Designing Virtual Memory System of MCM GPUs

Pratheek B*
Indian Institute of Science
Bangalore, India
pratheekb@iisc.ac.in

Neha Jawalkar*
Indian Institute of Science
Bangalore, India
jawalkarp@iisc.ac.in

Arkaprava Basu
Indian Institute of Science
Bangalore, India
arkapravab@iisc.ac.in

***Abstract*— Multi-Chip Module (MCM) designs have emerged as a key technique to scale up a GPU's compute capabilities in the face of slowing transistor technology. However, the disaggregated nature of MCM GPUs with many chiplets connected via in-package interconnects leads to non-uniformity.**

We explore the implications of MCM's non-uniformity on the GPU's virtual memory. We quantitatively demonstrate that an MCM-aware virtual memory system should aim to ① leverage aggregate TLB capacity across chiplets while limiting accesses to L2 TLB on remote chiplets, ② reduce accesses to page table entries resident on a remote chiplet's memory during page walks. We propose MCM-aware GPU virtual memory (MGvm) that leverages static analysis techniques, previously used for thread and data placement, to map virtual addresses to chiplets and to place the page tables. At runtime, MGvm balances its objective of limiting the number of remote L2 TLB lookups with that of reducing the number of remote page table accesses to achieve good speedups (52%, on average) across diverse application behaviors.

***Keywords*-Graphics Processing Units, Multi-Chip Module, Chiplet, Virtual Memory, Address Translation, Page Table Walkers, Translation Look-aside Buffers**

I. Introduction

The slowing of transistor scaling makes it hard to build larger monolithic single-chip processors [29], [37]. At the same time, the need to increase compute capabilities of processors has never been higher. The industry is increasingly adopting multi-chip-module (MCM) designs to cope with these twin challenges [2], [5], [6], [22], [25], [56], [58].

MCM designs embrace a disintegrated design where a processor package contains multiple smaller *chiplets*, with each chiplet housing only a fraction of the resources compared to a traditional monolithic SoC. Fabricating individual chiplets is easier and cheaper since it needs only a fraction of the transistors that a larger monolithic design would otherwise contain. These chiplets are then connected through a fast, high-bandwidth *in-package* interconnect to scale up to the computational needs of modern software. MCM designs are thus also called chiplet-based processor designs.

AMD's Epyc and Ryzen CPU families were the first commercial products to adopt an MCM design [2]. MCM designs are now making their way to commercial GPUs (Graphics Processing Units). For example, NVIDIA's upcoming Grace, AMD's Aldebaran MI200, and Intel's Ponte Vecchio GPUs, all embrace the MCM design to meet ever-growing compute needs in the era of slowing transistor scaling [22], [43], [56]. Further, broader industry support toward a common chiplet interconnect (UCIe) will likely make MCM-based processors commonplace in the near future [6], [25], [57].

MCM designs, however, present new design challenges due to the non-uniformity in accessing resources across chiplets. To an application, an MCM GPU may appear as a logically unified monolithic GPU. However, its physical resources are distributed across the chiplets. For example, each chiplet has its own memory hierarchy, including a slice of High-Bandwidth Memory or GDDR (HBM). Threads executing on a chiplet may need to access data residing on another chiplet's HBM. The *remote* memory access is slower than accessing data residing on the memory of its own chiplet. The remote accesses need to cross the inter-chiplet interconnect twice. While inter-chiplet bandwidth is often adequate, traveling to another chiplet can add $\sim$32 ns, one-way [7].

Several recent works have researched ways to limit the overheads of remote data access in MCM GPUs [7], [31], [32], [51]. These works explored the scheduling of GPU threads and placement of data for co-locating threads, and the data those threads access, onto the same chiplet. For example, a recent work by Khairy et al. showed that static analysis of GPU kernel yields a handful of patterns that capture the relation between the groups of threads (called Cooperative Thread Array or CTA) and the data they access [31]. The authors then propose Locality-Aware Scheduling and Placement (LASP) that leverages these patterns to schedule CTAs and place data pages on the same chiplet [31].

However, no prior work has explored the impact of MCM's non-uniformity on the GPU's virtual memory. We find that even when the data accesses are local, address translations may need to access resources on a remote chiplet, limiting the benefits of local data access.

We start by exploring the obvious design choices for the virtual memory in an MCM GPU. In one design, the L2 TLB and the page table walkers (PTWs) could be *private* to a chiplet. The L2 TLB and the PTWs serve requests only from their own chiplet. Alternatively, in a *shared* design, the L2 TLBs and PTWs of all chiplets are logically shared. On an L1 TLB miss, the virtual address of the missing translation is hashed to one of the chiplet's L2 TLB (also

*Both authors contributed equally.

called 'home' L2 TLB 'slice') that should serve the L1 miss. On a miss in the L2 TLB slice, chiplet's local PTWs service the request by walking the in-memory page table.

We find that MCM's inherent non-uniformity affects the virtual memory in two ways. First, a hit in the L2 TLB slice of another chiplet (i.e., *remote* L2 TLB hit) is slower than a hit in the chiplet's local L2 TLB slice. This affects only the shared TLB design since all hits are local in a private TLB. However, a shared TLB can leverage the aggregate L2 capacity across chiplets, unlike a private TLB.

Interestingly, MCM designs can significantly affect the page walk subsystem. When a PTW walks the in-memory page tables on an L2 TLB miss, it may have to access page table entries (PTEs) residing on the memory of another chiplet (i.e., remote memory access). A page walk may require up to four memory accesses for typical four-level radix-tree page tables. Each of those can be a remote memory access, incurring the additional latency of crossing the inter-chiplet interconnect twice. Thus, page table walks can significantly slow down due to remote accesses to PTEs.

We quantitatively analyze how these factors affect the design choices in an MCM GPU's virtual memory. We find that no single design point works well for all applications. Some applications perform better with private TLB thanks to faster local L2 TLB hits. Others prefer shared TLB due to lower L2 TLB miss rates, thanks to larger aggregate TLB capacity. We quantitatively find that several applications could speed up if remote L2 TLB hits were instead local hits.

We also notice that if an application does not speed up with shared TLB design over private TLB, it does *not* necessarily mean that the application has no use of the aggregate TLB capacity. In regular GPU applications, it is common for CTAs to compute on mutually exclusive portions of the dataset. If the CTAs and the data they access are mapped onto the same chiplet, then the aggregate L2 TLB capacity of the chiplets is well-utilized even with private TLB since there is no duplication of entries across the L2 TLB slices. Therefore, CTA scheduling and data placement are important considerations for L2 TLB utilization.

We find that the slower walks due to remote accesses to PTEs cause significant performance degradation under both private and shared TLB for several applications. Slower walks make servicing each L2 TLB miss significantly costlier and keep the walkers occupied for longer. This, in turn, increases the queuing at the PTWs and TLBs.

Driven by the analysis, we propose MCM-aware GPU virtual memory (MGvm) with the following goals. ① Leverage the aggregate L2 TLB capacity across the chiplets, ② Render remote L2 TLB hits to local hits, and ③ Reduce remote memory accesses to PTEs during page table walks.

MGvm proposes three enhancements for the purpose. First, it extends LASP [31]'s static analysis used for CTA scheduling and data page placement for better aggregate use of L2 TLB capacity while limiting remote L2 TLB accesses. If data access is local, then MGvm aims to ensure the corresponding address translation request is local too.

In a conventional design, the virtual address (VA) of an L1 TLB miss is hashed to determine the chiplet's L2 TLB slice (and thus, PTWs) should service the request. We call the hash function the home slice selection function (HSL). For private TLB, the HSL maps all VAs to the local L2 TLB slice, while in shared TLB, typically, an XOR or MOD of the VA determines its 'home' L2 TLB slice. In contrast, MGvm employs a *dynamic* HSL (dHSL) where the function is determined for each kernel. During a kernel's launch, the static analysis in LASP that determines its CTA scheduling and data placement to reduce remote memory access also determines the dHSL in MGvm. Thus, if a data access is local, the corresponding L2 TLB access is also local. Since dHSL maps a given VA to only one L2 TLB slice, it does not waste aggregate L2 TLB capacity due to duplication either.

Next, we observe that remote accesses to PTEs are avoided if a page containing PTEs is placed on the same chiplet whose L2 TLB slice (and thus, PTWs) are responsible for servicing the VA range mapped by PTEs residing on that page. The GPU driver, therefore, uses the dHSL to decide where to place a page with leaf-level PTEs. We focus on placing leaf-level PTEs as upper-level PTEs are often well-cached in page walk caches. A challenge, however, is that a page (4KB) typically holds 512 8-byte long PTEs, mapping 2MB of contiguous VA region. Thus, to avoid remote accesses to leaf-level PTE, the dHSL must map (multiples of) a 2MB VA region to a chiplet.

We also observe that in a few applications, a relatively small VA range ($<$ 2MB) is concurrently accessed from several chiplets. The coarse grain mapping of VAs by dHSL can lead to flooding of one of the L2 TLB slices with requests while others remain lightly loaded. We extend the GPU hardware to detect such imbalance at runtime and switch to finer-grain (4KB) mapping of VA amongst the chiplets to distribute L2 TLB traffic better.

In an evaluation with 15 workloads on a GPU with four chiplets, MGvm improves performance over private and shared TLB designs by 52% and 30%, on average, respectively. In summary, we make the following contributions.

- We, for the first time, quantitatively analyze and explore design choices for the virtual memory in MCM GPUs.
- We propose MCM-aware GPU virtual memory that achieves a balance between using aggregate L2 TLB capacity, local L2 TLB accesses, and local PTE accesses on page walk.

II. Background and Related work

Processor vendors have been building ever-larger GPUs to cater to the growing computing needs of today's data-driven world. However, with a GPU's die area already reaching $800mm^2$ [41] and the decreasing yield due to shrinking transistor sizes, it is economically unviable to continue to build larger dies [63].

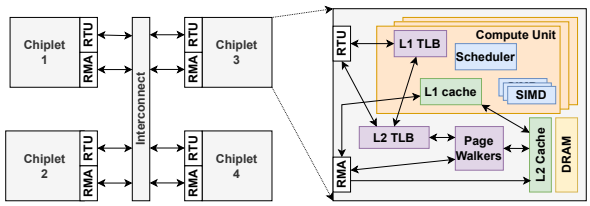


Figure 1: Architecture of an MCM GPU

Processor vendors are thus increasingly adopting MCM designs for both CPUs and GPUs [2], [22], [44], [56]. An MCM-based processor comprises multiple smaller dies or chiplets glued together with fast in-package interconnects. Each chiplet houses only a fraction of the resources of a large monolithic die. Thus, each chiplet has a much smaller die size, with a higher likelihood of getting a working chiplet. The aggregate computing capability of all chiplets is typically similar to or more than a large monolithic chip. Chiplets are connected with each other with high bandwidth, low latency (e.g., 768 GB/sec, 32ns [8]) in-package interconnects. The chiplets work together as a single logical unit and may present themselves as a *single GPU* to the programmer.

