2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00031

Enhancing GPU Performance via Neighboring Directory Table Based Inter-TLB Sharing

Yajuan Du*†, Mingyang Liu†, Yuqi Yang†, Mingzhe Zhang‡, Xulong Tang§
† Wuhan University of Technology, Wuhan, China
* Shenzhen Research Institute of Wuhan University of Technology, Shenzhen, China
‡ State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
§ University of Pittsburgh, Pittsburgh, Pennsylvania, USA
{dyj, lmy569}@whut.edu.cn, yangyuqi97018@163.com, zhangmingzhe@iie.ac.cn, tax6@pitt.edu

***Abstract*—Modern discrete GPUs support Unified Virtual Memory (UVM), simplifying GPU programming. However, UVM entails address translation on each memory access, which introduces expensive performance overhead during address translation. In this work, we select various workloads and conduct experiments on GPU performance. Our investigation shows that many workloads have low L1 TLB hit ratios of less than 40% on average. Even for a particular workload, the hit ratio is as low as 15%, which leads to significant performance degradation. Through further analysis, we find that a lot of common entries exist between neighboring private L1 TLBs, showing clear inter-TLB sharing behavior. To leverage the sharing, we propose a Neighboring Directory table based hardware scheme, named NeiDty. In NeiDty, L1 TLBs can probe physical addresses from neighboring L1 TLBs through a lightweight interconnect network. And NeiDty uses neighboring directory tables to keep track of the shared entries among neighboring L1-TLBs. In addition, we find it better to update address translation after two consecutive neighboring TLB hits than one hit. We run eight typical workloads with Gem5-GPU, and the results show that NeiDty increases the average hit ratio of L1 TLB TLB by 14% and improves the average performance by 10%.**

***Index Terms*—GPU, Address translation, Hit ratio, TLB sharing**

I. Introduction

With the promotion of machine learning, artificial intelligence, cloud computing, and other technologies, the global data volume is experiencing explosive growth. Modern CPUs already cannot meet the demand for such large-scale data. As the most popular accelerator, GPUs contain thousands of cores and have super parallel processing ability.

Recently, GPU vendors added Unified Virtual Memory (UVM) support. UVM provides a virtual address space abstraction similar to CPUs, eliminating manual memory management [1], [2]. Furthermore, 49-bit virtual addresses are adopted in UVM, which enables GPUs to access the entire system memory [3], [4], thus enabling oversubscribing GPU memory. These features of UVM are essential for abstract programming.

Unfortunately, UVM is not free. To support UVM, each virtual address must be translated to physical address before accessing data in GPU L1 caches. To accelerate the translation, some hardware and software [5], [6] are required, including private L1 TLBs, a shared L2 TLB, a multi-threaded page table walker, and so on. More specifically, each Streaming Multiprocessor (SM), consisting of many cores (e.g., 64 in Tesla P100 [3]), is equipped with a private L1 iTLB (instruction TLB), a private L1 dTLB (data TLB) and a TLB coalescer. All the L1 TLBs are backed by a shared L2 TLB. And a multi-threaded page table walker is responsible for handling L2 TLB misses. Similar to CPUs, page table walks in GPUs are time-consuming to read contents of multiple memory locations. Efficient TLBs can mitigate page table walk penalty.

However, recent research found that GPUs suffer a severe performance decrease for large-scale database processing with irregular memory access and insufficient TLB reach are the bottleneck for irregular access patterns [7], [8]. In this work, we investigate the address translation performance in GPU. We observe that L1 TLBs suffer a low miss rate and have obvious sharing behaviors, i.e. many missed address translation in one L1 TLB can be found in other L1 TLBs. However, L1 TLBs are private, and they cannot communicate with each other in traditional GPU. A straightforward method is building an interconnect network among all the L1 TLBs, but it is inefficient due to high area and power overhead. Besides, we find that among the TLB entries possessed by several TLBs, many are possessed by all the L1 TLBs. So building direct communication ports between neighboring TLBs would be more efficient.

In this paper, we observe the sharing behavior between neighboring TLBs and propose NeiDty by using neighboring directory tables to keep track of the common entries of different L1 TLBs. NeiDty adds necessary communication ports between neighboring L1 TLBs so they can be accessed on local TLB misses. For further improvement, we study the updating policies and find that on local TLB miss but remote TLB hit, it is better to load the entry from remote TLBs only when the virtual address is accessed twice in a row. We call the update policy to be NeiDty-twice. Finally, after applying NeiDty-twice update policy with Gem5-GPU simulator [9], NeiDty can increase the average hit ratio of L1 TLB by 14% and improve the average performance by 10%.

The main contributions of this work include:

- We provide a detailed analysis of inter-L1-TLB sharing and neighboring-L1-TLB sharing behavior. The results

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00031

show that there exists obvious sharing between neighboring L1 TLBs.

