Efficient organization of TLB hierarchy for GPUs

Aman Choudhary

MTech Project Report

Abstract

Modern GPUs are seeing much widespread adoption, as programming them has been made considerably easier with the introduction of CPU-like virtual memory features in them. However, supporting virtual memory on a GPU comes with its own overheads, with address translation being one of the major bottlenecks. The fact that a massive number of threads run concurrently on a GPU means that the translation lookaside buffers (TLBs) are oversubscribed most of the time. Prior work of TLB-sensitive workloads reveals a high degree of page sharing across the different cores of a GPU. This causes the same translation entries to get replicated across cores, thus leading to wastage of precious TLB capacity. To address these inefficiencies, we propose reorganizing the conventional GPU TLB hierarchy by decoupling the L1-TLB from the GPU core, and aggregating them to some degree, thus making them a shared resource. In order to further reduce duplication, we also use the technique of clustering, which involves grouping L1-TLBs and ensuring no-replication across the entire group. Our evaluation shows that our new design significantly brings down redundancy of entries across the L1-TLBs and achieves appreciable performance gains for the GPU applications.

1 INTRODUCTION

With steady increase in GPU compute density and advancement in high-level programming tools, we witness a wider adoption of GPUs by general purpose applications in high-performance computing. In such devices, each computing node typically consists of a CPU with one or more GPU accelerators. Modern-day large scale applications, including deep learning, molecular dynamics, genome sequencing and cryptocurrency mining, have a high degree of data-level parallelism. Hence, GPUs have become the preferred choice to accelerate these applications.

This widespread adoption of GPUs has been enabled by the GPU hardware manufacturers, such as AMD and NVIDIA, by lowering the barrier to entry. Modern GPU platforms are equipped with optimizing compilers, efficient runtimes and advanced driver support. An important feature that has been recently added by GPU vendors is Unified Virtual Memory or UVM. This feature minimizes the effort required by programmers to program a single GPU, integrated CPU-GPU systems, and even multi-GPU systems. They eliminate the need to perform explicit memory copies, as all page transfers to/from the GPU are implicitly taken care of by the GPU driver and the runtime. UVM also makes it possible for the GPU memory to be oversubscribed. To support all these capabilities, GPU vendors have added the needed virtual memory support, by providing the required hardware and software. In terms of hardware, this support includes efficient multi-level translation lookaside buffers (TLBs), multi-threaded hardware page table walkers (PTWs) and input-output memory management units (IOMMUs). On the other hand, software support includes UVM API calls, which are directly under the control of the programmer, as well as support from the GPU driver and runtime for handling tasks such as page migration, TLB shootdowns, and page swapping. Thus, the set of virtual memory features that have been predominantly available on CPUs have now been extended to support GPUs.

Unfortunately, enabling Virtual Memory support on GPUs introduces significant performance overhead. More specifically, address translation on GPUs turns out to be a major bottleneck. This is due to the limited size of the private L1-TLBs, which in turn, generates severe pressure on the shared L2-TLB. To understand the impact of these problems, we conducted a set of experiments that capture the TLB miss rates of a range of GPU applications. In our experiments (details in Section 3), we observe that GPU applications that possess poor temporal locality in terms of page accesses can have L1-TLB as well as L2-TLB miss rates that are as high as 99 %. To solve this problem, improving address translation efficiency has been a focus of recent work.

In the recent work by *Baruah et. al.* [2], it has been found out that the parallel nature of some GPU applications causes threads running on different cores to access the same page or even the same cache block. Sharing in applications is common due to threads running on different GPU cores accessing the same common data structure, such as matrices and arrays. The profiling of these workloads reveals that there exists significant sharing of pages across different CUs of GPU. This behavior can range from applications where pages are being shared by only a few GPU cores, to applications where pages are shared by all the cores on the GPU. Based on these observations, we propose to re-architect the TLB hierarchy in an effort to reduce TLB miss rates and ultimately achieve performance gains for these workloads. The key contributions of our work have been outlined in Section 3.2.

2 BACKGROUND

In this section, we review the general mechanics of address translation on a GPU.

2.1 GPU Architecture

In order to describe the GPU architecture, we will use the AMD GPU terminology. A GPU consists of multiple cores, also known as *Compute Units* (or CUs) to process data. A CU can execute a large number of GPU threads (also known as, *work-items*) in parallel. In a CU, a *wavefront* of 64 work-items always executes the same instruction in lock-step manner. CUs are grouped according to the hardware resources they share. In a GCN-based GPU such as the Radeon VII [1], a set of four CUs forms a *Shader Array* (SA). Four SAs form a *Shader Engine* (SE), with 16 CUs sharing the graphics-related components such as the Geometry Engine. Every CU is connected to its own private L1-vector cache. Each SA has one L1-scalar cache (a read-only cache that is mainly designed for caching addresses and constants) and one L1-instruction cache. The multi-banked L2 cache is shared by all the CUs in a GPU and is multi-ported to achieve high throughput. The L2 caches are connected to the main memory controllers, which typically use high bandwidth solutions (e.g., HBM, GDDR) to best meet the demands of memory-hungry GPU applications. A group of SEs, when combined, form the whole GPU. NVIDIA-based GPUs, such as the Turing architecture, follow a similar hierarchical design where groups of Streaming Multiprocessors (SMs) are connected to form Texture Processing Clusters (TPC), and six of such TPCs are combined to form a Graphics Processing Cluster (GPC).

2.2 Virtual address translation in GPUs

The CUs in modern GPUs use virtual addresses. This virtual addressing mechanism abstracts away the physical location of the data, thus enabling many useful features. As an example, virtual addressing enables Unified Virtual Memory (UVM), which relieves the programmer's burden of performing explicit memory management. Apart from this, it allows for computation across datasets that exceed the GPU memory capacity. The GPU hardware and driver can effectively move data from device to device by changing the virtual to physical address mapping. Virtual addressing also provides address-space isolation for running concurrent applications on the GPU.

Virtual addresses need to be translated to physical addresses before accessing the data in the GPU L1-cache. The key to virtualizing memory is to map each virtual address to a physical address in GPU memory. Address translations are maintained at page-granularity that are stored in a multi-level page table. Whenever, a CU issues an access to a virtual memory address, a page table walk is performed where a page table walker traverses through each level of the page table, until the walker locates the page table entry for the requested mapping in the last level of the table. The physical address is extracted from the page table entry, which can be used for further lookup in the memory hierarchy.

2.3 Translation Lookaside Buffer

In order to avoid the long latency incurred during a page walk, GPUs have caches for storing recently used address mappings, known as TLBs (Translation Lookaside Buffers). With the introduction of TLB into the address translation mechanism, any request that hits in the TLB does not require a page walk. However, similar to CPUs, TLB misses trigger a page walk. A multilevel page table requires many memory accesses to translate a single address, hence it is a long latency operation. Therefore, a TLB miss hurts performance as it can stall multiple threads, all waiting for the page walk to finish for the requested page.

Similar to the cache hierarchy, the TLB hierarchy on a GPU consists of multiple levels. Each GPU core or compute unit (CU) is equipped with a private L1-TLB that is typically fully associative to eliminate conflict misses. The L1-TLBs are typically backed by a larger L2-TLB, which is shared between all the available CUs in the GPU and is usually multi-ported to allow for concurrent lookups. The TLBs typically include Miss Status Holding Registers (MSHRs), to handle requests that have already missed in the L1-TLB and are in process of fetching translations from the L2-TLB.

When a miss occurs in both the L1 and L2-TLB, a page table walk is initiated, which is handled by the hardware page table walkers. Page table walkers need to traverse the entire multi-level page table while searching for the desired entry corresponding to the virtual address, thus incurring high latency. GPUs access an order of magnitude more pages than CPUs, requiring a commensurate number of translations. In light of this, the page table walker is highly threaded.