However, MCM designs introduce non-uniformity. Accessing resources on a remote chiplet is slower than accessing resources on a local chiplet. Thus, the key challenge is to leverage this non-uniformity. We, for the first time, explore the impact of non-uniformity on a GPU's virtual memory.

GPU architecture: Figure 1 shows a typical 4-chiplet MCM GPU, with the components of a single chiplet shown in detail. Each chiplet in an MCM GPU houses the typical compute and memory of a monolithic GPU, but in smaller portions. A chiplet contains several Compute Units (CUs), a.k.a, SMs in NVIDIA's terminology. Each CU is composed of several SIMD units. SIMD units in a CU share an L1 cache and an L1 TLB. The SIMD units of a CU typically execute a group of 32-64 threads in lockstep. A hardware coalescer combines memory accesses that fall on the same cache line before looking up the L1 cache. A programmer is required to divide the grid of threads executing their kernel (GPU program) into Cooperative Thread Arrays (CTAs), a.k.a, workgroups or threadblocks. A CTA can contain up to 1024 threads, and a given CTA executes entirely on a single CU. A GPU kernel (program) is launched for execution with many such CTAs.

Each chiplet has a highly banked L2 cache shared by all the CUs in the chiplet. Each chiplet also hosts a stack of HBM or GDDR [3]. It is typical for MCM GPUs to logically share the L2 cache and HBM across chiplets to aggregate the on-board cache and memory capacity [7], [31]. Thus, threads executing on a CU have access to the L2 cache and memory on *all* chiplets. However, accesses to L2 caches or HBMs on a remote chiplet are slower than accesses to the local L2 cache or HBM.

Each CU has a private L1 TLB. A larger L2 TLB is shared across all the CUs of the chiplet. The L2 TLB is followed by a set of parallel page table walkers (PTWs), also shared among the CUs. Typical 4-level radix-tree page tables resident on HBM(s), hold the virtual-to-physical address translation [42]. On a load/store, the associated L1 TLB of the CU is looked up. On a miss, the L2 TLB is looked up.

On an L2 TLB miss, a PTW walks the page tables to obtain the desired translation. A page walk cache (PWC) helps speed up page table walks by caching the higher levels (levels 1-3) of the page tables [11]. It performs a longest-prefix match on the virtual page number (VPN) to be translated. Based on the length of a prefix match, 1-4 memory accesses are required for a walk. In our design, page table entries are also cached in the L2 cache along with data.

While L1 TLBs are private to each CU, the L2 TLBs, PTWs, and PWCs of chiplets can either be logically shared or remain private to their chiplet. In the next section, we will quantitatively discuss the implications of these designs.

In our design, address translation requests that need to travel to another chiplet are routed through per-chiplet Remote Translation Units (RTUs). Data accesses to other chiplets are similarly routed through Remote Memory Access (RMA) Units (Figure 1). RTUs and RMAs are connected to each other via a high-speed in-package interconnect that provides 32ns latency from any chiplet to any other [7].

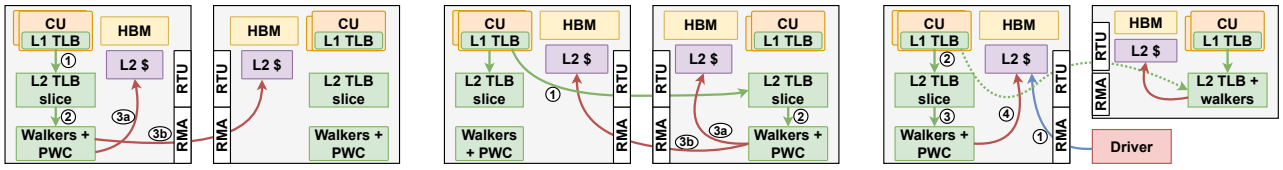
The GPU driver decides the allocation of CTAs to chiplets. It is also responsible for placing the pages containing data or PTEs across chiplets' memory.

CTA scheduling and data placement in MCM GPUs: Remote memory accesses, from a CTA executing on one chiplet, to memory on another chiplet, incurs additional latency compared to local memory accesses (to the same chiplet). Prior works have explored ways to reduce the number of slower remote memory accesses by coordinating CTA scheduling amongst chiplets with the placement of data pages across chiplets [7], [31].

Arunkumnar et al. pioneered the work on optimizing data locality for CTAs in MCMs GPUs [7]. They distribute contiguous groups of CTA to chiplets with the assumption that neighboring CTAs are likely to access similar regions of data. They propose a first-touch policy for data placement whereby a page is placed on the chiplet from where it is first accessed. Unfortunately, the first touch policy relies on GPU page faults, which are at least an order of magnitude slower than CPU page faults at 20-50 microseconds [62], [68].

Researchers also enabled software to explicitly express data locality to be exploited by the hardware [59], [65]. While a promising approach, it is not always possible for programmers to capture data locality in their program correctly and modify their program to communicate the same.

Recent works propose to use compile-time static analysis of kernels to infer data access patterns and their relation to CTAs for coordinating CTA scheduling and data placement [31], [32]. Inferring data access patterns through static analysis is plausible for GPU kernels as all memory is allocated before



(a) Private L2 TLB (b) Shared L2 TLB (c) Our proposal: MGvm

Figure 2: TLB lookup and page table walk under various configurations. Dotted lines show less common paths

a kernel starts execution, and threads are often arranged in CTAs to facilitate parallel access to data.

Khairy et al. [31] use static index-analysis of GPU kernels to minimize remote memory accesses. In their work, **LASP** (Locality Aware Scheduling and Placement), GPU kernels are classified according to their data access patterns. LASP then schedules CTAs to chiplets and tries to place the data pages those CTAs will access on the same chiplet, to maximize local memory accesses.

LASP classifies kernels into four classes using static analysis. In kernels such as Jacobi-1D, CTAs access overwhelmingly mutually exclusive data regions. LASP partitions the data across different chiplets' memory, and the CTAs that exclusively access them are scheduled on the same chiplets. Thus, data accesses remain local. Such kernels are classified 'no locality' (NL) since there is no locality (sharing) across CTAs. The kernels where a set of neighboring CTAs access contiguous sets of rows (columns) of a matrix are classified 'RCL' (row-column locality). Here, LASP stripes CTAs and data across rows (columns) amongst the chiplets to limit remote memory accesses. For example, in SYRK, CTAs along the y-axis of the kernel grid access consecutive rows of the input matrix. Thus, CTAs are divided along the y-axis of the grid, and the data is distributed row-wise.

The kernels in ITL (intra-thread locality) category demonstrate access locality within each thread, but there is a little discernible pattern across CTAs. Finally, kernels whose access patterns do not fit any of the patterns are termed 'unclassified'. The set of CTAs and data pages are divided equally, and contiguous blocks of CTAs and data pages are distributed evenly across chiplets.

Our baseline design uses LASP's CTA scheduling and data placement. We independently found LASP to be effective in reducing remote data accesses – it reduced remote data accesses from 67% to 18%, on average, over round-robin CTA scheduling and data placement. Khairy et al. further proposed controlled caching of data from remote chiplets in the L2 cache. However, it complicates cache coherence. We do not incorporate it in our baseline as it does not directly impact the virtual memory.

III. MCM's Impact on GPU Virtual Memory

We study the impact of MCM's non-uniformity on the GPU's virtual memory subsystem. First, the non-uniformity can affect the L2 TLB lookups in an MCM GPU. Each chiplet has an L2 TLB, which may be configured primarily in two ways. The L2 TLB on a chiplet may cache *only* the address translations requested by CUs on the same chiplet (private TLB). Alternatively, the L2 TLBs can be part of a logically shared TLB across the entire GPU (shared TLB). We refer to the L2 TLB on a chiplet as an L2 TLB *slice*. Figures 2a and 2b pictorially illustrate the private TLB and shared TLB designs.

In a private TLB design (Figure 2a), all L2 TLB accesses occur locally at the L2 TLB slice of the CU where the request originated ①. On an L2 TLB miss, page table walkers (PTWs) on the same chiplet service the walk request ②. In the shared TLB design (Figure 2b), on an L1 TLB miss, an address translation request is routed to one of the L2 TLB slices ('home') based on the hash of the missing virtual address ①. We call this hash function the *Home Slice Selection function* (HSL). Under shared TLB, translations in the L2 TLB are cached only in the home slice. On an L2 TLB miss, the PTWs of the home slice perform the page walk ②. We refer to the L2 TLB requests that travel to another chiplet as remote requests. Remote requests are slower than local requests, as they incur additional interconnect latency. While the private TLB does not incur remote requests, it may not necessarily leverage the aggregate capacity of the L2 TLB slices of all chiplets.

The non-uniformity of MCM design uniquely affects page table walks. A PTW can perform up to four memory accesses (assuming a four-level page table) based on hits/misses in the page walk caches. The GPU driver populates the PTEs and can place the pages containing PTEs in any chiplet's memory. A page walk may access PTEs residing on a remote chiplet's memory, incurring extra latency. Page walks can incur remote memory access in both private TLB and shared TLB (③b in Figures Figures 2a and 2b). Walks that access local memory are faster (③a). The fraction of remote memory accesses to PTEs is a key factor in determining the latency experienced on page walks in an MCM design. Further, slower walks can stall the L2 TLB by occupying MSHR entries at the L2 TLB for longer. On an MSHR stall, no new TLB misses can be served, ultimately putting back pressure on the entire address translation system.

We quantitatively analyze these effects on an MCM GPU's virtual memory. We first detail the baseline design. We follow Khairy et al.'s LASP for CTA scheduling and data placement [31]. However, their proposal is silent on how the pages

containing PTEs are placed. A naive policy would be to spread the pages with PTEs uniformly across chiplets in a round-robin fashion. However, this leads to many remote memory accesses on page walks. Instead, in our baseline, the PTE placement follows the data placement policy. Memory can be allocated amongst chiplets by the driver at the granularity of a page (default, 4KB) or its multiples. A single 4KB page contains 512 8-byte PTEs [42]. Thus, a single page with leaf-level PTEs contains translations for 2MB of contiguous virtual address region. We place a page containing leaf-level PTEs on the chiplet where any data page from the corresponding 2MB VA region is first placed.

The above-mentioned policy for PTE placement is also the same strategy used by Linux in placing pages with PTEs in multi-socket NUMA machines [1]. Pages containing upper-level PTEs follow the same principle, although each higher level covers 512 times more contiguous VA region. We empirically found that this policy reduced the fraction of remote memory accesses on page walks by 64%, on average, over the naive approach of uniformly distributing PTEs.

