2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Salus: Efficient Security Support for CXL-Expanded GPU Memory

Rahaf Abdullah†, Hyokeun Lee†, Huiyang Zhou† and Amro Awad†
North Carolina State University, Raleigh, USA†
{rmabdul2, hlee48, hzhou, ajawad}@ncsu.edu

***Abstract*—GPUs have become indispensable accelerators for many data-intensive applications such as scientific workloads, deep learning models, and graph analytics; these applications share a common demand for increasingly large memory. As the memory capacity connected through traditional memory interfaces is reaching limits, heterogeneous memory systems have gained traction in expanding the memory pool. These systems involve dynamic data movement between different memory locations for efficient utilization, which poses challenges for existing security implementations, whose metadata are tied to the physical location of data. In this work, we propose a new security model specifically designed for systems with dynamic page migration. Our model minimizes the need for security re-calculations due to data movement, optimizes security structures for efficient bandwidth utilization, and reduces the overall traffic caused by security operations. Based on our evaluation, our proposed security support improves the GPU throughput by a geometric mean of 29.94% (up to 190.43%) over the conventional security model, and it reduces the security traffic in the memory subsystem to 47.79% on average (as low as 17.71% overhead).**

I. Introduction

With the growing emergence of memory-intensive applications, graphics processing units (GPUs) have become essential accelerators in computing systems. Specifically, the extensive parallelism and high memory bandwidth of GPUs unprecedentedly speed up these applications by incorporating thousands of processing elements (e.g., streaming multiprocessors) and dedicated high-bandwidth memory (HBM). In recent years, general-purpose GPUs (GPGPUs) can execute not only graphics applications (e.g., video codec, image processing) but also more general tasks, for example, scientific computations [60], big data analytics [41], and deep learning (DL) [6].

However, GPUs face a "memory capacity wall" problem, as the memory footprints of modern applications have been continuously increasing. For example, training a state-of-the-art DL model (e.g., GPT3) requires more than 600GB of memory capacity to handle 175 billion parameters [8], whereas the memory capacity of current GPUs (i.e., NVIDIA H100) can only reach up to 80GB [54]. Typically, unified memory [12], [18], [34], [67] and multi-GPU [7], [48], [50] are used to overcome the capacity wall in GPUs. In unified memory, the host memory is leveraged in conjunction with the GPU device memory; however, transferring data from external memory is considered inevitable in addition to frequent host interactions for coordinating memory management, thereby incurring significant performance overhead [34]. On the other hand, multi-GPU distributes the working set over different GPU device memories, and communication between GPUs becomes a significant bottleneck [48].

Recently, several works [19], [27], [28], [40], [43] have explored the potential benefits of system memory expansion using Compute Express Link (CXL) [1], and also studied page placement and paging policies that can be adopted for efficient use of the expanded memory system. Thereafter, CXL becomes a promising solution to scale memory capacity and bandwidth in GPU systems in a cost-effective manner. Specifically, *CXL-expansion memory* allows accessing different memory technologies (e.g., DDR or NAND [29]) using the abstract and common load/store syntax without intrusive modifications of operating systems or device drivers. By storing data in the CXL-expansion memory, GPU can manage the data that cannot fit in its local GDDR/HBM memory without interrupting the host or traversing long links to remotely access host memory or remote GPUs. Furthermore, CXL features high-performance, low-latency cache-coherent accesses compared to other remote memory technologies, as CXL is defined over the PCIe physical layer with finer access granularity. Consequently, the CXL memory expansion for GPUs propels the population of cloud GPUs by offloading application execution to GPUs without incurring tremendous traffic overheads among different GPUs, ensuring high performance and quality-of-service.

However, offloading critical applications to large-scale systems, such as cloud GPUs, puts them under a higher security vulnerability of attacks targeting their data or computations. Thus, providing confidential computing is essential in such an environment. Confidential computing provides security guarantees against such attacks by introducing *Trusted Execution Environments (TEE)*. In addition to TEEs targeting general-purpose processors [5], [44], there also have been several TEE studies for GPUs [22], [23], [62]. TEE isolates application data and computations from other sharing applications or external entities. In the physical world, TEE defines a trusted computing base (TCB) that represents the trust zone, which is commonly the *GPU chip*; hence, any components beyond the TCB, including off-chip memory, are considered untrusted, and the use of these components requires security guarantees. Particularly, ensuring memory security stands as a fundamental cornerstone in creating a trusted environment for GPUs.

Unfortunately, the asymmetric bandwidths between CXL-expansion memory and GPU device memory pose a new fundamental challenge concerning both performance and se-

curity. Conventionally, heterogeneous memory systems (in our case, CXL-expansion memory and GDDR/HBM) require caching or page migration schemes to ensure high performance by moving frequently accessed/to-be-accessed data to higher bandwidth memory (e.g., HBM). Although extensive research has been conducted to investigate memory security in GPUs [2], [51], [65], [66], directly applying these security schemes incurs significant performance overheads. This is because, every data movement from the slow memory (i.e., CXL-expansion memory) to the fast memory (i.e., GDDR/HBM) triggers security-related operations (e.g., encryption and authentications). For example, swapping data between the slow memory (i.e., CXL-expansion memory) and the fast memory (i.e., GDDR/HBM) requires both memories to perform security operations when each memory sends/receives the data; furthermore, both memories are requested not only for the data but also security metadata. Consequently, the main challenge of achieving high performance in a *secure* GPU that heterogeneously adapts *CXL memory expansion* is designing an efficient security metadata management to reduce the security-related traffic by considering both security requirements and dynamic data movement.

To overcome this challenge, this paper presents a novel security model tailored for GPUs with heterogeneous memories that necessitate dynamic data movement in runtime. We loosely decouple the security metadata from the physical location of data by unifying the security metadata for both memories, eliminating the need for re-encryption during data relocation. On top of that, we restructure encryption counter blocks to efficiently share major counters among minor counters of the same interleaving granularity, yielding further minimal traffic on data relocations. Additionally, counter blocks in less frequently accessed memory are compacted, ensuring both efficient storage and traffic. Lastly, the traffic associated with metadata accesses and writebacks is significantly reduced by tracking the dirty information as bitmask format in CXL-to-GPU mappings, occurring only when necessary. Based on our evaluation, the proposed model effectively enhances performance, resulting in a notable improvement of 29.94%. In essence, our contribution can be summarized as follows:

- We propose a unified security model that decouples security calculations from the physical data location.
- We design interleaving-friendly encryption counters with restructured encryption counter blocks.
- Our proposed scheme compacts security metadata in the last-tier memory (CXL) to reduce traffic.
- We optimize security metadata accesses and writebacks by leveraging some bits in CXL-to-GPU mapping.

The remaining sections of the paper are organized as follows. Section II provides an overview of the relevant background concepts. Section III presents the motivation behind the work. The design and evaluation results are discussed in Section IV and V, respectively. Section VI provides a summary of related prior work. Finally, Section VII concludes the paper.

II. Background

In this section, we provide information about the baseline system and establish a foundation for understanding the related security concepts.

A. Memory Security

With the proliferation of cloud environments as platforms for running critical applications, the protection of confidential data from both internal and external attackers is of utmost importance. It is crucial to safeguard the data against unauthorized access or tampering attempts that may compromise the integrity or functionality of the running system or program. Therefore, the implementation of robust memory security measures becomes a necessity to prevent such malicious activities.

Memory security aims to protect data by providing three main guarantees, ***confidentiality***, ***integrity protection***, and ***freshness***. Confidentiality obscures data through encryption to block any attacker from understanding the data to prevent snooping. In contrast, integrity verification protects data from illegal changes performed by any entity other than the trusted one; it verifies the authenticity and integrity of data by employing techniques such as message authentication codes (MAC). However, ensuring data integrity alone does not guarantee that the received data is the correct and up-to-date version; data can undergo multiple updates during runtime. To address this, integrity trees [20], [57] are commonly utilized for freshness.

1) Encryption: Memory encryption restricts data understandability to the trusted computing base (TCB), which refers to the GPU chip in our threat model. By employing encryption techniques, the data is transformed into a ciphertext that is unintelligible without the appropriate decryption keys. This renders physical attacks, e.g., bus snooping, ineffective in gaining access to the original, readable data. Encryption can be done in two ways, ***direct*** or ***counter-mode-encryption (CME)***. In the former, the encryption algorithm (e.g., AES) is applied directly to the data itself. Typically, data is encrypted at the memory access granularity, e.g., 32B (or referred to as *sector* henceforth). The direct encryption algorithm exposes the encryption/decryption latency to the critical path of write/read operations consecutively. While write operations could tolerate such extra latency, read operations are more sensitive as their results are needed by the front-end of the system. On the other hand, CME, shown in Fig. 1, applies the encryption algorithm on a unique spatio-temporal initialization vector (IV). This vector consists of the data address as a spatial uniqueness factor and a special counter for providing temporal uniqueness, referred to as either the encryption counter or freshness counter. On each dirty cache block eviction, the associated counter is incremented and used to encrypt the evicted block before sending it back to memory. The encryption algorithm applied to this IV results in One-Time Pad (OTP), which, as its name suggests, should never be repeated while holding the same security key to prevent attacks. Thereafter, the OTP is XOR'ed with the plaintext data to get the final ciphertext. In opposite to direct encryption, the OTP could be pre-generated

before data arrival at the memory controller, leading to lower latency and higher throughput.