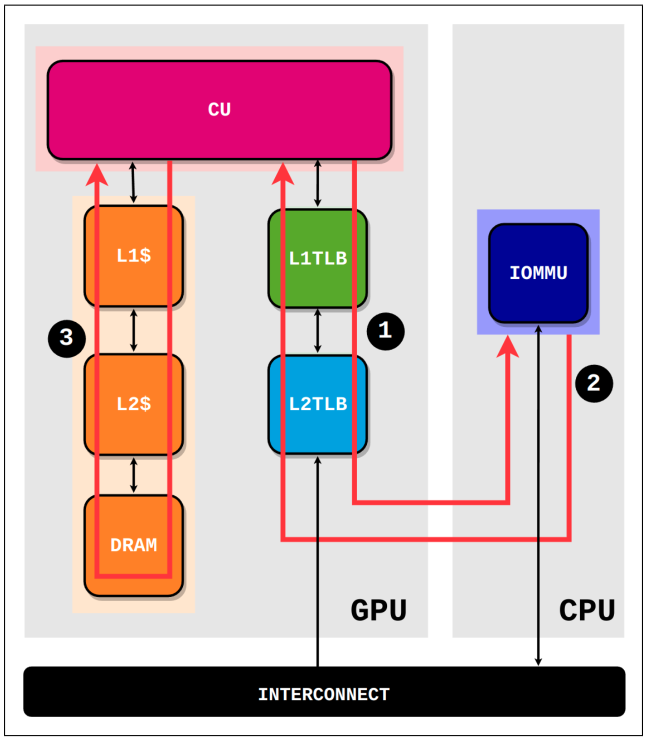

Figure 1: *The address translation process on a GPU*

The entire process of address translation is shown in Figure 1 and follows a set of steps. The process begins with the execution of a memory instruction by a CU, triggering a memory request to an L1 cache.

1. On a translation request from the CU, that misses in both the L1-TLB and L2-TLB, the request is forwarded to the page walk buffer on the IOMMU which is located on the CPU die.

2. Once a hardware page table walker is available, this request is picked up by the walker to perform a page table walk.

3. Once the page table walk completes and a valid translation is found, the request is returned to the L2-TLB and the L1-TLB. Both levels store the translation.

4. Finally, the L1-TLB returns the virtual-to-physical translation response to the L1 cache. Then, the L1 cache can issue the memory access with the physical address to its local storage and, if there is a cache miss, to the rest of the memory hierarchy.

3 MOTIVATION

3.1 Performance Bottleneck

GPU applications can have thousands of in-flight threads running concurrently (e.g.,163,840 maximum concurrent threads in an *AMD Radeon VII* GPU [1]). A large number of threads generate a large number of concurrent memory requests, causing severe pressure on the caches and the TLBs. To quantify this, we ran a few applications derived from diverse benchmark suites. The details of the applications evaluated along with their memory footprints are listed in Table 1. From Figure 2, it is evident that many workloads experience extremely high TLB miss rates in the private L1-TLBs and the shared L2-TLB (as high as 99 %). The MPKI (*Miss Per Kilo Instruction*) for L1-TLB as well as L2-TLB have also been listed in Table 2. An interesting fact to note about Table 2 is that the instructions in MPKI are warp-level instructions. Consequently, each such instruction can generate up-to 64 memory transactions. That is why, some application like *atax*, *bicg* and *spmv* have MPKI ≥ 1000.

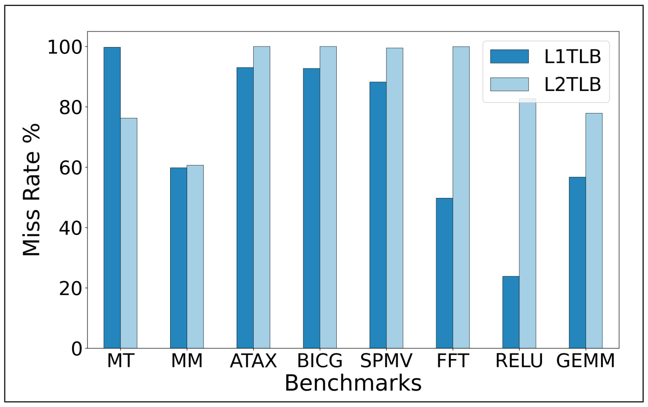

Figure 2: *TLB Miss rates for all benchmarks*

Even though GPUs are more latency tolerant as compared to CPUs, having such high TLB miss rates, which arises both in part due to the massive amount

Abbv	***Benchmark***	***Benchmark Suite***	***Memory Footprint***
MT	Matrix Transpose	AMDAPPSDK	32 MB
MM	Matrix Multiplication	AMDAPPSDK	58 MB
ATAX	Matrix Transpose and Vector Multiplication	Polybench	9 MB
BICG	BiCG Sub Kernel of BiCGStab Linear Solver	Polybench	9 MB
SPMV	Sparse Matrix Vector Multiply	SHOC	62 MB
FFT	Fast Fourier Transform	SHOC	64 MB
RELU	Rectified Linear Units	DNN Mark	128 MB
GEMM	General Matrix Multiply	DNN Mark	34 MB

Table 1: *List of benchmarks used*

of threads running on the GPU as well as the streaming behavior of the workloads, inevitably, lead to performance degradation. This is because, a significant number of wavefronts can stall upon TLB misses. To avoid the performance penalty associated with TLB misses that trigger long-latency page walks, we must reduce the TLB miss rates in GPUs.

Benchmark	***L1-TLB MPKI***	***L2-TLB MPKI***
MT	536.2	174.1
MM	61.5	12.1
ATAX	2205.7	2101.7
BICG	2199.2	2101.5
SPMV	6397.2	3687.4
FFT	15.3	0.5
RELU	35.3	5.9
GEMM	35.9	17.8

Table 2: *TLB MPKI for all benchmarks*

3.2 L1-TLB Sharing Behavior

The work by *Baruah et. al.* [2] reveals that the parallel nature of some GPU applications causes threads running on different cores to access the same page or even the same cache block. Sharing in applications is common due to threads running on different GPU cores accessing the same common data structure, such as matrices and arrays. Their work has found out via profiling of TLB-sensitive workloads, that there exists significant sharing of pages across different CUs of GPU. This behavior can range from applications where pages are being shared by only a few GPU cores, to applications where pages are shared by all the cores on the GPU.

We have borrowed from this insight, and come up with a more efficient organization for the existing TLB hierarchy which aims to tackle this issue. To be more specific, the above sharing pattern causes replication of TLB entries across private L1-TLBs of GPU cores. This in turn, leads to wastage of precious TLB storage, and is the root cause of high TLB miss rates. Our solution leverages this fact and tries to reduce replication by reorganizing the TLB hierarchy. The key design ideas of our work are outlined below:

1. **Aggregated L1-TLB Design:** Our first change involves decoupling the L1-TLB from their private cores, and then aggregating them. In a way, a many-to-one relationship is established between the CUs and TLBs in our design, contrary to the default one-to-one. This sharing of TLBs amongst CUs is aimed to curb the replication of TLB entries in L1-TLB.

2. **Clustered L1-TLB Design:** In order to further eliminate redundancy, we group the aggregated L1-TLBs into *clusters*, and enforce that there will be exactly one copy of any entry, if it exists, within a cluster. In this design, the CUs are connected with the cluster of aggregated L1-TLBs via an interconnect network. This design establishes a many-to-many relationship between the CUs and TLBs in our design, as opposed to the many-to-one in the previous design. The ideas of aggregation and clustering have been borrowed from a similar work done on L1 Caches [3].

3. **VIVT L1-Caches:** Although aggregation and clustering of L1-TLBs help to improve TLB miss rates, on the downside, they increase the address translation cost, since the L1-TLBs requests are now compelled to pass through an interconnect network. In order to reduce the impact of this, we explored a VIVT (Virtually Indexed Virtually Tagged) design for the L1 Data Cache instead of the traditional PIPT (Physically Indexed Physically Tagged) design. This altogether removes the address translation from the critical path of memory access, and provides runtime speedup, by avoiding TLB

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

Table 3: *Baseline GPU system configuration*

lookups when we have cache hits in the L1 Data Cache. We note that in [5], the authors have proposed VIVT design for the entire cache hierarchy (both L1 and L2 level). In our design we explore this only at L1, as our objective is to move the translation time from the critical path of L1 accesses.