Private versus shared TLB design: We start our quantitative analysis by exploring whether the private TLB or shared TLB design is preferable. For the analysis, we choose applications that can utilize at least four chiplets. Section VI-A details the methodology, including the applications evaluated. The applications cover diverse behavior, including those that stress the virtual memory and those that do not.

Figure 3 shows the throughput of applications under the private TLB and shared TLB designs (normalized to private TLB for each application). We notice that neither private TLB nor shared TLB always performs better than the other.

The advantage of shared TLB is its use of aggregate L2 TLB capacity. Several applications, e.g., MIS, SPMV, and SYRK, benefit from larger TLB capacity. Table III confirms the same. It reports the number of L2 TLB misses per thousand SIMD instructions (MPKI). We observe that kernels of applications such as SPMV (sparse matrix-vector multiplication) suffer very high MPKI. The larger aggregate capacity of shared TLB significantly lowers the MPKI.

In contrast, applications like page rank (PR) have significant MPKI but shared TLB does not reduce L2 TLB misses significantly. The irregular memory access pattern of the graph application, coupled with its larger memory footprint that overflows even the aggregate L2 TLB capacity, is to blame. While PR is unable to leverage aggregate capacity, it suffers from increased remote TLB accesses in the shared TLB design. Applications such as Stencil2D (S2D) also fail to benefit from the aggregate capacity. The CTAs in the kernel access data that can be well-partitioned. LASP's data placement and CTA scheduling policy leverage this to ensure that CTAs access data placed on their local chiplet. Consequently, an L2 TLB entry is typically looked up only by the CTAs executing on the same chiplet. Thus, private TLB does not lose capacity relative to a shared TLB as

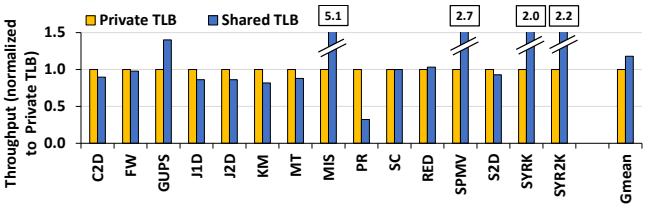


Figure 3: Throughput normalized to Private TLB design.

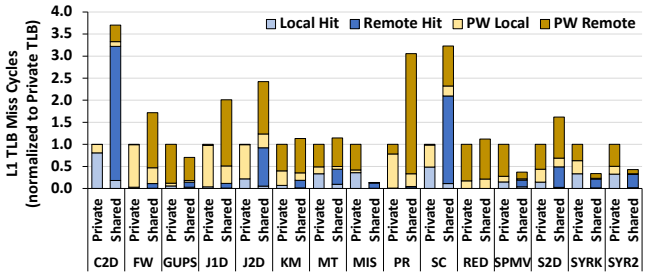


Figure 4: Breakdown of L1 TLB miss latency

there is little duplication of entries across L2 TLB slices of individual chiplets.

Effects of non-uniformity on virtual memory: We now take a deeper look at the effect of the non-uniformity of MCM on a GPU's virtual memory. The address translation requests that hit the L1 TLB are not affected by MCM's non-uniformity since L1 TLBs are private to each CU. In shared TLB, an L1 TLB miss may require a remote L2 TLB lookup if the HSL maps the virtual address onto a remote chiplet. In the case of private TLB, though, only the local L2 TLB slice is accessed. However, a page walk may need to access PTEs resident on a remote chiplet's memory. Thus, page table walks in both the designs can suffer from the non-uniformity of the MCM design.

To quantify the contributions of these factors on both the private TLB and shared TLB designs, we break down the total L1 TLB miss latency cycles into four parts. ① **L2 TLB local hit cycles** (`Local hit`). These cycles account for L1 TLB misses that hit in the L2 TLB of the same chiplet. ② **L2 TLB remote hit cycles** (`Remote hit`). These account for L2 TLB hits on a remote chiplet. These cycles are absent in private TLB. ③ **Local page walk cycles** (`PW local`). These are L2 TLB miss cycles spent by a walker to access PTEs resident on the same chiplet as the walker. ④ **Remote page walk cycles** (`PW remote`). These are the L2 TLB miss cycles a walker spends accessing PTEs that reside on a different chiplet from the walker. Figure 4 shows stacked (vertical) bars of the above breakdown of the L1 TLB miss latency for each application under the private TLB and shared TLB designs. Consequently, each application has two stacked bars. The height of each stack is normalized to the L2 TLB miss cycles under private TLB for a given application.

Several applications, e.g., C2D, J2D, and SC, spend a significant amount of cycles in 'L2 TLB remote hit' under shared TLB. The address translation overhead of

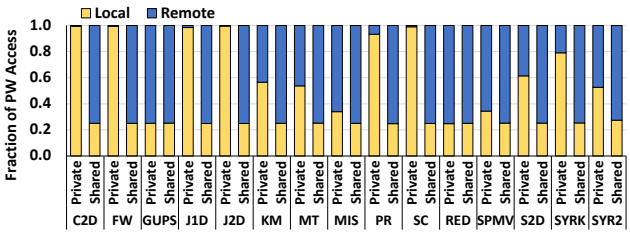


Figure 5: Split of page walk accesses into remote and local

these applications would reduce if these remote hits could be converted to local hits. This also explains why these applications perform better with private TLB, particularly since the aggregate L2 capacity with shared TLB does not reduce MPKIs (Table III). Next, we observe that applications such as J1D, J2D, PR, and S2D suffer under shared TLB due to remote memory accesses during page walks.

For a deeper analysis, we report the fraction of memory accesses due to page walks that are local versus those that are remote in Figure 5. We observe that under shared TLB, often a more significant fraction of memory accesses due to page walks end up on a remote chiplet compared to private TLB. This is expected in hindsight. LASP is often able to schedule CTAs and place the data pages those CTAs access on the same chiplet. LASP, coupled with our baseline's PTE placement policy, ensures that PTEs corresponding to the data are also likely to be placed on the same chiplet.

For example, suppose that CTAs executing on chiplet 1 mostly access data on the same chiplet. The corresponding PTEs are also likely to be placed on chiplet 1. Under private TLB, the L2 TLB slice on the same chiplet (chiplet 1) services address translation requests from the CTAs, and thus, the PTWs on chiplet 1 will perform page walks on L2 TLB misses. Since most PTEs corresponding to the virtual addresses of that data would also reside on the chiplet 1, page walks will mostly look up the local memory. In contrast, under shared TLB, the virtual addresses can map onto the L2 TLB slice of any of the chiplets (say, chiplet 2) as per the HSL. If there is a miss on the remote L2 TLB slice, that chiplet's walkers would have to access PTEs resident on chiplet 1, inducing remote accesses.

We also observe that a few applications incur many remote PTE accesses (a.k.a., remote page walk) even under private TLB. For example, in MT, the input matrix is accessed row-wise, while the output matrix is accessed column-wise. Accesses to the output matrix are largely remote. Since the baseline places page tables along with the data, the corresponding page table accesses also tend to be remote. Similarly, for SPMV, all the elements of a vector are accessed by all the CTAs. Thus, some amount of remote data accesses and thus, corresponding remote page walks are unavoidable. **Summary:** ① Neither private TLB nor shared TLB suits all applications. ② For several applications, e.g., SPMV, harnessing the aggregate L2 TLB capacity across chiplets, as in shared TLB, is important. ③ However, applications such as J2D slow down due to remote L2 TLB lookups in shared TLB design. ④ Importantly, remote memory accesses on page walks significantly affect both private TLB and shared TLB designs; more so in the latter.

IV. Goals and Key Design Ideas

Driven by the quantitative analysis, we set out to design a virtual memory system for MCM GPUs with the following three goals. ① Leverage the aggregate capacity of the L2 TLB slices across chiplets when needed, ② Limit accesses to remote L2 TLB slices, and ③ Limit remote memory accesses due to page table walks.

Toward these, we enhance GPU's virtual memory in three ways. First, we dynamically change the HSL at the time of *each kernel launch*, guided by LASP's placement of data pages. We call this optimization dynamic HSL (dHSL). Recall that the HSL decides the home L2 TLB slice responsible for serving a given virtual address. For example, suppose LASP finds that the CTAs of a kernel mostly access non-overlapping address regions (data), i.e., classified as NL. In that case, LASP schedules the CTAs and the data pages it would access on the same chiplet. We leverage the same insight gathered by LASP to ensure that the HSL maps the corresponding virtual address regions onto the same chiplet. Consequently, kernels enjoy local L2 TLB accesses (hits) as in a private TLB. In contrast, if the LASP decides to stripe data across chiplets, for example, when it encounters a kernel with ITL or is unclassified (Section II), the HSL would also accordingly map the virtual address ranges across chiplets.

This approach of LASP-guided HSL addresses the first two goals leveraging the aggregate capacity of the TLB slices and limiting remote TLB accesses. Notice that private TLB sacrifices capacity vis-a-vis shared TLB as it allows duplicate entries across L2 TLB slices of chiplets. However, dHSL maps a given virtual address onto a *single* L2 TLB slice only. When LASP decides to spread the data across chiplets, dHSL will also leverage the aggregate capacity of all chiplets' L2 TLB slices. At the same time, by ensuring L2 TLB slices are responsible for servicing translation requests from CTAs on the same chiplet wherever possible, LASP-guided HSL minimizes remote L2 TLB accesses.

Next, we focus on limiting remote memory accesses during page walks (a.k.a. remote page walks). Recall that the HSL maps a virtual address to a *home* chiplet. The PTWs on the home chiplet are responsible for translating virtual address regions mapped onto it by the HSL. The pages containing the PTEs corresponding to those address regions are placed on the home chiplet's memory to limit remote page walks. We focus on placing PTEs from the leaf level of the page table since PWCs often filter out the accesses to the upper levels of the page table.

It is evident from the above discussion that the HSL and the PTE placement strategy must work together to limit remote

page walks. However, notice that HSL can map VA ranges at the minimum granularity of a page, here, 4KB. Further, a single 4KB page with PTEs contains 512 PTEs. Thus, a page containing leaf-level PTEs covers the address translation for a 2MB of the contiguous virtual address range. Therefore, to ensure synergy between HSL and PTE placement, we enhance the dHSL to map virtual addresses at a coarser granularity of 2MB across L2 TLB slices of chiplets (named dHSL-coarse). At a kernel launch, the GPU driver consults dHSL-coarse to find the home chiplet for a given 2MB VA range. It then places the 4KB page containing the leaf-level PTEs for that 2MB region on that home chiplet's memory.

