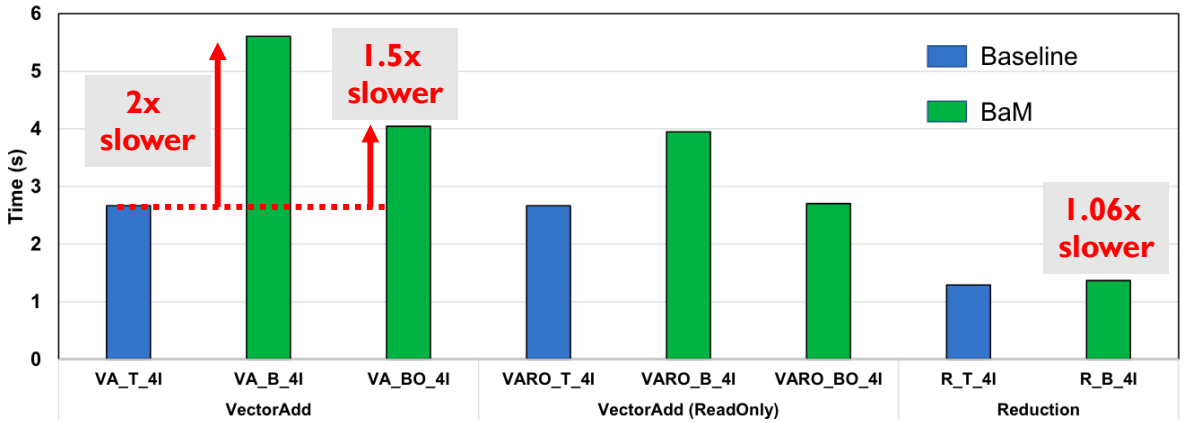
RAPIDS: **pin** the entire dataset file in the Linux CPU page cache.
BaM: data in 1-4 SSDs, 8GB cache, 4KB cache-line

RAPIDS experiences **software overheads** on the CPU to find and move data and manage the GPU memory

Evaluation on VectorAdd Workload

30GB

Take two input arrays with four billion elements (eight-Byte) each.
Input vectors are in storage, output vector requires to written to the storage.



Baseline: **VA_T**, vector add with **proactive tiling approach**, split four billion elements into five tiles.
BaM: **VA_B**, 8GB cache, 4KB cache-line, four SSDs (4I), w/ cache-line aware optimization (BO).

Conclusion

BaM, the first **accelerator-centric** system architecture that enable **GPUs to orchestrate high-throughput**, **fine-grained** access to storage without the CPU software bottleneck.

BaM Related Works

From the BaM authors

[VLDB'21] EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs
[VLDB'21] Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture
[ASPLOS'23] GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture
[ARXIV'23] Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses
[PhD Dissertation] Infrastructure to Enable and Exploit GPU Orchestrated High Throughput Storage Access on GPU
[PhD Dissertation] Application support and adaptation for high-throughput accelerator orchestrated fine-grain storage access

Other Resources

GPUDirect demo by NASA: https://youtu.be/GAZP1NcdWMo?si=OgY5WyxucYK29178
BaM source code: https://github.com/ZaidQureshi/bam
I(nterface) S(pecialization) A(pproaches) for storage system: https://www.youtube.com/watch?v=LBRRm3IfttM