- We propose NeiDty, a hardware solution to leverage neighboring-L1-TLB sharing. We design a neighboring directory table to keep track of the sharing information. And we design an interconnect network between neighboring L1 TLBs, to achieve the the sharing at a low price.
- We study three different update policies for neighboring-L1-TLB sharing and implement them with Gem5-GPU. The results show that for many workloads, it is better to load the entry to local TLB only when the virtual address is accessed twice in a row.

II. Background And Motivation

In this section, we first review the architecture of modern GPUs. Next, we discuss the importance of UVM and describe the address translation architecture after UVM. Then, in preliminary study, we explore the overall TLB hit ratio, inter-TLB sharing behavior, and neighboring-TLB sharing characteristic. Finally, we discuss how to leverage the sharing behavior to improve GPU address translation performance.

A. Background

1) GPU Architecture: GPU is a highly parallel processor. With the development of technology and programming models [10], the original dedicated GPU has become the current general-purpose accelerator in recent years. Currently, GPUs are widely used in many fields, including high-performance computing, computer graphics, machine learning, data analysis, etc. For Nvidia GPU, it is composed of Streaming Multiprocessors (SMs). Each SM contains a lot of cores, which are the functional units that execute instructions and the part of the GPU that runs our CUDA kernels [11]. Within SM, there are different caches, including private L1 data cache, read-only texture cache, constant cache, and software-managed shared memory. All SMs share a L2 cache, which is connected to the GPU main memory.

2) Unified Virtual Memory: Modern GPUs introduce Unified Memory (UM) and Virtual Memory (VM) [1]–[4]. UVM is a single memory address space accessible for any processor in a system, and it is transparent for programmers. With UVM, programmers can build more efficient dynamic data structures without the need to perform explicit memory copies, as the GPU driver and the runtime handle all page transfer automatically. Besides, UVM support memory oversubscription. However, UVM is not a technique to make well-written GPU codes run faster. To support UVM, for each instruction or data from caches, it is unavoidable to obtain the corresponding physical address by accessing TLB hierarchy, which is time-consuming.

3) Address Translation in GPU: As Fig. 1 shows, in modern GPUs, there are multi-level hardware components on address translation architecture [5], [6], [12], [13]. Each SM has one private L1 iTLB (instruction TLB) and one private L1 dTLB (data TLB), both equipped with TLB coalescer. L1 iTLB and L1 dTLB are used for caching instruction and data

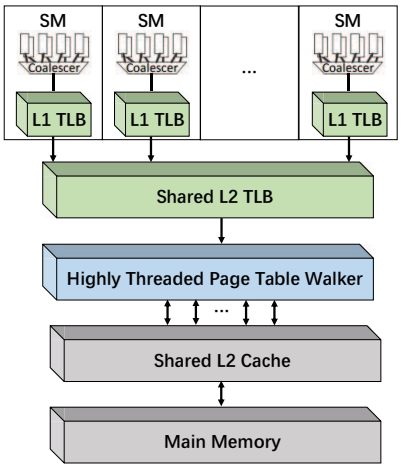

Fig. 1. GPU Address Translation Architecture

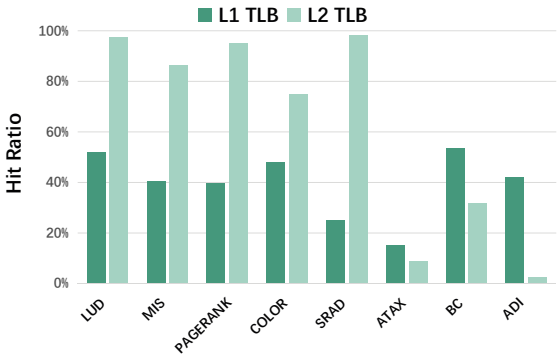

Fig. 2. L1/L2 TLB Hit Ratio

address translations. Because it is in the critical path, L1 TLB is fully associative to eliminate conflict, and it has a small capacity to provide low latency. All the available L1 TLBs are backed by one shared L2 TLB, much larger than L1 TLB. Besides, a highly-threaded shared Page Table Walker (PTW) will handle miss when a miss occurs in both L1 TLB and L2 TLB by traversing the Page Table (PT) entries. In GPU address translation architecture, there are three main steps during the translation. (1) SM issues the coalesced address translating request to L1 TLB. (2) If L1 TLB can serve the request, the physical address would be directly returned, or if a L1 TLB miss occurs but L2 TLB can handle the miss, the physical address would be returned and stored in L1 TLB. (3) On both L1 TLB and L2 TLB miss, PTW will retrieve page walk cache or the whole travel page table entries, then return the address mapping, which would be stored in both L1 TLB and L2 TLB.