The enhancements mentioned above work well for most applications but not for all. There are kernels where nearly every CTAs from all chiplets concurrently access a relatively small VA range ($<$ 2MB). Since dHSL-coarse maps at least 2MB of VA range onto one L2 TLB slice, address translation requests for those small but frequently accessed data structures, queue up at a *single* TLB slice. This queuing overwhelms the L2 TLB slice's resources, including the ports and MSHRs on one chiplet, while the other L2 TLB slices remain unloaded. Unfortunately, such behavior of a kernel is not necessarily discernible at the time of compilation. Thus, we extend the GPU hardware to monitor the system for imbalance in the L2 TLB traffic amongst the chiplets at runtime and switch to finer grain (4KB) mapping to ensure that one TLB slice does not become a choke point.

Note that giving up coarser granularity mapping (dHSL-coarse) sacrifices the benefits of local memory accesses on page walks. However, if the L2 TLB hit rate is high, then ensuring balance in L2 TLB utilization across chiplets could be more important than reducing remote page walks. Therefore, this enhancement, acronym-ed dHSL-balance, switches to finer grain mapping of virtual addresses across the L2 TLB slices if it encounters an imbalance in traffic among the L2 TLBs *and* finds the L2 TLB hit rate is high.
Summary: We achieve the three goals by: ① LASP-directed per-kernel configuration of the HSL to leverage the aggregate L2 capacity when needed, while benefiting from local L2 TLB lookup where possible (dHSL). ② Placing pages containing PTE on chiplets in coordination with the dHSL and the coarser grain (default at 2MB) mapping of virtual addresses to L2 TLB slices to limit remote memory accesses on page walks (dHSL-coarse). ③ Switching to finer-grain (default, 4KB) HSL at runtime to avoid congestion in one of the L2 TLB slices (dHSL-balance). We name the overall solution MCM-aware GPU virtual memory (MGvm).

V. Design and Implementation

We elaborate on how MGvm realizes the ideas discussed in the previous section. There are three major components in the design and implementation. ① At a kernel launch, the HSL is decided based on the static analysis for data page placement as in LASP. This information is conveyed to the GPU to configure the HSL in the hardware that determines which chiplet's L2 TLB (and, thus, page walkers) are responsible for translating a given virtual address. The static analysis is performed once per kernel, exactly as in LASP [31]. While LASP uses it for data page placement, MGvm uses it for deciding the HSL. ② At a kernel launch, the pages with page tables are placed as per the HSL chosen in the previous step. The CTAs are scheduled on the chiplet where their data resides, as in LASP. ③ During the execution of a kernel, the enhanced GPU hardware monitors possible imbalance in the L2 traffic. On encountering severe imbalance, the hardware employs dHSL-balance whereby the HSL falls back to fine-grain interleaving of virtual addresses across L2 TLB slices. Figure 2c depicts the operation of dHSL and dHSL-coarse, and figure 6 shows the switching logic for dHSL-balance.

Now, we describe the actions that happen upon a kernel launch and during the kernel's execution.

Upon a kernel launch: MGvm's first task is to decide the HSL function guided by LASP's data placement. An application may have multiple kernels, and MGvm can set a different HSL function for each kernel. LASP performs static analysis of a GPU kernel's code at compile time to infer its memory access pattern. Based on the analysis, LASP classifies the kernel into different categories (Section II). It interleaves data structures across the chiplets based on the expected access pattern and schedules the CTAs on the same chiplet where its data is placed.

MGvm leverages LASP's analysis to determine the most suitable HSL for a kernel to limit remote L2 TLB accesses and remote page table accesses. A page containing PTEs maps (at least) 2MB of contiguous virtual address (VA) region [1]. Thus, MGvm sets the HSL to map a VA region in multiples of 2MB onto a single chiplet while following the interleaving pattern selected by LASP. The need for HSL to interleave across a coarser granularity (2MB vs. 4KB) stems from its goal to reduce remote memory accesses on page walks. If LASP had selected an interleaving granularity for the data pages that is not a multiple of 2MB, then MGvm rounds it up to the closest multiple of 2MB for HSL (a.k.a, dHSL-coarse). If LASP's data interleaving granularity is a multiple of 2 MB, then MGvm guarantees that if a data access is local, the corresponding address translation request would be local too. However, if MGvm rounds up LASP's data interleaving granularity to a multiple of 2MB, then some of the accesses to L2 TLB can be remote while the data accesses are local.

Pseudo-code in Listing 1 shows the high-level algorithm used by MGvm to determine the HSL and place page tables (discussed later). Upon a kernel launch, the driver first queries the available LASP information about different memory allocations (line 1). Then it picks up the *largest*

[1]We assume 4KB default page size. Later in this section, we discuss how MGvm handles larger page sizes.

memory allocation of the kernel and queries LASP for the data placement policy used for the particular allocation (lines 2-3). Notice that MGvm focuses on ensuring local L2 TLB accesses for the largest data structure only, as it is likely to have the most significant effect on the performance of the kernel [31]. LASP decides the interleaving for each memory allocation independently. However, MGvm cannot similarly decide the HSL for each allocation independently without significantly complicating the HSL in the hardware. When an L1 TLB miss occurs, the HSL should quickly find which L2 TLB slice (thus, chiplet) the VA of that translation request maps to. To avoid hardware complexity and yet be effective in practice, MGvm employs a common HSL for the entire VA space of a kernel.

Notice that while LASP can distribute each memory allocation at a different granularity across chiplets, it focuses primarily on avoiding remote data accesses for the largest data structure. This is because data page placement alone is insufficient to ensure local accesses. LASP schedules the CTAs on chiplets to minimize remote data access to the largest data structure. We follow the same rationale to optimize for the largest data structure.

In listing 1, we define **LASPBlockSize** as the block size used by LASP for interleaving data across chiplets, for the largest memory allocation. If LASPBlockSize is a multiple of 2MB, then MGvm also uses the same as the HSL. Otherwise, MGvm chooses the closest multiple of 2MB as the HSL (lines 4-7). The output of this algorithm is the selected interleaving granularity (dHSL-coarse) used by the hardware to set its HSL before the given kernel starts executing (line 22).

The alignment of the virtual address ranges allocated to different data structures can impact the effectiveness of MGvm in reducing remote TLB lookups, and remote page walks. Suppose there are two data structures of sizes 4MB and 8MB for a given kernel. Say LASP chooses to use block sizes of 1MB and 2MB, respectively, for the two allocations. Since the latter allocation is the larger, MGvm decides to use 2MB granularity in HSL. Now say the allocation starts at VA 0 (zero). The first memory allocation (4MB) starts at VA 0, and the second memory allocation (8MB) gets the VA from 4MB to 12MB. In this case, the HSL would map the first 2MB chunk of the 8MB data structure onto the third chiplet's L2 TLB. However, the corresponding data pages will be placed on the first chiplet by LASP. Assuming LASP's static analysis correctly finds CTAs that will access that VA region and schedules them on the first chiplet, it will ensure local data accesses. However, the L2 TLB lookups for the given VA region will travel from the first chiplet (where the execution occurs) to the third chiplet. Thus, if the allocated VA ranges are improperly aligned, HSL may be ineffective in reducing remote L2 TLB accesses.

MGvm's driver takes care of the alignment needs as follows. ① The starting virtual address for the memory allocations in the program is aligned to the nearest power-of-2, larger than the largest memory allocation in the program. This is easy since the virtual address space is abundant. ② MGvm rearranges the VA assignments for each memory allocation request such that the largest allocation request is assigned VA first (lines 9-12). These ensure that the selected HSL will produce local L2 TLB lookups for the largest allocation, even with multiple allocations of varying sizes, as long as LASP analysis is accurate.

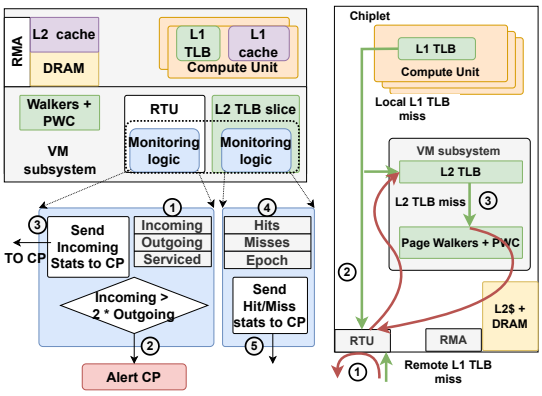
```
1  listOfAllocs = QueryLASP (LIST_OF_ALLOCS)
2  largestAlloc = max(listOfAllocs)
3  LASPBlkSize = QueryLASP(BLOCKSIZE, largestAlloc)
4  if LASPBlkSize % 2M == 0
5    dHSLcoarse = LASPBlkSize
6  else
7    dHSLcoarse = closestMultiple(LASPBlkSize, 2M)
8
9  # Allocate virtual and physical memory
10 alignTo = closestPow2(largestAlloc)
11 StartingVPN = newAlignedVAddr(alignTo)
12 sort(listOfAllocs)
13 for alloc in listOfAllocs
14   vAddr = allocateVAddr(alloc)
15   allocatePhyMemLASP(vAddr, alloc)
16
17 # Each page table page covers 2MB VA
18 foreach 2MB region
19   VPN = startAddr(region)
20   block = VPN / dHSLcoarse
21   homeNode = block % numChiplets
22   allocatePTEPageOnChiplet(VPN, homeNode)
23
24 GPU.SetdHSL(dSHLcoarse)
```

Listing 1: Pre-kernel launch steps

Next, the driver allocates the virtual addresses and physical memory to the different allocations (lines 13-15). Finally, the driver places the pages with PTEs such that remote memory accesses to the leaf-level PTEs are avoided during page walks (step ① in Figure 2c). Lines 17-22 show the pseudo-code for the same. The allocation for the PTEs is made simple by the fact that dHSL-coarse always allocates contiguous VA regions of multiples of 2MB on the same chiplet. For every allocated 2MB VA region, the driver finds the chiplet whose L2 TLB slice would be responsible for its address translation as per the chosen HSL function (lines 19-21). It then allocates the corresponding PTEs such that the page containing the leaf-level PTEs resides on the chosen chiplet. The upper-level PTEs may reside on any chiplet as their placement is not critical to the performance.

Operation during kernel execution: The command processor (CP) in a GPU schedules CTAs to different chiplets as per LASP to reduce remote data accesses. MGvm does not make any modifications to this. The chosen HSL and PTE page placement limits remote L2 TLB accesses and remote page walks, utilizing LASP's CTA scheduling.

