Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance

Trinayan Baruah[1] Yifan Sun[1,2] Saiful A. Mojumder[3] José L. Abellán[4] Yash Ukidave[5] Ajay Joshi[3]
Norman Rubin[1] John Kim[6] David Kaeli[1]

{tbaruah, kaeli}@ece.neu.edu, ysun25@wm.edu, {msam, joshi}@bu.edu, yash.ukidave@mlp.com,
nrubin3@gmail.com, jlabellan@ucam.edu, jjk12@kaist.edu

[1]Northeastern University, [2]William & Mary, [3]Boston University, [4]Universidad Católica San Antonio de Murcia,
[5]Millennium USA, [6]KAIST

ABSTRACT

Programming on a GPU has been made considerably easier with the introduction of Virtual Memory features, which support common pointer-based semantics between the CPU and the GPU. However, supporting virtual memory on a GPU comes with some additional costs and overhead, with the largest being from the support for address translation. The fact that a massive number of threads run concurrently on a GPU means that the translation lookaside buffers (TLBs) are oversubscribed most of the time. Our investigation into a diverse set of GPU workloads shows that TLB misses can be extremely high (up to 99%), which inevitably leads to significant performance degradation due to long-latency page-table walks. Our profiling of TLB-sensitive workloads reveals a high degree of page sharing across the different cores of a GPU. In many applications, a page can be accessed in temporal proximity by multiple cores, following similar memory access patterns. To support the inherent sharing present in GPU workloads, we propose *Valkyrie*, an integrated cooperative TLB prefetching mechanism and an inter L1-TLB probing scheme that can efficiently reduce TLB bottlenecks in GPUs. Our evaluation using a diverse set of GPU workloads reveals that Valkyrie is able to achieve an average speedup of 1.95×, while adding modest hardware overhead.

CCS CONCEPTS

• **Computing methodologies** → **Graphics processors**; • **Software and its engineering** → **Virtual memory**.

KEYWORDS

GPU; TLB Design; Virtual Memory; TLB Prefetching; TLB Probing

ACM Reference Format:
Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, David Kaeli. 2020. Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414639

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414639

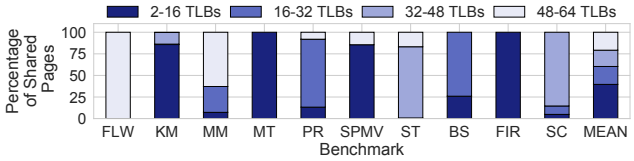

Figure 1: Distribution of page sharing behavior across different L1-TLBs, in an example GPU with 64 L1-TLBs. X-Y TLBs means a range between X to Y L1 TLBs share pages.

1 INTRODUCTION

Modern-day large scale applications, including deep learning [23], molecular dynamics [20], genome sequencing [30] and cryptocurrency mining [47], have a high degree of data-level parallelism. GPUs have become the preferred choice to accelerate these applications. This widespread adoption of GPUs has been enabled by the GPU hardware vendors, such as AMD and NVIDIA, by lowering the barrier to entry. Today's GPU platforms are equipped with optimizing compilers, efficient runtimes and advanced driver support. One key feature that has been recently introduced by GPU vendors is Unified Memory (UM) [34] in CUDA terminology or Shared Virtual Memory (SVM) [5] in OpenCL terminology. UM and SVM lower the effort required by programmmers to program a single GPU, fused CPU-GPU systems (e.g., APUs [3]), as well as multi-GPU systems. They eliminate the need to perform explicit memory copies, as the GPU driver and the runtime handle all page transfers to/from the GPU. They lower programmer burden by managing CPU-to-GPU and GPU-to-GPU data transfers [1, 11] and support oversubscription of memory [29, 34]. To support all these capabilities, GPU vendors have added virtual memory support, providing the required hardware and software. On the hardware side, this support includes efficient input-output memory management units (IOMMUs), multi-threaded hardware page table walkers (PTWs), and multi-level translation lookaside buffers (TLBs). In terms of software support, the features include Unified Memory (UM) API calls, which are directly under the control of the programmer, as well as support from the GPU driver and runtime for handling tasks

such as page migration, TLB shootdowns, and page swapping [34]. The set of virtual memory features that have been predominantly available on CPUs have been extended to support GPUs [37, 38].

Unfortunately, enabling Virtual Memory support on GPUs introduces significant performance overhead. Features such as demand paging and page eviction create performance bottlenecks [11, 18, 29]. Address translation on GPUs is a major bottleneck, given the limited size of the private L1-TLBs, which in turn, generates severe pressure on the shared L2-TLB. For example, cryptocurrency miners have reported that they suffer significant performance degradation due to TLB misses on GPUs [13]. To solve this problem, improving address translation efficiency has been a focus of recent work [9, 52].

To understand the impact of these problems, we conducted a set of experiments that capture the L1-TLB miss rates of a range of GPU applications. In our experiments (details in Section 4), we observe that GPU applications that possess poor temporal locality in terms of page accesses can have L1-TLB miss rates that are as high as 99% and L2 TLB miss rates that are as high as 93%. The limited bandwidth of the shared L2-TLB [52] also renders it ineffective, because GPU applications can have thousands of memory transactions in flight and if those requests miss in the L1-TLB, they would eventually have to be serviced by the L2-TLB. A naive way of combating this problem is to simply increase the size and port count of the L1 and L2 TLBs. However, such solutions increase the size and port count of SRAM-based arrays, inevitably introducing area and power overhead [44]. Prior work on GPU TLB design enhancements have mostly focused on the shared L2-TLB [9] or on page-coalescing mechanisms [8]. Baseline designs for L1-TLBs on a GPU have been established in prior work [37, 38]. However, there has not been much work on designing and architecting the L1-TLBs to leverage the unique characteristics of GPU applications, which in turn has the potential to increase application performance. In this work, we make the following three key observations about the L1-TLB behavior of GPU applications, motivating our efforts to improve the L1-TLB design:

1.) The first observation (as shown in Figure 1) is that there is a high degree of page sharing across the cores in a GPU. The sharing behavior can range from applications where pages are being shared by only a few GPU cores, to applications where pages are shared by all the cores on the GPU.

2.) The second key observation is that the TLB translation of many different pages across cores follow similar patterns. This means if we can track which cores share similar page table entries, we can potentially prefetch these entries ahead of time to increase the L1-TLB hit rate, in turn improving overall system performance. As shown in Figure 2, if we can identify that the same set of TLBs shares pages A, B, C and D, we can use this information to prefetch entries into the sharing TLBs, after one of the sharers accesses pages E, F, and G.

3) Finally, we observe (as shown in Figure 3) that in many instances, whenever there is an L1-TLB miss, the same page table entry can also be found in another L1-TLB on the GPU (about 60% of the time). On average, about 33% of these misses can also be found in the same Shader Engine (the Shader Engine is described in Section 2). If we can design a low-cost network for inter-TLB communication and develop an efficient mechanism to leverage a portion of this inter-L1-TLB locality, we can potentially retrieve a

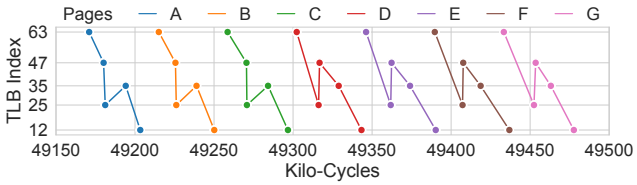

Figure 2: Accesses to multiple pages by the same set of cores (represented by TLB index on the y-axis) during the execution of KMeans benchmark. A, B, C, D, etc. represent different unique pages.

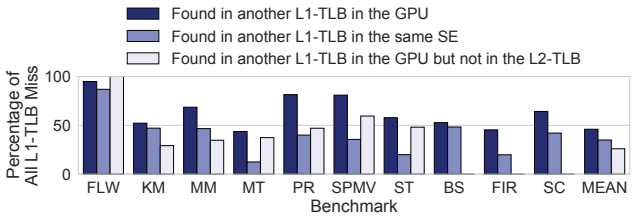

Figure 3: Percentage of time an L1-TLB miss entry was found in another L1-TLB.

translation from a neighboring L1-TLB without having to go the L2-TLB or to the MMU. In Section 2 we demonstrate how we can reduce significantly the number of these misses by accessing these translations in neighboring TLBs.

