LATOA: Load-Aware Task Offloading and Adoption in GPU

Hossein Bitalebi
KTH Royal Institute of Technology
Sweden, Stockholm
hobita@kth.se

Vahid Geraeinejad
KTH Royal Institute of Technology
Sweden, Stockholm
vahidg@kth.se

Farshad Safaei
Shahid Beheshti University
Iran, Tehran
f_safaei@sbu.ac.ir

Masoumeh Ebrahimi
KTH Royal Institute of Technology
Sweden, Stockholm
mebr@kth.se

ABSTRACT

The emerging new applications, such as data mining and graph analysis, demand extra processing power at the hardware level. Conventional static task scheduling is no longer able to meet the requirements of such complicated applications. This inefficiency is a major concern when the application is supposed to run on a Graphics Processing Unit (GPU), where millions of instructions should be distributed among a limited number of processing cores. A non-optimal scheduling strategy leads to unfair load distribution among the GPU's processing cores. Consequently, while busy cores are stalled due to the lack of resources, waiting for their data from the main memory, other cores are idle, waiting for busy cores to complete their tasks. Our study introduces LATOA, a Load-Aware Task Offloading and Adoption method that tackles this problem by reducing both stall and idle cycles. LATOA is the first study moving from static to dynamic task scheduling based on run-time information obtained from the Miss Status Holding Register (MSHR) tables. In LATOA, all processing cores are dynamically tagged with critical, neutral, or relaxed states. Then, irregular warps with low locality properties are detected and offloaded from critical cores (going to the stall state) to relaxed ones (going to the idle state). Based on our experiments, LATOA reduces the number of stall cycles on average by 24% and increases the neutral states on average by 38%. In addition, with negligible hardware overhead, LATOA improves system performance and power efficiency on average by 26% and 7%, respectively.

KEYWORDS

GPU, cache access, offloading, adoption, unbalanced load distribution, irregularity, locality, stall cycle

ACM Reference Format:
Hossein Bitalebi, Vahid Geraeinejad, Farshad Safaei, and Masoumeh Ebrahimi. 2023. LATOA: Load-Aware Task Offloading and Adoption in GPU. In *15th Workshop on General Purpose Processing Using GPU (GPGPU '23), February 25, 2023, Montreal, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589236.3589243

GPGPU '23, February 25, 2023, Montreal, Canada

ACM ISBN 979-8-4007-0776-6/23/02.
https://doi.org/10.1145/3589236.3589243

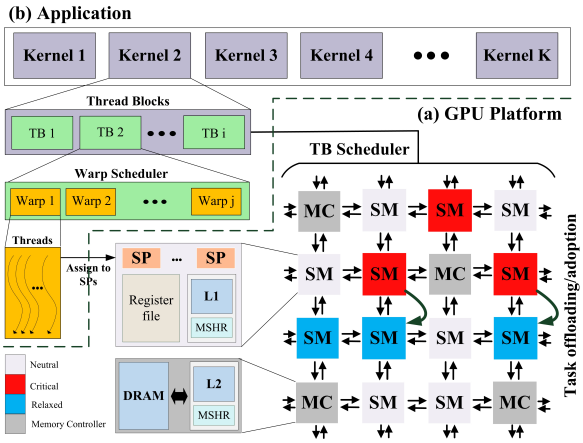

Figure 1: An overall scheme (a) GPU platform; (b) Application running on GPU

1 INTRODUCTION

Graphics Processing Unit (GPU) is renowned for its ability to run thousands of threads at the same time in the form of Thread Level Parallelism (TLP) [10]. Figure 1(a) shows a conventional GPU architecture according to the NVIDIA term. The architecture consists of Streaming Multiprocessors (SMs), each with a register file, an L1 private cache, and several Streaming Processors (SPs) to form Single Instruction Multiple Thread (SIMT). If data cannot be provided by the L1 cache, requests would be sent to the memory controller through the interconnection network to access the L2 cache and the main memory.