4 BASELINE SYSTEM

Before we begin to outline our optimizations to the TLB hierarchy, we first describe our baseline system. The entire system is modelled and simulated on [4]. We model and evaluate an *AMD R9 Nano* GPU (the parameters are listed in Table 3). The GPU consists of 4 Shader Engines (SE), where each SE has 4 Shader Arrays (SA). Each SA further has 4 CUs, resulting in a total of 64 CUs in the GPU. Each CU has 4 SIMD units and each SIMD unit can have up to 10 wavefronts, resulting in a total of 40 wavefronts (maximum) that can be run on a CU. On AMD GPUs, threads belonging to the same wavefront (64 threads) execute in a lockstep fashion. Therefore, at any given point in time, the maximum number of threads in-fight on a CU is 2560 threads. The actual number of threads that can run at a given time depends on adequate resources being available, such as registers and shared memory.

The architecture features a multi-level cache hierarchy. The L1 Data caches are private to each CU, whereas the L2 cache is shared among the CUs of the GPU. The L2 cache is 8-way banked and interleaved at a cache-line level. Also, each Shader Array has an Instruction and a Scalar cache, which is common for all 4 CUs of the Shader Array. It models a virtual address space with full support for address translation hardware that includes a set of private fully-associative L1-TLBs, a set-associative non-banked L2-TLB that is shared by all the CUs, and an IOMMU. A basic layout of the TLB hierarchy corresponding to the baseline GPU system has been illustrated in Figure 3. Similar to the caches, each SA has an Instruction and a Scalar TLB, which is shared between all 4 CUs of the SA. Any address translation request that misses in the local GPU hierarchy is forwarded to the IOMMU, which is physically located on the CPU side. The IOMMU has a multi-threaded page table walker which can perform 8 searches in the page table in parallel. All of our experiments are run with a 4KB page size.

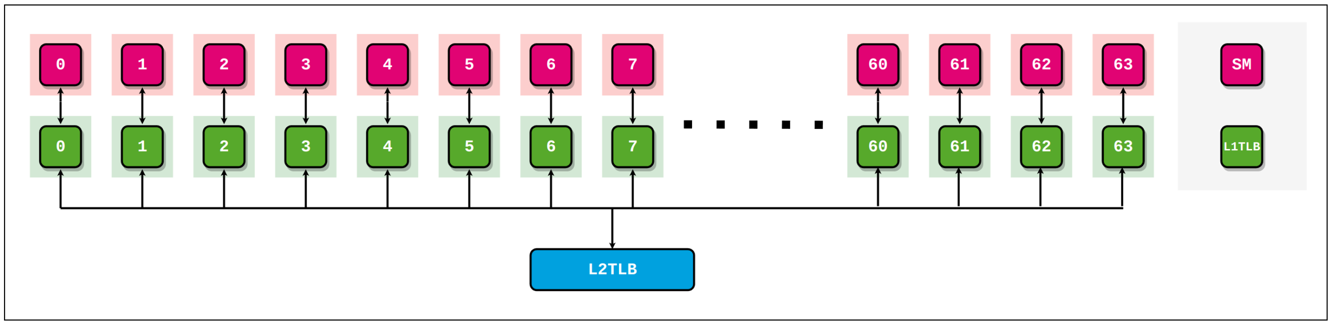

Figure 3: *TLB hierarchy in baseline GPU*

#TLBs per GPU	*#CUs sharing each TLB*	*#Entries per TLB*	*#Sets per TLB × Ways*	*#MSHR Entries per TLB*	*Access Latency*	*Xbar Latency*
64	1	128	1 X 128	4	1 cycle	-
32	2	256	2 X 128	8	2 cycles	1 cycle
16	4	512	4 X 128	16	4 cycles	2 cycles
8	8	1024	8 X 128	32	6 cycles	3 cycles
4	16	2048	16 X 128	64	8 cycles	4 cycles
2	32	4096	32 X 128	128	10 cycles	5 cycles
1	64	8192	64 X 128	256	12 cycles	6 cycles

Table 4: *Description of 7 different designs using Aggregated TLBs*

5 AGGREGATED L1-TLBs

5.1 Design

As described in Section 3.2, there is a significant amount of redundant entries across the private L1-TLBs. In order to deal with this problem, we decoupled the L1-TLBs from the cores and merged them at varying degrees, for e.g. 64 L1-TLBs merged to form 32 larger L1-TLBs, or 16 L1-TLBs and so on. The first consequence is that now, the L1-TLB is not private to the cores, instead each L1-TLB is now shared by a group of dedicated cores. The schematic diagram for the same has been illustrated in Figure 4. The details of the design are outlined below:

- Starting from 64 private L1-TLBs, we have tried aggregating the TLBs up to 7 different designs, with 64 private L1-TLBs at one extreme, representing no aggregation, and a single shared L1-TLB for all the cores, at the other extreme, representing complete aggregation. Table 4 summarizes the details for the same.
- If the number of aggregated L1-TLBs in the GPU is T, then each aggregated L1-TLB is now shared between $(64/T)$ CUs. In order to facilitate the communication between the CUs and the L1-TLBs, we propose a $(64/T) \times 1$ crossbar between the CUs and the TLB. This introduces additional latency while accessing the TLB. For a crossbar of size $M \times N$, we estimate the crossbar latency using the following formula: $\log_2(M) + \log_2(N)$. A $M \times N$ crossbar can be thought of as a $M \times 1$ multiplexer followed by a $1 \times N$ demultiplexer. The delay for a combinational circuit is proportional to $\log_2(\#Inputs)$, hence the above formula.
- The key point to note is that the total number of TLB entries across the GPU remains unchanged, i.e. we are in no way increasing the TLB storage at L1 level. It is just a reorganization of the resources in a different fashion.
- The default TLB configuration in the baseline system has a 128-entry full-associative cache. However, as we aggregate the TLBs, we model them as set-associative caches, with each of them having $(64/T)$ sets and 128-way associativity.

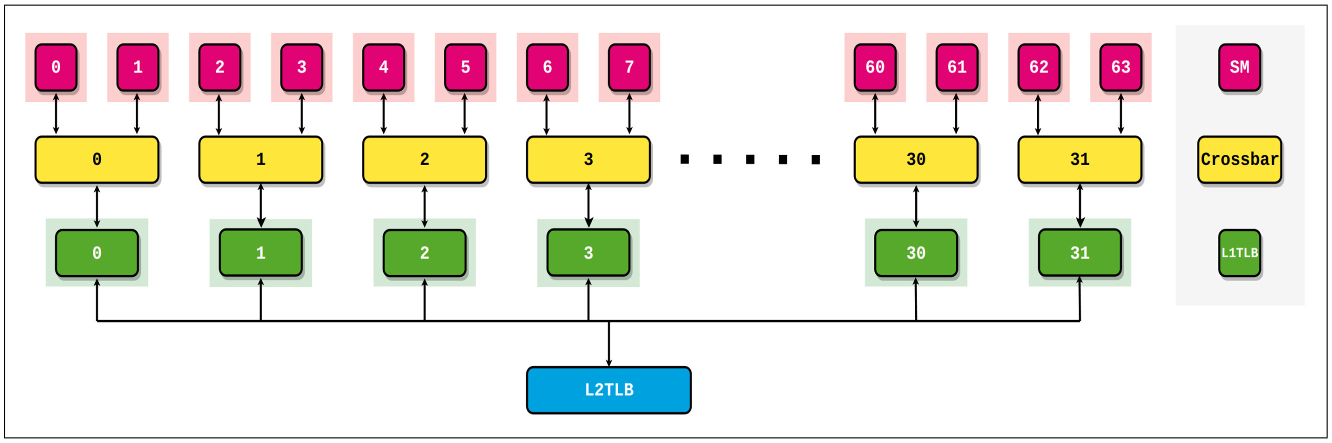

Figure 4: *A GPU with 64 CUs and 32 aggregated L1-TLBs*

- In the baseline system, each private L1TLB is equipped with 4 MSHR entries. They are meant to deal with those requests that miss in the L1-TLB, while a request for the same address has already been forwarded to L2-TLB. As we aggregate the TLBs, the MSHR entries also get merged, resulting in $(64/T) * 4$ MSHR entries per aggregated TLB.

- The trade-off that exists while merging the TLBs, is that while a larger shared L1-TLB may improve the hit-rate, however, now its access-time increases. Accordingly, if the number of sets in the TLB is S, we model modified access latencies for the aggregated L1-TLBs using the following formula: $\log_2(S) * 2$. The set indexing structure can be approximated by a $\log_2(S) \times S$ decoder. Following the same rationale as for crossbar latency, we came up with the above formula. We also verified its sanity using the *CACTI* tool for cache-modelling [6].