Based on these observations, we propose *Valkyrie*, an integrated solution consisting of two mechanisms. Our first mechanism is the design of a hardware prefetcher which takes advantage of inter-core locality and prefetches TLB entries based on runtime detection of dynamic page table entry sharing behavior. Our second mechanism performs L1-TLB probing, taking further advantage of this locality behavior by enabling neighboring L1-TLBs to communicate with each other for TLB miss resolution. The ultimate goal of *Valkyrie* is to improve the performance of GPU applications that are highly sensitive to TLB misses.

The main contributions of this work include:

- We perform an in-depth characterization of the inter-L1-TLB page sharing behavior across a diverse set of GPU applications. Our analysis reveals new insights (discussed earlier) into the TLB sharing behavior of GPU applications.
- We propose and design *Valkyrie*, a programmer-transparent hardware solution to leverage inter-L1-TLB locality and improve the performance of TLB-sensitive applications by means of two mechanisms ***Prefetching*** and ***Probing***. To support prefetching, we use a Locality Detection Table and a runtime feedback mechanism to determine the L1-TLBs to which we prefetch the translation information. To implement Probing, we co-design the interconnect within a set of neighboring L1-TLBs using a bidirectional ring NoC, leveraging inter-core locality. This allows L1-TLB misses to resolved by neighboring L1-TLBs.
- We implement and evaluate *Valkyrie* in MGPUSim [45]. Our evaluation shows that Valkyrie can improve the overall GPU system

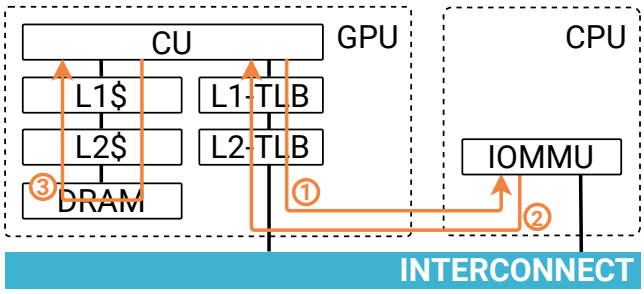

Figure 4: The address translation process on a GPU.

performance by up to 1.95× on an average when compared to a system without TLB prefetching and inter-TLB probing mechanism, with modest hardware overhead.

2 BACKGROUND & MOTIVATION

In this section we review the general mechanics of address translation on a GPU, describe the performance bottlenecks associated with GPU address translation, and discuss how we can exploit application-level characteristics to alleviate these bottlenecks.

2.1 GPU Architecture

We will be using AMD GPU terminology in the following description of GPU architecture. A GPU uses multiple cores (i.e., Compute Units or CUs) to process data. A CU can execute a large number of GPU threads (i.e., work-items) in parallel. In a CU, a wavefront of 64 work-items always executes the same instruction. CUs are grouped according to the hardware resources they share. In a GCN-based GPU such as the Radeon VII [6], a set of four CUs form a Shader Array (SA). Four SAs form a Shader Engine (SE), with 16 CUs sharing the graphics-related components such as the Geometry Engine. Every CU is connected to its own private L1-vector cache. Each SA has one L1-scalar cache (a read-only cache that is mainly designed for caching addresses and constants) and one L1-instruction cache. The multi-banked L2 cache is shared by all the CUs in a GPU and is multi-ported to achieve high throughput. The L2 caches are connected to the main memory controllers, which typically use high bandwidth solutions (e.g., HBM, GDDR) to best meet the demands of memory-hungry GPU applications. A group of SEs, when combined, form the whole GPU. NVIDIA-based GPUs, such as the Turing architecture [35], follow a similar hierarchical design where groups of Streaming Multiprocessors (SMs) are connected to form Texture Processing Clusters (TPC), and six of such TPCs are combined to form a Graphics Processing Cluster (GPC).

2.2 Virtual address translation in GPUs

Modern GPUs use virtual addresses in their CUs. Virtual addressing abstracts away the physical location of the data, enabling many useful features. For example, virtual addressing enables Unified Memory (UM), which relieves the programmer's burden of performing explicit memory management. The GPU hardware and driver can effectively move data from device to device by changing the virtual to physical address mapping. Virtual addressing also enables address-space isolation for running concurrent applications on the GPU [9].

Virtual addresses need to be translated to physical addresses before accessing data in the GPU L1-cache. Modern GPUs provide dedicated hardware for address translation, which includes multi-level TLBs, multi-threaded page table walkers (PTWs), and memory-management unit caches [37, 38]. This support avoids the overhead of depending on the CPU to perform address translation, which can significantly impact performance [38].

Similar to the cache hierarchy, the TLB hierarchy on a GPU consists of multiple levels [41]. Each GPU core or compute unit (CU) is equipped with a private L1-TLB that is typically fully associative to eliminate conflict misses [9, 41]. The L1-TLBs are typically backed by a larger L2-TLB, which is shared between all the available CUs in the GPU and is usually multi-ported to allow for concurrent lookups [9]. The TLBs typically include Miss Status Holding Registers (MSHRs), so that an L1-TLB can handle other requests while fetching translations from the L2-TLB [37]. Multi-level TLBs also tend to be organized in a *mostly-inclusive* hierarchy where the translation may or may not be necessarily present in both the L1-TLB and L2-TLB. Mostly-inclusive TLBs are both easier to implement and do not suffer from the overheads of back invalidations when the L1 to L2-TLB bandwidth becomes a bottleneck [14, 51].

A miss in both the L1 and L2-TLBs triggers a page table walk, which is handled by the hardware page table walkers [38]. Page table walkers traverse entries in the entire page table to search for the virtual address, incurring high latency. To deal with the high number of concurrent requests over a short time window, these page table walkers are typically multi-threaded [37, 38].

Combining these elements, the address translation process of a GPU is shown in Figure 4 and follows a set of steps. The process starts with the execution of a memory instruction by a CU, triggering a memory request to an L1 cache. ①: On a translation request from the CU, that misses in both the L1-TLB and L2-TLB, the request is forwarded to the page walk buffer on the IOMMU which is located on the CPU die [41, 50]. Once a hardware page table walker is available, this request is picked up by the walker to perform a page table walk. ②: Once the page table walk completes and a valid translation is found, the request is returned to the L2-TLB and the L1-TLB. Both levels store the translation. ③: Finally, the L1-TLB returns the virtual-to-physical translation response to the L1 cache. Then, the L1 cache can issue the memory access with the physical address to its local storage and, if there is a cache miss, to the rest of the memory hierarchy.

GPU applications can have thousands of in-flight threads (e.g., 163,840 maximum concurrent threads in an AMD Radeon VII GPU [6]) running concurrently. A large number of threads generate a large number of concurrent memory requests, causing severe pressure on the caches and the TLBs. From Figure 5, we observe that many workloads experience extremely high TLB miss rates in the private L1-TLBs (as high as 99%) and the shared L2-TLB (as high as 93%). Even though GPUs are more latency tolerant as compared to CPUs, having such a high TLB miss rate, which arises both in part due to the massive amount of threads running on the GPU as well as the streaming behavior of the workloads, inevitably, leads to performance degradation as a significant number of wavefronts can stall on serving TLB misses [9]. To avoid the performance penalty

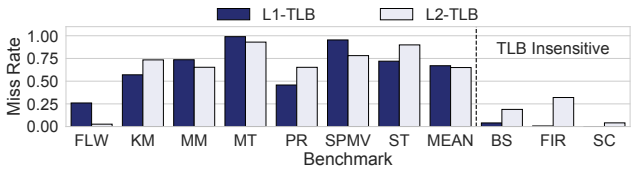

Figure 5: L1-TLB and L2-TLB miss rates across GPU applications.

associated with TLB misses that trigger long-latency page walks, we must reduce the TLB miss rate in GPUs. While the actual latency of servicing a miss from the L2-TLB is only about 10 cycles [9], the queuing latency which arises due to the large number of threads stalled waiting on L1-TLB misses to be serviced from the L2-TLB means that the actual latency can be much longer.

Another challenge with current GPU TLB designs is that the L2-TLBs experience high pressure due to the large number of concurrent threads running on the GPU, but have limited bandwidth. Unlike caches, banking the GPU's TLBs does not offer much benefit [52]. Adding more ports to the shared L2-TLB also leads to significant area and power costs.