From the application perspective, an application is composed of several kernels, as shown in Figure 1(b). Each kernel is divided into multiple thread blocks (TB). TB is the smallest unit of assignment to the cores. During the execution of an application, one or several TBs may be assigned to a processing core by the TB scheduler. Further, each TB is divided into several warps where the execution of warps is independent of each other. Each warp is composed of a number of threads that, in the utilized architecture, is equal to the number of SPs. Then, each thread is assigned to one SP for parallel execution. The warp scheduler aims to order warps to reduce execution time and enhance thread-level parallelism.

If the execution of a warp stalls due to the L1 miss, the processing core proceeds with the execution of the next warp. In case all

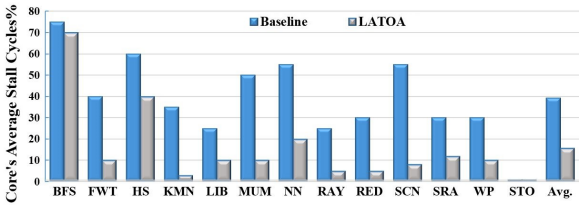

Figure 2: Average processing core's stall cycles during the application execution

warps stall, waiting for their data from the lower levels of the memory hierarchy, the entire core goes to the stall state. Our primary analysis (see Table 2 for simulation setup), depicted in Figure 2, shows that the average stall cycles of cores are very large under different examined applications (listed in Table 1). As can be seen from this figure, the stall rate ranges from 25% in LIB and RAY to around 75% in the BFS benchmark. On average, 40% of cores are stalled, waiting for their memory requests to be responded to. The execution of an application will not be finished until all processing cores complete their duties. Thereby, slow cores become a barrier to performance. This has motivated us to propose a solution to reduce the number of stall states of the processing cores and thus improve GPU performance. We propose a Load-Aware Task Offloading and Adoption method (LATOA) to balance the load among the GPU processing cores based on run-time information. LATOA provides a simple yet effective approach to determine irregular warps for offloading from critical cores (red SMs in Figure 1(a)) to the neighboring relaxed cores (blue SMs in Figure 1(a)). With this scheme, LATOA not only affects the system's performance by providing balanced load distribution among the SMs but also improves the locality property of a critical SM by offloading the irregular warps at run-time.

To the best of our knowledge, this is the first effort to dynamically determine the status of the processing cores at run-time, determine suitable warps for offloading, and send the selected warps to the relaxed cores for adoption. The main contributions of this paper are as follows:

- We monitor the execution behavior of the GPU cores to determine their states in terms of load pressure at run-time and whether a core is close to its stall or idle state. Accordingly, cores are dynamically tagged with one of the three states: Relaxed, Neutral, and Critical. We define threshold values based on the information of the L1 missed status table.
- We propose a mechanism to determine the warps that should be offloaded, with the aim of bringing the most benefit to the critical core and avoiding them from entering the stall state. In addition, we select a relaxed core among the neighboring cores to adopt the warps with the goal of preventing the core from entering an early idling state.
- We perform an extensive set of analyses on a large set of benchmark suits, confirming the superior benefits of LATOA over the baseline and a state-of-the-art method in achieving dynamic load balancing. LATOA is able to reduce the number of stall cycles and improve the system performance on average by 24% and 26%, respectively.

2 RELATED WORK

In recent years, GPU has played a significant role in accelerating new emerging applications such as speech recognition, image and video classification, and data analysis. Enormous studies and efforts have been dedicated to improving GPU efficiency in processing these intricate applications [17, 18]. In the following, we review some of these studies.

Task Scheduling: Efficient task and resource scheduling and management have always been an important area of research. Several studies have tried to optimize the warp scheduler within a processing core to maximize TLP [7, 8, 20]. Various techniques have been proposed to efficiently allocate TBs to the processing cores [14, 22, 23]. However, these works have mostly focused on distributing TBs among the processing cores based on static compile-time information. In other words, they have neglected the consequences of static TB distribution on different cores at run-time.