5.2 Evaluation

Workloads: For our evaluation, we selected a diverse set of workloads. Our workloads are selected from four different benchmark suites that include: AMDAPPSDK, Polybench, SHOC and DNN Mark. The memory footprints of our applications are listed in Table 1. The memory footprints are sufficiently large to stress the memory subsystem, including both the caches and the TLBs. Figure 5 reports the performance of aggregated L1-TLBs design for different benchmarks. We report L1-TLB and L2-TLB miss rates, L1-TLB and L2-TLB MPKI, and finally speedup improvement. In order to explain the results, we classify the applications into two groups:

(i) **Replication Sensitive:** This category includes the applications characterized by significant sharing of L1-TLB entries across the CUs. The performance of these application react to change in the degree of replication, hence the name replication-sensitive. The application in this class are: *mt*, *mm*, *atax*, *bicg* and *gemm*.

(ii) **Replication Insensitive:** The applications in this class don't show any appreciable sharing across the private cores. These applications are not bound to benefit from techniques which are aimed to reduce replication. The applications in this class are: *spmv*, *fft* and *relu*.

- From Figure 5 (a), it is clear that L1-TLB miss rates drop rapidly, as we aggregate the L1-TLBs from 64 to 1. This decrease is majorly due to reduction in replication of L1-TLB entries as we aggregate the L1-TLBs. They almost drop to $\leq$ 5 % for all benchmarks except for *spmv*. One exception to this decreasing trend is seen for *atax* and *bicg*, where there is $\sim$ 25 % increase in miss rate as we aggregate the L1-TLBs from 16 to 8.

- On the contrary, the trend for L2-TLB miss rates is quite different for each application (Figure 5 (a)). The L2-TLB miss rates don't portray much meaningful information and it is more insightful to look at the L2-TLB MPKI numbers. This is because miss rates are calculated as the portion of TLB accesses that don't hit in the TLB. Now, unlike L1-TLB where the number of TLB accesses is same for all configurations, the number of accesses for L2-TLB is derived from L1-TLB misses, which is a variable across different configurations. Since the L2-TLB miss rate is calculated against a non-uniform base metric i.e. L1-TLB misses, it makes it difficult to judge L2-TLB miss rates.

- As shown in Figure 5 (b), the trend for L1-TLB MPKI completely mimics that of L1-TLB miss rates, which is expected. On average, the decrease in L1-TLB MPKI across all applications is close to 90 %.

- If we look carefully in Figure 5 (b), L2-TLB MPKI is quite constant for replication insensitive applications, unlike that of replication sensitive applications where it decreases with aggregation. Now, L1-TLB miss can be classified as: (i) *Lookup misses* and (ii) *MSHR hits*. Out of these two, only the lookup misses constitute L2-TLB accesses, which in turn determines the L2-TLB MPKI. Replication sensitive applications benefit from aggregation, since it limits duplication of entries, and cause their lookup misses to drop. A decrease in lookup misses translates into lesser number of L2-TLB accesses and finally, a lower L2-TLB MPKI. This ultimately leads to performance gains.

- The reverse is true for replication insensitive applications. These applications have very less duplication of entries across the L1-TLBs. Hence, they do not benefit in terms of lookup misses upon aggregation. Their lookup misses show resistance to aggregation, and show no major change. The same is translated into an almost constant L2-TLB MPKI, as pointed out above. This is again confirmed by the fact that there

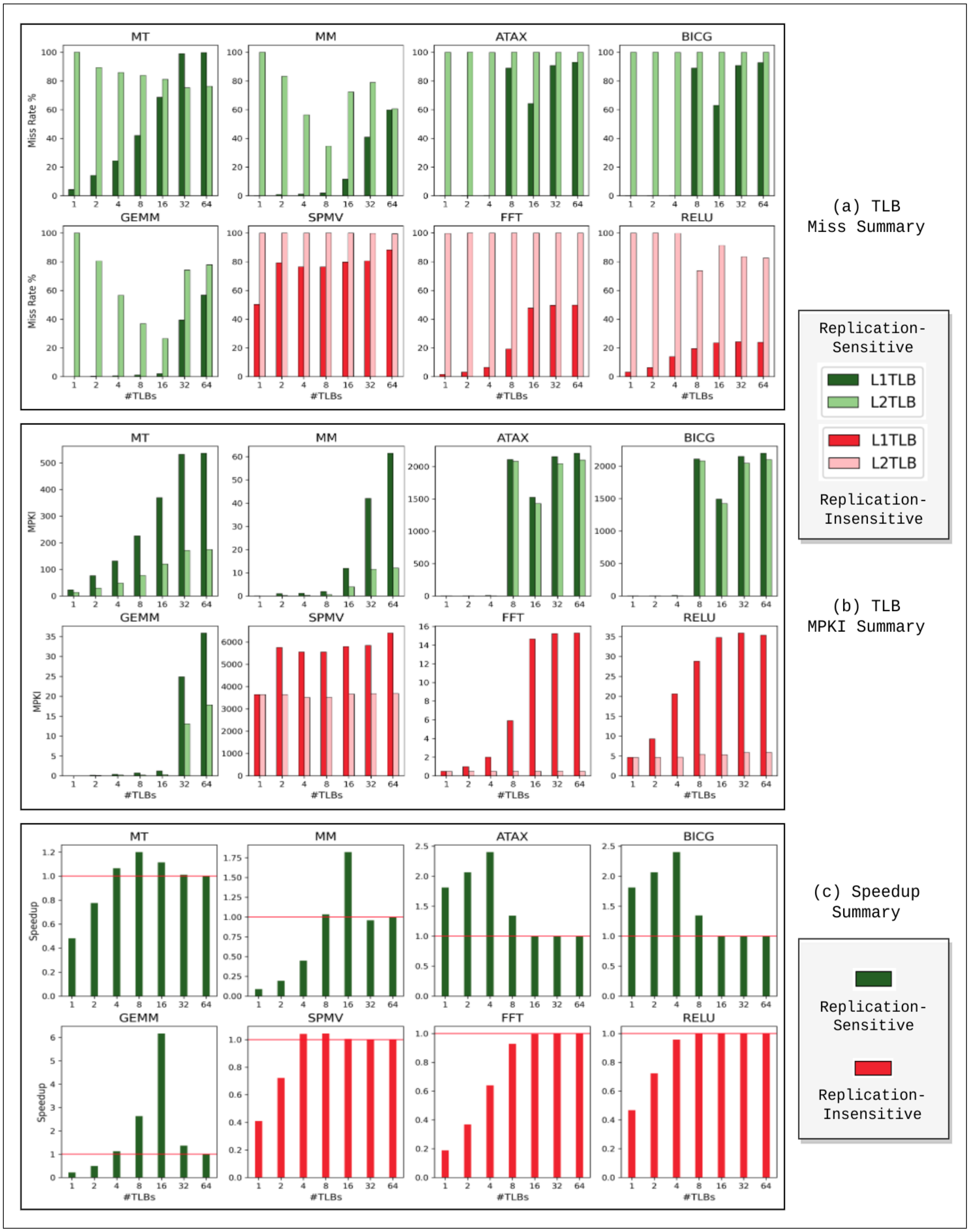

Figure 5: *Performance summary for Aggregated TLBs design*

is not much of speedup improvement for these applications .

- Another important point to note is that although the L1-TLB MPKI decreases for all applications, irrespective of their replication-sensitivity, the speedup gain is only observed for replication-sensitive workloads. This is because the drop in L1-TLB miss rates for replication-insensitive applications is due to drop in MSHR Hits, and not lookup misses. However, it it is these L1-TLB lookup misses that contribute to L2-TLB access, which incurs high delays, unlike MSHR hits. Prior work also confirms that a major source of GPU address translation overheads is the significant bandwidth demand for the shared L2-TLB due to frequent private L1-TLB misses [5]. While the actual latency of servicing a miss from the L2-TLB is only about 10 cycles in our system, the queuing latency which arises due to the large number of threads stalled waiting on L1-TLB misses to be serviced from the L2-TLB means that the actual latency can be much longer. That is why, even though there is drop in MSHR hits and consequently L1-TLB miss rates, they don't translate into significant performance gains.