Before the kernel starts executing, the CP broadcasts the HSL's granularity to all the L1 TLBs and the RTUs. The L1 TLBs and the RTUs use the same to find any given VA's home TLB slice (chiplet). Under MGvm, L1 TLB misses are



(a) Per-chiplet monitoring logic (b) Routing logic

Figure 6: Switching logic

```
global bool prevImbalance;

let numIncoming[4] be num. incoming reqs at each
     chiplet
let hitRate = totalNumHits / totalNumAccesses
let imbalance = False

total = sum(numIncoming)

for nI in numIncoming
    if (nI/total) > 0.8
        imbalance = True

if imbalance == True and hitRate > 0.9
    if prevImbalance == True
        send SwitchMsg(TLBs, RTUs)
    else prevImbalance = true
else prevImbalance = false
```

Listing 2: Decision flow at the CP

typically directed to the local L2 TLB slice in the common case, thanks to dHSL (step ② in Figure 2c). Further, dHSL-coarse ensures that leaf-level PTE accesses remain local (④ in Figure 2c).

Mitigating L2 TLB imbalance: The primary task of MGvm during the execution of a kernel is to monitor for potential imbalance in the L2 TLB traffic that can hurt performance. As noted in Section IV, coarse granularity mapping in dHSL-coarse is to blame in such cases. Kernels that typically enjoy higher L2 hit rates are often most affected by imbalance. If the L2 MPKI is high, then the address translation overhead is dominated by the long latency page walks, and thus, imbalance at L2 TLBs matters less.

MGvm extends the GPU hardware to monitor the signs of imbalance in the presence of high L2 TLB hit rates. Upon detecting such a scenario, MGvm abandons dHSL-coarse and switches to fine-grain (here, 4KB) interleaving (dHSL-balance) to distribute the L2 traffic evenly across all chiplets. MGvm defines imbalance at a chiplet as follows.

$$imbalance := \frac{IncomingRequests}{TotalRemoteRequests} > threshold$$

We assume the number of chiplets in the GPU to be 4 in the following discussion without loss of generality. Figure 6a shows the per-chiplet monitoring logic. MGvm monitors the number of incoming and outgoing translation requests with a pair of counters at each RTU (① in Figure 6a). Recall that all translation requests that travel to another chiplet pass through the RTU (Section II). Another counter maintains the total number of requests (incoming and outgoing) serviced. At the end of an epoch (default, 5000 requests), each RTU saves the values of the counters and resets them to 0. Then, based on the saved values of these counters, each RTU determines if there is any possibility of imbalance across the system.

$$Possible := IncomingReqs > 2 * OutgoingReqs$$

If the RTU detects a possible imbalance for two consecutive epochs, the RTU sends a *trigger* message to the CP (② in Figure 6a). Upon receiving the message, the CP requests all the RTUs to share their counter values of incoming requests. Further, the CP requests all L2 TLB slices to send the number of hits and misses over the last epoch (④ in Figure 6a). The RTUs and the L2 TLB slices reply with the requested information (③, ⑤ in Figure 6a). Once the CP receives the data from all the RTUs and TLBs, it combines the information to infer the existence of imbalance (if any) across chiplets. Listing 2 shows the algorithm used by the CP to decide on switching. If the imbalance amongst the chiplets is higher than a threshold (default, > 0.8), and the L2 TLB hit rate is high (default, > 0.9), for two consecutive epochs, the CP decides to switch to fine-grain dHSL-balance.

When the CP decides to switch to dHSL-balance, it broadcasts an HSL *switch* message to all the L1 TLBs, RTUs and L2 TLBs. Once the RTUs and L1 TLBs receive it, they *asynchronously* switch to the fine-grain HSL.

Mechanism to switch to dHSL-balance: Switching to a different HSL is not instantaneous. Translation requests already in the network must be rerouted appropriately according to the latest HSL. Figure 6b illustrates the additional paths that a translation request can traverse in MGvm (in red) in addition to the baseline (in green). Due to asynchronous switching, an RTU may receive requests from other chiplets that should not have been routed to it as per the updated HSL. The RTU assumes that its currently effective HSL (e.g., dHSL-coarse) is the HSL currently used by all components and reroutes the request according to its current HSL (① in Figure 6b). Since all chiplets eventually receive the switching message from the CP, rerouting happens only a bounded number of times (typically, once). Further, delays within a chiplet may lead to a request being (incorrectly, as per the latest HSL) routed to the RTU instead of the local L2 TLB. The RTU assumes its HSL to be the latest and routes the request back to the local L2 TLB slice (② in Figure 6b).

The L2 TLB slice on each chiplet also maintains the latest HSL. The L2 TLB looks up all the requests it receives. If the lookup is a hit, no additional action is needed. On a miss,

the L2 TLB slice sends the request to the local page walk system *only* if its copy of the HSL determines that it should serve that request. Otherwise, it redirects it to the RTU (③), which, in turn, sends it to the request's home L2 TLB slice.

As Figure 6b shows, the asynchronicity of components (even within the same chiplet) can cause translation requests to reroute. Suppose the RTU and the L2 TLB slice on a chiplet temporarily disagree on the HSL. Then a translation request may be rerouted between the RTU and L2 TLB slice. A similar situation may arise when RTUs on different chiplets disagree about the HSL in use. However, this is a *transitory* phase without the possibility of deadlock since the RTUs, and L2 TLB slices of each chiplet will eventually receive the switching message from the CP. We empirically found that switching messages constitute a minuscule fraction ($< 0.1\%$) of the network traffic due to address translation.

Switching back to dHSL-coarse: MGvm can switch back from fine-grain dHSL-balance to the original driver-selected dHSL-coarse if the imbalance disappears. Every L2 TLB entry is tagged with the chiplet ID that the corresponding VPN would have hashed under dHSL-coarse during the TLB fill operation. Additionally, we keep four counters, one representing each chiplet on the GPU. Whenever an L2 TLB entry is accessed, the counter corresponding to the chiplet ID stored in that L2 TLB entry is incremented. At the end of every epoch, these counters are checked to see if the imbalance has disappeared (i.e., no chiplet has an imbalance greater than a threshold, default 0.5). If two consecutive epochs show that the imbalance has disappeared, we switch back to dHSL-coarse. We do not observe the need to switch back in our evaluated applications.

Notice that since TLBs are read-only caches, there can be no correctness issue even when different slices use different HSL during switching. Switching also does not require a TLB shootdown since the page tables remain unaltered. We empirically found that it has little performance impact as it is a transitory phase. We evaluated a hypothetical configuration that magically avoids all costs of switching. We found a minuscule performance difference ($< 1\%$) between this magical configuration and MGvm.

Hardware overheads: MGvm requires one additional register (to store the HSL parameter) per each CU, RTU, and L2 TLB to compute the HSL. Each RTU requires five 32-bit counters, one for maintaining the epoch, two for maintaining the number of incoming and outgoing requests, and two more for maintaining the number of incoming and outgoing requests in the previous epoch. Each L2 TLB entry is augmented with $log(numChiplets)$ bits. Each L2 TLB also maintains $numChiplets$ counters. In an MCM GPU with four chiplets, MGvm needs an additional 32 bits per CU (4096 across the entire GPU), 1152 bits across the RTUs and L2 TLB counters, and 4096 bits for the L2 TLB entries. In aggregate, the total state overhead is 9344 bits.

MGvm with larger page sizes: Hitherto, we assumed 4KB pages. However, MGvm's design is not tied to the page size. The page table's radix tree structure remains the same when using large pages (say 64 KB pages) for data. The page tables pages continue to use base 4KB pages [42]. Thus, the basic algorithms remain the same. The most significant changes are: ① The dHSL-coarse would now have to be the multiples of 32MB instead of 2MB, since each page with PTEs would now cover 32MB of contiguous virtual address space. ② The thresholds for switching the HSL may change due to higher expected TLB hit rates. In Section VI, we evaluate MGvm with 64KB pages.

Table I: Simulation Parameters

Chiplets	4 chiplets, 32 CUs per chiplet 16 memory controllers per chiplet
CU	4 exec. units per CU, 64 threads per wavefront
Cache (per CU)	64KB vector cache, 5 cycle lookup 32KB Inst. cache shared b/w 4 CUs 16KB Scalar cache shared b/w 4 CUs
TLB (per CU)	32 entry fully assoc., 1 cycle lookup
L2 TLB	512 entry TLB per chiplet, 8 way, 10 cycle lookup 64 entry MSHR per chiplet
L2 Cache	4MB per chiplet, 16 banks, 16 way, 12 cycle lookup
DRAM	1 TBps, 100 ns latency
Page walking	16 page walkers per chiplet (default) 32 entry fully assoc. page cache, 10 cycle lookup
Interconnect	768 GBps, bi-directional, ~32 ns latency

VI. Evaluation

A. Methodology

We use the MGPUSim [59] simulator to model the MCM GPU. MGPUSim is designed to model multiple interconnected GPUs. We simulate a GPU with characteristics similar to AMD's GCN architecture. Our simulated GPU has 4 chiplets connected to each other via a fast, high-bandwidth interconnect. Each chiplet has 768 GBps of bi-directional bandwidth to every other. The latency to cross from one chiplet to another is 32ns (default). We follow the numbers published by NVIDIA for these parameters [7]. Table I details the simulation parameters. We follow the strategy detailed in LASP [31] for data placement and CTA scheduling.

We use a mix of workloads from various suites, e.g., Polybench [48], AMD App SKD [4], Heteromark [60], SHOC [23], Pannotia [20]. Table II shows the workloads, their classification according to LASP, and their memory footprints. To present a holistic evaluation, we include workloads that stress the virtual memory and those that do not.

B. Results and analysis

Figure 7 shows the normalized throughput (higher is better) for four different configurations – private TLB, shared TLB, MGvm-no-balance, and MGvm. MGvm-no-balance is our proposed solution, but without balancing, i.e., dHSL-balance is disabled and only dHSL and dHSL-coarse enabled. The throughput is measured as the total number of instructions executed divided by the total time. The height of each bar is normalized to the private TLB configuration.