2.3 L1-TLB Sharing Behavior

The parallel nature of some GPU applications causes threads running on different cores to access the same page or even the same cache block [21]. Sharing in applications is common due to threads running on different GPU cores accessing the same common data structure, such as matrices and arrays. Three interesting facts can be distilled from this observation of the L1-TLB sharing behavior:

First, as we can observe from Figure 1, the amount of page sharing can vary from application to application. For example in the Matrix Transpose benchmark (MT), 100% of the pages are shared by 2-16 L1-TLBs. For Page Rank (PR), on the other hand, 78% of the pages are accessed by 16-32 L1-TLBs. Thus the number of sharer L1-TLBs can vary from application to application.

Second, the shared pages are accessed in temporal proximity to each other. Figure 2 shows the accesses to multiple pages (A, B, C, D, etc.) from the same set of TLBs (12, 25, 35, 47 and 63) at different points in time for the KMeans benchmark. If we look at these accesses to the different pages, we can see the pattern that the same set of TLBs are making access to different pages throughout the execution of the application. Taking the example of this scenario, originally translations for pages A, B and C were brought into the shared L2-TLB on an L1-TLB miss from TLB 63. When TLBs 47, 25, 35, and 12 attempts to load this translation, they have a probability of getting the translation information from the shared L2-TLB (assuming it has not been evicted yet). However, since the L2-TLB is shared by all the L1-TLB's in the GPU, the limited bandwidth of the L2-TLB can still impact the speed of the L1-TLB to L2-TLB accesses. If we know beforehand that TLBs 63, 47, 25, 35, and 12 share a large number of pages, then on an L1-TLB miss from TLB 63, the translation can be potentially prefetched into L1-TLBs 47, 25, 35 and 12 ahead of time. The accesses to shared pages can then easily turn into L1-TLB hits for TLBs 47, 25, 35, and 12, reducing translation latency and potentially improving

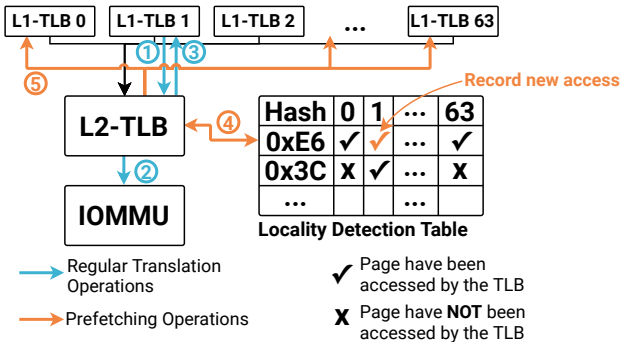

Figure 6: The mechanism of Prefetching. ① Translation arrives at L2-TLB. ② Translation misses in L2-TLB requires a page table walk at the IOMMU. ③ Translation response returned to the requester. ④ Concurrently with ③, the L2-TLB updates the Locality Detection Table and fetches the list of L1-TLBs that may also potentially need this page. ⑤, The page entry is then sent to the L1-TLBs.

overall performance. The key challenge is to track the TLB entry-sharing behavior across the entire GPU and then issue prefetches accordingly to maximize performance.

Third, based on our observation that GPU cores access a similar set of pages, some of the L1-TLB misses on a core can potentially be serviced by another core's L1-TLB (see Figure 3). Using the KMeans benchmark from Figure 3 as an example, 52% of the misses can be potentially served by another L1-TLB on the same GPU if there was a mechanism to communicate between L1-TLBs. If this form of inter-TLB communication was restricted to TLBs on a single SE, it will still allow 47% of such misses to be resolved. The mostly-inclusive hierarchical design for multi-level TLBs that are commonly used [14, 51] due to their simplicity leads to another interesting observation. On average across the workloads we studied, 26% of the L1-TLB misses that could potentially to be resolved by another L1-TLB were not even found in the shared L2-TLB. This means that enabling some form of inter L1-TLB communication will help to resolve bottlenecks arising due to both the limited bandwidth and limited capacity of the L2-TLB.

3 VALKYRIE SYSTEM DESIGN

Valkyrie provides a hardware solution to improve the performance of TLB-sensitive GPU applications. *Valkyrie* uses two mechanisms: **Prefetching:** Here, we introduce a Locality Detection Table (LDT) that tracks runtime inter-TLB locality information. A prefetcher that resides in the L2-TLB then uses this information from the LDT to alleviate TLB bottlenecks by prefetching pages to the L1-TLBs. **Probing:** With L1-TLB Probing, we integrate a lightweight intra-SE on-chip ring that interconnects the L1-TLBs, enabling neighboring L1-TLBs on the ring to service L1-TLB misses from each other.

3.1 Prefetching Mechanism

We design *Valkyrie's* prefetching mechanism to exploit commonly observed behavior in GPU applications, where multiple L1-TLBs access the same page-table entries in temporal proximity (as observed

in Figure 2). The first challenge in designing the prefetching mechanism is to dynamically determine if there are any other L1-TLBs that are likely to share a particular page-table entry. This calls for a table that can keep a record of such page-sharing behavior. To that end, we introduce the Locality Detection Table (LDT). The LDT is placed alongside the L2-TLB to keep track of the L1-TLBs that tend to share the same page table entries. The LDT is a finite-sized (100 entries in our implementation), fully associative table, as shown in Figure 6.[1].

For a 64-bit virtual address using 4KB pages, the virtual page number (VPN) has 52 bits. For every memory access, in case of an L1-TLB miss, we send a request to the L2-TLB. The L2-TLB processes the request as usual and the translation information is sent to the requesting L1-TLB. In parallel to the regular address translation process, we hash the 52-bit VPN to generate an 18-bit tag. The tag bits are then compared with the tags of each row in the fully-associative LDT to determine if there is a match. In the case of a tag match, we read the corresponding entry in the LDT. Each entry in the LDT is a 64-bit mask (1 bit each for the 64 L1-TLBs in our evaluated system). We set the bit corresponding to the L1-TLB that currently has a miss to 1. A value of 1 in any other bit position indicates that the corresponding page has been accessed by one or more other L1-TLBs. In the case of an LDT miss, a new entry corresponding to the current memory access is added to the LDT. If the LDT is full, the oldest entry is evicted and the new one inserted (i.e., FIFO replacement). Evicting old tags encourages the prefetcher to forget old page access patterns and learn new ones.

We chose 18 as the number of tag bits for the LDT. This value is large enough to minimize collisions over a short execution duration. Although the total number of possible hashed values is high (an 18-bit hash can have up to 2^{18} entries), we only keep 100 entries in the LDT to limit the area and power overhead of the LDT. The insertions and fetches from the LDT are not on the critical path and so we do not take a performance hit. Using a fully-associative structure allows us to make updates and retrieve elements from the table in a single cycle.

Next, the L2-TLB needs to use the information from the LDT to decide if we should prefetch translation information into any of the L1-TLBs. Ideally, the L2-TLB should send the page entry to all the L1-TLBs marked in the LDT which have been identified to have used that page before. However, sending too many prefetched TLB entries can evict useful TLB entries and waste energy. While the applications we evaluated have some degree of L1-TLB page sharing behavior (see Figure 1), not all of them will benefit from prefetching since some of these applications do not suffer from high L1-TLB miss rates (see Figure 5). Therefore, it is important to use some form of feedback mechanism at runtime to throttle the number of L1-TLBs to whom the prefetched translation is sent to at runtime. Thus, we design a simple hysteresis-style mechanism to throttle and tune the number of "partner" L1-TLBs (NPT) at runtime. When the number of sharing L1-TLBs suggested by the LDT is greater than the NPT value, we randomly select a total of NPT L1-TLBs and prefetch the translation entries only to them.

[1]Typical L1-TLBs are 32-128 fully associative structures. So, our dedicated LDT structure for the entire GPU with 100 entries and with fewer bits per entry can be easily realizable.

To determine the maximum number of L1-TLBs that can receive a prefetched page entry (as represented by NPT), we use a confidence-based parameter tuning mechanism. Every 100K cycles (i.e., an epoch), the parameter tuner either increments or decrements the NPT value by 4. Before making the decision, the parameter tuner collects the total number of L1-TLB accesses and the number of L1-TLB hits from all the L1-TLBs. This process reuses the L1-TLB to L2-TLB network and splits the address bus bits for both total access counts and hit counts. If the L1-TLB hit rate is increasing, we increase the confidence level (from 0 to 3, saturating at level 3) and repeat the action taken in the previous epoch. If the L1-TLB hit rate is decreasing, the confidence level is reduced. If the L1-TLB hit rate continues to decrease when the confidence level is 0, the parameter tuner will start to adjust the NPT value in the opposite direction. Adopting this throttling mechanism, the NPT value can dynamically adapt to the characteristics of the running application.