- From Figure 5 (c), we observe that for replication-sensitive applications, the speedup first increases, reaches a maximum, and starts to drop as we perform aggregation of L1-TLBs from 64 to 1. There are two trend at play here: (i) As we do aggregation, the duplication of TLB entries reduces and L1-TLB lookup misses drop, a positive trend which contributes to performance improvement. (ii) The second trend, a negative one is that of rising L1-TLB access latency due to larger sized L1-TLBs as well as bigger crossbars, as we undertake aggregation. This hurts the performance as address translation which lies on the critical path of memory access starts incurring severe costs. In short, there is a sweet spot in terms of number of aggregated L1-TLBs for each application, where it gives optimal performance. On either extremes, the performance starts to degrade. Considering the best speedup for each application, the average speedup for replication sensitive applications is 2.8x, with maximum being a impressive 6.17x for *gemm*, and minimum being 1.2x for *mt.*

- For the replication-insensitive applications, the speedup shows almost no response at beginning and quickly deteriorates as we perform aggregation of L1-TLBs from 64 to 1. In the best case only *spmv* shows a meagre speedup of 1.04x. The maximum speedup achieved by other two applications is 1x i.e. no speedup.

- Finally, we need to talk about the impact of extreme aggregation. On average, there is approximately 32 % drop in performance for the configuration of fully shared L1-TLB. At worst, the performance drops by $\sim$ 91 % for *mm*, when extreme aggregation is done.

6 CLUSTERED L1-TLBs

In order to further reduce the replication of TLB entries across L1-TLBs, and thus improve TLB miss rates, we next use the technique of clustering.

6.1 Design

- The aggregated L1-TLBs as well as CUs are now grouped together in varying granularities to form a **cluster**. A single cluster of L1-TLBs is a *no-replication zone* for TLB entries. To elaborate this point, for a given virtual address, we

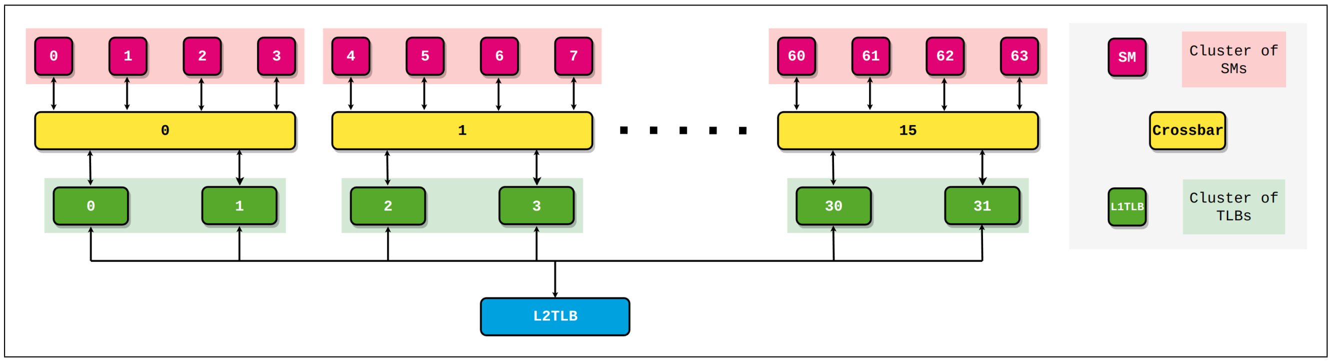

Figure 6: *A GPU with 64 CUs, 32 aggregated L1TLBs and 16 clusters*

make sure that there will be atmost a single copy of its translation in the entire cluster. To enforce this, the complete virtual address space is interleaved across the sets of member TLBs that are part of a cluster. This ensures that every virtual address is assigned a unique set and its entry, if present, will be present in only that set.

#TLBs	*#Clusters*	*Xbar Size*	*Xbar Latency*
64	**64**	-	-
	32	2 × 2	2 cycles
	16	4 × 4	4 cycles
	8	8 × 8	6 cycles
	4	16 × 16	8 cycles
	2	32 × 32	10 cycles
	1	64 × 64	12 cycles
32	**32**	2 × 1	1 cycle
	16	4 × 2	3 cycles
	8	8 × 4	5 cycles
	4	16 × 8	7 cycles
	2	32 × 16	9 cycles
	1	64 × 32	10 cycle
16	**16**	4 × 1	2 cycle
	8	8 × 2	4 cycles
	4	16 × 4	6 cycles
	2	32 × 8	8 cycles
	1	64 × 16	10 cycles
8	**8**	8 × 1	3 cycle
	4	16 × 2	5 cycles
	2	32 × 4	7 cycles
	1	64 × 8	9 cycles
4	**4**	16 × 1	4 cycle
	2	32 × 2	6 cycle
	1	64 × 4	8 cycle
2	**2**	32 × 1	5 cycle
	1	64 × 2	7 cycle
1	**1**	64 × 1	6 cycle

Table 5: *Description of Xbars in Clustered TLBs design*

- In our previous design, a group of CUs was accessing a single aggregated L1-TLB. In contrast, in the new design, a cluster of CUs interfaces with a cluster of aggregated L1-TLBs (see Figure 6). In essence, the one-to-one relationship between CUs and L1-TLBs in baseline system, transitioned to many-to-one in aggregated L1-TLBs design, and is now gets converted to many-to-many relationship. It can also be viewed as a one-to-one relationship between a cluster of CUs and a cluster of aggregated L1-TLBs.

- Clustering can be viewed as the second dimension in our the design of TLB hierarchy, where the number of aggregated L1-TLBs would be the first dimension. Essentially, we first fix the number of aggregated L1-TLBs, and on-top of that we perform clustering. If the number of aggregated L1-TLBs in the GPU is T, then the number of clusters C can be an integer in the range: $1 \leq C \leq T$. The details for different configurations have been listed in Table 5.

- In the case when the number of clusters is same as the number of aggregated L1-TLBs, i.e. $C = T$, then there is only one L1-TLB in one cluster. This configuration is identical to our previous design, where we didn't group the L1-TLBs into a cluster. Hence, there is a one-to-one correspondence between the 7 configurations in our previous design, and the ones in our current design where, $C = T$ is satisfied.

- In order to facilitate the communication between a cluster of CUs with its corresponding cluster of L1-TLBs, we propose a crossbar between the cluster of CUs and the cluster of L1-TLBs (see Figure 6). The number of crossbars required is same as the number of clusters C, i.e. one crossbar for every cluster.

- For C clusters and T aggregated L1TLBs, the size of each crossbar is $(64/C) \times (T/C)$.

- Like in the previous design, the presence of crossbars introduces additional latency while accessing the TLB. For a crossbar of size $M \times N$, we estimate the crossbar latency using the following formula: $\log_2(M) + \log_2(N)$.

6.2 Evaluation

- Similar to aggregation, L1-TLBs miss rates show a declining trend (Figure 7 (a)) as we introduce larger sized-clusters (or decrease the number of clusters), since clustering acts on top of aggregation to further reduce duplication of L1-TLB entries. A cluster increases the no-replication zone from a single L1-TLB to a group of L1-TLBs. This trend is consistent across all applications except for a few exceptions like *atax* and *bicg*, where a spike is seen when the number of clusters is decreased from 16 to 8. For replication-insensitive applications, this decrease in L1-TLB miss rates is attributed only to decline in MSHR Hits, as noted in Section 5.2. They don't show much improvement in L1-TLB lookup misses, like replication-sensitive applications. Figure 7