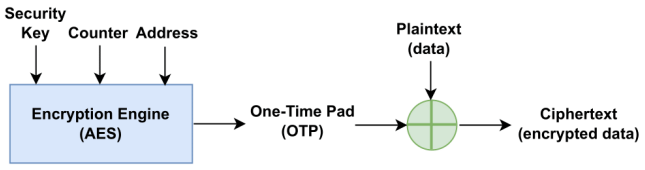

Fig. 1. Counter-Mode Encryption (CME)

Counters used for CME could be monolithic counters as in Intel SGX [20], which uses 56 bits for each. The state-of-the-art counters organization is called split counters [64] as counters are divided into two parts, minor and major. The advantage is that counters can be compacted in one counter block with a group of minor counters sharing the major one.

2) Message Authentication Code (MAC): The trusted processor writes data that eventually gets stored off-chip in untrusted memories. These off-chip memories are susceptible to physical attacks such as snooping, spoofing, splicing, and replay attacks. Encryption can prevent the snooping but not any of the rest. Hence, the integrity of the data produced by the processor needs to be protected against any alteration or tampering. Message authentication codes (MACs) are typically used for this purpose. Basically, they are cryptographic hashes generated over the data and a secure key. MAC codes could be of different lengths depending on the required security level, which increases as the length of the MAC code increases. Intel SGX uses 56-bit MAC codes per data block [20], while Partitioned and Sectored Security Metadata (PSSM) uses 32-bit MAC per sector [66].

3) Integrity Trees: Although MACs ensure the authenticity of the data read from memory, there is room for attackers to replay old versions of the data, counter, and MAC, in a way that makes all security operations pass; the memory controller cannot tell that this data is stale. Thus, integrity trees are used for tracking freshness. They achieve this by keeping the root of a tree, hierarchically computed over all memory data, in the system's TCB. Integrity trees are built over either the data itself [45] or one of the security metadata (i.e., counters or MACs [20], [57]). Level by level, the tree grows upwards until all of the data is covered by a single node, namely the root. Thus, any replay attack attempt fails as it leads to different tree nodes and root. Tree nodes could be simply hashes or MACs as in Bonsai Merkle Tree (BMT) [57] shown in Fig.2, or it could be a combination of version counters and a MAC per node as in Intel SGX integrity tree [20]. BMT is used in systems using counter-mode encryption for confidentiality, where it is built over encryption counters. To prevent all possible replay attacks, it is crucial to establish a link between tree nodes and MACs using the counters in the MAC computation. This linkage ensures that fresh counters are used in conjunction with valid MACs and data, thereby preventing situations where a fresh counter is present but the MAC and data are stale, leading to potential security vulnerabilities.

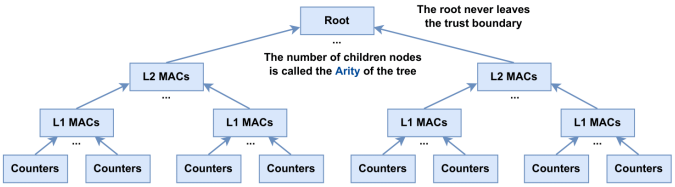

Fig. 2. Bonsai Merkle Tree over Encryption Counters

B. Memory Expansion via CXL

Compute Express Link (CXL) [1] is a high-speed, high-capacity cache-coherent interconnect utilizing a custom PCIe interface. It enables low-latency communication between the host and connected devices and memories. CXL protocol supports three different types of devices: type-1 stands for accelerators lacking memory, and they are connected to a host through CXL. Type-2 is the same as type-1 but with local memory, and finally, type-3 is a mere memory device used for memory system expansion. The CXL transaction layer consists of three protocols that are dynamically multiplexed over a single link: `CXL.io`, `CXL.cache`, and `CXL.mem`. Each of these protocols provides different capabilities and functionalities in the CXL systems. The `CXL.io` protocol provides the input/output interface operations facilitating data transfer between the host and connected devices. The `CXL.cache` protocol works on maintaining the coherency features when shared data is cached. The `CXL.mem` protocol manages memory-related operations and facilitates the utilization of CXL-attached memories in multiple modes, including: `Memory Expansion`, `Memory Pooling`, and `Memory Sharing`. The memory expansion mode enables the host system to expand its memory capacity by leveraging additional memory modules connected via CXL as type-3 devices. This allows systems to process larger datasets and memory-intensive applications. Both memory pooling and memory sharing modes focus on efficient resource utilization and cost optimization. In the pooling mode, multiple devices connected through CXL can pool their memories, while offering exclusive access for each pooled portion. Memory sharing mode, as the name suggests, enables the coherent sharing of memory by a group of CXL-connected devices. With the advancements in PCIe technology and the adoption of PCIe 5.0, the bandwidth offered by a CXL port is comparable to the bandwidth achieved using conventional DDR5 memories [40]. This highlights the potential of CXL to deliver high-performance memory access and data transfer rates, further enhancing system capabilities and performance. Due to these attractive characteristics of CXL, many vendors are developing and incorporating CXL-supported systems and memories, and various products are already available [37], [46], [58].

C. Baseline System and Threat Model

Our system consists of a GPU connected to a High Bandwidth Memory (HBM) device or GDDR memory through its memory interface. To enhance the overall memory capacity of the system, we incorporate memory expander modules by utilizing the CXL protocol as type-3 devices through the PCIe interface. The trust boundary of the system includes the processor chip and the on-chip components, while off-chip components, including memories, are untrusted and excluded from the trust domain. In our threat model, we focus on defending against physical attacks, such as bus snooping and data tampering, which can be executed on data stored in memory. Therefore, the system implements methodologies to defend against such attacks, whereas side-channel attacks (e.g., power, timing, and electromagnetic [22], [30], [52]) are all excluded in our assumed threat model. Data confidentiality in the system is ensured by employing counter-mode encryption, while the data integrity is protected through the use of message authentication codes. Plus, data freshness is guaranteed using bonsai merkle trees over the encryption counters. Each of the local memory and expanison memory has its own security metadata linked to its addresses and stored locally within the respective protected memory. These metadata are stored in a reserved fixed region of each memory. Addressing these metadata depends of the start offset of this region plus the local data address in the memory channel as defined in PSSM [66]. To summarize, every memory unit independently stores encryption counters, organized using a split counter structure, along with MACs and BMT nodes computed over its encryption counters. To enhance performance, as discussed in literature, we install security metadata caches within the controllers responsible for executing security operations.

D. Memory Organization

Generally, GPUs are designed to handle large amounts of data in parallel, such as graphics processing and scientific computations. To efficiently handle massive parallel computations and thus data transfers from and to the memory, GPUs tend to have a high number of memory channels boosting the achieved memory-level parallelism. Memory channels are pathways for data transfer between GPU cores and memory modules. The more memory channels a GPU has, the higher the memory bandwidth it can achieve. This increased memory bandwidth allows faster data transfers, improving overall system performance.

Having more memory channels alone does not guarantee high performance; these channels need to be utilized efficiently by interleaving accesses across the available channels [10]. With optimal interleaving, accesses that happen close in time head for different memory channels in parallel, maximizing the harvested system bandwidth. Different interleaving techniques [21], [32], [56] have been explored to minimize channel access conflicts and prevent partition camping. Generally, they all aim to distribute memory traffic evenly across the available memory channels to increase memory-level parallelism.

In previous work [10], the authors observed that the finer interleaving granularity could improve the performance; this is basically at the scale of one to few cache lines. Current GPUs employ sub-page granularity for fine-grained memory interleaving across memory channels [47]. Kim. H et al. [33] showed that interleaving at the cacheline level performs 1.48X better than page-level interleaving. In our model, the fine-grained interleaving chunk size of 256B is assumed; hence, moving a page from the CXL-connected memory to the GPU device memory requires interleaving across multiple channels.

III. Motivation

A. Memory Capacity Wall in GPUs

GPUs have gained significant popularity in the fields of machine learning (ML) and data science due to their powerful parallel computing capabilities. However, the limited memory capacity of GPUs can indeed pose a challenge for memory-intensive operations involved in ML and scientific applications. For example, ML algorithms often require large amounts of memory to store and manipulate intermediate results, model parameters, and input data. These data grow at the order of hundreds of Gigabytes to Terabytes [9], [17]. When the memory requirements exceed the available device memory of the GPU, the memory scarcity issue can result in performance degradation as it requires accessing remote memories, e.g., host memory, when unified memory support is enabled. Due to the aforementioned capacity limitations, the advancement in the performance and accuracy of data science and ML applications is restricted by the availability of memory capacity.