3.2 Probing Mechanism

As we discussed in Section 2 and demonstrated in Figure 1, there is significant opportunity to fetch translations from neighboring L1-TLBs. To leverage inter-L1-TLB locality, we design a mechanism for inter-L1-TLB communication called *Probing* that operates over a lightweight intra-SE on-chip ring network that interconnects the 16 L1-TLBs that are part of the SE.

Address Translation Information Search Scope: First, we need to decide how many other L1-TLBs should be probed on an L1-TLB miss. Figure 3 shows the percentage of time a CU found the address translation information in another L1-TLB in the same SE. On average, ~35% of the L1-TLB misses can be served from another L1-TLB within the same SE. Hence, we restrict the scope of searching for address translation information to L1-TLBs located within the same SE. The Compute Units (CUs) in an SE are physically placed close to each other, so we can probe the L1-TLBs within an SE with a low-overhead interconnect solution. Probing L1-TLBs from other SEs requires long wires and complex switches, which is expensive in terms of both area and the energy. Moreover, searching in each L1-TLB on the entire GPU may generate a lot of redundant L1-TLB look-ups.

Interconnect Network Between L1-TLBs: To connect the L1-TLBs within the same SE, we propose using a bidirectional ring network, as shown in Figure 7. A bidirectional ring is well-suited for this purpose as it allows for efficient communication across neighboring L1-TLBs without high complexity. Another motivation for choosing a ring is that the number of CUs in an SE is typically low. The two most recently introduced GPUs on the HPC market (i.e., the NVIDIA TU102 GPU [35] and the Radeon VII [6]) have a total of 12 streaming multiprocessors (SMs) and 16 CUs in each SE, respectively. It is unlikely that the number of SMs or CUs per SE will increase dramatically in the future. The ring is 64-bits wide in each direction and it is wide enough to accommodate communications that only require single-flit packets. Since the interconnected L1-TLBs can receive requests from each other, a 16-entry buffer is sufficient. The buffers are placed at each L1-TLB to store the incoming requests from neighboring L1-TLBs.

Search Mechanism: Given that we do not have *apriori* knowledge of which L1-TLBs in an SE can provide the translation, it is

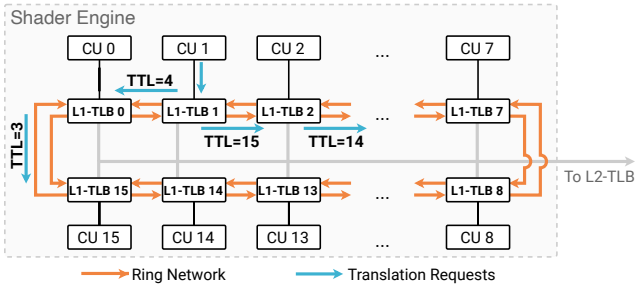

Figure 7: Block diagram of the intra-SE on-chip network (bi-directional ring topology) that interconnects the L1 TLBs together. An example of handling a miss in CU-1's L1-TLB is shown by the blue arrows. The translation request misses in L1-TLB1 which results in requests being forward in both directions of the ring with different Time To Live (TTL) values. On the counterclockwise ring, the L1-TLB1 probes at most 4 TLBs (TTL=4), while for the clockwise ring, the L1-TLB1 probes at most 15 TLBs (TTL=15).

critical to have smart mechanisms to avoid the penalty of searching every L1-TLB in the SE. Mechanisms that detect frequently shared data between GPU L1-data caches have been proposed in the past [21]. Such mechanisms are useful when predicting cache-line sharing behavior since a typical cache line is smaller than a page and therefore the percentage of shared cache-lines between a pair of L1-caches will typically be lower than the percentage of shared pages between a pair of L1-TLBs. In our observations, the percentage of shared pages for the application can be as high as 99%, which means that a predictor is not helpful. Predicting whether a page is shared or private for applications that have such a high degree of page sharing does not provide any benefits. Also, prediction mechanisms have difficulty predicting whether the shared page is present at a particular instant in time on another L1-TLB. Therefore, to reduce the complexity and the chances of mispredictions that occur with a prediction-based scheme, we follow a different probing mechanism as outlined below.

In *Valkyrie*, in case of an L1-TLB miss, the L1-TLB sends out two requests concurrently in each of the rings (both clockwise and counterclockwise) with different *Time To Live* (TTL) values, added to the packet header. The first request, defined as the *primary request*, is sent with a TTL value of 15 for a 16-node ring. The scope of this request is to search the full ring (all 15 L1-TLBs) until there is either a hit or the TTL reaches 0 (the TTL value is decremented by 1 at each hop). The second request is sent in the opposite direction with a TTL value of 4 and is called the *secondary request*. We select a value of 4 because our experiments show that, on average, about 50% of the requests will hit in a neighboring L1-TLB that is less than 4 hops away in the ring. The secondary request can perform lookups in a maximum of 4 L1-TLBs.

If both the primary and secondary requests hit in any L1-TLB, a reply is sent through the ring network with the translation information. The request which returns the translation first is used and the one arriving later is simply discarded. If the secondary request which is sent with the TTL value of 4 encounters a miss after 4 hops, the request returns with a negative acknowledgment, and the translation request is forwarded to the L2-TLB. If the primary request cannot find a valid translation, the request is simply squashed since a translation miss has already been forwarded to the L2-TLB when the secondary request returned with a negative acknowledgment. One challenge with a ring network is the higher network hop count. However, by using this two-pronged communication mechanism, we minimize the wait time before sending a request to the L2-TLB.

Frequency of Inter-L1-TLB Communication: Another important design question is how to arbitrate between sending a request to the L2-TLB and sending a request to the other L1-TLBs. If all misses are forwarded to the other L1-TLBs, this can lead to poor utilization of the L2-TLB bandwidth and overload the inter-TLB communication network. We need a runtime mechanism to decide if the request should be sent to the L2-TLB or to the other L1-TLBs via the ring. Broadly speaking, we do not send out requests to neighboring L1-TLBs if the L2-TLB is highly effective in serving translation misses. There are two ways to measure this behavior. One mechanism is to measure the effectiveness at each L1-TLB controller by maintaining a running average of the latency of translation requests that have been sent to the L2-TLB and comparing it to a preset threshold. Another mechanism is to look at the buffer/link utilization of all the L1-to-L2 TLB links and infer this information. This second mechanism requires the L2-TLB to collect the buffer/link utilization information repeatedly, and broadcast it to the L1-TLBs since the L1-TLBs cannot infer the utilization of the other L1-to-L2 TLB links directly. This will, in turn, introduce additional network traffic. Therefore, we choose the first approach, as the running average latency can be calculated by each L1-TLB controller locally without having to rely on repeated communication with the L2-TLB. With the former method, we experimentally chose to use a threshold value of 150 cycles, since an L2-TLB hit will be serviced in less than 150 cycles (a page table walk takes at least 150 cycles in our configuration). If the L1-TLB controller detects that the latency is higher than a preset threshold, it will start using the ring to send out translation requests to neighboring L1-TLBs.

Request Arbitration: When using the probing mechanism, an L1-TLB receives requests from neighboring L1-TLBs and the local CU. We use a round-robin arbitration scheme in the L1-TLB to choose between serving requests from the local CU and requests from neighboring L1-TLBs (requests from either ring direction are treated equally). Round-robin arbitration is used to provide local fairness between the requests. Requests from neighboring L1-TLBs queue up in a buffer that can hold a maximum of 16 entries in our implementation. If the queue is full, and a request arrives, it is simply ignored and passed on to the next L1-TLB in the ring.