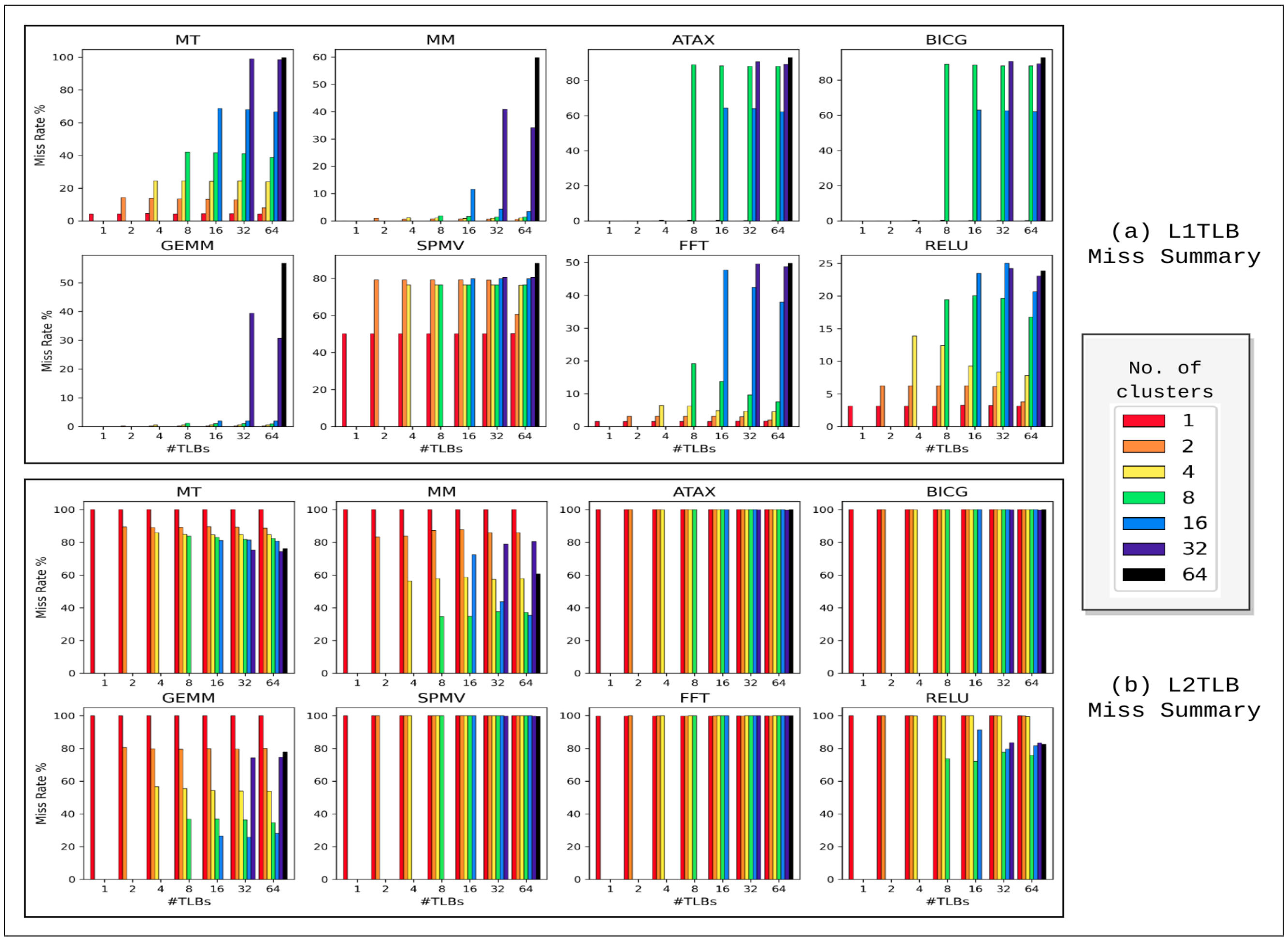

Figure 7: *Performance summary for Clustered TLBs design: I*

(b) shows the L2-TLB miss rates, which vary with each application, as discussed earlier.

- From Figure 8 (a), we note that L1-TLB MPKI completely mimics the trend portrayed by L1-TLB miss rates, as it did in case of aggregation. Considering the configuration which gives the least L1-TLB MPKI relative to the baseline, on average, the decrease in L1-TLB MPKI is approximately 90 %.

- For replication-sensitive applications, we note considerable reduction in L2-TLB MPKI as we increase the cluster size (Figure 8 (b)), a trend also observed during aggregation. Considering the configuration which gives the least L2-TLB MPKI relative to the baseline, on average, the decrease in L2-TLB MPKI for replication-sensitive applications is $\sim$ 98 %, as compared to only $\sim$ 9 % for replication-insensitive applications. Like previously, replication-sensitive applications benefit from clustering since it further eliminates replication of L1-TLBs entries, which cause L1-TLB lookup misses to diminish. The same is reflected in their L2-TLB MPKI, and later in their speedup gains, refer 8 (c).

- On the contrary, it is evident that L2-TLB MPKI for replication-insensitive applications almost stays the stagnant across all configurations, a trend similar to what we observed during aggregation. Their is an exception to this trend, namely in *relu*, but the change is very minor. It is clear that increasing the cluster size, does not cause much lowering in the L1-TLB lookup misses for these applications due to lack of any significant replication across L1-TLBs. As a result, the L2-TLB traffic is similar to as it was for the baseline, resulting in the observed behaviour for L2-TLB MPKI. This is confirmed by very

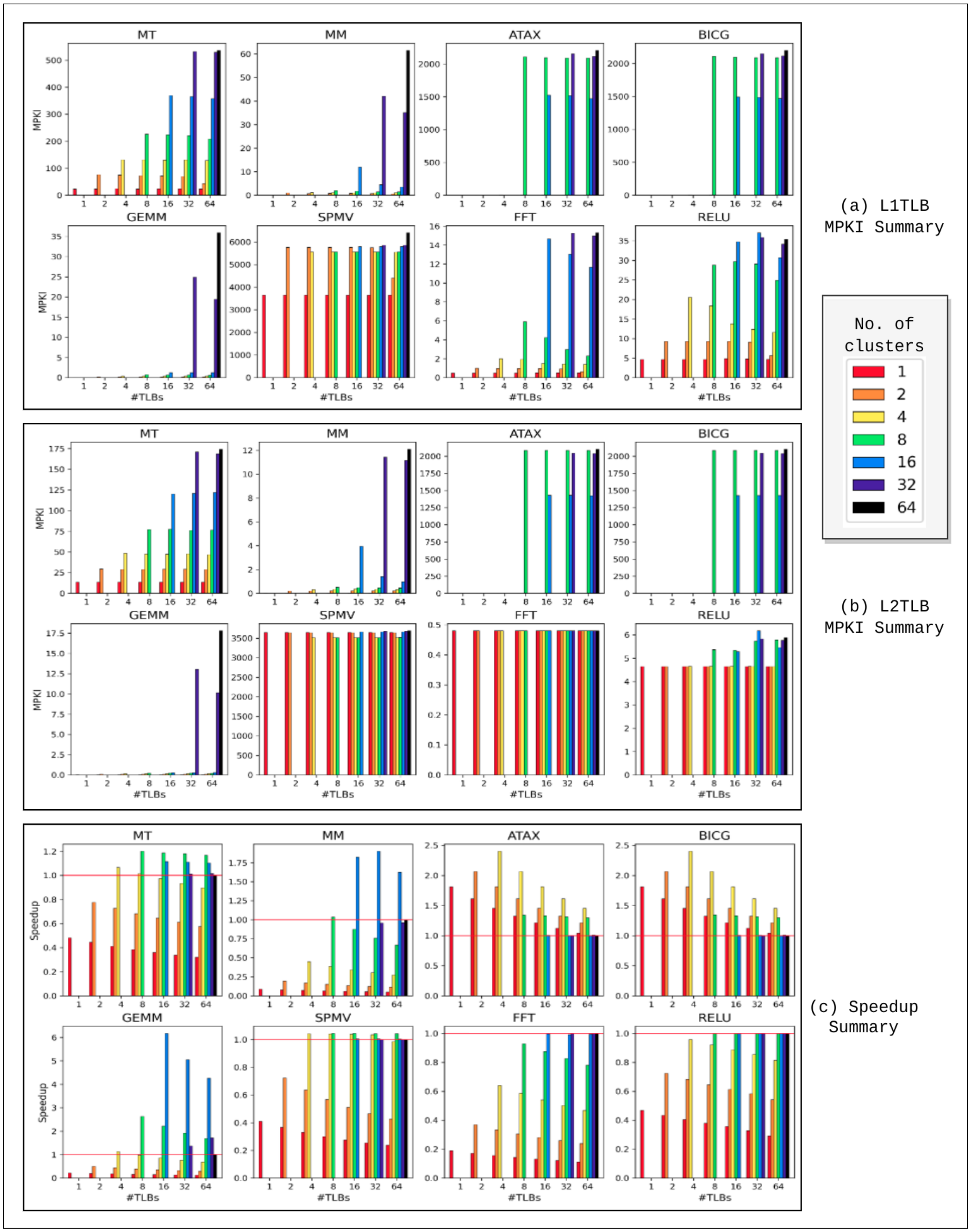

Figure 8: *Performance summary for Clustered TLBs design: II*

poor speedup gains for replication-insensitive applications in 8 (c).

