CS-Assist: A Tool to Assist Computational Storage Device Offload

Lokesh N. Jaliminche[1], Yangwook Kang[2], Changho Choi[2], Pankaj Mehra[3], Heiner Litz[1]
University of California Santa Cruz[1], Samsung Semiconductor Inc.[2], Elephance Memory Inc.[3]

1 INTRODUCTION

The exponential growth of data has made data movement an obvious target of performance and power optimization for data processing applications. This has fueled a growing interest in Computational Storage Devices(CSDs) that can mitigate data movement overhead between host and storage devices in modern data-intensive applications [3]. CSDs such as Samsung's SmartSSD enable this capability by integrating a hardware accelerator in every device. With ongoing technological advancements while we get faster ways of interfacing physically to CSDs (PCIe 5 and 6), better ways of interacting with it (via CXL) [5], and better software mechanisms for programming offload [1], identifying the functions to offload remains the principal, and ever-present technical problem every current and future application of computational storage must address.

Existing methodologies follow an iterative implementation and evaluation cycle, which is slow and cost-prohibitive. We propose a systematic and general methodology for automated application offload analysis to address this issue. In particular, we propose CS-Assist to determine candidate kernels that should be offloaded to CSDs.

Recognizing the distinct nature of CSD's hardware capabilities and its position in system architecture, we first identify essential hardware and kernel characteristics contributing to performance (Section 2). Then, Section 3 provides an overall workflow to identify candidate kernels for offloading. Section 4 shows our initial evaluation of an analytics workload running atop PostgreSQL DB, demonstrating accurate estimations (less than 7% prediction error) from applying our methodology.

2 KEY HW AND KERNEL CHARACTERISTICS

Performance Characteristics of Hardware Components: The computational and memory (DRAM) resources of CSDs are inherently limited by factors such as physical space, cost, and power consumption. These constraints directly influence the potential performance benefits of offloading specific functions/kernels to these devices. The Roofline Model [8] has traditionally been utilized to perform bound and bottleneck analysis of such hardware limitations. We propose an extended roofline model (equation 1) that first accounts for computing resources by focusing on the **Peak Compute Performance (PCP) and Operational Intensity (OI)** of the kernel under analysis. Unlike previous studies that exclusively utilize floating-point operations(FLOPS) within the Roofline Model framework, we also consider integer operations as the FLOPS efficiency of the data center workloads is only 0.1%, rendering it an inappropriate measure for data center computing [6].

Next, we utilize **Peak DRAM (PDB), Storage(PSB), and Host to CSD's Interconnect Bandwidth(PIB)** to account for data transfer performance of the components involved in the data path of the kernel execution. Our extended roofline model can be scaled to consider kernel offloads on multiple CSDs by appropriately scaling the model input values. Model input values can be obtained by benchmarking hardware components with appropriate benchmarks and utilizing hardware performance counters.

$$\text{Est. Perf. (CSD)} = \min((OI \times \text{PDB}), (OI \times \text{PSB}), (OI \times \text{PIB}), \text{PCP}) \quad (1)$$

Working Set Size (WSS) : Besides the performance characteristics of different hardware components, CSDs have limited DRAM capacity because of memory cost and power consumption. Limited DRAM capacity can significantly reduce the performance of offloaded kernels whether it is used for NAND Flash caching or for holding the working set of an offloaded computation. To account for this limitation, we utilize Working Set Size (WSS), which is defined as the amount of data the application kernel requires to be present in the memory(DRAM) to avoid suffering from excessive IO overheads. Ideal WSS includes kernels with regular data access patterns with Reuse distance and average stride distance [7] smaller than DRAM available on the target CSD; otherwise, it can suffer lower performance.

Analyzing whole-application WSS is challenging due to overheads and difficulty in tracing specific kernel memory accesses. We propose an approximation technique involving Linux Cgroups to restrict DRAM and observe kernel execution slowdowns. These slowdowns, measured with Linux perf utility, indicate higher WSS for the kernel. This approach is applied to PostgresSQL, contrasting with the direct analysis used for independent kernels (Section 4).

Kernel Dependencies : Kernel A depends on Kernel B if A uses B's output. Offloading decisions, like in Table 1, consider both dependencies and WSS (Working Set Size). For example, offloading a kernel with low WSS but dependent on a host-resident kernel with high WSS incurs high communication costs. Ignoring these dependencies in offloading decisions can lead to performance issues due to increased communication overhead.