Resource Utilization: Despite the potential of parallel processing in GPU, the TLP rate could drop significantly due to stall cycles. There are numerous studies to facilitate GPU application execution by improving TLP [16]. Most of these works demand extra processing and storage resources despite the fact that there are noticeable unutilized resources available due to the processing cores' stall cycles. Another group of studies has suggested utilizing available resources while waiting for the response from the main memory [12, 15, 19, 25]. NURA [6] has proposed a resource-sharing process to take advantage of the stalled cores' resources such as shared memory and register file in favor of active cores. Zorua et al. [24] has developed a compiler-based approach to utilize the shared memory as the swap space for context switching among TBs through virtualizing the resources. Kim et al. [13] proposed warp instruction reuse, allowing repeated warp instructions to reuse the results of previous computations instead of re-executing similar instructions. Chao et al. [26] has focused on increasing the register file size to improve the TLP rate by utilizing the shared memory space as a register file. In [2], the impact of distributing the processing cores on the memory hierarchy levels has been studied, which showed to enhance near-memory processing and thus performance.

3 BACKGROUND AND MOTIVATION

In this section, we first explain the functionality of the MSHR table which is an important resource to extract run-time information. Then, we shortly explain the issue of static task scheduling in GPU.

3.1 MSHR Table

Each cache bank at different levels of the memory hierarchy holds a table called Miss Status Holding Register (MSHR), which keeps the status of the missed cache accesses. Figure 3 shows an MSHR table that is used in a conventional GPU. MSHR is composed of several entries, where each entry contains the cache block address and the number of slots. A slot holds the information on the missed accesses, such as requester ID, format bits, offset bits, and the data buffer.

Cache misses can be classified into two categories: 1) Primary miss: indicating that the existing cache lines do not contain the newly requested address, and thus a new MSHR entry must be assigned to the new request. *Entry-Pointer* (in Figure 3) is used

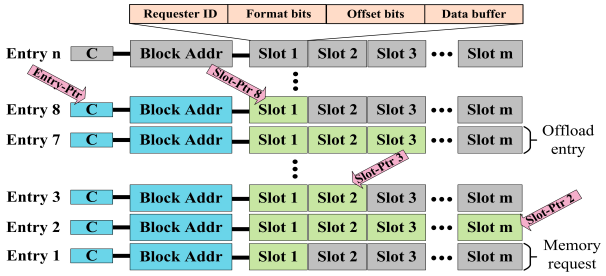

Figure 3: A conventional MSHR table in GPU

to track the number of primary misses; and 2) Secondary miss: showing that the newly requested address shares the same cache line as the previous miss(es). In that case, no new entry will be assigned to the request in MSHR since the cache line is already in progress to be provided. Instead, a new slot in the matching entry is allocated to the request. *Slot-Pointer* (in Figure 3) is used to trace the number of filled slots of an entry [9]. As shown in the figure, memory requests are issued in the FIFO fashion, with Entry 1 always being the next memory request to be transmitted. After a request is issued, all entries will be shifted down.

As long as there are empty entries in MSHR, the cache can accept new misses from the processing cores. However, if all entries are filled up, the core has to be stalled as it cannot proceed with the next instruction with a miss access. Irregular memory requests are one of the main reasons leading to early MSHR overflow and core stalls. Irregularity refers to a memory request that is not synchronized with most existing requests (i.e., low locality rate among different memory requests). In this case, an entire entry should be allocated to the address block needed by the irregular request, while due to request irregularity, this address block might not be used by current and probably future memory requests. This is similar to entries 1 and 8 in Figure 3 with a single slot filled in. In this study, the warps that produce irregular memory requests are named irregular warps. As another consequence, when the data block is returned back, the cache controller may have to evict one of the current data lines, which might be highly demanded, and bring in a new one with a lower chance of reuse [27].

3.2 Processing Cores' Load Unbalance

As previously mentioned, the execution of an application is finished when all processing cores have completed their duties. This puts the slower cores as a barrier to performance. Our study confirms the fact that different processing cores within a GPU do not follow the same execution time. While some cores are busy with handling irregular requests (e.g., branch predictions), others are relaxed. These busy cores usually spend a considerable number of stall cycles waiting for a response from the main memory associated with their irregular memory requests. On the other hand, cores that are not stuck in the trap of requests' irregularity perform their duties normally without facing any specific pipeline stalls. Nevertheless, these cores have to remain idle until busy cores complete their tasks, thus suffering from under-utilization.