Table II: Workloads

Abbr.	Benchmark	Size (MB)	Locality Type
C2D	2-D convolution [48]	512	NL
FW	fast Warshall transform [4]	32	RCL
GUPS	multi-threaded, random access	16	unclassified
J1D	1-D Jacobi solver [48]	512	NL
J2D	2-D Jacobi solver [48]	128	NL
KM	kmeans clustering with 20 clusters [60]	128	ITL
MIS	max. independent set [20]	16	NL+ITL
MT	matrix transpose [4]	32	NL
PR	Page Rank algorithm [60]	256	ITL
SC	simple convolution on a matrix [4]	512	NL
RED	reduction kernel [23]	256	NL
SPMV	sparse matrix vector multiplication [23]	360	ITL
S2D	stencil operation on 2-D matrix [23]	32	NL
SYRK	rank of a symmetric matrix [48]	32	RCL
SYR2	rank-2k of a symmetric matrix [48]	16	RCL

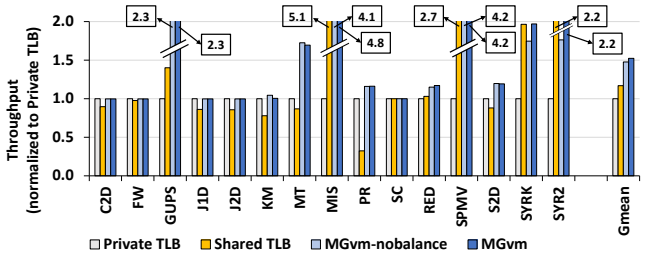


Figure 7: Throughput of private TLB, shared TLB, MGvm-no-balance and MGvm, normalized to private TLB

Table III: L2 TLB MPKI with different configurations

Workload	Private TLB	Shared TLB	MGvm
C2D	1.07	1.07	1.07
FW	2.28	2.28	2.28
GUPS	698.32	480.82	513.27
J1D	3.21	3.21	3.21
J2D	2.16	2.15	2.15
KM	11.04	10.25	21.70
MT	69.31	62.00	68.5
MIS	260.52	2.11	8.50
PR	91.33	90.38	90.83
SC	0.40	0.40	0.40
RED	6.01	5.93	5.93
SPMV	1531.47	422.73	451.94
S2D	12.32	10.24	12.18
SYRK	201.46	53.03	53.17
SYR2	178.35	54.72	54.95

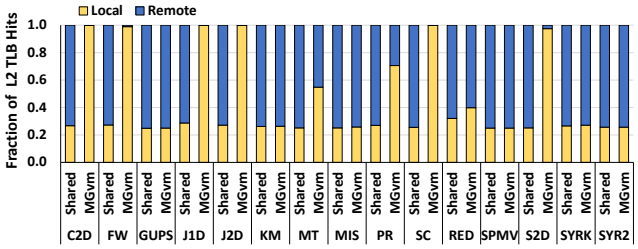


Figure 8: Reduction in remote TLB hits with MGvm compared to shared TLB

We first observe that MGvm outperforms private TLB by 52% and shared TLB by 30%, on average (geometric mean). Even if we compare MGvm to the better-performing configuration among private TLB and shared TLB for individual applications, MGvm outperforms the better alternative by 12%, on average. In short, MGvm not only matches the better among private and shared TLB for each application but also performs better than any of them. While comparing MGvm-no-balance and MGvm, we observe the need to detect and correct imbalance in three of the applications – MIS, SYRK, and SYR2.

Next, we dive into the sources of these improvements. Note that MGvm aims to achieve a balance between three objectives: ① lower L2 TLB miss rate by leveraging aggregate L2 TLB capacity, ② increase local L2 TLB hit fraction to limit remote TLB lookups, and ③ limit remote memory accesses due to page walks to make walks faster. We will analyze MGvm against each of these objectives.

Table III shows the L2 TLB misses per kilo SIMD instructions (MPKI). As expected, the best aggregate capacity, and thus, the lowest MPKI, is achieved under shared TLB. We observe that, in most cases, MGvm achieves MPKI close to that of the shared TLB design. For example, applications such as MIS, SPMV, SYRK, and SYR2, which suffer many L2 misses in private TLB, leverage the aggregate L2 capacity under MGvm. Surprisingly, the MPKI of KM is higher with MGvm than private TLB. On further investigation, we find that MGvm reduces the page walk latency of KM by a large fraction (Figure 10). This leads to frequent L2 TLB fill as each completed page walk brings a new entry into the TLB. This, in turn, leads to frequent eviction of useful TLB entries, i.e., thrashing of the L2 TLB. We further confirm that the MPKI of KM increases with a hypothetical configuration where all remote page walks are made local (and hence faster). The phenomenon of thrashing due to faster page walks has been observed before [49].

Figure 8 shows the breakup of local versus remote L2 hits for shared TLB and MGvm. Note that all L2 hits are local for private TLB and thus are not shown. The figure shows that applications that benefit from a high local L2 TLB hit rate in private TLB also experience good local L2 hit rate under MGvm relative to shared TLB. This is evident in applications such as C2D, J1D, J2D, S2D, SC, and MT. In short, MGvm can leverage the benefits of private TLB where it suits the application.

Finally, we evaluate if MGvm is able to reduce remote memory accesses during page walks. Figure 9 shows the breakdown of local versus remote accesses to PTEs for private TLB, shared TLB, and MGvm. Across all applications, except SYRK and SYR2, MGvm provides the lowest fraction of remote accesses to PTEs, establishing its success in achieving the third goal. For SYRK and SYR2, the remote access fraction with MGvm is similar to shared TLB. This is because these applications switch to using finer grain (4KB) to avoid

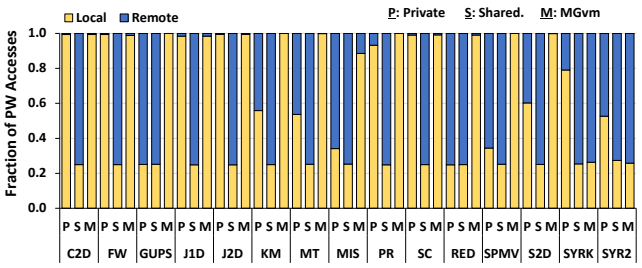


Figure 9: Ratio of local vs. remote page table accesses

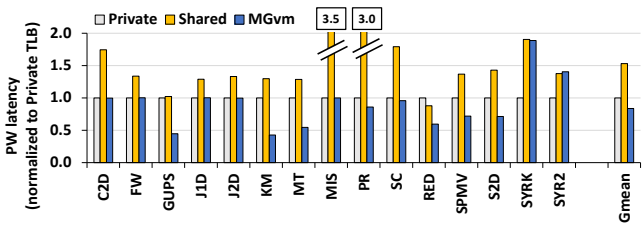


Figure 10: Page walk latency with various configurations

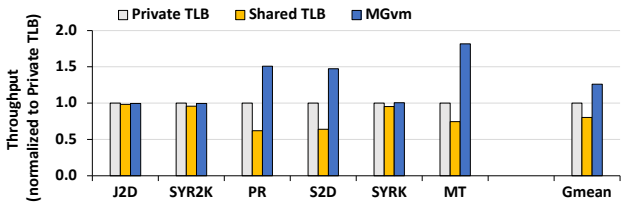


Figure 11: Throughput with large pages

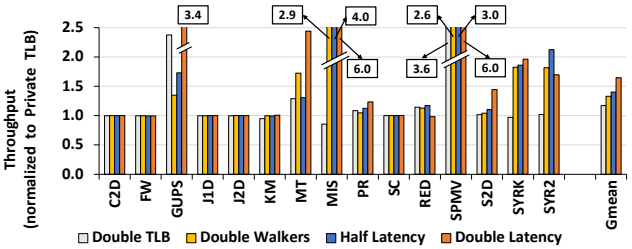


Figure 12: Throughput of MGvm normalized to private TLB

imbalance in L2 TLB traffic early in their executions. Giving up coarser grain (2MB) mapping to L2 TLB slices sacrifices the ability to keep accesses to leaf-level PTEs local. However, MGvm finds this necessary to limit imbalance for maintaining good overall performance.

In Figure 10, we further show how reducing remote PTE accesses speeds up page walks across configurations. This confirms the benefits of limiting remote accesses to PTEs.

In summary, MGvm matches or betters the best of both worlds – private TLB and shared TLB, even when the baseline contains state-of-the-art optimizations for CTA scheduling and data placement, as well as smart page table placement that follows data placement.

C. Sensitivity studies and generality

Large pages: We study the impact of MGvm with 64KB and 2MB large page sizes that NVIDIA GPUs support [42]. We increased the memory footprints of workloads to generate enough TLB misses to stress the virtual memory subsystem. Unfortunately, these large-footprint simulations run for many days. Figure 11 shows a subset of applications running private TLB, shared TLB, and MGvm with 64KB pages. These applications finished simulation within a reasonable time. Even with larger 64KB pages, MGvm remains effective, outperforming the better alternative for each application by 26%, on average. The speedup is as high as 82% for matrix-transpose (MT). We also experimented with 2MB pages (not shown). As expected, the headroom for improvement reduces with a larger page size. However, MGvm remains useful by consistently providing more than 15% performance improvement over private TLB or shared TLB for applications with high address translation overheads.

Sensitivity to TLB sizes, number of walkers, and interconnect latency: We studied the impact of larger L2 TLB sizes, more page table walkers, lower and higher interconnect latencies. Specifically, we studied the impact of doubling the size of each L2 TLB slice to 1024 entries (a total of 4096 entries). We studied the impact of doubling the number of page table walkers to 32 per chiplet (a total of 128). We also studied the impact of halving (16 ns) and doubling (64 ns) the inter-chiplet interconnect latencies.

Figure 12 shows the throughput of applications with MGvm under these varying configurations, with the height of each bar, normalized to the throughput under private TLB (higher is better). Figure 13 shows the same with the heights of the bars normalized to shared TLB. As expected, with more virtual memory resources (TLBs, page walkers), the relative improvement with MGvm moderates but remains significant. For example, even after doubling the TLB size, MGvm outperforms private and shared TLB by 12% and 20%, respectively, on average (geomean). After doubling the number of walkers, the improvements are 38%, and 21%.

As expected, when the interconnect latency is halved, the effect of non-uniformity in the MCM design reduces. Thus, improvements with MGvm moderate at 43% and 17%, on average, over the private and shared TLB designs. The opposite trend is observed when the interconnect latency doubles. We see 68% and 54% improvements over private TLB and shared TLB with higher interconnect latency.

The generality of MGvm: Thus far, we assumed LASP [31] for the CTA scheduling and data page placement policy in the baseline design. While MGvm leverages LASP's static analysis, it can provide benefits even if the static analysis is unavailable.

We simulate an *alternative* baseline design where CTAs and the data pages are distributed in a round-robin fashion across the CUs and the memory banks in the absence of LASP. We call this the *naive* baseline. The optimization in

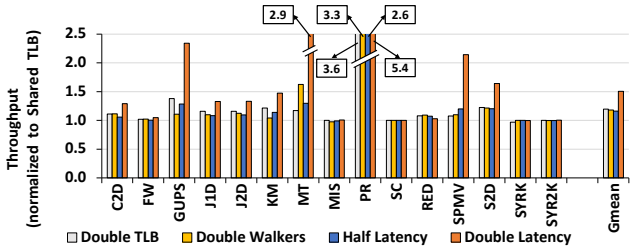


Figure 13: Throughput of MGvm normalized to shared TLB

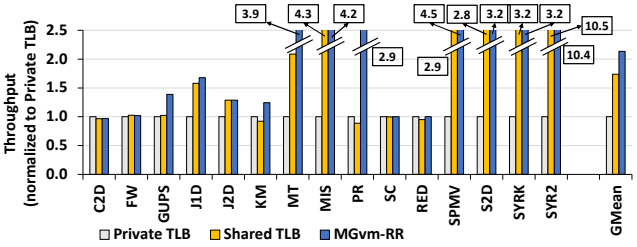


Figure 14: Application of our techniques to naive baseline with round-robin scheduling

MGvm that leveraged LASP's analysis to dynamically select HSL to avoid remote TLB lookups is inapplicable. However, the other optimization to place PTEs for reducing remote page walks remains relevant. We name this constrained version of MGvm, MGvm-RR (round-robin).