Selectivity : Defined as the input-to-output data size ratio of a kernel, higher selectivity results in more data reduction. It's crucial in offloading decisions, especially when multiple kernels are candidates. Low selectivity can strain IO bandwidth, while higher selectivity improves data reduction and bandwidth efficiency.

Kernel A WSS	Kernel B WSS	Host System	CSD offload
High	High	A and B	None
High	Low	A	B
Low	High	A and B	None
Low	Low	A or None	B or A and B

Table 1: Offload Decisions Considering Dependency and WSS

3 CS-ASSIST WORKFLOW

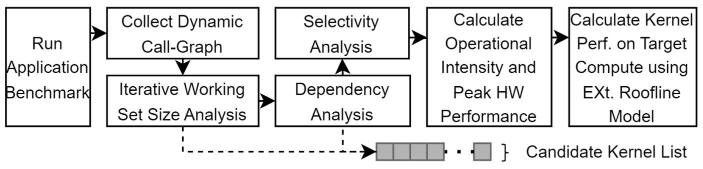
Run Application Benchmark
Collect Dynamic Call-Graph
Iterative Working Set Size Analysis
Selectivity Analysis
Dependency Analysis
Calculate Operational Intensity and Peak HW Performance
Calculate Kernel Perf. on Target Compute using EXt. Roofline Model
Candidate Kernel List

Figure 1: CS-Assist: Overall Workflow

Having identified the hardware and application characteristics that affect the performance, we next discuss our overall workflow for leveraging them to identify candidate kernels for offloading(Figure 1).

Identifying Candidate Kernels: As shown in Figure 1, to identify candidate kernels, we examine a set of representative benchmarks, that stress a particular application functionality on the host system. We collect execution stack traces of these applications via the perf utility to examine the dynamic call graphs. Next, we perform an iterative WSS analysis on the call graph starting from the leaf node. First, we check whether the leaf node kernel's WSS is < CSD's DRAM in which case it is added to the candidate kernel list. Then, we add its parent node to the candidate kernels list and calculate their cumulative WSS. This process is continued until the cumulative WSS exceeds the DRAM size available for use. If the WSS exceeds the DRAM size, we remove the recently added kernel from the candidate kernels list and continue WSS analysis on all the remaining kernels. After WSS analysis, dependency analysis performed to check if any candidate kernels depend on kernels with higher WSS. If yes, we remove them from the candidate kernel list to avoid higher communication cost. Finally, we analyze selectivity to choose the kernels with the highest selectivity and equation 1 is used to calculate estimated offload performance.

4 INITIAL EVALUATION

In this section, we illustrate the usage of our methodology to analyze and identify candidate kernels of the PostgreSQL database application. Our Target System is a CSD that utilizes FPGA as a compute unit with 4GB of DRAM, which is significantly smaller than the Host system that has Intel Xeon Processor with up to 32 active cores being utilized with 500GB DRAM. The effectiveness of our methodology is determined by the ability to identify candidate kernels for offloading and by comparing our estimated performance with real-world implementations of candidate kernels on the target device.

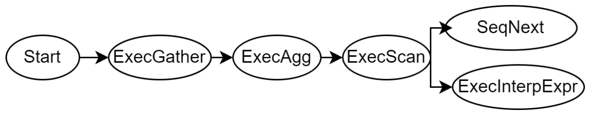

Figure 2: Dynamic Call Graph(TPCH Query 6)

Tentative Candidate Kernels and their dependencies	WSS corresponding to DRAM Capacity
ExecInterpExpr	Low
SeqNext	Low
ExecScan: ExecInterpExpr, SeqNext	Low
ExecAgg: ExecScan	Low
ExecGather: ExecAgg	High

Table 2: Kernels with Dependency and WSS (TPCH: Query 6)