The issue is due to the fact that conventional TB scheduling methods rely on compile-time information, trying to equally distribute TBs among the processing cores [1]. However, the execution behavior and the processing time of a TB cannot be accurately predicted at the compile time due to the uncertainty of input values and conditional branches. Therefore, we argue that tasks should be fairly assigned to cores based on run-time information rather than distributed equally based on compile-time information. LATOA takes advantage of the MSHR table to make run-time decisions in order to uniform load balancing among all processing cores.

4 WARP OFFLOADING/ADOPTION

Despite the enormous processing potential of GPU owing to the power of task parallelism, the processing cores might switch to the stall state for many execution cycles. This might happen due to resource limitations where all entries of the MSHR table are filled in and waiting for the data from the lower memory hierarchy levels. To reduce the stall states, we detect the processing cores that are more likely to get stalled and offload their irregular warps to the relaxed cores, which might face idle cycles. LATOA consists of three major steps, explained as follows:

4.1 Candidate Cores for Offloading

We tag the cores with three states as Critical, Neutral, and Relaxed. Whenever a core is likely to switch to the stall state, we name it a **critical** core. On the contrary, the core that is away from the irregularity issues and may enter an idle state is named a **relaxed** core. There is a third type of core that we call a **neutral** core. These cores are neither critical nor relaxed. Our ultimate goal is to enable all cores to operate in their neutral state. According to our evaluations, the number of unresponded memory requests in a processing core is directly linked to the state of the core. The number of filled entries in the MSHR table is a reliable indicator that a core is close to go to the stall or idle state.

We divide the number of MSHR entries (i.e., 64 in our experiments) into three equal levels to categorize processing cores into one of the three aforementioned processing states. The cores with the filled entries between zero and 1/3 of the total number of MSHR entries (i.e., 21 in our experiment) are tagged as relaxed; the cores with filled entries between 1/3 to 2/3 are tagged as neutral, and the cores with filled entries of more than 2/3 are tagged as critical. To hold the core state, a 2-bit flag is considered to store one of three states. The processing core is relaxed, neutral, and critical if the flag is set to 00, 01/10, or 11, respectively.

When a core is tagged as critical, certain warps should be selected for offloading, which could benefit the critical core the most. This is done by exploring the irregularity rate of the L1 missed memory requests through the MSHR table information, which will be discussed next.

4.2 Candidate Warps for Offloading

Now that we have identified the critical cores, we must select suitable warps for offloading, as random warp offloading may hurt the overall performance by imposing extra overhead. We provide a mechanism to determine the warps that generate irregular memory accesses within a critical processing core and offload them into the relaxed cores.

4.2.1 Instruction Type. Various types of instructions are required to complete the execution of an application. For example, conditional

instructions, such as Jump, are performed without needing any operands. Furthermore, instructions responsible for calculating effective addresses do not need to access any memory hierarchy levels. In this study, we consider warp as a potential candidate for offloading if all of the threads within the warp are running the same instruction with two independent operands which need to access one of the memory hierarchy levels (L1, LLC, or main memory). An example of a two-operand instruction is "Addition". With two-operand instructions, four cache patterns exist as hit-hit, hit-miss, miss-hit, and miss-miss. Except for the hit-hit pattern, which will be immediately responded to, the address block(s) of the cache miss access(es) has to be stored in the corresponding entry(s)/slot(s) of the MSHR table. In sum, two-operand instructions with at least one miss access are considered the candidates for offloading. Among these candidates, the ones with irregular memory access requests are selected for offloading as will be discussed in the next subsection.

4.2.2 Warps Irregularity. During task scheduling, some processing cores may receive the portion of the application that imposes divergent execution flow and repeatedly calls for data blocks that are not located in the cache lines. In other words, the rate of the locality property among the memory requests becomes very low [3, 4]. As a result, data blocks that are currently located in the cache (after the costly data movement from the main memory) cannot be reused effectively and should be evicted from the cache lines due to the variety of the demanded data blocks. Under this circumstance, the cache fails to work efficiently in dealing with irregular parts of the application, and the rate of cache misses will dramatically increase. Consequently, MSHR entries are quickly filled up, which leads to the core stall.