Figure 14 shows the performance of MGvm-RR over private TLB and shared TLB with the naive baseline. First, we observe that without LASP, private TLB is not a good design choice since it suffers from high L2 TLB miss but fails to fully harness local TLB lookup due to its inability to exploit locality. MGvm-RR outshines private TLB by 113%, on average. The shared TLB performs relatively better as it leverages the aggregate capacity of L2 TLB slices. However, it also suffers from a high percentage of remote memory accesses to PTEs. MGvm-RR improves throughout over the shared TLB by 26%, on average, as it limits remote page walks by smartly placing the PTEs.

In general, we find that when the data access locality is improved through smart data placement and CTA scheduling, the average data access latency reduces. This brings forth the address translation as a more significant performance bottleneck. However, even in the absence of smart data placement and CTA scheduling, and even without dynamic HSL optimization, MGvm can provide significant speedups.

VII. Discussions

Why not replicate page tables? A key goal of MGvm is to avoid remote memory accesses while walking the page table. An alternate way to achieve the same is by replicating the entire page table onto each chiplet's memory. We simulated replication of the page table by making all accesses to the page table local (i.e., zero remote accesses to PTE). We evaluated this for both private TLB and shared TLB designs.

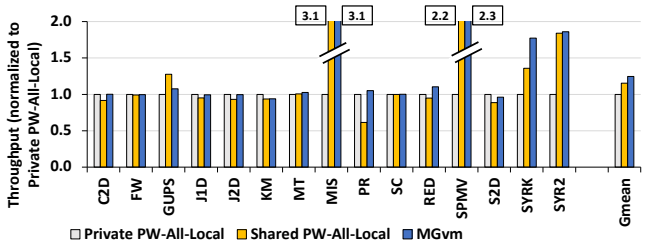


Figure 15: Comparison with Page Table Replication (PTR). Throughput normalized to private TLB with PTR.

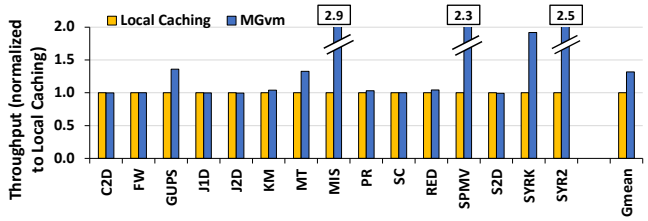


Figure 16: Impact of local caching of remote L2 TLB entries

Page table replication can improve performance by 23% and 20%, over private and shared designs. Importantly, MGvm outperforms private and shared TLB designs with page table replication by 24% and 8%.

Figure 15 shows three bars for each application – representing throughput with page table replication with private TLB and shared TLB (Private PW-All-Local and Shared PW-All-Local in the figure), and with MGvm. Recall that MGvm also limits remote L2 TLB lookups and leverages aggregate L2 TLB capacity of the chiplets, besides limiting remote accesses to PTEs during page walks. MGvm outperforms private TLB with page table replication due to better L2 TLB hit rate. It outperforms shared TLB with page table replication thanks to fewer remote L2 TLB lookups. Importantly, MGvm achieves this without the complexity of keeping all the replicas of the page table consistent. Today's MCM processors already have 4 chiplets [2], and this number will only increase [57]. Keeping many replicas of a page table coherent, especially with unified virtual memory, would be challenging.

Caching remote TLB entries: Another design alternative is to cache remote L2 TLB entries in the local L2 TLB slice of the requesting chiplet. Caching remote TLB entries locally can reduce remote L2 TLB lookups. We studied whether caching remote TLB entries in the requesting chiplet's L2 TLB slice can improve performance.

Figure 16 reports our findings and compares page table replication with MGvm. We observe that caching remote TLB entries hurts performance compared to MGvm by 24%, on average. We find that the duplication of entries across L2 TLB slices due to caching of remote entries is to blame. Duplication decreases the effective aggregate capacity of the L2 TLB, thus, increasing the L2 TLB MPKI by 60% over MGvm. In particular, applications that benefited from larger

TLB capacity (MT, MIS, SPMV, SYRK, and SYR2) show significant performance loss with caching of remote TLB entries. In contrast, MGvm, by design, ensures that there is no duplication of entries, since an L2 TLB entry is mapped only onto one L2 TLB slice.

Applicability to unified virtual memory (UVM): MGvm's two key optimizations – dynamically selecting HSL to ensure local TLB lookups for local data accesses and page table placement to ensure local accesses to PTEs, are equally applicable under UVM, albeit with a slightly different implementation. The static analysis that determines the data placement, CTA scheduling (as in LASP [31]), and the dHSL in MGvm, remain unchanged under UVM. At a kernel launch, the driver sets up dHSL as before. Unlike typical pinned memory allocation in GPU (e.g., using cudaMalloc), the data pages may be allocated during a kernel's execution on encountering page faults. Thus, UVM's page fault handler should be extended for MGvm to allocate the pages containing PTEs following MGvm's principle, i.e., on the chiplets whose L2 TLB slice is responsible for translating the VA region mapped by those PTEs. In summary, MGvm's principles are equally applicable under UVM, but its implementation should extend to the page fault handler.

VIII. Related work

The background section describes the most closely related prior works. We discuss several more here.

HMG [51] tackles the problem of coherence across a multi-(MCM)GPU system. Vijaraghavan et al. [64] discuss an AMD APU which uses MCM for scaling. Loh et al. [37] discuss the benefits and drawbacks of various approaches to building chiplet-based processors. Simba is a prototype MCM deep learning accelerator [53]. These prior works did not study the impact of the MCM design on virtual memory.

Researchers have proposed various techniques to reduce address translation overheads in GPUs, such as the coalescing of TLB requests [38], warp scheduling [47], direct mapping [27], and virtual caches [66]. Vesely et al. studied the effect of memory access divergence on a GPU's virtual memory [62]. Mosaic explored the overheads of demand paging and issues with huge pages [9]. Ducati used last-level caches to store address translations [28]. Shin et al. studied the re-ordering [54] and coalescing [55] of page walks. Tang et al. leverage the compressibility of TLB entries to improve hit rates [61]. Valkyrie proposed probing and pre-fetching into L1 TLBs [14]. Kotra et al.leverage under-utilized resources in CUs to cache address translations [34] . Li et. al study the TLB hierarchy of a multi-GPU system and suggest methods for effectively utilizing multiple levels of TLBs under multi-tenancy [35]. MASK used L2 TLB tokens and selective bypassing of the L2 TLB to avoid thrashing under multi-tenancy [10]. DWS studied the impact of multi-tenancy on page walkers [10]. ETC studied pre-fetching and eviction policies under unified memory over-subscription [36]. Kim et al. proposed batching page faults for reducing unified memory overheads [33]. ActivePointers is a software mechanism for mapping files onto GPU address space [52]. None studied the virtual memory in MCM GPUs, though.

CARVE caches remote data for multi-GPU systems in the local DRAM for lowering memory access latency [67]. Milic et al. modify caching and interconnect policies according to application phase behaviour [40]. Griffin balances memory distribution by migrating pages across GPUs [13]. These works explore multi-GPUs, not MCMs.

Mitosis replicates page tables across all sockets in a multi-socket CPU system [1], while vMitosis extends it to work under virtualization and enables PTE migration [45]. In contrast, we avoid the complexity of replicating page tables but still achieve mostly local page walks. MIXTLB explored mechanisms to support multiple page sizes on a single TLB [21]. CoLT exploited virtual address space contiguity to map multiple address translations to a single TLB entry [46]. SpecTLB uses address translation speculation to hide page walk latency [12]. Reactive NUCA and Cooperative Caching [19], [26] carefully place and spill cache lines in NUCA caches to improve cache hit rates and performance. Bhattacharjee et al. suggested exploiting commonality in TLB miss patterns across CPU cores to pre-fetch TLB entries [18], while Mazumdar et al. proposed predicting dead entries in TLBs to reduce TLB misses [39]. Previous works have also explored using segments to avoid page table lookups or use very large page sizes [15], [16], [17], [24], [30], [50]. However, none of these works explore the effect of MCM's non-uniformity on the virtual memory system.

IX. Conclusion

We quantitatively analyzed the impact of MCM designs on a GPU's virtual memory system. We demonstrated that remote TLB lookups and remote memory accesses to PTEs during page walks can limit performance on MCM GPUs. We proposed MGvm to reduce such remote L2 TLB lookups and remote memory accesses to PTEs. We then extended it with dynamic switching for applications to switch back to fine-grained interleaving if needed. Our solution speeds up 15 applications by 52%, on average, over the private TLB baseline and by 30% over the shared TLB baseline.

Acknowledgment

The authors thank anonymous reviewers for their valuable feedback and help in improving the article. The authors also thank the artifact evaluators from MICRO 2022. The authors thank members of Computer Systems Labs at IISc for their valuable comments and discussions. This work is partially supported by research grants from VMware and Google. Neha is supported by Prime Minister's Research Fellowship, while Arkaprava is partially supported by a Young Investigator Fellowship by Pratiksha Trust, Bangalore.

APPENDIX

A. Abstract

Our artifact provides a Docker image with simulator code, required compilation tools (Go language), and scripts to compile, run and parse the metrics. We have included the code and scripts for reproducing figures 7,8,9, and 10 from the paper, corresponding to throughput, the ratio of remote to local TLB access, the ratio of remote to local page walk accesses, and page walk latencies, respectively.

B. Artifact check-list (meta-information)

- **Compilation: golang**
- **Run-time environment: Tested on Linux**
- **Hardware: Tested on x86**
- **Metrics: Throughput, Page walk latency, Ratio of local to remote accesses.**
- **How much disk space required (approximately)?: 20 GB**
- **How much time is needed to prepare workflow (approximately)?: One (1) hour**
- **How much time is needed to complete experiments (approximately)?: Forty Eight (48) hours**
- **Publicly available?: yes**

C. Description

We evaluated different virtual memory configurations for MCM GPUs using a modified version of MGPUSim. We have 15 benchmarks in our evaluation. We have containerized our entire setup to allow easy evaluation.