Increasing the available memory capacity in GPUs faces inherent limitations due to technological constraints. GPU memory expansion encounters challenges such as reaching the pin count limit for Graphics Double Data Rate (GDDR) or the space limitations of High Bandwidth Memory (HBM) modules [15]. Unlike CPUs, which can accommodate larger memory capacities, GPUs prioritize maintaining high memory bandwidth to meet the demands of their compute cores. As a result, enlarging GPU memory capacity while preserving the required bandwidth becomes a significant constraint, emphasizing the need for alternative approaches like external memory expansion to overcome memory capacity limitations and meet the growing demands from memory-intensive applications.

Expanding the available GPU memory using external memories provides a viable solution for the capacity wall problem. Unified memory and multi-GPUs are used to overcome the capacity wall hit by applications' demands. Unified memory uses the host memory in conjunction with the device memory; however, accessing data frequently from the host causes severe performance degradation. Normally, page faulting is used to move data between the CPU and GPU based on demand. When a page fault occurs, the host's operating system is interrupted to remap the accessed page so that it can be migrated to the GPU. This process can negatively impact performance, as it introduces overhead and latency due to the context switching and page migration [38], [42]. Multi-GPUs exploit the aggregate memory of the connected GPUs to accommodate

larger datasets. However, data movement between GPUs can create a performance bottleneck. Although multiple works optimize the performance of multi-GPU systems [48]–[50], these studies still require several accesses to remote GPUs. Therefore, expanding the memory capacity *locally* is considerably necessary while avoiding/minimizing remote access.

B. CXL-expanded Heterogeneous GPU Main Memory

The growing need for expanded memory capacity in CPUs and GPUs has increased the adoption of heterogeneous memory subsystems [11], [24], [39], [40], [43]. These subsystems consist of memories with diverse access characteristics, requiring the implementation of specialized techniques to ensure efficient utilization. Recently, CXL has emerged as a promising solution for expanding the main memory of not only CPUs but also GPUs due to its low latency and load/store syntax that abstracts the underlying memory media types.

When a system is composed of heterogeneous memories, without efficient distribution of the data during the application runtime, the system suffers from serious performance degradation. While CXL-connected memories offer lower latency compared to traditional remote memories [19], [40], [43], [59], they still introduce higher latency compared to memories connected directly through memory interfaces. In the case of GPUs, memories are often optimized to provide higher levels of parallelism compared to conventional CPU systems; hence, relying solely on expanded memories without careful memory management can lead to significant performance degradation. Previous work [7] has proved that fixing data location for GPU applications when remote memories are used is detrimental to performance. This is due to the observation that the distribution of data accesses for different pages changes over time. Therefore, maintaining a static data placement strategy for GPUs can lead to suboptimal performance, as the hotness of data accesses fluctuates. To address this issue, dynamic data movement techniques [3], [7], [34] have been proposed to efficiently adapt to the changing access patterns and hence improve overall performance, as the main objectives of these studies are ensuring the majority of data accesses to be directed to the fast high-bandwidth memory to keep the system performance near the peak.

Prior proposals on heterogeneous memories advocate using the fast high-bandwidth memory as a cache of the other memory [24]–[26], [35], [39], [55]. This configuration results in the automatic containment of the hot data within the high-bandwidth memory, thereby harvesting the best bandwidth of the system. Previous studies on DRAM caching have explored two options: caching data at the cacheline granularity or the page granularity [24]–[26]. These studies have found that caching at the cacheline granularity can be beneficial for capturing temporal locality. However, in the case of DRAM caching, where temporal locality is already effectively captured by higher-level caches, the focus shifts towards capturing spatial locality. It has been observed that larger granularities, such as memory pages, are more effective in capturing spatial locality. This is particularly relevant in the context of GPUs, where multiple threads work in parallel on the same data structures, increasing the likelihood of spatial locality patterns. Similarly, GPU device memory is used to cache the most recently accessed pages of the system data from the lower-bandwidth memory (e.g., CXL-expansion memory).

C. Overheads of Security Management in GPU Heterogeneous Memory Systems

Memory security support is crucial to protect information from unauthorized access and manipulation. However, it is important to acknowledge that security implementations introduce non-trivial performance overheads. One such overhead arises from the use of metadata, which must be accessed and processed whenever data is read from or written to its untrusted off-chip storage location. The metadata is indexed based on the address of the associated data, which is also used in generating security metadata and ciphertexts. When the location of the data changes, the corresponding metadata and ciphertext must be updated accordingly. This introduces additional complexity and computational costs to the system.

In *GPU systems with heterogeneous memories* where dynamic data relocations are performed for better memory utilization, performance impacts originated from security considerations become even costlier than traditional GPUs with homogeneous memory. The movement of data between different memories can introduce additional security overheads because security metadata is closely tied to the data location. Anytime data is moved between different memories, data decryption is required using the metadata associated with the old location, and subsequently re-encrypting the decrypted data with the metadata associated with the new location. This process involves accessing and potentially updating the security metadata for both locations.

In addition to the aforementioned overhead, the interleaved nature of data across multiple memory channels further complicates the situation. Recent works on GPU security [2], [66], have organized security metadata locally per channel due to several considerations, e.g., the coherence of security metadata, avoiding duplicates in memory. However, this approach results in a single page having its corresponding security metadata distributed across all the channels it is interleaved over, requiring the retrieval of multiple security metadata blocks from all the channels for a single page movement.

Consequently, using the conventional security metadata organization in a heterogeneous memory system can significantly impact performance. The continuous relocation of data driven by application access patterns introduces frequent security metadata accesses and updates, resulting in unnecessary overheads. Moreover, in certain scenarios, a page may be evicted from DRAM cache before all its distributed chunks in multiple channels are accessed, rendering any security-related traffic and operations for those unaccessed parts wasted.

In our evaluations of a CXL-expanded GPU memory system, we observed that the security operations associated with the data location adjustments during runtime result in acute system slowdown. Specifically, as Fig. 3 depicts, the system

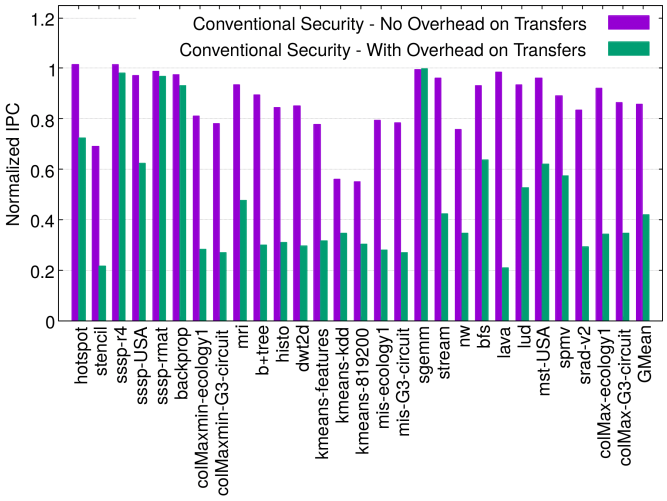

Fig. 3. Data Movement Security Overheads

experiences a performance slowdown of 2.04× compared to a system that assumes no security overheads due to data movement. This slowdown arises because security measures are directly tied to the physical location of the data, besides these security measures are distributed over multiple channels rather than centralized; hence, such a factor causes inefficiencies in managing security in a dynamic memory environment. In other words, these overheads stem from two main sources: **(a)** the re-encryption computation needed when the data location changes, and **(b)** the security traffic associated with the re-encryption process.

In fact, similar to the rationale behind moving the data to the higher-bandwidth memory for better performance, accessing the security metadata from the CXL every time would introduce a high penalty. On the other side, choosing the high-bandwidth memory as the permanent host of the whole system metadata, indeed, wastes much of the precious high-bandwidth memory capacity. Moreover, the current organization of the security metadata blocks, especially encryption counters, is unfriendly for interleaving with dynamic re-locations. Referring to the design of split counters discussed in Section II-A1, the page interleaving results in the counter block/sector being shared by several channels in the high bandwidth memory. To overcome these challenges, smarter security implementations are desired to ensure data protection seamlessly while accommodating dynamic data relocations over multiple channels.

IV. Design

This section discusses the design choices and optimizations employed by our data-relocation-friendly security design. The key is to show how security metadata can be efficiently handled in the context of a CXL-expanded GPU memory.

A. Security Optimizations: Unified Security Model

Our proposed scheme addresses the security management challenges discussed in Section III-C.