To identify irregular warps, we utilize the number of filled slots in an MSHR table, which we believe is a beneficial criterion of the (ir)regularity of the requested address block. The larger number of filled slots in an entry means a better regularity of the corresponding memory block, whereas the small number of filled slots indicates a poor locality. Accordingly, we count the number of filled slots of an entry and compare it to a threshold value. The threshold value is considered to be 25% of the total number of slots in an entry, confirmed experimentally to be a suitable threshold.

Thereby, an entry is counted as unpopular if it is filled by less than or equal to 25% of its total number of slots. Considering an MSHR table with 16 slots per entry, similar to our experiments, the threshold value will be 4. If the number of filled slots of the examined entry is greater than 4, it implies the memory access is regular, and thus the importance of the data block. So, it is desirable to bring the required data block to the cache rather than offloading the corresponding warp to another core. However, a value less than 4 indicates the unpopularity of the data block, and thus warps belonging to all slots in that entry are counted as irregular and thus offloaded. Consequently, the entire entry will be emptied. In case the cache access pattern of an offloaded warp is miss-miss, then the second slot should also be discarded from the corresponding entry and slot of the MSHR table.

4.3 Offloading Warps

As was mentioned already, warps' execution is independent of each other, so there is not any dependency problem with offloading.

Table 1: Benchmarks' specification

Name	Abrr.	Suite	Name	Abrr.	Suite
Breadth first search	BFS	Rodinia	LIBOR	LIB	CUDA
Kmeans	KMN	Rodinia	Ray tracing	RAY	CUDA
Neural network	NN	Rodinia	Scan	SCN	CUDA
SRAD	SRA	Rodinia	Fast walsh transform	FWT	CUDA
Reduction	RED	Rodinia	Weather prediction	WP	CUDA
Mummergpu	MUM	Rodinia	StoreGPU	STO	CUDA
Hotspot	HS	Parboil			

Table 2: GPU configuration

Parameter	Baseline GPU configuration
Total cores	56 processing cores
Per core	32 warp width, 8 TBs, 1536 threads, 32768 registers 48 KB scratchpad memory, round robin TB scheduling
L1 data cache	32 KB, 4-way, 128B block size, MSHR: 64 entry, 16 slots
LLC (L2) cache	8 x 128K, 16-way,
Memory	FR-FCFS scheduler, DDR3-1333H, 8 memory channel
Core, L2 clock	700 MHz, 700 MHz
Interconnect	2D mesh, XY routing, 1 core/node, 4 VCs, 4 routing latency, 1 channel latency

When a warp is offloaded to another SM, all its valid data is also offloaded to the new SM. On the host side, the offloading behaves like a normal warp that has finished all its executions and should be terminated. On the guest side, it looks like a new warp has been assigned to the SM with some initial values.

LATOA suggests offloading warps from Entry 7 of the MSHR table (see Figure 3). The reason is that offloading from the oldest requests (i.e., Entry 1 to 6) may cause pipeline stalls in the architecture (the concept is out of the scope of this paper). On the other hand, offloading from one of the newest filled entries (e.g., Entry 8) is unsuitable as the later requests may demand the same address block, leading to more filled slots for that entry in the future.

4.4 Candidate Core for Warp Adoption

Now that we have identified the critical cores and the irregular warps for offloading, we need to find a suitable relaxed core for adopting warps. This, in turn, prevents the relaxed core from entering an early idling state. Offloading a warp without considering the core's position in the network may lead to additional network traffic. To minimize the offloading overhead to the interconnection network, relaxed cores will only be selected among the neighboring cores.

Considering a mesh topology (see Figure 1), each router is connected to four neighbors except for the border and corner cores. To this end, each core is equipped with a 4-bit register to hold the state of its neighbors, where 1-bit is allocated to each neighbor. When a core switches to the relaxed state, it propagates a positive signal to its neighbors, showing that the core is ready to adopt warps. On the contrary, when a core exits the relaxed state, it propagates a negative signal to the neighbors, announcing that the core is no longer interested in adopting a new warp. The critical core checks its 4-bit status register in a specific order (e.g., East, North, West, and South) to find the first relaxed core for offloading its warps. To maintain locality in the new host, the warps will be offloaded to the selected core until it is no longer in its relaxed state.