3.3 Combining Prefetching and Probing

While using either Locality-Based TLB Prefetching or L1-TLB Probing alone is beneficial to performance, combining these two mechanisms provide the maximum benefits. However, using both mechanisms at the same time can lead to destructive interference. Prefetching directly into the L1-TLBs has a negative impact on the probability of an L1-TLB miss of a neighbor being resolved by L1-TLBs. Essentially, although the entry evicted from an L1-TLB on a prefetch does not harm that L1-TLB itself, the evicted entry could be useful

Component	Configuration	Quantity
CU	1.0 GHz	64
L1 Vector Cache	16KB 4-way, 5-cycle latency, LRU	64
L1 Inst Cache	32KB 4-way, 5-cycle latency, LRU	1 per SA
L1 Scalar Cache	16KB 4-way, 5-cycle latency, LRU	1 per SA
L2 Cache	256KB 16-way, 8-cycle latency	8
DRAM	512MB HBM, 100-cycle latency	8
L1 TLB	1 set, 128-way, 1-cycle latency, LRU	64
L2 TLB	32 sets, 16-way, 10-cycle latency, 2 ports, LRU	1
IOMMU	8 Page Table Walkers, 150-cycle latency	-
Intra-GPU Network	Single-stage XBar	1

Table 1: GPU system configuration.

Abbv.	Application	Benchmark Suite	TLB Sensitive	Memory Footprint
KM	KMeans	Hetero-Mark	✓	66 MB
MT	Matrix Transpose	AMDAPPSDK	✓	72 MB
PR	PageRank	Hetero-Mark	✓	55 MB
MM	Matrix Multiplication	AMDAPPSDK	✓	32 MB
ST	Stencil 2D	SHOC	✓	42 MB
FLW	Floyd-Warshall	AMDAPPSDK	✓	72 MB
SPMV	Sparse Matrix Vector Multiply	SHOC	✓	50 MB
SC	Simple Convolution	AMDAPPSDK	×	40 MB
FIR	Finite Impulse Response Filter	Hetero-Mark	×	61 MB
BS	Bitonic Sort	AMDAPPSDK	×	30 MB

Table 2: Workloads used to evaluate Valkyrie.

for another neighbor L1-TLB. As our analysis suggests (see Figure 9), the potential hit rate in neighboring L1-TLBs degrades to 8% (the baseline has a neighbor potential hit rate of 52%), if we prefetch directly into the L1-TLBs.

One way to avoid this problem is to add extra victim buffers [49] and/or prefetch buffers [2, 15, 25], which will add to the area and power overheads of the GPU core. For a 128-entry L1-TLB, a size that is highly effective for GPUs [37], the most practical option is to simply reorganize the 128-entry L1-TLB, allocating 104 entries of the L1-TLB for handling translations and reserving 24 entries to serve as a separate prefetch buffer. We build on prefetch buffer design from the prior cache and TLB designs [15, 25], inserting the prefetched TLB entries into the prefetch buffer. The prefetch buffer also uses LRU-based replacement. When a prefetched TLB entry arrives, the buffer is updated using LRU. Upon an L1-TLB lookup, the L1-TLB and the prefetch buffer will be checked in parallel. If the translation information is found in the prefetch buffer, then the corresponding entry is moved from the prefetch buffer into the L1-TLB. An LRU replacement policy is used to decide which entry from the L1-TLB should be replaced. Splitting the entries of the L1-TLB between demand fetched translations and prefetches, we can increase the neighbor hit rate potential to 37% (see Figure 9) on an average, reclaiming much of the lost inter L1-TLB locality.

4 EVALUATION METHODOLOGY

We implement *Valkyrie* using the MGPUSim [45] simulator, which faithfully models the AMD GCN3 ISA and has been validated against GCN3 hardware.

Simulator: We have extended MGPUSim to model Valkyrie. We integrated the prefetcher into the TLB hierarchy and added the Locality Detection Table (LDT). For inter-L1-TLB probing mechanism, we modeled the ring-network that connects the different L1-TLBs. For the prefetching, searching the LDT entry takes 1 cycle. Each hop from one TLB to another takes 1 cycle [12]. Other execution latencies (e.g., the network delay incurred during inter-L1-TLB communication, prefetching from the L2-TLB level to the L1-TLB and the queuing latency) are fully accounted in our simulation model.

GPU System: We model and evaluate an AMD R9 Nano GPU (the parameters are listed in Table 1). In Section 5, this will be our baseline system. The GPU consists of 4 Shader Engines (SE), where each SE has 16 Compute Units (CUs), resulting in a total of 64 CUs in the GPU. On AMD GPUs, threads belonging to the same wavefront (64 threads) execute in a lockstep fashion. Each CU has 4 SIMD units and each SIMD unit can have up to 10 wavefronts, resulting in a total of 40 wavefronts (maximum) that can be run on a CU. Therefore, at any given point in time, the maximum number of threads in-flight on a CU is 2560 threads. The actual number of threads that can run at a given time depends on adequate resources being available, such as registers and shared memory. Execution latency in the SIMD pipeline is 6 cycles and throughput is 1 wavefront every 4 cycles.

The architecture features a multi-level cache hierarchy. The L1 caches are private to each CU, whereas the L2 cache is shared among the CUs of the GPU. The L2 cache is 8-way banked and interleaved at a cache-line level. We model a virtual address space with full support for address translation hardware that includes a set of private fully-associative L1-TLBs, a set-associative L2-TLB that is shared by all the CUs, and an IOMMU. The L2-TLB is not banked since prior work [52] reveals that banking of the L2-TLBs does not offer benefits. Any address translation request that misses in the local GPU hierarchy is forwarded to the IOMMU, which is physically located on the CPU side. The IOMMU has a multi-threaded page table walker which can perform 8 searches in the page table in parallel. All of our experiments are run with a 4KB page size, which is the common page size used in prior studies on address translation hardware design on GPUs [9, 41, 42]. While larger pages (e.g., 2MB) have the potential of reducing L1-TLB misses, they have large page migration latencies [8, 11, 19] and can also increase the average number of stalled wavefronts on TLB misses to 100% [8, 9] and hence are not always optimal to use.

Workloads: For our evaluation, we selected a diverse set of workloads that includes both TLB sensitive and TLB insensitive applications. Our workloads are selected from three different benchmark suites that include Hetero-Mark [46] and SHOC [16] and AMDAPPSDK [4]. The memory footprints of our applications are listed in Table 2. The memory footprints are sufficiently large to

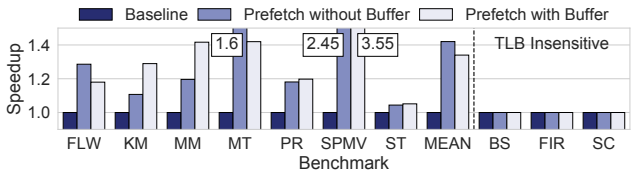

Figure 8: Performance speedup over the baseline (no prefetching) when prefetching directly into the TLB and when prefetching into the 24-entry prefetch buffer that is split from the L1-TLB.

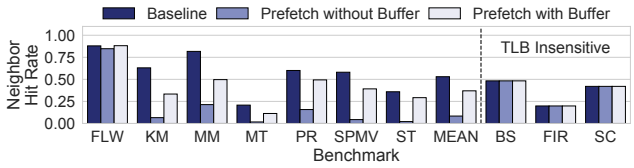

Figure 9: Comparison of the change in neighbor hit ratio potential when splitting a part of the L1-TLB into a 24-entry prefetch buffer. It is observed that without the prefetch buffer, the neighbor hit rate of the prefetching mechanism is significantly lower than the baseline(no prefetching).

stress the memory subsystem, including both the caches and the TLBs.

5 EVALUATION RESULTS

In this section, we present our evaluation of *Valkyrie* using the different workloads mentioned in Section 4. As shown in Figure 5, seven of our ten workloads (Matrix Transpose, Matrix-Multiplication, KMeans, Stencil2D, Page Rank, Sparse-Matrix Vector Multiplication and Floyd Warshall) show high TLB miss ratios (both at the L1-TLB level and L2-TLB level), and so they can benefit from our proposed scheme. We also include three other workloads that have low L1-TLB miss ratios (Finite Impulse Response, Bitonic Sort and Simple Convolution) to show that our scheme does not degrade the performance of such applications.