B. Preliminary Study

Enabling unified memory support in GPU is essential for programmers, but it is not conducive to GPU performance. Actually, it is harmful due to the expensive address translation. To obtain detailed information about GPU address translation, we run eight different workloads using Gem5-GPU simulator,

TABLE I
CORRELATION BETWEEN NEIGHBORING TLBS

Workload \ TLB id	TLB0	TLB1	TLB2	TLB3	TLB4	TLB5	TLB6	TLB7	TLB8
LUD	0.22	0.21	0.21	0.21	0.21	0.21	0.21	0.22	0.22
MIS	0.19	0.18	0.18	0.17	0.18	0.18	0.19	0.19	0.19
PAGERANK	0.21	0.17	0.19	0.20	0.20	0.20	0.20	0.20	0.21
COLOR	0.20	0.19	0.19	0.19	0.20	0.19	0.20	0.21	0.21
SRAD	0.17	0.17	0.15	0.17	0.17	0.17	0.18	0.17	0.20
ATAX	0.20	0.14	0.28	0.14	0.22	0.18	0.19	0.20	0.17
BC	0.43	0.50	0.53	0.42	0.43	0.37	0.34	0.31	0.30
ADI	0.03	0.63	0.07	0.07	0.07	0.60	0.06	0.05	0.08

TABLE II
CORRELATION BETWEEN NEIGHBORING TLBS

Workload \ TLB id	TLB9	TLB10	TLB11	TLB12	TLB13	TLB14	TLB15	Average
LUD	0.23	0.23	0.23	0.24	0.23	0.23	0.22	0.22
MIS	0.19	0.19	0.19	0.19	0.20	0.20	0.18	0.19
PAGERANK	0.20	0.22	0.21	0.22	0.22	0.23	0.20	0.20
COLOR	0.21	0.22	0.20	0.22	0.20	0.22	0.20	0.20
SRAD	0.19	0.21	0.22	0.20	0.21	0.20	0.17	0.18
ATAX	0.17	0.18	0.19	0.20	0.19	0.15	0.18	0.19
BC	0.31	0.31	0.30	0.29	0.30	0.29	0.38	0.36
ADI	0.10	0.01	0.01	0.01	0.62	0.03	0.06	0.16

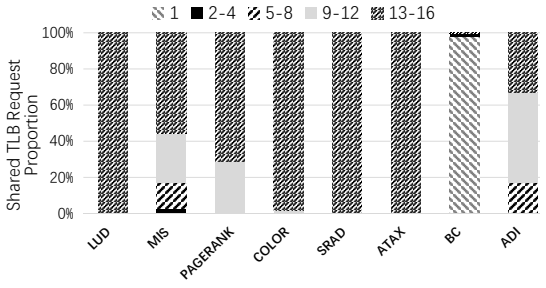

Fig. 3. Inter-TLB Sharing Characteristic

from Rodinia [14], Pannotia [15], PolyBench [16] benchmark suits, including LUD, MIS, PAGERANK, COLOR, SRAD, ATAX, BC, and ADI. And we mainly collect statistics about TLB hit ratio, inter-TLB sharing, and neighboring-TLB sharing.

1) TLB Hit Ratio Characteristic: From Fig. 2, we can obtain two critical observations:

First, many workloads suffer low L1 TLB hit ratios. The average hit ratio of L1 TLB is less than 40%, and even for those workloads possessing poor temporal locality, the hit ratio of L1 TLB is just 15% (ATAX) and 25% (SRAD), which indicates that the hit ratio of L1 TLB can be further improved. The main reason is that the capacity of L1 TLB is not enough. The high miss ratio will lead to frequent stalls for warps, which significantly hurts the GPU parallelism and degrade performance.

Second, the average hit ratio of shared L2 TLB is as high as 80% except for three workloads (ATAX, BC, and ADI), which are significantly higher than L1 TLB. Even for SRAD, the L2 TLB hit ratio is 98% while the L1 TLB hit ratio is only 25%. This means that many translations cannot be served by private L1 TLB but by shared L2 TLB.

Considering the high parallelism of GPU workloads, we infer that there are many common translations among L1 TLBs. To expose the inter-L1 TLB patterns in GPU workloads, we do further experiments.

2) Inter-TLB Sharing Characteristic: The sharing of GPU is common. Different GPU cores often access the shared data structure in GPU applications, such as array and matrix. Some threads in different GPU cores will access the same pages, even the same cache lines. To analyze the sharing behaviour, we calculated the number of TLBs possessing duplicate entries for each address translation request. Fig. 3 shows the distribution of the proportion of address translations shared by different numbers of L1 TLBs. The x-axis represents eight workloads, and the y-axis represents the proportion of the different sharing numbers (how many L1 TLBs possess the sharing entry at the same time). There are five possible intervals for y-axis: ”1”, ”2-4”, ”5-8”, ”9-12”, and ”13-16”. For example, ”1” indicates only one TLB has the matched entry, i.e., no sharing; ”2-4” indicates there are two to four TLBs with the matched entry.