5 EVALUATION

In this section, LATOA is evaluated and analyzed in various aspects, including performance, power efficiency, processing core states, and cost overhead. In addition to the baseline, LATOA is compared with one of the state-of-the-art schemes, CLAMS [11]. In CLAMS, each core is given a ranking, calculated by dividing the number of ready-warps by the total number of warps (i.e., ready and block warps) in a core, which is a value between 0 and 1. This ranking is carried out by packets to the memory controller where memory requests are prioritized based on their rankings (or ranking range). In this way, requests from critical cores will be served earlier, balancing the execution time of all cores.

5.1 Simulation Setup

LATOA is implemented by GPGPU-Sim v3.2 [1], a cycle-accurate simulator according to the Fermi-liked architecture. We have used various benchmarks from different benchmark suits such as Rodinia [5], Parboil [21], and CUDA SDK [28], listed in Table 1. As shown in Table 2, the baseline architecture has 56 processing cores, and each core has a 32 KB private L1 data cache. Cores have access to 8*128 KB shared L2 cache through the interconnection network.

5.2 Performance Evaluation

Figure 4 illustrates performance reports normalized to the baseline, evaluated on various benchmark suites. The system performance is reported based on the Instruction-Per-Cycle (IPC) criterion. In comparison to the baseline, LATOA has successfully increased the system performance by a maximum of 48% and an average of 26%. On average, LATOA has achieved 15% better performance improvement than CLAMS over all the benchmarks. Applications like SCN, MUM, NN, KMN, and FWT are considerably memory intensive. Which means, they frequently generate memory requests during the execution. In addition, the load pressure of their processing cores follows an unbalanced behavior where some cores are relaxed while others are critical. Such applications with unbalanced load pressure characteristics at the run time are more likely to take advantage of LATOA. SCN and MUM, as the representatives of this group, successfully achieved 48% and 43% performance improvement compared to the baseline, respectively. As is clear from the figure, even though the BFS application is also memory intensive, LATOA has not been capable of improving the performance by more than 4%. The reason is that during the execution of BFS, almost all processing cores are in their critical states and there are not enough relaxed cores for adopting their warps.

Among the tested applications, RAY is heavily sensitive to locality property among the memory requests. As was mentioned, LATOA improves the locality property of processing cores by offloading irregular requests, thus leaving more space to hold in-use data blocks. That is why the performance of RAY has significantly improved by around 20%. On the other hand, LATOA is unable to improve the performance of STO as STO is severely process intensive, and most of its requested data is provided by the shared memory.

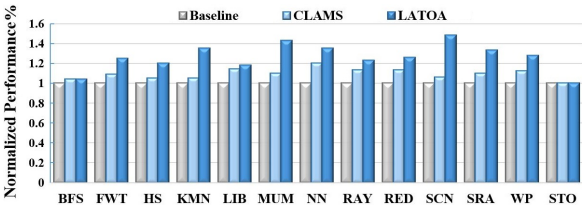

Figure 4: Performance evaluation

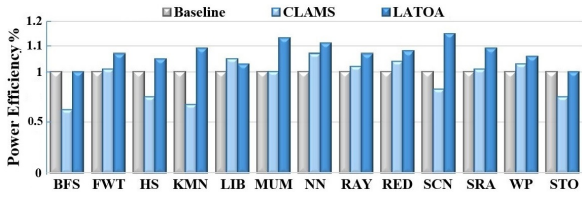

Figure 5: Power evaluation

5.3 Power Consumption

Figure 5 demonstrates the power efficiency based on Instruction per Joule (Ins/J) that is normalized to the baseline. LATOA succeeds in reducing power consumption, on average, by 7% (Ins/J). This achievement is mainly due to increased data block reusability in critical cores, lower queuing delay in injection buffers, and lower static and leakage power consumption because of the reduced stalled and idle cycles. Contrary to LATOA, CLAMS imposes extra power overhead on the system by an average 3%. This inefficiency is due to the complexity of calculating the criticality ranking of processing cores and additional bytes added to the memory request packets.