Splitting the L1-TLB into a smaller L1-TLB and a prefetch buffer: Figure 8 shows the speed up that can be achieved through prefetching without splitting the L1-TLB and through prefetching when splitting the L1-TLB into a smaller L1-TLB and prefetch buffer[2]. Even without splitting the L1-TLB, just prefetching provides an average[3] speedup of 1.34×. However, if translation entries are prefetched directly into the L1-TLB, the potential for finding page-translation entries in the neighboring L1-TLBs reduces to 8% on an average (see Figure 9). This happens because, although the actions of the prefetcher do not hurt the L1-TLB hit rate of the local TLB, it evicts entries that could have potentially been referenced by a neighboring L1-TLB in the future. By splitting the 128-entry L1-TLB into a 24-entry prefetch buffer and 104-entry L1-TLB, we

[2]As explained in Section 3.3, we reserve 24 L1-TLB entries from each L1-TLB as a prefetch buffer.
[3]All average values are calculated as geometric means.

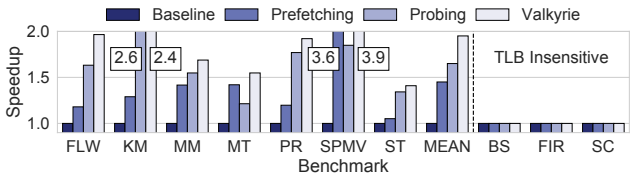

Figure 10: Performance speedup over the baseline for the different mechanisms of Valkyrie.

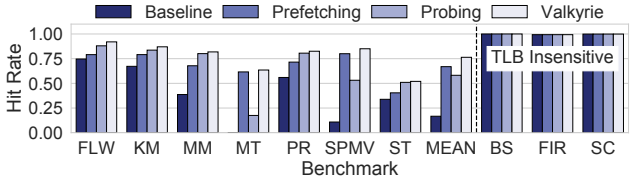

Figure 11: Improvements in L1-TLB hit ratio over the baseline for the different mechanisms of Valkyrie.

can improve our chances of finding page-translation entries in the neighboring L1-TLB to 37% on average (see Figure 9), while also achieving a small performance gain of 1.08× on average, when compared to prefetching directly into the L1-TLB. Therefore, from here onwards, we use the L1-TLB which has been split into a 104-entry L1-TLB and 24-entry prefetch buffer for the rest of this section[4], as this configuration provides both performance gains, as well as improves the neighbor-hit ratio potential. We present results for alternate L1-TLB split configurations later in this section.

Prefetching and Probing: Figure 10 shows the performance gained when using both prefetching and probing mechanisms of *Valkyrie* as compared to the baseline. The proposed prefetching mechanism can provide an average speedup of 1.45× over the baseline, whereas our proposed L1-TLB probing mechanism can provide an average speedup of 1.65×. When both mechanisms are used, *Valkyrie* can provide an average speedup of 1.95×. Note that the total speedup when the two mechanisms are combined is smaller than the sum of the speedups achieved in isolation. The reason is that the prefetching and the probing mechanisms are working to resolve a similar type of problem (i.e., L1-TLB misses with inter-core locality). Valkyrie achieves a speedup of 1.55× (not shown in the figure) if we do not reorganize the L1-TLB using a 104/24 split. This is due to reduced probing hits, which in turn reduces the benefits of the probing mechanism. When compared with a large and heavily multiported ideal MMU design, Valkyrie achieves a speedup of 0.54× as compared to the baseline which only achieves a speedup of 0.28× over the ideal MMU design. Interestingly, the performance impact of the two mechanisms on the different workloads varies. From Figure 10 we can observe that the Floyd Warshall (FLW) workload benefits more from probing (1.63×) than from prefetching (1.17×). The reason is that FLW has the possibility of resolving 86% (see Figure 3) of its L1-TLB misses from L1-TLBs within the

[4]The baseline L1-TLB still has 128-entries since it does not use the prefetch buffer

same Shader Engine (SE) and therefore probing is immensely helpful. On the other hand, other benchmarks such as the Sparse Matrix Vector Multiplication workload (SPMV) gain more from prefetching than probing (3.55× and 1.8×, respectively). This is because, for SPMV, translations are accessed by different cores in extremely close temporal proximity, making it more probable that the required translation information is already prefetched in the prefetch buffer. Moreover, if prefetching is done correctly, it has the potential to provide the best performance (since the data is already there when the access arrives) and therefore SPMV achieves huge performance gains from prefetching. The only application which suffers some performance degradation when both mechanisms are combined is the KMeans (a small drop from 2.6× achieved by probing to 2.4× when combined). We traced back the source of this problem and found that when the two mechanisms are combined, the L2-TLB hit rate decreases by 5%, as opposed to when only probing is used. We believe this is because, at runtime, the prefetcher can end up prefetching into certain CUs more than it does into others. This improves the L1-TLB hit rate of a subset of CUs. However, a side effect of this is that the L2-TLB entries can end up getting evicted (recall that we are using a mostly-inclusive TLB hierarchy) due to a lower number of references from the L1-TLBs where the hit-rates are higher. Due to this behavior, when another L1-TLB encounters a miss on a particular page-entry, it can no longer find that entry in the L2-TLB as the LRU replacement policy replaced it due to fewer references arriving at the L2-TLB. This, in turn, decreases the L2-TLB hit rate, which causes slight performance degradation.

Improvement in L1-TLB Hit Ratios: From Figure 11 we can observe that, when compared to the baseline, our proposed prefetching and probing mechanisms can improve the L1-TLB hit ratio on an average by 4× and 3.624× respectively. When combined in *Valkyrie*, the average L1-TLB hit ratio further improves by 4.60×, which is a significant improvement and is highly correlated with performance improvement.

Sensitivity of *Valkyrie* to L1-TLB size: While 128-entry L1-TLBs have been shown to provide the best performance [37], 64-entry L1-TLBs for GPUs are also quite common [9, 41, 42]. In a spirit similar to how we propose to split the L1-TLB area into a smaller L1-TLB and a prefetch buffer, we evaluated a design with 52-entry L1-TLB and a 12-entry prefetch buffer (we scaled down both the L1-TLB size and prefetch buffer size). From Figure 12, we can observe that we improve performance by 1.43× with prefetching. This gain is comparable to when using a 128-entry L1-TLB (1.45×). The performance benefits of probing are slightly lower (1.38× as opposed to 1.65× when using a 128-entry L1-TLB). This is expected because a smaller L1-TLB will reduce the chances of finding a valid translation in the neighboring L1-TLBs. When the prefetching and probing schemes are combined for the 64-entry L1-TLB, *Valkyrie* still outperforms the baseline by 1.71×, which is slightly lower than the 1.95× benefit when using a 128-entry L1-TLB. This is due to the decrease in the benefits of probing when using a smaller L1-TLB.

Sensitivity of *Valkyrie* to L2-TLB size: A larger L2-TLB has the potential to improve performance, as it reduces the L2-TLB misses and can avoid long-latency page-table walks. Therefore, we did a sensitivity study (see Figure 13) by comparing the performance when using a larger L2-TLB of 1024, 2048, 4096 and 8192 entries. While varying the size, we keep the associativity of the

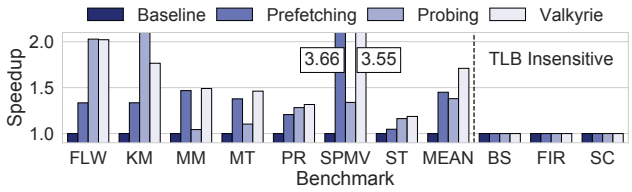

Figure 12: Performance speedup over the baseline for the different mechanisms of Valkyrie when using a smaller L1-TLB of 64 entries.

TLB the same and only vary the number of sets. We observed that Valkyrie still outperforms the baseline when scaling to larger L2-TLB sizes. We obtain a geometric mean speedup of 1.82×, 1.27×, 1.18× and 1.06×, respectively. The gains become less pronounced when increasing the L2-TLB size, due to the ability of a larger L2-TLB to reduce the number of misses. Overall, we can observe that around 8192 L2-TLB entries, the performance of the baseline when compared to Valkyrie is close (6% difference). Since Valkyrie still outperforms the baseline for the evaluated configurations, it shows that our solution is future-proof, even if the L2-TLB size keeps scaling in the future.

Sensitivity split of L1-TLB: We study alternate organizations of the L1-TLB, which is split into an L1-TLB and prefetch buffer. The configurations evaluated have a 32-entry L1TLB + 96-entry prefetch buffer, 64-entry L1TLB + 64-entry prefetch buffer, 96-entry L1TLB+ 32-entry prefetch buffer, which we term as SplitA, SplitB and SplitC, respectively. The default configuration of Valkyrie assumes the L1-TLB is split into a 104-entry L1-TLB and a 24-entry prefetch buffer. We observe that, as compared to the default split, the other splits have a geometric mean speedup of 0.74×, 0.84× and 0.92×, respectively over the default split. SplitA has the lowest performance because allocating more space for the prefetch buffer reduces the chances of the probing mechanism to find TLB entries from neighboring TLBs, which impacts the performance.