As shown in Fig. 3, inter-TLB sharings vary with workloads, but almost all the workloads show significant sharing. Especially for LUD, SRAD, ATAX, and COLOR, most address translations can be found in 13 to 16 L1 TLBs. If we can leverage the sharing, the performance of GPU would be improved. The direct solution is to build an interconnect network among all the L1 TLBs. But making all the L1 TLBs communicate with each other will lead to many problems. For

example, it would increase area, power overhead, and access conflicts. Based on the observation that the interval "13-16" is the largest proportion in Fig. 3, if the physical address can be obtained from far TLBs, it should be also be obtained from neighboring TLBs in most cases. Moreover, a direct communication port with neighboring TLBs can be built at a lower price. Then, we conduct experiments to explore the neighboring-TLB sharing characteristic to support our idea.

3) Neighboring-TLB Sharing Characteristic: To describe the neighboring sharing behavior, we calculate the proportion of a kind of requests. These requests are missed in the local L1 TLB but can be handled by its two neighboring L1 TLBs (i.e., the correct physical address is cached in neighboring L1 TLBs). The results are shown in Table. I and Table. II. Almost all the proportion is over 20% except for part of ADI. This means that 20% of the missing requests of each L1 TLB can be served by its neighboring L1 TLBs. Even for BC, the average sharing degree is 36%. Especially for ADI, the maximum value of sharing ratio is 0.63, while the minimum value is 0.01, showing an unbalanced neighboring-TLB sharing behavior.

C. Motivation

Based on the preliminary study, we can obtain three common observations. Firstly, the low hit ratio of L1 TLB is limited by the capacity of L1 TLB. Second, many common entries exist among L1 TLBs, of which a significant portion is shared by "13-16" TLBs. Directly building an interconnect network among all the L1 TLBs can leverage the sharing behavior, but it will lead to area cost, power cost, and contentions. Third, about 20% L1 local TLB misses can be handled by neighboring L1 TLBs. This inspires us to build an interconnect network among each TLB with its two neighboring TLBs. And because each TLB only communicates with its neighboring TLBs, we can simplify the interconnect network as much as possible. Each TLB only needs to add two ports and a 2-entry-size buffer. Besides, the wire latency and length can also be optimized. However, it's still challenging to obtain the sharing information at a low price. The direct method is that on a L1 TLB miss, L1 TLB directly accesses neighboring TLBs through the interconnect network. The method does not need to add more components but would lead to time overhead. The access to L1 TLB is in the critical path, so low latency is necessary. Therefore, we add a neighboring directory table for each L1 TLB to keep track of the common entries of neighboring L1-TLBs. The directories will be retrieved in parallel when an address translation arrives at L1 TLB. Then on a L1 TLB miss, the sharing information can be quickly obtained from those directories.

III. Design and Implementation of NeiDty

In this section, we describe NeiDty, a programmer-transparent hardware scheme leveraging neighboring-TLB sharing to enhance GPU performance. We first present the overview of NeiDty. Then, we present its structure and workflow. Furthermore, we study different update policies for L1 TLB. Finally, we analyse the overhead of NeiDty.

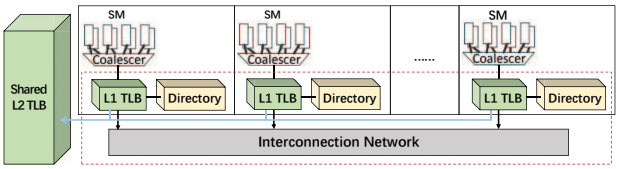

Fig. 4. System Design Architecture Overview

Valid	Hash	Left TLB	Local TLB	Right TLB
1	tag1	1	0	1
0	tag2	0	1	1
…	…	…	…	…

Fig. 5. Neighboring Directory Table Structure

A. Architectural Overview of NeiDty

As Fig. 4 shows, on the basis of traditional architecture, NeiDty adds two components: a neighboring directory table (called directory for short) and an interconnect network. The directory is used to keep track of the common entries of neighboring L1 TLBs. The interconnect network is used to connect neighboring L1 TLBs. With them, we can effectively leverage neighboring-TLB sharing, by quickly obtaining TLB sharing information from neighboring directory and directly accessing neighboring TLB through the interconnect network.

B. Implementation of NeiDty

1) The Structure of Neighboring Directory Table: As shown in Fig. 5, a directory is composed of many entries (32 entries in our experiment, the same as L1 TLB). Each entry is composed of a tag and a bitmap. For a 64-bit virtual address using 4KB page, the virtual page number (VPN) has 52 bits. To save space, we hash the 52-bit VPN to an 18-bit tag. If the 18-bit tag is matched but VPN is not matched, local TLB would continue to access L2 TLB. However, the unmatched situation will not occur in most cases, because 18-bit is big enough to decrease hash conflict. The bitmap has 3 bits, corresponding to left TLB, local TLB and right TLB. The first TLB's left TLB is the last TLB. In the bitmap, "1" indicates that the corresponding address translation is stored. The local TLB bit is mainly used for eviction strategy. Moreover, to eliminate conflict misses, the directory is designed to be fully-associative.