The coupling of the security metadata with the exact data address, forces all these re-encryptions to happen when data is transferred. Hence, by decoupling the security measures from the physical location of data, we can avoid the need for re-encryption when data is moved to different locations. In our approach, we propose to compute a unique security measure for each data unit that remains valid regardless of data location in the memory system. This is achieved by utilizing the GPU device memory as a cache for the CXL-connected memory, allowing us to treat the CXL address as a permanent physical address that remains unaffected by data movements. Consequently, this address can be used for all security computations.

In our approach, we eliminate the need for separate security metadata models per physical memory. Any data unit has a single security-related metadata, which can be used correctly in any heterogeneous memory. When a page is moved to the GPU device memory, this transition occurs seamlessly without requiring additional security computations. The security computations take place as in traditional systems with fixed memory locations, triggered on last-level cache (LLC) misses and writebacks.

Limitations of Previous GPU Memory Protection Schemes: While data is transferred at the page granularity, moving the metadata at the granularity of a single block is more beneficial. Moving an entire metadata page would encompass multiple data pages, which may not all be cached in the GPU memory, resulting in wasted memory and loss of locality in the security metadata. Previous research [66] has shown that organizing security metadata in a contiguous manner for sequential page blocks can result in difficulties when dealing with memory partitions that have interleaved pages. Thus, they proposed that each memory partition contains the security metadata for its respective data. In other words, each page has its data blocks distributed across multiple memory partitions, and their associated security metadata are also present in multiple partitions. However, this approach introduces a challenge for the major encryption counter management. The dynamic movement of blocks between the CXL memory and the GPU device memory results in different blocks belonging to various CXL pages but residing contiguously in a single memory partition in the device memory sharing a single major counter, but they may not have the same access pattern, resulting in different major counter values. As a consequence, the necessity to share this counter with different values would require re-encryptions to unify their counters. The same problem arises when a CXL page is to be evicted from the GPU device memory. The page is collected from a group of partitions, each using a different major counter for securing the page portion it has, potentially having different values that would require consolidation through re-encryptions. To address this challenge, we propose interleaving-friendly split counters (Section IV-A1). Furthermore, collapsed counter (Section IV-A2), fine-grained fetching (Section IV-A3), and fine-grained dirty tracking (Section IV-A4) are proposed for bandwidth optimizations.

1) ***Interleaving-Friendly Split Counters****:* The conventional design of split counters used for encryption and freshness in-

volves pairing a major counter with a group of minor counters. The major counter is shared among minor counters, and each minor counter is associated with one of the consecutive data blocks in the physical memory. Each counter tracks the number of writes to its respective memory location. The indexing of these counters is based on the address of the memory location, specifically the data address in the CXL-connected memory. The current design of a counter block for GPUs [66], as optimized for sectored caches, includes a major counter shared among 32 minor counters / 8 data blocks. However, this design makes counter management more complex when the CXL page is distributed among multiple partitions in the GPU device memory. As the single shared major counter, covers data that exceeds the typical interleaving granularity of 2 blocks, resulting in the challenges described above in IV-A.

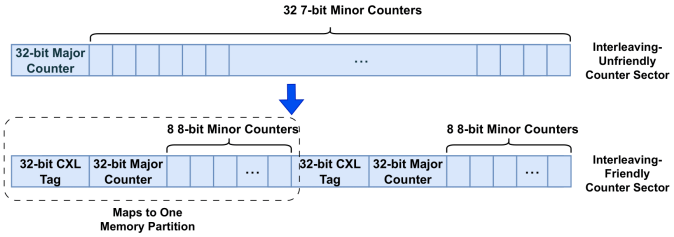

Fig. 4. Interleaving-Friendly Split Counters in the GPU-Device-Memory Side

As shown in Fig. 4, our interleaving-friendly split counters divide minor counters into groups based on the interleaving granularity. Each group of minor counters shares an individual major counter. The aim is to have a unique different major counter per data that resembles one interleaving chunk. This eases the movement of counters between memories and eliminates the need for extra complex processing to duplicate the major counter in multiple locations and handle overflows in a non-localized manner. In our system, in the GPU device memory, one major counter is shared among 8 minor counters. This arrangement results in two groups of counters within a single counter sector. Each of these counter groups is associated with one interleaving chunk of data. These chunks, despite belonging to different CXL pages, coexist in one partition within the GPU device memory. A 32-bit tag is associated with each counter group to identify the CXL page it belongs to, this is needed for the optimization explained in IV-A3.

With this interleaving-friendly counter block organization, the transferred counters align perfectly with the transferred data. This facilitates us to store the counters in the GPU memory at specific addresses that can be indexed based on the destination location of the transferred data. As a result, the addressing of the counters in the GPU memory is independent of the expansion memory addresses so that we keep maintaining locality among the stored data in the GPU memory, while security calculations use the expansion memory addresses to allow for unified metadata across all system memories.

Fig. 5. MAC Sector with Collpased Major Counter at Transfer

*2) **Collapsed Checkpointed Counters**:* To minimize the traffic between two heterogeneous memories even further while benefiting from the fact that data is cached at the high-bandwidth GPU memory, we propose that the fine-granularity minor counters can be eliminated and collapsed into their single major counter at the CXL memory side. This implies zero value per each of the minor counters. As a result in the CXL, we only have a single major counter of 32 bits per interleaving chunk. However, the counters in the high-bandwidth memory still apply the fine-grained split design to allow for finer-grained per-sector tracking of data writes without incurring extra re-encryptions and security operations due to the sharing of a single counter. The difference is that the minor counters are reset when the major counter is filled from the expansion memory on page transfer. While on the opposite transfer to CXL, if any of the minors is not zero, the major is incremented and all minors are reset with the required re-encryptions for the collapse process. In contrast to counters, MAC values are independent of each other and have no shared values. This makes them more friendly for interleaving with dynamic relocations. A single MAC sector holds MACs for a single data block. Additionally, 56-bit MAC can be used as Gueron [20] has proved that it provides a sufficient security level. This leaves room for embedding the corresponding major counter in the same MAC sector at transfer between memories. In other words, in our proposed scheme, the counter sector is collapsed into the corresponding MAC sector, as shown in Fig. 5. Hence, our approach completely eliminates the traffic of counter blocks between memories, and only MAC blocks need to be transferred, with corresponding counters held internally. Freshness trees, on the other hand, are maintained locally within each memory. This approach allows us to minimize the amount of data that needs to be transferred while still preserving the same security level.

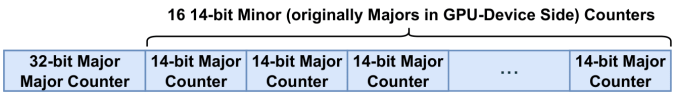

Fig. 6. Counter Sector Design in CXL-side (Split Design of Collapsed Majors)

At the expansion memory, one option is to merge counters into MAC blocks and use MAC blocks directly for building the BMT. However, the BMT's depth and width would significantly increase, as MAC blocks span a much larger memory range than counters. To keep the BMT traffic in the already scarce-bandwidth memory as small as possible, we propose to maintain separate counter blocks, on top of which the BMT is built. And as counters and MAC information needs to move together to other memory, the required counter is loaded to the corresponding transferred MAC sector for saving bandwidth. The design of counter blocks in the CXL memory

side adopts the split counter design, but due to the collapsed major counters, and to limit the minor counter overflow, the size of each minor counter is doubled as shown in Fig. 6. It functions as conventional split counters, as long as minors share the same major, no extra operations are needed. Once a minor overflows to use a different major, all minors are reset and major is incremented and all affected data is re-encrypted.

*3) **Fetch Only On-Access**:* Prior works on DRAM cache investigated both moving the whole page completely or using prediction to move only the parts that are expected to be accessed [25], [26], [63]. Our proposal works with any of these, and regardless if all of the page data is moved at once or not, the fetch on access for security metadata is applicable. A page is distributed over multiple memory channels, and each channel requires different security traffic to access its local security blocks. However, not all moved parts of the page receive access before the page is selected to be moved back to the other memory. Therefore, there is an opportunity to save some of the bandwidth wasted on transferring MAC blocks for data that would be untouched/unchanged.