Configurations

- Private TLB.
- Shared TLB.
- MGvm (our proposal).
- MGvm-nobalance (variant of our proposal).

Evaluation

Our primary results are in figures 7, 8, 9, and 10.

- Figure 7 shows the throughput of different configurations, normalized to private.
- Figure 8 shows the ratio of remote vs local TLB hits.
- Figure 9 shows the ratio of remote vs local page walk accesses.
- Figure 10 shows the average page walk latency, normalized to private.

1) How to access: Please download the Dockerfile and the code archive (tar-gz) from the following URL.
https://doi.org/10.5281/zenodo.6937470.

2) Hardware dependencies:

- A single run (configuration) requires around 300-400 GB of RAM. Running all the configurations in parallel requires 1.2 TB to 1.5 TB.
- Each run requires 15 cores (for 15 workloads).

We ran our simulations on x86-64 machines (both Intel and AMD). We expect our simulations to behave identically if compiled and run on other architectures.

3) Software dependencies: We have containerized our setup and provided a Docker image for ease of evaluation. Therefore, we only have a software dependency on Docker. To install Docker on an Ubuntu machine, use the following command.

```
sudo apt install docker.io
```

D. Installation

Please download the Docker file and code archive (tag-gz). Ensure the Dockerfile and code archive (tar-gz) are in the same directory. Build the Docker image from the Docker file with the following command. Note: please run the command from the same directory as the Dockerfile and code archive.

```
docker build -f mgvm.Dockerfile . -t mgvm
```

The above command also copies the code archive inside Docker and extracts it to `mgvm` directory.

Run the Docker container in interactive mode using the following command. This command opens up a bash shell in the Docker image.

```
docker run -it mgvm
```

Change directory to the `mgvm` directory. Then again to `scripts` sub-directory.

```
cd mgvm
cd scripts
```

The `scripts` directory contains several scripts for compiling and running the workloads.

Run the following scripts in order to setup the workloads.

```
./0_clean.sh
./1_compile_benchmarks.py
./2_copy_benchmarks.sh
./3_gen_runners.py
```

The above commands create four new sub-directories inside `scripts` folder, namely `private`, `shared`, `mgvm`, `mgvm-nobalance`. These correspond to the various configurations under evaluation. Each of these sub-directories will contain a `samples` folder, which contains the benchmark binaries, and scripts for running each benchmark. For e.g., a file called `convolution2d.sh` is responsible for running the benchmark convolution2D with the right parameters.

We have provided more scripts to run these workloads in batches. Please see the next section.

E. Experiment workflow

To run the experiments, we provide the following scripts :

- 4_run_benchmarks_private.sh
- 4_run_benchmarks_shared.sh
- 4_run_benchmarks_mgvm-nobalance.sh
- 4_run_benchmarks_mgvm.sh

For different configurations, please run the appropriate scripts. The outputs would be stored in the corresponding sub-directories in the `scripts` directory. If the machine has sufficient RAM and cores, one may run more than one configuration at a time. Each configuration requires around 300-400 GB RAM at peak, and 15 cores to run.

For e.g., to run the private configuration, please run,

```
./4_run_benchmarks_private.sh
```

The scripts launch the jobs as background processes. Most jobs finish within 12 hours. A few (SPMV in particular) may take up to 24 hours. We track the progress of our jobs using `ps` or `top`.

After **all** the configurations have finished running, please use the parsing scripts to generate a csv with the results. The output will be stored in the file `results.csv`. Please use the same order of arguments to the parsing script.

```
./5_collect_stats.py private shared \
        mgvm-nobalance mgvm
```

Finally, running the normalization script over `results.csv` as follows, will produce the normalized numbers as in the paper. The output will be stored in `normalized.csv`.

```
./6_normalize_results.py results.csv \
        normalized.csv
```

F. Evaluation and expected results

The contents of `normalized.csv` match the corresponding figures in the paper.

G. Notes

Please ensure that the scripts have executable permisssion set. If not, please use chmod to set the executable bit.

```
chmod u+x root <scriptname.ext>
```

Please use `TMUX` when running Docker using interactive mode. If the Docker container dies while in interactive mode, then the entire runs launched from the Docker image will be lost. Alternatively, please start the Docker container in detached mode.

References

[1] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently self-replicating page-tables for large-memory machines," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 283–300.

[2] AMD, "Amd epyc products," https://www.amd.com/en/products/epyc, 2021.

[3] AMD, "Amd high bandwidth memory," https://www.amd.com/en/technologies/hbm, 2021.

[4] AMD, "Tools and sdks," https://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/, 2021.

[5] Apple, "Apple unveils m1 ultra, the world's most powerful chip for a personal computer," https://www.apple.com/in/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/, 2022.

[6] ARM, "Why chiplets and why now?" https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/why-chiplets-why-now, 2022.

[7] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 320–332. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080231

[8] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

[9] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "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*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 136–150. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123975

[10] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency," *SIGPLAN Not.*, vol. 53, no. 2, p. 503–518, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173169

[11] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 48–59. [Online]. Available: https://doi.org/10.1145/1815961.1815970

[12] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 307–318. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000101

[13] 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)*. IEEE, 2020, pp. 596–609.

[14] T. Baruah, Y. Sun, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Valkyrie: Leveraging inter-tlb locality to enhance gpu performance," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 455–466. [Online]. Available: https://doi.org/10.1145/3410463.3414639

[15] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[16] Basu, Arkaprava, "Revisiting virtual memory," 2013, http://research.cs.wisc.edu/multifacet/theses/arka_basu_phd.pdf.

[17] Basu, Arkaprava and Hill, Mark D. amd Swift, Michael M., "Virtual memory management system with reduced latency," 2015, https://patents.google.com/patent/US9158704B2/en.

[18] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[19] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. USA: IEEE Computer Society, 2006, p. 264–276. [Online]. Available: https://doi.org/10.1109/ISCA.2006.17

[20] S. Che, B. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *2013 IEEE International Symposium on Workload Characterization, IISWC 2013*, 09 2013, pp. 185–195.

[21] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 435–448. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

[22] I. Cutress, "Intel: Sapphire rapids with 64gb of hbm2e, ponte vecchio with 408 mb l2 cache," https://www.anandtech.com/show/17067/intel-sapphire-rapids-with-64-gb-of-hbm2e-ponte-vecchio-with-408-mb-l2-cache, 2021.

[23] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[24] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[25] Google, "A chiplet innovation ecosystem for a new era of custom silicon," https://cloud.google.com/blog/topics/systems/open-chiplet-ecosystem-powering-next-era-of-custom-silicon, 2022.

[26] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 184–195.

[27] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 637–650. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173194

[28] A. Jaleel, E. Ebrahimi, and S. Duncan, "Ducati: High-performance address translation by extending tlb reach of gpu-accelerated systems," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3309710

[29] S. N. Jonathan Koomey, "Moore's law might be slowing down, but not energy efficiency," https://spectrum.ieee.org/moores-law-might-be-slowing-down-but-not-energy-efficiency, 2015.

[30] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 66–78. [Online]. Available: http://doi.acm.org/10.1145/2749469.2749471

[31] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-centric data and threadblock management for massive gpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1022–1036.

[32] 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 Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 1–23, 2018.

[33] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, *Batch-Aware Unified Memory Management in GPUs for Irregular Workloads*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[34] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing gpu translation reach by leveraging under-utilized on-chip resources," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1169–1181. [Online]. Available: https://doi.org/10.1145/3466752.3480105

[35] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-gpus via sharing and spilling aware tlb design," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1154–1168. [Online]. Available: https://doi.org/10.1145/3466752.3480083

[36] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–63. [Online]. Available: https://doi.org/10.1145/3297858.3304044

[37] G. H. Loh, S. Naffziger, and K. Lepak, "Understanding chiplets today to anticipate future integration opportunities and limits," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 142–145.

[38] J. Lowe-Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of International Symposium on High-Performance Computer Architecture*, ser. HPCA '14, 02 2014, pp. 568–578.

[39] C. Mazumdar, P. Mitra, and A. Basu, "Dead page and dead block predictors: Cleaning tlbs and caches together," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 507–519.

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

[41] Nvidia, "Nvidia ampere architecture in-depth." [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[42] Nvidia, "Nvidia pascal mmu," 2019, https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[43] Nvidia, "Nvidia grace cpu," https://www.nvidia.com/en-in/data-center/grace-cpu/, 2022.

[44] NVIDIA, "Nvidia hopper architecture," 2022, https://nvdam.widen.net/s/9bz6dw7dqr/gtc22-whitepaper-hopper.

[45] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi, "Fast local page-tables for virtualized numa servers with vmitosis," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 194–210.

[46] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 258–269.

[47] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942

[48] L.-N. Pouchet and T. Yuki, "Polybench," 2010, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[49] B. Pratheek, N. Jawalkar, and A. Basu, "Improving gpu multi-tenancy with page walk stealing," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 626–639.

[50] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1106–1120. [Online]. Available: https://doi.org/10.1145/3466752.3480062

[51] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 582–595.

[52] S. Shahar, S. Bergman, and M. Silberstein, "Activepointers: A case for software address translation on gpus," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 596–608.

[53] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[54] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular gpu applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 180–192.

[55] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular gpu applications," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Piscataway, NJ, USA: IEEE Press, 2018, pp. 352–363. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00036

[56] R. Smith, "Amd announces mi200 accelerator family," https://www.anandtech.com/show/17054/amd-announces-instinct-mi200-accelerator-family-cdna2-exacale-servers, 2021.

[57] R. Smith, "Universal chiplet interconnect express (ucie) announced: Setting standards for the chiplet ecosystem," https://www.anandtech.com/show/17288/universal-chiplet-interconnect-express-ucie-announced-setting-standards-for-the-chiplet-ecosystem, 2022.

[58] I. Spectrum, "Behind intel's hpc chip that will pierce the exascale barrier," https://spectrum.ieee.org/intel-s-exascale-supercomputer-chip-is-a-master-class-in-3d-integration#toggle-gdpr, 2022.

[59] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[60] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[61] X. Tang, Z. Zhang, W. Xu, M. T. Kandemir, R. Melhem, and J. Yang, "Enhancing address translations in throughput processors via compression," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 191–204. [Online]. Available: https://doi.org/10.1145/3410463.3414633

[62] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 161–171.

[63] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and analysis of an apu for exascale computing," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 85–96.

[64] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi *et al.*, "Design and analysis of an apu for exascale computing," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 85–96.

[65] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 829–842.

[66] H. Yoon, J. Lowe-Power, and G. S. Sohi, "Filtering translation bandwidth with virtual caching," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 113–127. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173195

[67] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 339–351.

[68] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 345–357.