2) The operation of Neighboring Directory Table: When an address translation arrives at a L1 TLB, the directory would be retrieved in parallel with L1 TLB. On the L1 TLB miss, the sharing information would be returned immediately, which contains whether and which neighboring TLBs have the corresponding physical address. If both neighboring TLBs have the corresponding physical address, the local TLB would access left TLB. After obtaining address mapping from neighboring TLB, the local L1 TLB would directly stored the entry by default.

The directories sniff each update from neighboring L1 TLBs. When neighboring L1 TLBs add one entry, the directory would update the sharing information. If the corresponding tag

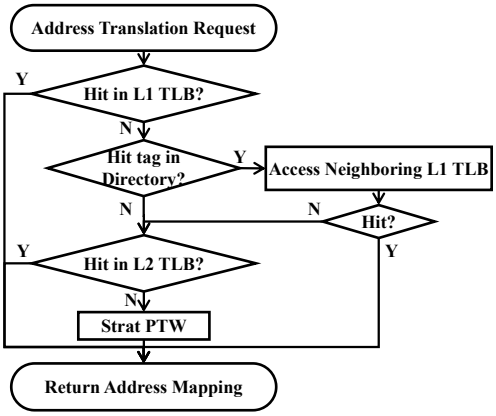

Fig. 6. Address Translation Flow Chart with NeiDty

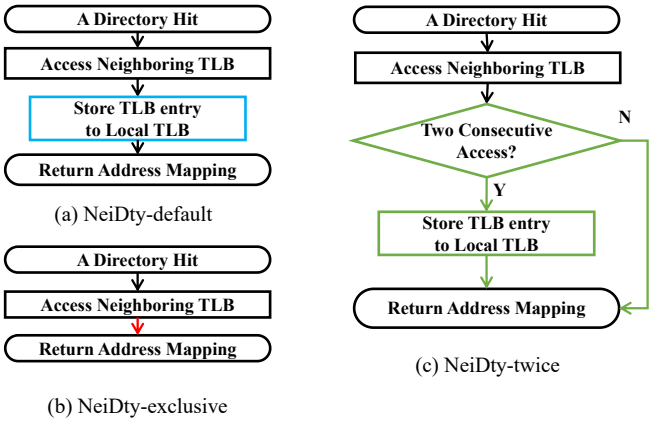

Fig. 7. Three Update Policies. NeiDty-default always stores TLB entry to local TLB, NeiDty-exclusive never stores TLB entry to local TLB, and NeiDty-twice would store TLB entry to local TLB only when it was accessed twice in a row.

of VPN has been in the directory, then update it. Otherwise, the directory will try to find a spare space to store the sharing information. Similarly, when neighboring L1 TLBs evict one entry, the directory would update the bitmap if the entry has been recorded. Because the number of directory entries is the same as one L1 TLB, it cannot record all the share information. If one directory is full, it would preferentially evict the entry that has been stored in local L1 TLB, or evict one entry according to FIFO (First In, First Out).

3) The Design of Interconnect Network: To connect each L1 TLB with its neighboring TLBs, we add three ports to L1 TLB, for left TLB, right TLB, and directory respectively. The first two ports are two-way communications to implement two-way retrieval. The overall topology of the interconnect network is like a bi-directional ring, but the scope is different. In NeiDty, the scope of the interconnect network is limited to the two neighbors. The benefit is that we do not need extra large buffers to cache the address translation request from neighboring TLBs. A two-entry buffer is sufficient. Moreover, as for scalability, the latency of the network will not become higher with the increase in SM number.

4) The Workflow of NeiDty: Fig. 6 shows the processing workflow of address translation requests in NeiDty. When receiving an address translation from GPU cores, the system first searches in the local L1 TLB. If local L1 TLB can serve it, the system would directly return the matched physical address. Otherwise, its directory would be checked. On a miss in the directory, the request would be forwarded to L2 TLB, the same as traditional schemes. On a hit in the directory, in most cases, the matched address translation would be obtained from at least one remote L1 TLB, and the matched physical address would be returned through the interconnect network, significantly decreasing the latency. The insertion and lookup of the directories can be executed in parallel with L1 TLB's operations, which are not on the critical path. Therefore, it has no impact on performance. Furthermore, fully associative structures allow NeiDty to update the directory table in a single cycle, the same as L1 TLB.

C. Study on Update Policies of Neighboring TLBs

By default, when a L1 TLB successfully obtains an address mapping from neighboring L1 TLBs, it would store it to local entries directly. This update policy (called NeiDty-default for convenience) aims at leveraging temporal locality. However, NeiDty-default cannot improve some workloads' L1 TLB local hit ratio efficiently. The reason is that NeiDty cannot improve L1 TLB reach efficiently. For those workloads, L1 TLBs are usually full due to the oversubscription. NeiDty-default will produce frequent updates and evictions, leading to severe thrashing.