- For replication-sensitive applications, the speedup patterns which were observed for aggregation manifest exactly as is, when we perform clustering. Like previously, similar positive and negative trends act in case of clustering as well. For a fixed number of aggregated L1-TLBs, increasing the cluster size causes further decline in duplication of L1-TLB entries, than aggregation alone was capable of. This is the positive trend since it boosts performance, by reducing L1-TLB lookup misses. On the contrary, a negative trend comes into play due to high latency associated with the ever larger crossbars required to support these bigger clusters. These penalties are incurred on the critical path of address translation, which hurt the application performance. In summary, these two competing trends first lead to a performance gain upon increasing cluster size, reach a plateau, and then quickly start degrading the performance. The average speedup for replication sensitive applications is 2.8x, with maximum being 6.17x for *gemm*, and minimum being 1.2x for *mt*.

- Similar to replication sensitive applications, the patterns in speedup for replication insensitive applications also mimic the behaviour which they displayed during aggregation. Since these applications do not possess significant redundancy of TLB entries across the private L1TLBs, the performance only degrades as we arrange the aggregated TLBs into larger clusters. These application do not get the benefit of positive trend as discussed above, and only get penalized by the negative trend. Expect for *spmv*, which shows slight performance improvement (4 % in the best case), the other two applications show direct performance deterioration upon building larger clusters.

- Table 6 lists the configuration corresponding to which we achieve the best speedup, for each application. Interestingly, the best configuration turns out to be the ones where there is only pure aggregation, and no clustering, the only exception to this rule being *mm*, where the best speedup is achieved from a clustering configuration i.e. 32 aggregated L1-TLBs arranged into 16 clusters. The speedup with 16 L1-TLBs and and an equal number of clusters for *mm* is only marginally lower, equal to 1.82x. This, prima facie, seems to imply that we achieve no benefit from clustering. The truth is that clustering indeed improves the performance, but only till the number of aggregated L1-TLBs is on the higher side. This number is unique to each application, for e.g. the benefit of clustering is evident till 32 aggregated L1-TLBs for *mm* and *gemm*, for *mt* it is 16, and finally till 8 for *atax* and *bicg*. In fact, when we focus on systems with lower number of aggregated TLBs, they are already stressed by large crossbars and high L1-TLB access latencies. Further clustering only aggravates this issue. In summary, the best speedups are achieved when the number of aggregated L1-TLBs are roughly between 4-16, and this is where clustering starts to perform poorly. Consequently, in most cases, the maximum speedups correspond to configurations where their is only aggregation, without any clustering.

App.	***#TLBs*** $\times$ ***#Clusters***	***Speedup***
MT	8×8	1.20x
MM	32×16	1.90x
ATAX	4×4	2.40x
BICG	4×4	2.40x
GEMM	16×16	6.17x
SPMV	8×8	1.04x
FFT	64×64	1.00x
RELU	64×64	1.00x

Table 6: *Configurations which give best speedup for each benchmark*

7 VIVT L1Caches

The process of address translation lies on the critical path of L1 Data Cache access, in the case of a *Physically-Indexed Physically-Tagged (PIPT)* cache. As aggregation and clustering of L1-TLBs further increases TLB access time, it hurts the performance of application, even with improved L1-TLB hit rates. An alternate form of cache design is *Virtually-Indexed Virtually-Tagged(VIVT)* cache. A VIVT cache is indexed using virtual addresses, thus needing no translation if the data is found in cache. Only in cases of a cache miss, address translation needs to be performed, in order to lookup for data in the lower levels of cache. This significantly filters the address translation traffic, when we hit in the data cache, and thus boosts performance of the system. A VIVT organization for GPUs is proposed, for both L1 and L2 cache in [5]. However, there are several underlying problems associated with purely virtual caches:

- **Homonyms:** These occur when the same virtual address maps to different physical addresses. This is the standard situation in multi-programmed CPU systems. However, if there is only a single address space, which is generally the common case in GPUs, then homonyms don't occur, since all processes use the same mapping of virtual addresses to physical addresses.

- **Synonyms:** These arise when different virtual addresses map to the same physical address. However, as pointed out in [5], synonyms are much less likely to occur in context of GPUs, as is the case with CPUs. This is because: (i) GPUs are designed to accelerate a single application at a time, they are not general-purpose devices like the CPU. Hence, there is usually only a single virtual address space resident at a given time in the case of GPUs. (ii) Second, GPUs don't execute OS kernel code; they only execute user-level applications. (iii) Third, GPUs rarely access I/O devices. Hence, most causes of synonym accesses are eliminated in case of GPUs.

As noted above, the issue of homonyms and synonyms is not so significant in the context of GPUs, as is the case of CPUs. Hence, we decided to explore a purely virtual design for the L1-Data cache, with the aim of removing address translation off the critical path of memory access.

7.1 Design

- The *AMD R9 Nano* GPUs have a dedicated module called *Address Translator*, shown abbreviated as *AT* in Figure 9. Its job is to first, forward memory access requests to the TLB hierarchy, and on arrival of the requested translations, redirect them to the cache subsystem. Our implementation rearranges this module as shown in Figure 9 (b) to achieve the desired VIVT cache configuration.

- Our design assumes a single active address space as of now, which is generally the common use-case in GPUs. In case of multi-tenancy, we can tag the virtual address with address-space identifiers which would disambiguate the virtual addresses (mak e them globally valid), in order to avoid the problem of homonyms.

- As of yet, we didn't detect any synonym for our applications. However, this does not rule out the possibility of their occurrence. If synonyms are not dealt with correctly, then it might cause correctness issues. This is currently a limitation of our implementation.

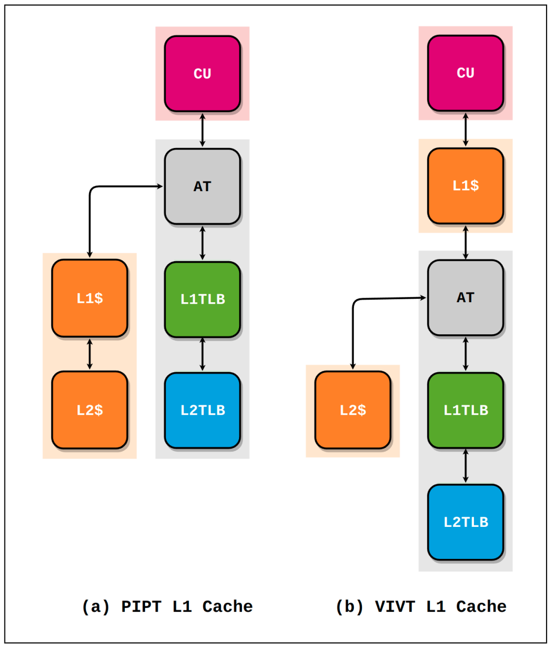

Figure 9: *PIPT and VIVT Cache design*

7.2 Evaluation

Figure 11 compares the best speedup achieved across all configurations (obtained using aggregation and clustering), for PIPT L1 caches against their VIVT version. In order to explain the results obtained, we classify our application as follows:

(i) **Type I** : These include applications which experience severe L1-Cache miss rates, as high as 99 %. *mt*, *fft* and *relu* come under this type.

(ii) **Type II** : The applications in this type still have high L1-Cache miss rates, but slightly lower than that of Type I, i.e. around 90 - 95 %. This class includes *atax*, *bicg* and *spmv*

(iii) **Type III** : The final category represents applications with moderate L1-Cache miss rates. It includes *mm* and *gemm*, with miss rates of 47 % and 65 % respectively.