Sensitivity to larger page sizes: We also evaluated Valkyrie with page sizes of 8KB, 16KB, 64KB and 2MB. We observed a geometric mean speedup of 1.92×, 1.89× and 1.61× and 1×, respectively. The benefits of Valkyrie are less pronounced when scaling up to larger page sizes as using a large page will inevitably reduce the TLB miss rate. However, since large pages come with their own set of problems such as false-sharing in NUMA systems [19, 53], having solutions to reduce TLB miss rates for base page sizes is still important.

Area and Power Overhead of *Valkyrie*: We evaluated the hardware costs required to implement the two prefetching and probing mechanisms in *Valkyrie*. To support the prefetching mechanisms, there are two 32-bit registers for storing the average L1-TLB hit rates for the past and present phases, and one 32-bit register for storing the last action that was taken. The confidence counter requires a 2-bit register. A set of comparators is required to compare the average TLB hit rates across an epoch, as well as to identify the last action that was taken. There is a single Locality Detection Table (LDT) for the entire GPU that resides alongside the L2-TLB. The LDT is designed with 100 entries. Each entry holds a 64-bit

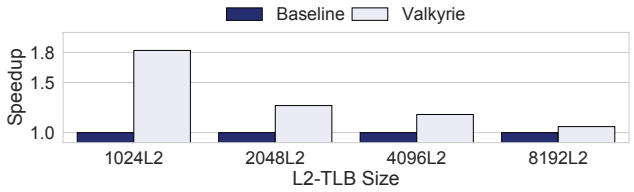

Figure 13: Performance speedup of Valkyrie over the baseline when using a larger L2-TLB size of 1024, 2048, 4096 and 8192 entries, respectively

.

mask to identify the L1-TLB-ID that has accessed a particular page. Besides, since the LDT is also fully-associative, it requires a comparator for each entry. However, the comparator in the LDT only needs to compare 18 bits, resulting in much smaller comparators in the LDT than in the L1-TLB, where we need to compare a 16-bit process ID and a 52-bit virtual page number. Therefore, the overall hardware cost of the LDT is smaller than a single L1-TLB in terms of both the storage space, as well as the number of comparators required. Splitting the L1-TLB into a smaller L1-TLB and a prefetch buffer has negligible overhead.

To support our probing mechanism, we need one intra-SE ring per SE in the GPU. The ring topology does not require any complex switches, saving both area and energy. The only area overhead comes from the bi-directional channels connecting the L1-TLBs. As the 16 L1-TLBs per SE are placed physically close to each other, the length of these channels can be kept short to reduce energy and area overhead. Each L1-TLB has a 16-entry queue, which stores the requests for arbitration. Each request is 64-bit wide, resulting in a total area of 128 bytes for the queue in each L1-TLB. In terms of power consumption, which we calculate using CACTI [33] and parameters obtained for circuit-level simulations [24], Valkyrie introduces an additional 0.06 watts of power over the baseline. This constitutes a total of 0.37% of the 175W-TDP [48] of the R9 Nano chip, which is the GPU simulated in our work.

6 RELATED WORK

There have been a number of studies aimed at improving virtual memory management on CPUs and GPUs.

GPU Address Translation: With the introduction of Unified Memory, a large body of research has focused on the design of the GPU's TLB architecture. Power et al. [38] and Pichai et al. [37] were the first to explore the design space for GPU address translation hardware and motivated the need for TLB-aware architectural enhancements for GPUs. Jaleel et al. [22] proposed a mechanism to store TLB entries in the last-level cache and DRAM. Vesely et al. [50] studied the overhead associated with address translation on GPUs and found that the TLB miss latency can take up to 25× to resolve, as compared to a CPU, due to the high bandwidth demands of GPU applications.

Yoon et al. [52] proposed a mechanism to reduce the impact of TLB misses by using virtually-indexed virtually-tagged (VIVT) L1 caches. While using a VIVT L1 cache can be an effective solution to filter translations on a GPU, they come with their own sets of problems (e.g., managing co-located kernels from two different processes, managing shared physical memory regions, etc.). Besides, using physically tagged caches makes it easier and more feasible to maintain compatibility issues with CPU caches [37] and also eases cache-coherency mechanisms between the CPU and GPU [43]. Hence, we opted for physically tagged caches in our work.

Ausavarungnirun et al. [9] describe how to manage a GPU's shared L2-TLB when running concurrent applications. Their mechanism focused on improving the performance of concurrent applications at the L2-TLB level by using TLB-fill tokens and L2-TLB bypassing. Our scheme can be integrated with this prior work wherein their work can potentially improve the performance of the L2-TLB and *Valkyrie* can improve the performance of the L1-TLB.

Shin et al. [41, 42] proposed architectural changes to the IOMMU scheduler so that page-table walk requests are served more efficiently. *Valkyrie* accelerates address translation at the L1-TLB level by exploiting the TLB sharing behavior in GPU applications. Our scheme complements the approaches proposed in [9, 22, 41, 42] and can be integrated with theirs to further improve performance.

TLB Prefetching Mechanisms: Kandiraju et al. [26] proposed a Distance Prefetching mechanism to prefetch TLB entries for single-core CPUs, utilizing the strided behavior of memory accesses to reduce the TLB miss rate. Saulsbury et al. [39] proposed a recently-used based TLB preloading scheme for CPUs. Bhattacharjee et al. [15] were the first to explore Inter-Core Cooperative TLB Prefetching mechanisms based on the inter-core page sharing behavior in multi-core CPU applications. Unlike CPUs, such Inter-Core Cooperative TLB prefetching cannot be directly applied to GPUs because of the large number of cores involved. GPU systems require a more centralized mechanism for controlling and coordinating prefetches. *Valkyrie's* Locality Based Prefetching and Locality Detection Table provide a more centralized solution. *Valkyrie* also exploits sharing behavior by using an L1-TLB probing mechanism, a characteristics that has not been explored in prior TLB prefetching studies. Margaritov et al. [31] proposed a prefetching mechanism for CPUs where prefetches are issued directly to deeper levels of the page table. They do not however prefetch directly into the TLB. Integration with such mechanisms to design multi-level prefetching mechanisms is a rich area for further exploration.

Power et al. [38] explored a simple TLB prefetcher design for GPUs which improves performance by only 1%. Kuth et al. [27] studied mechanisms for compiler-directed TLB prefetching on heterogeneous SOCs, requiring modifications to the source code. Vesely et al. [50] observed that the existing TLB prefetcher on GPUs are ineffective and motivated the need for a more application-aware TLB-prefetching mechanisms, which our work clearly adopts with Valkyrie's locality-based prefetching scheme.

Inter-core GPU Communication Mechanisms: Ibrahim et al. [21] proposed a mechanism to allow L1 cache misses on a GPU to be resolved by another L1 cache on the same GPU. Their mechanism relies on accurate identification of the cores that share similar cache lines to increase application performance. In contrast, we have observed that even if *Valkyrie's* LDT can provide some hints on which L1-TLBs share similar page entries, it is impossible to know if the page entry resides in a particular L1-TLB at the exact time of servicing an L1-TLB miss. This renders predictive mechanisms ineffective for inter L1-TLB communication. Other prior

work on inter-core GPU communication addressed cache coherency mechanisms and how to reduce the coherence overhead [43]. In contrast to all of the above work on inter-core communication for data caches, we identify the requirements for improving address translation efficiency and design tailored solutions for TLB prefetching and inter-core TLB communication. To support communication within a GPU, most prior work [10, 55] has focused on designing on-chip networks for enabling communication between the GPU's SEs and the L2 cache (or memory partitions), opting for a 2-D mesh network. Recently, Ibrahim et al. [21] have also explored using a 2-D mesh network for core-to-core communication. In the design of these on-chip networks, a "flat" GPU architecture has been commonly assumed, where a single-level on-chip network topology is employed. However, as described in Section 2, modern GPUs are built hierarchically with CUs clusterized in SAs, and with SAs clusterized in the SEs. Hence, in this work, we assume a hierarchical organization where, apart from the regular L1 cache to L2 cache interconnect, we integrate a ring network within each SE for communication between L1 TLBs. Ring networks are commonly used as network-on-chip solutions in a number of commercial multicore processors [7, 40]. A ring topology has also been proposed for inter-L1 cache communication in GPUs [17] and heterogeneous CPU/GPU architectures [28]. We adopt a similar ring network, but the key difference is that we leverage the unique communication characteristics of inter-TLB sharing to design a cost-efficient ring architecture.