In order to decrease the number of update, we propose two update policies and compare them with NeiDty-default.

NeiDty-default As shown in Fig. 7(a), on directory hit, it would access address mapping from neighboring TLBs, then directly update the mapping to local TLB. The previous chapters of this paper used default policy.

NeiDty-exclusive As shown in Fig. 7(b), on directory hit, the matched entry would never be updated from remote TLB to local TLB.

NeiDty-twice As shown in Fig. 7(c), on directory hit, the matched entry would not be updated from remote TLB to local TLB until the virtual address is accessed twice in a row.

D. Overhead Analysis

Extra storage space overhead is needed to implement NeiDty. The main space cost is the storage for the directories. For GPU with 16 L1 TLBs, there are 16 directories. Assume the page size is 4KB. The tags bits are 18 bits. Each directory only records two neighboring L1 TLBs and itself, so 3 bits are needed. In our experiment, the number of L1 TLB and directory entries are both set to 32. Besides, each L1 TLB is added two buffers to cache the interconnect internet packets. NeiDty would not increase the program's complexity and would not affect the correctness of the program. NeiDty is a space-for-time scheme, and the extra small storage space is

TABLE III
GPU SYSTEM CONFIGURATION

Component	Configuration
GPU System	1.0 GHz,16 SMs
L1 Data Cache	64 KB, 4-way, 6-cycle latency
L2 Data Cache	1 MB, 16-way, 10-cycle latency
L1 TLB	1 set, 32-way, 1-cycle latency
L2 TLB	32 set, 16-way, 10-cycle latency
IOMMU	8 page table walkers, 150-cycle latency
DRAM	512 MB, 100-cycle latency

worth it. Furthermore, the size of directory can be adjusted according to the number of shared GPU cores.

IV. EVALUATION

In this section, we evaluate the proposed NeiDty with Gem5-GPU simulator. Next, we present and analyze the experimental results.

A. Experimental Setup

Gem5-GPU [9] combines two well-known simulators: Gem5 [17], [18] and GPGPU-Sim [19]. We modified Gem5-GPU to implement NeiDty. We integrated NeiDty into the system and added L1 TLBs ports to communicate with its neighboring TLBs. And the latencies of both neighboring directory table and communication with neighboring TLB are set to be one cycle.

Most configurations of the simulator are default parameters. Table. III shows some important configurations. We select eight workloads from Rodinia [14], Pannotia [15], Poly-Bench [16] benchmark suits for evaluation: LUD, SRAD, BC, COLOR, MIS, PAGERANK, ATAX, and ADI. We compare three schemes in experiments: Baseline (original scheme), Valkyrie [20], and NeiDty. In Baseline, a traditional GPU TLB design is used, whose L1 TLBs are private and cannot communicate with each other. The probing mechanism proposed by Valkyrie [20] is implemented as comparison, which uses an intra-SE ring network for inter-L1-TLB communication.

B. Measures

With private L1 TLBs interconnected, we define the hit ratio of one private L1 TLB as the sum of local hit ratio and remote hit ratios, where local TLB hit ratio is the same as traditional L1 TLB hit ratio, and remote hit ratios are the hit ratios of remote TLBs (neighboring TLBs in NeiDty). The detailed definitions are as follows:

$$\begin{aligned} r_{L1} &= r_{local} + r_{remote} \\ r_{local} &= n_{local}/n_{sum} \\ r_{remote} &= n_{remote}/n_{sum} \end{aligned} \tag{1}$$

r_{L1}, r_{local}, r_{remote} denotes the whole L1 TLB hit ratio, local hit ratio, and remote hit ratio respectively. n_{sum}, n_{local}, n_{remote} denotes the number of all the L1 TLB requests, the number of hits on local TLB, and the number of hits on remote TLBs respectively. Moreover, we use the average time of the address translation as the performance measure.