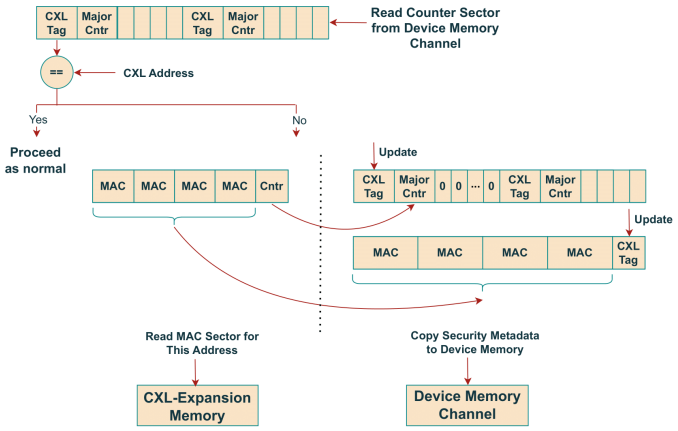

Fig. 7. Accessing Security Metadata in GPU Device Memory

Based on the observation above, we propose to fetch MAC blocks opportunistically. This means there is a possibility that the security metadata is not yet available in the high-bandwidth memory, even though the data is there. To track this, we add additional tags to the counter and MAC blocks in the high bandwidth memory, the CXL Tag in Fig. 7, which identifies to which data page they belong. This allows all security accesses to proceed optimistically, assuming that the metadata has already been moved to this memory. By performing a single comparison against the tag, it can be determined whether the access can continue or if a read from the expansion memory (to retrieve the MACs) is required. Whenever the security metadata is read from the expansion memory, the metadata is populated into the security metadata caches located in the device memory controllers and subsequently it is stored back in the CXL memory upon evicting the page from the GPU device memory to the CXL expansion memory. Page read and eviction operations are shown respectively in Fig. 7 and Fig. 8.

*4) **Fine-granularity Dirty Tracking**:* On evictions, dirty data has to be written back to the expansion memory. Typically, the dirty tracking happens at the page granularity utilizing one bit from the page table entry to identify if the page is dirty or not. This coarse tracking definitely results in write amplification, because not necessarily all blocks are dirty. Prior work Kona [11] has shown that most applications write to a much smaller set of the page's blocks, and they propose finer-grain tracking for dirty parts of the page to prevent write amplification. This can also be applied to GPU applications [51], [65]. In a GPU page table entries design [53], there is even no dirty bit. With unified memory support, it is possible to oversubscribe the GPU memory while using the host CPU memory as a swap space. Since the GPU page table entries do not support dirty tracking, all evictions result in writebacks to the host CPU. We utilize the fine dirty tracking as Kona [11], to reduce the traffic for both data and security metadata. The granularity of tracking used in our system is the same as the interleaving granularity. We utilize some bits in the address mapping to host the dirty tracking bits – more on this in Section IV-B.

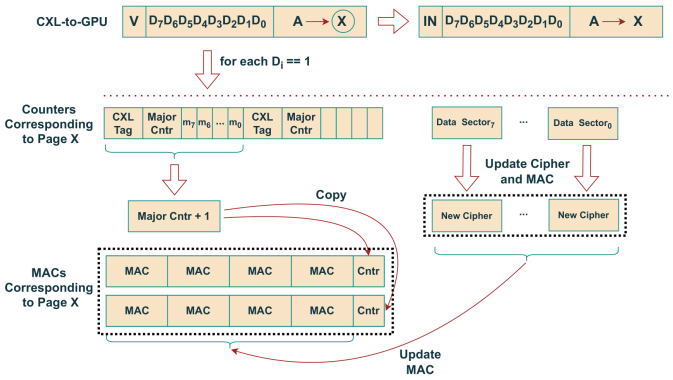

Fig. 8. Handling the Eviction to Expansion Memory

On a page eviction, the dirty tracking will be consulted to determine which chunks of the page need to be written back to the expansion memory, as shown in Fig. 8. The transferred parts are the ones surrounded by the dotted black line. This process requires collapsing counters into one major counter, and thus re-encrypting the chunk with the new counter before it is moved back to the expansion memory. This is indeed much less overhead compared to the re-encryption needed for an entire page at any movement.

B. Memory Cache Tags

In prior works on using memory as a cache [24], [35], [39], various approaches have been explored for managing memory caches and tagging the cached data. Some employ traditional caching techniques, where each cached data item is associated with a corresponding tag. In some cases, the tags are stored in separate SRAM caches, while in others, they are stored directly in the memory alongside the cached data [24]. Alternatively, other approaches utilize additional tables

or mappings to keep track of the addresses of the cached data in different memory locations [35], [39]. In the context of Tagless DRAM cache [39], the management of DRAM caching is achieved by designing the page tables to store the DRAM cache address whenever data is cached in the DRAM. Conversely, if the data is not cached in the DRAM, the page tables hold the address of the data in the last-tier memory. To track the location of cached data in the last-tier memory, a separate table called the global inverted page table is employed. This table maintains mappings from the DRAM cache to the corresponding addresses in the last-tier memory.

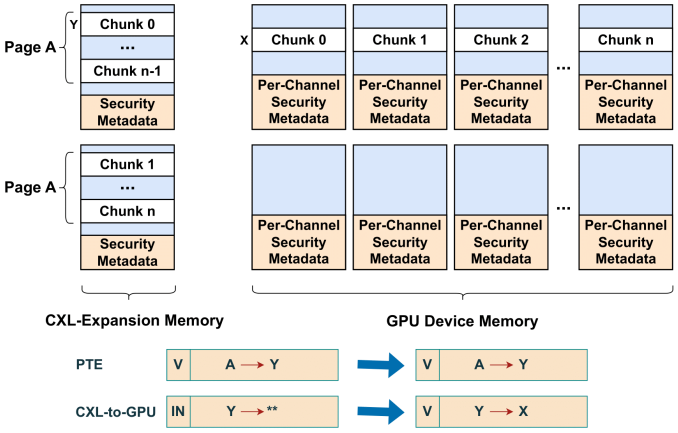

Fig. 9. Caching Pages in Device Memory

Our work adopts the auxiliary mappings table approach, rather than the traditional cache tags. This decision is made to avoid the storage overhead associated with caching tags in channels per a smaller granularity than a page. While storing a single tag per page would require accessing two different channels for each memory access, adding complexity and inefficiency to the system. Fig. 9 shows the page caching in the device memory.

While our work focuses on security optimizations and is orthogonal to memory caching techniques, retrieving the CXL address is crucial for security operations whenever off-chip memory access occurs. In other words, the CXL address must be obtained to perform necessary security checks. Given this requirement and to optimize the overheads in a GPU system with expanded memory, our work proposes flipping the address translation order. The page table entries permanently store CXL addresses, making them transparent to any page caching or eviction to and from the GPU device memory; hence, TLB shootdowns are avoided. Furthermore, the use of CXL addresses in L1 data caches eliminates the need for cache flushes, even if the data is no longer cached in the GPU memory. This is because the addresses remain unique and can be directly used for subsequent memory operations. When memory access fails to find the requested data in L1, it must be transmitted through the interconnect to the designated L2 slice coupled to a memory partition. The address of the data in the GPU device memory determines the routing decision. Therefore, the first need to access the second mappings table occurs before making the interconnect routing decision and also before accessing L2, thus, L2 can be addressed based on the GPU device memory addresses.

CXL mappings to the GPU memory are stored in a hashed table for space efficiency. Each mapping sector of 32 bytes holds 4 consecutive CXL mappings to preserve spatial locality benefits. These are stored in the GPU memory interleaved, similar to application data. Whenever a memory request passes through the interconnect, the corresponding mapping is required to route the request to the designated memory partition. Reverse engineering by Ahn, Jaeguk, et al. [4] revealed that all streaming multiprocessors (SMs) within a Graphics Processing Cluster (GPC) in the GPU share a single connection to the interconnect. To optimize the retrieval of address mappings, we augment each GPC connection with a CXL-to-GPU address mapping cache, capable of storing 128 entries. This design choice ensures reasonable logic complexity while maintaining compatibility with the existing hardware. Memory requests that hit the mapping cache can proceed with both the CXL and GPU memory addresses to the appropriate memory partition. However, if a memory request misses in the mapping cache, it is directed to a dedicated control logic responsible for handling mapping misses. This logic issues requests to read the mappings from memory and verifies if the corresponding page is already cached or not. If the page is resident in the GPU memory, the mapping is eligible to be sent to the requesting cache. On the other hand, if the page is not present in the device memory, a request is made to copy the page to the device memory, and a new mapping is established. Once the copying and mapping process are complete, the mapping cache request is fulfilled. To maintain free space at all times, evictions from the GPU memory may occur in the background as discussed in previous works [35], [39]. When evictions take place, stale mappings need to be invalidated from the mapping caches. The mapping miss handling logic could track expected caches to have the translation, so invalidation is sent only to a subset of the mapping caches to reduce generated traffic.

The control logic is also utilized for dirty tracking; it is augmented by a 32-entry buffer that holds CXL-to-GPU mappings of pages whose dirty bitmask has changed since the translation was last accessed from memory. Whenever a write access occurs, the control logic receives the CXL address along with a write signal. Then, it checks if this mapping is present in its buffer. If not, it reads the mapping from memory and sets the corresponding dirty bit. If the mapping is already buffered, it simply sets the corresponding dirty bit. The cached mappings in the CXL-to-GPU mapping caches can be evicted silently, as the most recent version of the dirty bitmask is either in the controller buffer or the memory. When the buffer gets full, the least recently used mapping gets evicted to memory without any other processing.