GPU Page Management: Ausavarungnirun et al. [8] and Sun et al. [45] proposed mechanisms to dynamically decide between using small or large pages for GPU applications. Zheng et al. [54] proposed microarchitectural enhancements to hide page-fault latency on GPUs. They also proposed prefetching pages from the CPU to the GPU memory. Agarwal et al. [1] proposed page-prefetching mechanisms for CPU-to-GPU page transfers. Ganguly et al. [18] studied the interplay between the prefetcher and page-eviction policies on systems supporting Unified Memory and proposed mechanisms to design an eviction aware page-prefetching policy. In contrast, our work addresses prefetching of page table entries at the L1-TLB, as well as inter L1-TLB probing, by leveraging the TLB entry sharing behavior found in GPU applications.

7 CONCLUSION

Given the advantages that Virtual Memory (VM) provides in terms of programming ease, the performance of VM is going to be critical for modern day GPU applications. However, address translation bottlenecks such as high TLB miss rates and long latency page walks can severely harm the performance of GPU applications. Mechanisms to mitigate these costs are going to be extremely important so that GPUs can enjoy the benefits of VM and not pay a large performance overhead.

In this work, we have observed that certain classes of TLB-sensitive GPU applications exhibit an inherent sharing behavior where the same page-table entry is shared across multiple L1-TLBs in the system. We proposed and evaluated two novel solutions to take advantage of this inter L1-TLB locality present in GPU applications, reducing address translation bottlenecks and improving performance.

Our proposed scheme is named *Valkyrie*, a programmer-transparent hardware solution that exploits inter L1-TLB locality using prefetching and L1-TLB probing schemes. *Valkyrie's* prefetching mechanism dynamically identifies inter-TLB locality and prefetches page translations ahead of time to take advantage of inter-TLB locality. *Valkyrie's* probing scheme operates within each Shader Engine to effectively allow L1-TLBs to retrieve data from neighboring L1-TLBs without paying the high performance penalty of going to the L2-TLB. *Valkyrie's* prefetching alone can provide an average speedup of 1.45×, whereas its probing mechanism can provide an average speedup of 1.65×. Combining these two schemes, *Valkyrie* enables the GPU hardware to take full advantage of inter L1-TLB locality, resulting in an average speedup of 1.95×.

To further improve the performance of *Valkyrie*, we plan to integrate it with other approaches that improve the page-walk performance [41, 42]. We also plan to study effective communication mechanisms and hierarchical network designs such as fat-trees [32] to support more efficient inter-L1 TLB communication schemes. Another promising direction to explore further is the performance impact of such schemes when integrated with coalesced TLBs for GPUs [36].

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback. This work was supported in part by NSF CNS-1525412, NSF CNS-1525474, NRF-2015M3C4A7065647, NRF-2017R1A2B4011457, and AMD.

REFERENCES

[1] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 354–365.

[2] Thomas Alexander and Gershon Kedem. 1996. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings. Second International Symposium on High-Performance Computer Architecture*. IEEE, 254–263.

[3] AMD. 2009. AMD APU. (2009). http://developer.amd.com/wordpress/media/2012/10/apu101.pdf

[4] AMD. 2015. AMD APP SDK OpenCL Optimization Guide.

[5] AMD. 2018. OpenCL Shared Virtual Memory. (2018). https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/sharedVirtualMemory.html

[6] AMD. 2019. AMD Radeon VII. (2019). https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf

[7] I. Anati, D. Blythe, J. Doweck, H. Jiang, W. Kao, J. Mandelblat, L. Rappoport, E. Rotem, and A. Yasin. 2016. Inside 6th gen Intel Core: New microarchitecture code named skylake. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–39. https://doi.org/10.1109/HOTCHIPS.2016.7936222

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 136–150.

[9] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. *ACM SIGPLAN Notices* 53, 2 (2018), 503–518.

[10] Ali Bakhoda, John Kim, and Tor M. Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 421–432. https://doi.org/10.1109/MICRO.2010.50

[11] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems.

In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 596–609.

[12] Bradford M Beckmann and David A Wood. 2004. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 319–330.

[13] Abhishek Bhattacharjee. 2017. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro* 37, 5 (2017), 6–10.

[14] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 62–63.

[15] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-core cooperative TLB for chip multiprocessors. *ACM Sigplan Notices* 45, 3 (2010), 359–370.

[16] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 63–74.

[17] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2016. Cooperative caching for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–25.

[18] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*. 224–235.

[19] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on {NUMA} Systems. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 231–242.

[20] Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, and Sharon C Glotzer. 2015. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications* 192 (2015), 97–107.

[21] Mohamed Assem Ibrahim, Hongyuan Liu, Onur Kayiran, and Adwait Jog. 2019. Analyzing and Leveraging Remote-core Bandwidth for Enhanced Performance in GPUs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 258–271.

[22] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-performance address translation by extending TLB reach of GPU-accelerated systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–24.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[24] Ajay Joshi, Christopher Batten, Yong-Jin Kwon, Scott Beamer, Imran Shamim, Krste Asanovic, and Vladimir Stojanovic. 2009. Silicon-photonic clos networks for global on-chip communication. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 124–133.

[25] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[26] Gokul B Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 195–206.

[27] Andreas Kurth, Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2018. Scalable and efficient virtual memory sharing in heterogeneous SoCs with TLB prefetching and MMU-aware DMA engine. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 292–300.

[28] Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. 2013. Design Space Exploration of On-Chip Ring Interconnection for a CPU-GPU Heterogeneous Architecture. *J. Parallel Distrib. Comput.* 73, 12 (Dec. 2013), 1525–1538. https://doi.org/10.1016/j.jpdc.2013.07.014

[29] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 49–63.

[30] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. 2006. Gpu accelerated smith-waterman. In *International Conference on Computational Science*. Springer, 188–195.

[31] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1023–1036.

[32] Hiroki Matsutani, Michihiro Koibuchi, and Hideharu Amano. 2007. Performance, cost, and energy evaluation of fat h-tree: A cost-efficient tree-based on-chip network. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–10.

[33] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[34] NVIDIA. 2018. NVIDIA Unified Memory. (2018). http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[35] NVIDIA. 2019. Turing GPU. (2019). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[36] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. Colt: Coalesced large-reach tlbs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 258–269.

[37] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 743–758.

[38] Jason Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 568–578.

[39] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. 2000. Recency-based TLB preloading. In *Proceedings of the 27th annual international symposium on Computer architecture*. 117–127.

[40] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-Core X86 Architecture for Visual Computing. *ACM Trans. Graph.* 27, 3 (Aug. 2008), 1–15. https://doi.org/10.1145/1360612.1360617

[41] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling page table walks for irregular GPU applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 180–192.

[42] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-aware address translation for irregular GPU applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 352–363.

[43] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 578–590.

[44] GS Sohi and TM Austin. 1996. High-bandwidth address translation for multiple-issue processors. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*. IEEE, 158–158.

[45] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*. 197–209.

[46] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–10.

[47] Michael Bedford Taylor. 2013. Bitcoin and the age of bespoke silicon. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 1–10.

[48] Tech Powerup. 2015. AMD R9Nano. (2015). https://www.techpowerup.com/gpu-specs/radeon-r9-nano.c2735

[49] Steven Vanderwiel. 2007. Multi-level cache architecture having a selective victim cache. US Patent App. 11/259,313.

[50] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 161–171.

[51] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 430–443.

[52] Hongil Yoon, Jason Lowe-Power, and Gurindar S Sohi. 2018. Filtering translation bandwidth with virtual caching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 113–127.

[53] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 339–351.

[54] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.

[55] Amir Kavyan Ziabari, José L. Abellán, Yenai Ma, Ajay Joshi, and David Kaeli. 2015. Asymmetric NoC Architectures for GPU Systems. In *Proceedings of the 9th International Symposium on Networks-on-Chip*. Article Article 25, 8 pages.