5.4 Processing Core States

Various applications usually show different execution behavior. This makes predicting their demands based on static information a difficult task to achieve. Due to static task scheduling, some processing cores have to tolerate more load pressure than others. Figure 6(a) depicts processing cores' states during the application execution for all the utilized benchmarks. In this figure, the upper portion of the bars (in red color) demonstrate the average critical state of all cores for the corresponding application, whereas the middle and lower portions of the bars (in white and blue colors, respectively) demonstrate the average neutral and relaxed states of all cores, respectively. Based on the obtained results, the processing cores are in their relaxed, neutral, and critical states on average by 29%, 23%, and 48%, respectively, in the baseline architecture. Despite the significant rate of cores in their critical state, there is a considerable number of processing cores that are relaxed. Therefore, some processing cores are highly susceptible to switching into the stall state due to a lack of resources while others may switch to the idle state with free resources. LATOA, on the other hand, explores the processing core states at run time and offloads some irregular instructions from critical cores to relaxed ones.

Figure 6(b) shows the advantage of LATOA in balancing the load distribution among processing cores by increasing the neutral states. The average relaxed state and critical state are decreased by 18% and 20%, respectively. Consequently, LATOA increases the

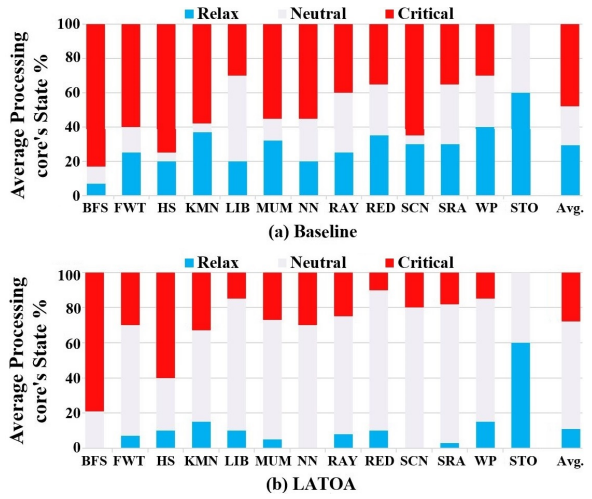

Figure 6: Core state during the applications' execution.

neutral state of the cores to 61% from that of 23% in the baseline architecture.

5.5 Hardware Overhead

LATOA imposes negligible cost overhead to the system due to its simple implementation both in offloading and adoption mechanisms. The number of filled entries should be compared with pre-defined threshold values (i.e. 1/3 and 2/3 of the total number of entries, as explained in Section 4.1) to determine core states. A 2-bit register per core should be added to hold one of the three core states. Moreover, for detecting the (ir)regularity of an entry in a critical core, the number of filled slots should be compared with a pre-defined threshold value (i.e., 1/4 of the total number of slots, as discussed in Section 4.2). In both computations, a 6-bit comparator is sufficient, considering an MSHR table with 64 entries and 16 slots per entry. We have utilized an already existing comparator in the MSHR baseline architecture, thus adding no extra overhead. To hold the state of the neighboring cores, each core has to be equipped with a 4-bit register.

6 CONCLUSION

This paper tackled the task scheduling issue in GPUs, which results in a considerable number of the stall and idle cycles in the processing cores. We observed that the cores with higher load suffer from stall cycles when all entries of their MSHR are filled. On the other hand, cores with lower loads undergo a short execution cycle and were forced to go to the idle state until all other cores completed their duties. We proposed LATOA to detect cores entering the stall or idle states using the rich run-time information of the MSHR table. Then, we proposed an approach to carefully detect and offload irregular warps with low locality properties. This strategy not only leaves some MSHR entries empty to prevent the core from switching to the stall state but also improves the locality property among current and future requests by avoiding unnecessary eviction of in-use data blocks from the cache. LATOA showed promising results by reducing the critical and relaxed states by 20% and 18% while improving the performance and power efficiency on average by, 26% and 7%.

ACKNOWLEDGMENTS

The research was supported by STINT project MG2018-8007.