Security Impact: Salus guarantees the same security level achieved by other memory security models [2], [51], [66]. It organizes the security metadata differently for efficient system performance. The only difference is that different counters

TABLE I
BASELINE GPU CONFIGURATION

SM Config	80 SMs, 1132 MHz
Register File	256 kB/SM, 20 MB in total
L1 D-Cache	32 KB/SM
Shared Memory	96 KB/SM
L2 cache	2 banks/memory partition, each L2 cache bank is 96 KB, 6 MB in total
DRAM (HBM)	850 MHz, 32 partitions, 868 GB/s, pseudo-random memory interleaving
CXL-Expansion	Capacity up to 16TB, $\frac{1}{16}$th of the device memory bandwidth
CXL-to-GPU Mapping Caches	128-entry, One per GPC, dual-ported

are used for the same memory address in the GPU memory, which could result in the same value being used more than once, however, this is protected against OTP reuse by using the unique address of the data in the CXL memory.

V. EVALUATION

A. Simulation Environment

We use GPGPU-Sim v4.0 [31] to evaluate our CXL-expanded memory system. The configuration of the baseline GPU is shown in Table I, which is modeled based on Nvidia's Volta architecture. To simulate the expansion memory connected via the CXL protocol, we modeled extra memory modules of an aggregate bandwidth that is equal to $\frac{1}{16}$th of the device memory bandwidth, which is nearly equal to PCIe 5.0 $\times$16 bandwidth.

The device memory is assumed to hold 35% of the application footprint. Both L1 and L2 data caches are sectored by default in Nvidia Volta architecture. Based on the proposals of prior works on GPU security [2], [65], [66], metadata caches are also sectored and use local partition addresses. Table II shows the detailed configurations of used metadata caches.

TABLE II
METADATA CACHES AND SECURITY CONFIGURATION

MAC Cache, Counter Cache, BMT Cache	Each 2kB/memory partition, 128B blocks, 4-way sectored, 256 MSHRs, allocate-on-fill policy
MAC Latency	40 cycles
Encryption Engine	1 pipelined AES/memory partition

We evaluate our system using a collection of benchmarks from various benchmark suites, including Rodinia-3.1 [14], Parboil [61], Lonestargpu-2.0 [36], and Pannotia [13]. The benchmarks used in our study are chosen from different memory intensity categories. Stencil, B+tree, Lava, and NW are categorized as low memory-intensity benchmarks, while the remaining benchmarks have a memory bandwidth utilization of at least 20% and are considered to have medium or high memory intensity.

B. Experimental Results

1) Performance Improvement: Fig. 10 demonstrates the instructions per cycle (IPC) improvement achieved by our proposed security design compared to the impact of the conventional security design on a GPU system equipped with heterogeneous memory configurations as in Tables I and II, where data is dynamically relocated. The instructions per cycle (IPC) are normalized to a system with the same memory configuration but without any security support. Our data-relocation-friendly security design achieves a 29.4% improvement over the conventional security model.

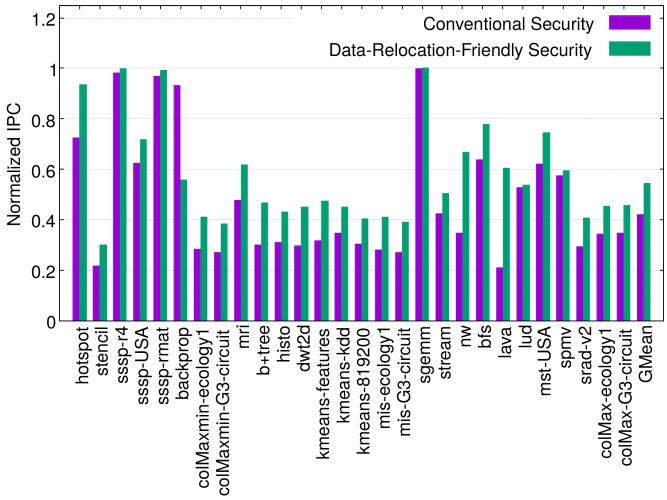

Fig. 10. Performance Improvement Driven by Our Security Design

This performance boost results from the reduced security overhead, on both data transfer directions. First, all security operations due to the transfer between the different memories are eliminated as the metadata is unified to be used regardless of the data location in the memory system. Second, the security traffic is optimized to be minimal as possible so that it occurs only when needed, utilizing minimal bandwidth. Moreover, building the BMT in the last tier memory over much coarser granularity counters shrinks its size and thus its traffic, especially since it covers a larger memory capacity. The workloads that experience the highest improvement are those accessing fewer memory channels during the lifetime of their pages in the device memory. Specifically, benchmarks like NW, B+tree, and Lave fall into this category. For these workloads, the majority of the pages have less than half of their memory channels accessed before they are evicted from the device memory. On the opposite side, benchmarks like Backprop and Segmm experience either no change or slowdown. This is because these benchmarks touch almost all channels for the majority of the transfers. Moreover, these accesses are spread out *over time*, resulting in more misses, especially when traversing the BMT to verify counters, in contrast to the baseline that verifies all page counters at once and thus exhibits better locality.

2) Security Traffic: Fig. 11 shows the normalized security traffic after applying our data-relocation-friendly security design. As a result of our approach, 52.03% of the security traffic is reduced since data that remains unaccessed during the page's lifetime in the GPU device memory does not trigger any security operations or traffic. This aligns with the explanations in V-B1; hence, with our proposal, applications that access

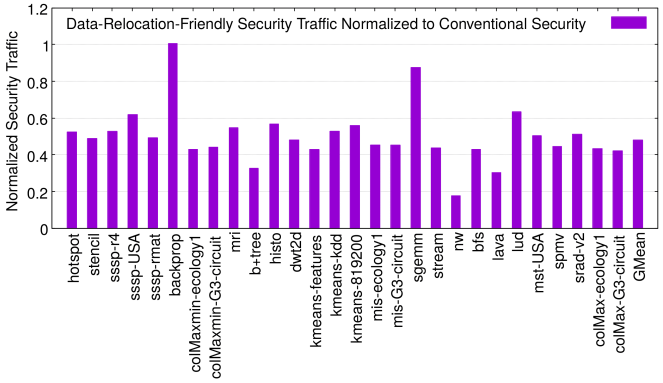

Fig. 11. Security Traffic

fewer channels during their lifetime have much lower traffic for security operations.

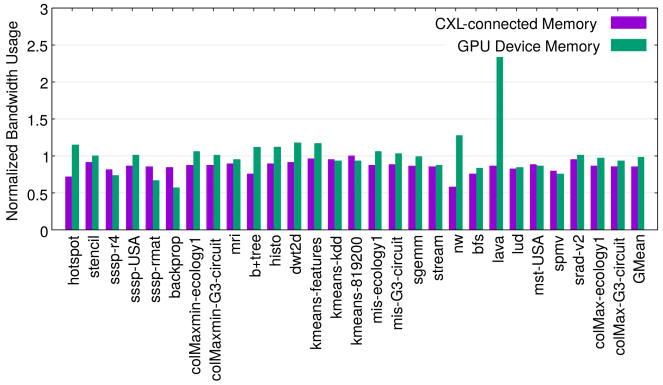

Fig. 12. Bandwidth Usage

3) Memory Bandwidth Utilization: Security metadata can contend with normal data on the memory bandwidth. The problem becomes more exacerbated with dynamic data relocation due to the increased security traffic. Our data-relocation-friendly security design aims to reduce this traffic as much as possible. Fig. 12 reflects the security bandwidth usage achieved with our security design normalized to that of the conventional security design. In our design, the utilization of the CXL bandwidth is 14.92% less than the conventional security design. Additionally, the security usage of the GPU device memory bandwidth is 2.05% less than that of the conventional design.

4) Sensitivity to CXL Bandwidth: The effectiveness of our proposed security design is studied using different CXL bandwidths, considering various bandwidth specifications of underlying memory devices or physical layers. The main evaluation is done using a bandwidth of $\frac{1}{16}$th of the GPU device memory bandwidth. Fig. 13 shows the performance results for different ratios of CXL bandwidth to the device memory bandwidth, while the device memory bandwidth is fixed as shown in Table I. Each of the values is normalized to the performance number of the same memory configuration with no security support. The figure indicates that Salus drives better performance even with higher memory bandwidth for

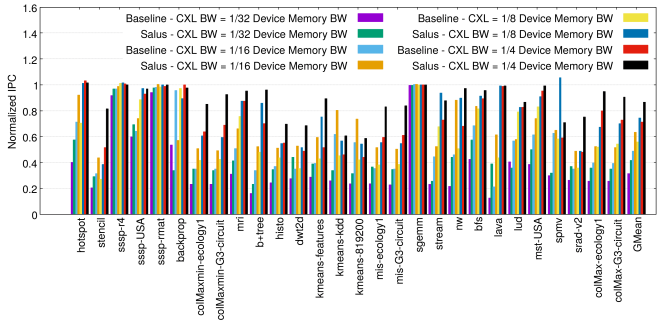

Fig. 13. Sensitivity to CXL Bandwidth

the CXL-connected memory. Salus achieves 32.79%, 29.94%, 32.90% and 21.76% performance improvement over the conventional security for CXL bandwidth of $\frac{1}{32}$th, $\frac{1}{16}$th, $\frac{1}{8}$th and $\frac{1}{4}$th of the device memory bandwidth, respectively.