- From Figure 11, it is clear that for *Type I* applications, no speedup improvement is achieved upon replacing the PIPT L1-cache with a VIVT one, which is as expected. Since these applications face severe miss rates, we almost always have to perform address translation, thus defeating the purpose of a virtual L1-cache.

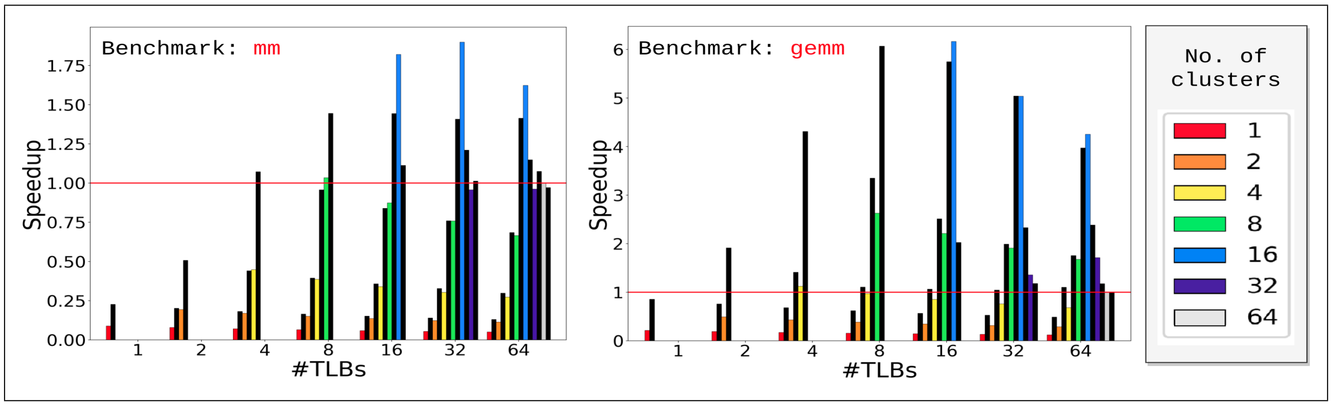

Figure 10: *Speedup summary for mm and gemm (The colored bars represent speedup for systems using PIPT L1-Cache. For each colored bar, the black colored bar immediately to its right represents the VIVT speedup for the same TLB configuration)*

- *Type II* applications also suffer from high L1-cache miss rates. However, they show modest gains unlike *Type I* applications, since ∼ 5-10 % of the memory accesses hit in the L1-cache. These accesses get filtered out from the address translation traffic. To be specific, the speedup gains for *atax* and *bicg* were around 7-8 %. For *spmv*, the gain is very meagre, i.e. 0.4 %.

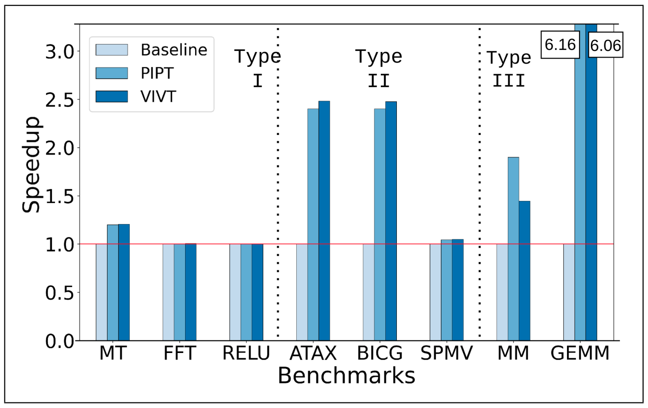

Figure 11: *Speedup summary for all design configurations*

- We would expect the VIVT version to perform better than its PIPT counterpart for *Type III* applications, since they have appreciable L1-cache hit rates, which create ample scope for improvement. However, prima facie, Figure 11 indicates the opposite. The best performing VIVT configuration performs slower than the configuration having best speedup when using PIPT cache. To be specific, the speedup difference between the best configurations is 44 % for *mm*, and 10 % for *gemm* respectively. To understand this further, we compare the performance of different L1-TLB configurations with PIPT and VIVT organisation in Figure 10.

- For *mm*, the VIVT configuration consistently outperforms its PIPT counterpart, except for the case, when we arrange the aggregated L1-TLBs into 16 clusters (represented by the blue bars). The same behaviour is observed for *gemm*, except for the cases when the aggregated L1-TLBs are grouped into 16 or 32 clusters. Interestingly, it is these anamalous cases, where PIPT design achieves its best speedup, while VIVT scheme performs badly. If we leave out these exceptions, the VIVT configurations achieve 30 % and 140 % more speedup on average, across all configurations, for *mm* and *gemm* respectively.

- The reason for this anomalous behaviour in these configurations, is that although the VIVT cache was preventing considerable traffic from going into the TLB hierarchy, the requests which eventually reached the TLB hierarchy incurred high miss rates. To be specific, lets consider the configuration with 32 aggregated TLBs, and 16 clusters for *mm*, where PIPT gives optimal speedup. In this case, the L1-TLB and L2-TLB MPKI was 5x and 8x times higher respectively, for the VIVT design as opposed to its PIPT counterpart. Similar behaviour is observed for *gemm* as well. As we had described earlier, L2-TLB traffic is a major bottleneck during address translation which hurts application performance badly, hence the unexpected behaviour.

8 CONCLUSION

Due to several key benefits, that Virtual Memory support has to offer for GPU programmers, its performance is going to be critical for modern day GPU applications. However, the issues pertaining to address translation act as bottlenecks and can severely harm the performance of GPU applications. Thus, it is imperative to design solutions to mitigate these costs, so that GPUs can enjoy the benefits of virtual memory support, and not pay high penalties at the same time.

In this work, we have leveraged the fact that certain GPU applications exhibit an inherent sharing behavior where the same page-table entry is shared across multiple L1-TLBs in the system. We proposed and evaluated the techniques of aggregation and clustering of L1-TLBs in order to deal with the same. Although we were able to significantly bring down the replication, and also achieve appreciable performance benefits, our design points have an inherent cost associated with the usage of interconnect networks. Hence, we also explored a VIVT scheme for the L1 data cache to move the address translation costs off the critical path of memory access. Our results show that an aggregated TLB hierarchy, along with a virtual cache subsystem promises to alleviate the problems of address translation to an appreciable degree.

9 FUTURE WORK

- The L1-TLB access latencies were cross-checked against *CACTI* [6], which is a cache modelling tool. The minimum supported block size by the tool was 64 bytes, very much unlike that of any TLB. TLBs are more specialized caches, and hence we plan to utilize a modelling tool tailored specifically to model TLBs, in order to accurately verify our TLB access latencies.

- Similar to TLB access latencies, we need to verify our crossbar latency numbers using tools like NOC (Network-On-Chip) simulators to test their correctness.

- We haven't considered the area and power cost aspects for our designs. We plan to verify the same using *AccelWattch* [7], which is a power modeling framework for modern GPUs.

- As is evident from Table 6, the configuration which gives the best speedup across all applications is not unique. The number of aggregated TLBs which provides the best speedup varies with the application. Therefore, we plan to devise an adaptive scheme which will change the L1-TLB configuration on the fly, to best suit the needs of the application.

- We need to extend our VIVT implementation to ensure that it handle the issues of purely virtual caches properly.

References

[1] AMD. 2019. AMD Radeon VII. (2019). *https://www.olcf.ornl.gov/wpcontent/uploads/2019/10/ORNL_Application_Readiness_WorkshopAMD_GPU_Basics.pdf*

[2] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, David Kael. *Valkyrie: Leveraging Inter-TLB Locality to Enhance GPU Performance.* In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2020).

[3] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh and A. Jog, *"Analyzing and Leveraging Decoupled L1 Caches in GPUs,"* 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021.

[4] Yifan Sun, Trinayan Baruah, Saiful Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. *MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization.* In The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019.

[5] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. *Filtering Translation Bandwidth with Virtual Caching.* In ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018.

[6] N. P. Jouppi, A. B. Kahng, N. Muralimanohar and V. Srinivas, *"CACTI-IO: CACTI with off-chip power-area-timing models"* 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2012, pp. 294-301.

[7] Vijay Kandiah, Scott Peverelle, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. *AccelWattch: A Power Modeling Framework for Modern GPUs.* In MICRO54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021.