REFERENCES

[1] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE symposium on ISPASS*. IEEE, 163–174.
[2] Hossein Bitalebi, Vahid Geraeinejad, and Masoumeh Ebrahimi. 2022. Near LLC versus near main memory processing. In *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*. 1–6.
[3] Hossein BiTalebi and Farshad Safaei. 2021. LARA: Locality-aware resource allocation to improve GPU memory-access time. *The Journal of Supercomputing* 77, 12 (2021), 14438–14460.
[4] Hossein Bitalebi and Farshad Safaei. 2022. Criticality-aware priority to accelerate GPU memory access. *The Journal of Supercomputing* (2022), 1–26.
[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE symposium on workload characterization (IISWC)*. Ieee, 44–54.
[6] Sina Darabi, Negin Mahani, Hazhir Baxishi, Ehsan Yousefzadeh-Asl-Miandoab, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2022. NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–27.
[7] Cong Thuan Do, Hong Jun Choi, Sung Woo Chung, and Cheol Hong Kim. 2020. A novel warp scheduling scheme considering long-latency operations for high-performance GPUs. *The Journal of Supercomputing* 76, 4 (2020), 3043–3062.
[8] Ahmed ElTantawy and Tor M Aamodt. 2018. Warp scheduling for fine-grained synchronization. In *IEEE Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 375–388.
[9] Yongbin Gu and Lizhong Chen. 2019. Dynamically linked MSHRs for adaptive miss handling in GPUs. In *Proceedings of the ACM International Conference on Supercomputing*. 510–521.
[10] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Symposium on ISCA*. 152–163.
[11] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2016. Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. 351–363.
[12] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. Regmutex: Inter-warp gpu register time-sharing. In *Symposium on ISCA*. IEEE, 816–828.
[13] Keunsoo Kim and Won Woo Ro. 2018. WIR: Warp instruction reuse to minimize repeated computations in GPUs. In *IEEE Symposium on HPCA*. IEEE, 389–402.
[14] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware CTA clustering for modern GPUs. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 297–311.
[15] Zhongjin Li, Victor Chang, Haiyang Hu, Maozhong Fu, Jidong Ge, and Francesco Piccialli. 2021. Optimizing makespan and resource utilization for multi-DNN training in GPU cluster. *Future Generation Computer Systems* 125 (2021), 206–220.
[16] Zhen Lin, Hongwen Dai, Michael Mantor, and Huiyang Zhou. 2019. Coordinated CTA combination and bandwidth partitioning for GPU concurrent kernel execution. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 3 (2019), 1–27.
[17] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. 2021. Batch-Sizer: Power-Performance Trade-off for DNN Inference. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC '21)*. 819–824.
[18] Seyed Morteza Nabavinejad, Sherief Reda, and Masoumeh Ebrahimi. 2022. Coordinated Batching and DVFS for DNN Inference on GPU Accelerators. *IEEE Transactions on Parallel and Distributed Systems* (2022).
[19] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella Ferrer, and Francisco J Cazorla. 2019. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier. In *Euromicro Conference on ECRTS*, Vol. 23.
[20] Jayati Singh, Ignacio Sañudo Olmedo, Nicola Capodieci, Andrea Marongiu, and Marco Caccamo. 2022. Reconciling QoS and concurrency in NVIDIA GPUs via warp-level scheduling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1275–1280.
[21] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.
[22] Devashree Tripathy, Amirali Abdolrashidi, Laxmi Narayan Bhuyan, Liang Zhou, and Daniel Wong. 2021. Paver: Locality graph-based thread block scheduling for gpus. *ACM Transactions on TACO* 18, 3 (2021), 1–26.
[23] Devashree Tripathy, Amirali Abdolrashidi, Quan Fan, Daniel Wong, and Manoranjan Satpathy. 2021. Localityguru: A ptx analyzer for extracting thread block-level

locality in gpgpus. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–8.

[24] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *IEEE/ACM Symposium on MICRO*. IEEE, 1–14.

[25] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 609–621.

[26] Chao Yu, Yuebin Bai, Qingxiao Sun, and Hailong Yang. 2018. Improving thread-level parallelism in GPUs through expanding register file to scratchpad memory. *ACM Transactions on TACO* 15, 4 (2018), 1–24.

[27] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, and Zhiying Wang. 2020. Coordinated page prefetch and eviction for memory oversubscription management in gpus. In *IEEE IPDPS*. IEEE, 472–482.

[28] Cyril Zeller. 2011. Cuda c/c++ basics. *NVIDIA Coporation* (2011).