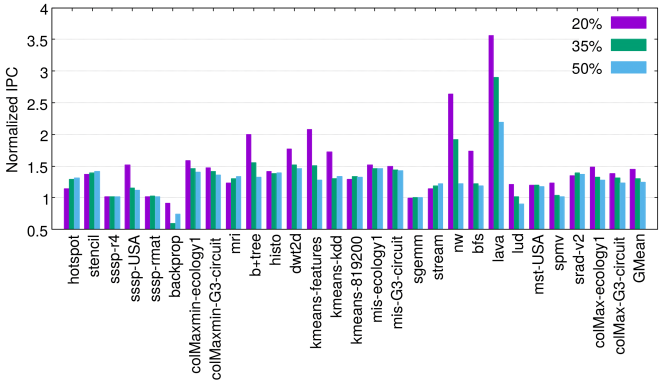

Fig. 14. Sensitivity to Device Memory Capacity Ratio to Footprint

5) Sensitivity to Application Footprint Percentage in GPU Device Memory: Fig. 14 shows the design sensitivity to the GPU device memory capacity ratio to the footprint. The more of the application's footprint that can fit in the high-bandwidth memory, the less frequent page movements we have. Consequently, less observed security traffic is due to transfers. Thus, the figure demonstrates how the improvements become more noticeable as less of the footprint can be contained in the GPU device memory. Specifically, our proposal achieves 51.64%, 34.48%, and 26.83% on average for device memory to the application footprint ratios of 20%, 35%, and 50%, respectively.

VI. Related Work

GPU Memory Security: Memory security is taking increased importance, especially with the offloading of applications to run remotely on clouds. In the context of GPUs, there are proposals for trusted execution environments [22], [23], [62], where the whole communication between the host and the GPU is confidential and integrity protected. Other works focus more on memory security against physical attacks. These works employ the same security measures used in conventional CPU systems but with adjustments and optimizations specialized for GPUs, such as common counters [51] work that optimizes encryption counters by grouping continuous data

that observe the same update pattern to use a single counter. PSSM [66] changes the organization of security metadata to be more friendly to the partitioned and sectored memory in GPUs. Plutus [2] proposes optimizations for the bandwidth usage due to metadata by using data similarity to perform MACless integrity verifications. However, all these works consider stable fixed data locations for the whole runtime, but with the advancement in the use of heterogeneous memory and the need for data migrations, there should be more optimized management for the security metadata.

CXL-connected Memory: The introduction of CXL has prompted researchers to explore various ways of integrating CXL-connected memory into the system's memory architecture, offering benefits such as expansion and pooling. For example, Pond [40] utilizes CXL to create memory pools, which serve as remote memory for computation nodes. By combining local memories with CXL-connected memory pools, the total cost of the system can be reduced without overprovisioning individual nodes with high-capacity memory. TPP [43], focuses on managing data movement between the main memory and CXL-connected memory. The goal is to optimize the placement of the hot set of data pages in the main memory, maximizing system utilization and performance. Other works [19] are also exploring the potential of CXL to enhance computing capabilities. These efforts aim to leverage CXL to drive advancements in large-scale computing systems. The majority of these works target CPU systems, whereas our work brings the power of CXL to the GPU world as well.

Heterogeneous Memory: The growing diversity of memory types and the need for efficient memory management have spurred research on heterogeneous memory systems. One area of focus has been DRAM caching, which has been extensively studied in various works. Initially, the cache management was performed at the cacheline granularity [55], similar to conventional CPU caches. However, this approach incurred high overheads for cache management and did not fully exploit spatial locality, as the temporal locality was already captured by higher-level caches [24], [25]. Subsequent works [24]–[26] explored caching at the page granularity, although it increased pressure on memory bandwidth. This approach showed improved performance by leveraging larger spatial locality and reducing tag overheads. With the increased capacity of DRAMs used as cache, using SRAM for tags became impractical. As a result, researchers proposed storing tags in DRAM itself, optimizing performance by colocating tags in the same row as the data [55]. Recent works [16], [35], [39] have replaced tags with changes to page table mappings that are aware of the DRAM cache locations. This approach relies on auxiliary tables to store mappings between the DRAM cache and the other memory.

VII. Conclusion

The increasing demand for memory resources in GPU applications, such as machine learning and scientific applications, has pushed the limits of traditional memory designs. As a result, new solutions have emerged, involving additional memories working alongside the local device memory, especially with the introduction of various memory technologies offering larger capacities but with limited speeds or bandwidth. The CXL protocol has played a significant role in advancing this domain, with ongoing research exploring its potential. However, the use of different memory types with distinct characteristics necessitates data movements, whether following caching procedures or custom algorithms. Security support in such systems poses significant challenges and can severely impact performance. To address this, our paper introduces a data-relocation-friendly security design, which reduces security traffic in the system and eliminates security operations related to data movements. Our design improves GPU performance by 29.94% compared to conventional security implementations used for traditional memory systems.

Acknowledgements

Part of this work was funded through Office of Naval Research (ONR) grants N00014-21-1-2809 and N00014-21-1-2811, NSF grant CCF-1908406, and an AMD gift fund. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

References

[1] (2021) Compute express link (cxl). [Online]. Available: https://www.computeexpresslink.org/

[2] R. Abdullah, H. Zhou, and A. Awad, "Plutus: Bandwidth-efficient memory security for gpus," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 543–555.

[3] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 354–365.

[4] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim, "Network-on-chip microarchitecture-based covert channel in gpus," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 565–577. [Online]. Available: https://doi.org/10.1145/3466752.3480093

[5] T. Alves, "Trustzone: Integrated hardware and software security," *White paper*, 2004.

[6] T. Baji, "Evolution of the gpu device widely used in ai and massive parallel processing," in *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*. IEEE, 2018, pp. 7–9.

[7] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-software support for efficient page migration in multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 596–609.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[10] F. Cabarcas, A. Rico, Y. Etsion, and A. Ramirez, "Interleaving granularity on high bandwidth memory architecture for cmps," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2010, pp. 250–257.

[11] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 79–92. [Online]. Available: https://doi.org/10.1145/3445814.3446713

[12] C.-H. Chang, A. Kumar, and A. Sivasubramaniam, "To move or not to move? page migration for irregular applications in over-subscribed gpu memory systems with dynamap," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456727.3463766

[13] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," 09 2013, pp. 185–195.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[15] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[16] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[17] J. Evans, M. Andersch, V. Sethi, G. Brito, and V. Mehta. (2022) Nvidia grace hopper superchip architecture in-depth. [Online]. Available: https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/

[18] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 451–461.

[19] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access, High-Performance memory disaggregation with DirectCXL," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 287–294. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gouk

[20] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 204, 2016.

[21] D. T. Harper and J. R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1440–1449, 1987.

[22] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud GPUs," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 817–833. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/hunt

[23] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," 04 2019, pp. 455–468.

[24] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 25–37.

[25] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 404–415, jun 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485957

[26] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[27] W. Jin, W. Jang, H. Park, J. Lee, S. Kim, and J. W. Lee, "Dram translation layer: Software-transparent dram power savings for disaggregated memory," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589051

[28] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 45–51. [Online]. Available: https://doi.org/10.1145/3538643.3539745

[29] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 45–51. [Online]. Available: https://doi.org/10.1145/3538643.3539745

[30] E. Karimi, Z. H. Jiang, Y. Fei, and D. Kaeli, "A timing side-channel attack on a mobile gpu," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 67–74.

[31] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[32] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004, pp. 288–299.

[33] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "Coda: Enabling co-location of computation and data for multiple gpu systems," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, sep 2018. [Online]. Available: https://doi.org/10.1145/3232521

[34] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[35] Y. Kim, H. Kim, and W. J. Song, "Nomad: Enabling non-blocking os-managed dram cache via tag-data decoupling," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 193–205.

[36] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. [Online]. Available: http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[37] A. Labs. (2022) Leo cxl™ memory connectivity platform. [Online]. Available: https://www.asteralabs.com/products/cxl-memory-platform/leo-cxl-memory-connectivity-platform/

[38] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[39] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless dram cache," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 211–222.