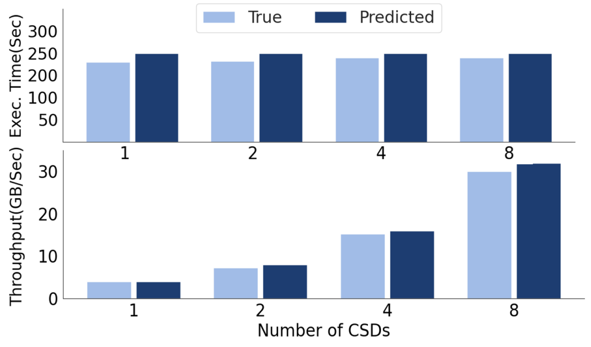

Figure 3: True Vs. Estimated Performance

Following our methodology discussed in section 3, we run Query 6 of the TPCH benchmark on the host system and collect a dynamic-call execution graph (Figure 2). Then we perform iterative WSS analysis on this dynamic call graph starting from the leaf nodes. Table 2 shows the results of the WSS analysis; apart from ExecGather, all the other kernels become candidates due to their lower WSS. Next, in our dependency analysis, since none of the candidate kernels depends on ExecGather, we proceed to the Selectivity analysis to decide whether all or subset of kernels to offload. Considering the Dependency and WSS information of the candidate kernels(Table 2), we can either offload both ExecAgg and ExecScan(ExecIntrExpr, SeqNext) or only ExecScan(ExecIntrExpr, SeqNext)(Methodology depicted in Table 1). However, offloading only ExecScan leads to lower Selectivity, while offloading ExecAgg leads to higher Selectivity since it performs aggregate operations on the data fetched from SSD/Storage. So we suggest offloading both ExecAgg and ExecScan kernels.

To estimate the performance of offloading candidate kernels (ExecAgg, ExecScan), we use equation 1. Operational Intensity is calculated following methodology provided by wang et. al [6]. The peak performance of hardware components is obtained using relevant benchmarks. We assume the peak performance for FPGAs relative to the host performance (in line with the actual FPGA implementation of the candidate kernels) as it is challenging to get the peak performance of FPGAs without implementation due to their reconfigurable architecture. We plan to address this limitation in our future work. To estimate the performance for multiple CSDs, we scale the input values corresponding to the number of CSDs. For instance, if the peak storage bandwidth of 1 CSD is 4GB/Sec, then we scale it to 8GB/Sec for two CSDs.

Figure 3 shows our results, comparing our estimated performance against the True performance of candidate kernels implementation on CSDs. We utilize our estimated performance to calculate execution time and throughput to process $\approx$ 1TB to 8TB data. (Note: The amount of data is scaled with the number of CSDs). Overall we can see that our estimated performance is sufficiently accurate(less than 7% percentage error) and can be used to perform initial application analysis to identify candidate kernels and make kernel offloading decisions.

5 CONCLUSION AND FUTURE WORK

We propose CS-Assist, which considers key hardware and application characteristics to help identify kernels suitable for beneficial computational storage offload (Sections 2, 3). Our initial evaluation (Section 4) shows sufficient performance estimation accuracy. However, we are unable to model the peak computing performance of FPGAs due to their reconfigurable architecture. We plan to address this limitation in the future. Other work in progress uses CS-Assist to identify offloadable kernels from Machine Learning workloads. NAND-based Parameter serving for large models[1] and sparsity optimization [2, 9] are the principal uses of flash storage there. In the future, in-memory processing [4] could additionally enable power and cost-efficient offload of embeddings calculation and search. We will continue to develop CS-Assist based on our experience with identifying and evaluating offload candidates in such use cases.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review*, 32(5):81–91, 1998.
[2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. 2023.
[3] Antonio Barbalace and Jaeyoung Do. Computational storage: Where are we today? In *CIDR*, 2021.
[4] Minsu Kim, Muqing Liu, Luke R Everson, and Chris H Kim. An embedded nand flash-based compute-in-memory array demonstrated in a standard logic process. *IEEE Journal of Solid-State Circuits*, 57(2):625–638, 2021.
[5] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. Database kernels: Seamless integration of database systems and fast storage via cxl.
[6] Lei Wang, Wanling Gao, Kaiyong Yang, and Zihan Jiang. Bops, a new computation-centric metric for datacenter computing. In *Benchmarking, Measuring, and Optimizing: Second BenchCouncil International Symposium, Bench 2019, Denver, CO, USA, November 14–16, 2019, Revised Selected Papers 2*, pages 262–277. Springer, 2020.
[7] Jonathan Weinberg, Michael O McCracken, Erich Strohmaier, and Allan Snavely. Quantifying locality in the memory access patterns of hpc applications. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 50–50. IEEE, 2005.
[8] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
[9] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems*, 2:412–428, 2020.

[1]https://docs.cerebras.net/en/latest/wsc/cerebras-basics/cerebras-execution-modes.html