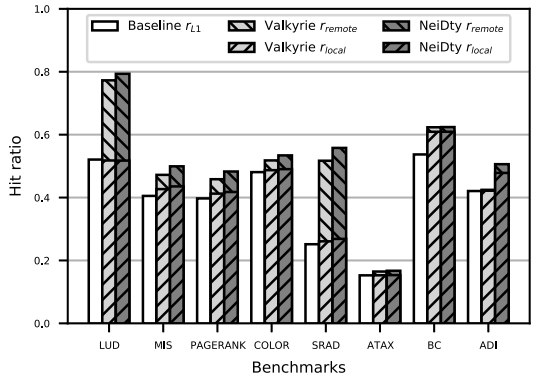

Fig. 8. TLB hit ratio over the baseline for the Valkyrie and NeiDty

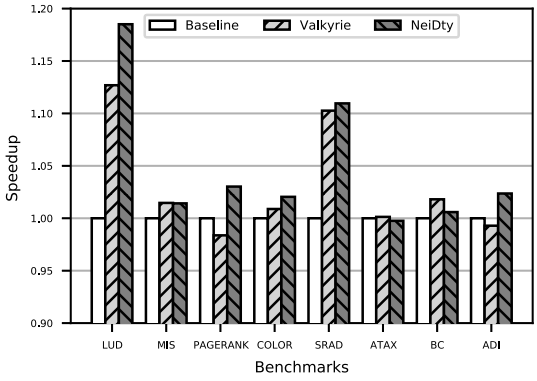

Fig. 9. Performance speedup over the baseline for the Valkyrie and NeiDty

C. Hit Ratio Results

Fig.8 shows the L1 TLB hit ratios of L1 TLB in three methods (Baseline, Valkyrie, and NeiDty), where the bottom of bars is local L1 TLB hit ratio (r_{local}), the top is remote L1 TLB hit ratio (r_{remote}), and the sum is L1 TLB hit ratio (r_{L1}), as mentioned in Section IV-B.

1) Comparison with Baseline: First, NeiDty does not affect the local TLB hit ratio for most workloads obviously. However, NeiDty improves remote TLB hit ratios. NeiDty provides an average 10% remote TLB ratio. Even for LUD, NeiDty provides a 29% remote TLB ratio. The high remote TLB ratio makes L1 TLB more efficient.

2) Comparison with Valkyrie: The probing mechanism proposed by Valkyrie is essentially based on inter-TLB sharing, similar to NeiDty, thus showing similar characteristics. However, NeiDty performs better than Valkyrie, especially for ADI, whose sharing behavior is unclear and unbalanced. Valkyrie usually needs to travel several L1 TLBs through the ring interconnect network for workloads like ADI. While NeiDty can quickly obtain sharing information from the directory, which is in parallel to the TLB access.

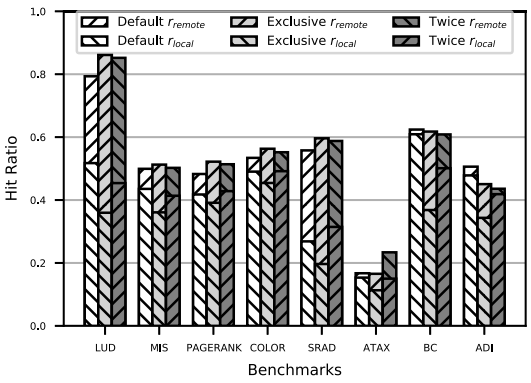

Fig. 10. TLB hit ratio for different Loading Methods

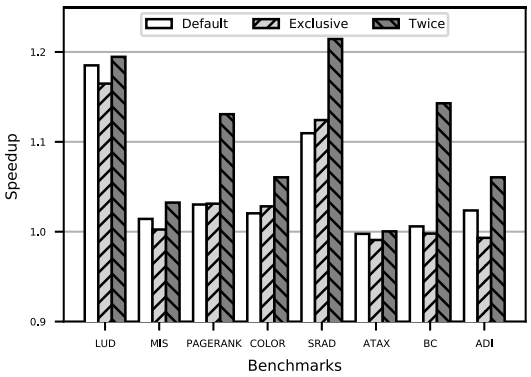

Fig. 11. Performance Speedup for different Loading Methods

D. Performance Results

We normalize the average delay of the address translation. Fig. 9 shows the speedup of the average delay of an address translation of three schemes (Baseline, Valkyrie, NeiDty). NeiDty can provide 5% average speedup over Baseline. Even for LUD, NeiDty provide 21% speedup over Baseline. Moreover, for most workloads, NeiDty is better than Valkyrie. However, both improvements of NeiDty and Valkyrie have limitations. For example, for MIS, PAGERANK and SRAD, the L1 hit ratios are improved, but the performance cannot benefit from them, which is limited by small TLB reach. To further improve L1 TLB reach, we do further improvements on update policy.

E. Update Policies of Neighboring TLBs

We conduct experiments for three update policies, as mentioned in Section III-C. Their hit ratios and performance speedup results are shown in Fig. 10 and Fig. 11. Among them, NeiDty-twice performs best for all workloads. (1) Compared with Baseline, NeiDty-twice can provide an average 10% speedup over baseline. Even for SRAD and LUD, NeiDty-twice provides 20% speedup; (2) Compared with NeiDty-default, NeiDty-twice improve TLB reach and alleviates frequent TLB insert and eviction overhead, thus making NeiDty more efficient; (3) Compared with NeiDty-exclusive, NeiDty-twice cannot provide a much higher L1 TLB hit ratio, but its local hit ratio has a higher proportion. For BC, similar to NeiDty-exclusive, NeiDty-twice has about 60% L1 TLB hit ratio, but NeiDty-twice has up to 50% local hit ratio.