[40] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[41] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics,"

Proc. VLDB Endow., vol. 9, no. 14, p. 1647–1658, oct 2016. [Online]. Available: https://doi.org/10.14778/3007328.3007331

[42] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 1092–1098.

[43] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[44] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[45] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy*, 1980, pp. 122–122.

[46] Microchip. (2022) Smart memory controllers. [Online]. Available: https://www.microchip.com/en-us/products/memory/smart-memory-controllers

[47] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: Numa-aware gpus," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 123–135.

[48] H. Muthukrishnan, D. Lustig, D. Nellans, and T. Wenisch, "Gps: A global publish-subscribe model for multi-gpu memory management," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 46–58. [Online]. Available: https://doi.org/10.1145/3466752.3480088

[49] H. Muthukrishnan, D. Lustig, O. Villa, T. Wenisch, and D. Nellans, "Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 516–529.

[50] H. Muthukrishnan, D. Nellans, D. Lustig, J. A. Fessler, and T. F. Wenisch, "Efficient multi-gpu shared memory via automatic optimization of fine-grained transfers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 139–152.

[51] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure gpu memory," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 1–13.

[52] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Side channel attacks on gpus," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 4, pp. 1950–1961, 2021.

[53] Nvidia. (2016) Pascal mmu format changes. [Online]. Available: https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf

[54] NVIDIA. (2021) Nvidia a100 80gb pcie gpu - product brief. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf

[55] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 235–246.

[56] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 74–83. [Online]. Available: https://doi.org/10.1145/115952.115961

[57] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 183–196.

[58] K. Sethi. (2022) Expanding the limits of memory bandwidth and density: Samsung's cxl memory expander. [Online]. Available: https://semiconductor.samsung.com/newsroom/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/

[59] D. D. Sharma, "Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022, pp. 5–12.

[60] M. Showerman, J. Enos, C. Steffen, S. Treichler, W. Gropp, and W.-m. W. Hwu, "Ecog: A power-efficient gpu cluster architecture for scientific computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 83–87, 2011.

[61] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[62] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/volos

[63] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, "An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[64] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 179–190.

[65] S. Yuan, A. Awad, A. W. B. Yudha, Y. Solihin, and H. Zhou, "Adaptive security support for heterogeneous memory on gpus."

[66] S. Yuan, Y. Solihin, and H. Zhou, "Pssm: Achieving secure memory for gpus with partitioned and sectored security metadata," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 139–151. [Online]. Available: https://doi.org/10.1145/3447818.3460374

[67] S. Zhang, Y. Yang, L. Shen, and Z. Wang, "Efficient data communication between cpu and gpu through transparent partial-page migration," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 618–625.

APPENDIX

A. Abstract

We evaluate Salus using GPGPU-sim v4.0 as described in section V. In this artifact, we provide the extended version of the simulator with Salus implementation along with required instructions to build and run experiments to reproduce the main performance results shown in Fig.10 - Fig. 12.
The artifact can be executed on any Linux distribution with 20GB disk space, though it is tested in Ubuntu 18.04 environment. It is recommended to run the artifact on a machine with a high number of cores (e.g. 64) with 40GB main memory at least for such a number of cores.

B. Artifact check-list (meta-information)

- **Program:** Extended GPGPU-SIM v4.0
- **Compilation:** GCC/G++ 7.5.0, python3
- **Run-time environment:** Ubuntu 18.04 or any other version
- **Metrics:** IPC, Memory Traffic
- **Output:** Files with statistics
- **Experiments:** Launched using the provided scripts
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes
- **How much time is needed to complete experiments (approximately)?:** Depends on the number of cores, for 64 cores, it takes one day and a half for all experiments
- **Publicly available?:** No

C. Description

We provide both a virtual machine with all dependencies installed and a zipped folder to be downloaded directly on the machine.

1) How to access: The two different options can be downloaded from this link https://drive.google.com/drive/folders/15UfYMRVOGaauKyf7U5YR31ribBeI4mIK?usp=drive_link

2) Hardware dependencies: For the virtual machine option, it is highly recommended to have many cores accessible for the VM because there are many workloads to run. It is recommended to allocate 40GB of memory at least for the VM along with 40GB of main memory. On the other hand, if the artifact is downloaded directly on the machine, at least 20GB of disk space and 40GB of main memory are needed.

3) Software dependencies: If the artifact is downloaded directly on the machine, then a Linux system is required (tested on Ubuntu 18.04). The following software dependencies are required, which are listed on the github page of the simulator:

- CUDA Toolkit (we used 11.8.0)
- GCC/G++
- make and makedepend
- xutils
- bison
- flex
- zlib
- python-pip

D. Installation

1) First, download Salus artifact from the provided link.
2) Run the following commands to install the required dependencies on Ubuntu 18.04, skip this step if you are using the VM option:

```
$ sudo apt-get install build-essential
      xutils-dev bison zlib1g-dev flex
      libglu1-mesa-dev vim
$ wget https://developer.download.nvidia
      .com/compute/cuda/
      11.8.0/local_installers/
      cuda_11.8.0_520.61.05_linux.run
$ sudo sh cuda_11.8.0_520.61.05_linux.run
$ sudo apt install python-setuptools
      python-dev build-essential
      python-pip
$ pip3 install --ignore-installed PyYAML
$ pip3 install psutil
$ pip install numpy
```

3) Setup the GPGPU-sim environment and compile the simulator by executing the following commands:

```
$ cd Salus_Artifact/gpgpu-sim_distribution/
$ source initialize_script.bashrc
$ source setup_environment
$ make
```

4) Compile the benchmarks and fetch the needed data to run them by running the following commands, skip the last line for the VM option:

```
$ cd ../gpu-app-collection/
$ source ./src/setup_environment
$ make all -i -j -C ./src
```

E. Experiment workflow

To run the experiments needed to generate the results shown in Fig.10-Fig.12, the job spawning manager of Accel-Sim is utilized, we created a script to launch the used workloads with the correct configurations, to run jobs concurrently, please make sure to have large number of cores on your machine, as the process manager by default launch jobs equal to the number of the cores in the system. To run the script, execute the following:

1) Move to the directory of the script:

```
$ cd ../util/job_launching
```

2) To run the workloads with no security at all:

```
$ ./running_script.sh N
```

To be able to extract the results, the numbers of the simulations have to be remembered, to do so, run the following, and write down the first and last jobs numbers:

```
$ ./job_status
```

3) To run the workloads with baseline security, run the following command and save the simulation numbers:

```
$ ./running_script.sh B
```

4) To run the workloads with Salus, run this command and do as above:

```
$ ./running_script.sh S
```

F. Evaluation and expected results

After running the experiments needed, the running time depends on the hardware resources available on the machine. It takes around one day to two to finish all experiments on a 64-core machine.

Once the simulations have finished execution, the output files generated by GPGPU-Sim are stored in a folder (automatically created if not existent) named **sim_run_[cuda_version]** in the main directory **Salus_Artifact**. To extract the needed values from the output files:

- Start by copying the collection scripts from the folder directly to the results folder:

```
$ cd ../../scripts
$ cp ./* ../sim_run_11.8/
```

- Copy to the start of each script the number of the simulations, in this format:

```
# No Security
simNums = range([first_job_no_securtiy],
          [last_job_no_security] + 1)
# Baseline Security
simNums = range([first_job_baseline],
          [last_job_baseline] + 1)
# Salus
simNums = range([first_job_salus],
          [last_job_salus] + 1)
```

For regenerating the results of Fig.10:

1) To get the IPC results of the simulations, run the **salus_results_collection.py** script once for each of the configurations, by commenting the simNums lines for the other configurations, do the following for each of the configurations:

```
$ cd ../sim_run_11_8
$ python salue_results_collection.py >
  some_output_file.txt
```

2) Fig.V-B1 shows these results by taking normalizing the IPCs of Baseline and Salus to the IPCs of No security.

For regenerating the results of Fig.11:

1) To get the security memory traffic (number of memory requests), run the **security_traffic_collection.py** script once for both Baseline and Salus configuration, by commenting the simNums lines for the other configurations, do the following for each of the configurations:

```
$ cd ../sim_run_11_8
$ python security_traffic_collection.py >
  some_output_file.txt
```

2) Fig.11 shows these results by taking normalizing the number of Salus traffic values to those of the Baseline.

For regenerating the results of Fig.12:

1) To get the bandwidth utilization of both CXL and HBM memories, run the **bw_util_collection.py** script once for both Baseline and Salus configurations, by commenting the simNums lines for the other configurations, do the following for each of the configurations:

```
$ cd ../sim_run_11_8
$ python bw_util_collection.py >
  some_output_file.txt
```

2) Fig.12 shows these results by taking normalizing the number of Salus traffic values to those of the Baseline.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html