V. Related Works

To leverage inter-TLB locality, Baruah et al. [20] proposed Valkyrie with a prefetching mechanism for L2 TLB using a Locality Detection Table (LDT) and a probing mechanism by interconnecting the L1-TLBs through an on-chip ring network. NeiDty is similar to Valkyrie's probing mechanism, but NeiDty focuses on neighboring TLBs rather than inter TLBs. Moreover, NeiDty is easier to implement without complex network policy designs. Besides, NeiDty uses directories to keep track of sharing of neighboring TLBs and adopt more effective update strategies to improve the TLB reach.

Besides, there are many other studies on GPU address translation. Power et al. [5] evaluate the main component in GPU, which shows that modest hardware changes can improve GPU address translation in low overhead. Pichai et.al [12] also designed a basic GPU MMU model but they focused on warp scheduling. Yoon et al. [21] proposed a software transparent virtual cache hierarchy for GPU, where applications can directly access data from caches bypassing TLB unless a miss occurs in caches. Shahar et al. [22] designed a software address translation, which covered a software TLB and a translation aggregation algorithm to support memory mapped file. Shin et al. [23] proposed a specific SIMT-aware scheduler, reordering the page table walks based on shortest-job-first scheduling and batching the requests, reducing the stalls and making page table walker work better.

Ausavarungnirun et al. [6] proposed a application-transparent GPU memory manager. Ausavarungnirun et al. [24] designed a memory hierarchy for supporting multi-application workloads, and improves performance of page table walker cache and shared TLB by using TLB-fill tokens, L2-TLB bypassing, and an effective dram scheduler. Yan et al. [25] proposed *Translation Ranger*, to provide operating system support for the contiguity-aware TLBs by migrating physical pages in a background daemon process. Jaleel et.al. [26] proposed DUCATI to store address translations in last-level cache and memory, increasing TLB reach. Skarlatos et.al. [27] proposed a page table design based on Elastic Cuckoo Hashing. Achermann et.al. [28] proposed Mitosis, to transparently replicate and migrate page-tables across sockets. Tang et.al. [13] designed a compression mechanisms for TLB to increase TLB reach. Kotra et.al. [8] proposed a mechanism to leverage those under-utilized instruction cache and shared memory to improve TLB reach without additional hardware modification. Li et al. [29] proposed a TLB hierarchy (*least-TLB*), employing least-inclusive policy and using a Cuckoo filter to track the translations in multiple GPUs' L2 TLBs to increase TLB reach.

VI. Conclusion

We proposed NeiDty to leverage Neighboring-TLB sharing effectively. The key idea of NeiDty is to enable L1 TLB can quickly access address mapping from neighboring TLBs at very low price. Furthermore, we study three different update policies of NeiDty and found that NeiDty-twice is more efficient for many workloads. Equipped with NeiDty-twice policies, NeiDty can increase the average hit ratio of L1 TLB by 14% and improve the average performance by 10%, significantly improving GPU performance across various workloads.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This research was funded by Shenzhen Fundamental Research Program (No. JCYJ20210324122406017), NSFC (No. 62002339) and the Strategic Priority Research Program of the Chinese Academy of Sciences (No. XDB44030200).

References

[1] NVIDIA, NVIDIA, 2011. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-40

[2] M. Harris, NVIDIA, 2013. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[3] Nvidia, NVIDIA, 2016. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[4] N. Sakharnykh, NVIDIA, 3 2018. [Online]. Available: https://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[5] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 568–578.

[6] 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: Association for Computing Machinery, 2017, p. 136–150.

[7] T. Karnagel, T. Ben-Nun, M. Werner, D. Habich, and W. Lehner, "Big data causing big (tlb) problems: Taming random memory accesses on the gpu," in *Proceedings of the 13th International Workshop on Data Management on New Hardware*, ser. DAMON '17. New York, NY, USA: Association for Computing Machinery, 2017.

[8] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing gpu translation reach by leveraging under-utilized on-chip resources," ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021.

[9] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[10] NVIDIA, NVIDIA, 2022. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[11] ——, "Nvidia turing gpu architecture," Website, 2019, https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[12] 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: Association for Computing Machinery, 2014, p. 743–758.

[13] 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.

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

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

[16] J. Karimov, T. Rabl, and V. Markl, "Polybench: The first benchmark for polystores," in *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*, R. Nambiar and M. Poess, Eds. Cham: Springer International Publishing, 2019, pp. 24–41.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[18] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[19] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[20] 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," ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 455–466.

[21] H. Yoon, J. Lowe-Power, and G. S. Sohi, *Filtering Translation Bandwidth with Virtual Caching*. New York, NY, USA: Association for Computing Machinery, 2018, p. 113–127.

[22] S. Shahar, S. Bergman, and M. Silberstein, "Activepointers: A case for software address translation on gpus," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 596–608.

[23] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular gpu applications," ser. ISCA '18. IEEE Press, 2018, p. 180–192.

[24] 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," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 503–518, 2018.

[25] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation ranger: Operating system support for contiguity-aware tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 698–710.

[26] 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.

[27] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, *Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1093–1108.

[28] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, *Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines*. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–300.

[29] 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.