PAPER

# LFWS: Long-Operation First Warp Scheduling Algorithm to Effectively Hide the Latency for GPUs

**Song LIU**†, ***Member***, **Jie MA**†, **Chenyu ZHAO**†, **Xinhe WAN**†, ***and*** **Weiguo WU**†a), ***Nonmembers***

**SUMMARY** GPUs have become the dominant computing units to meet the need of high performance in various computational fields. But the long operation latency causes the underutilization of on-chip computing resources, resulting in performance degradation when running parallel tasks on GPUs. A good warp scheduling strategy is an effective solution to hide latency and improve resource utilization. However, most current warp scheduling algorithms on GPUs ignore the ability of long operations to hide latency. In this paper, we propose a long-operation-first warp scheduling algorithm, LFWS, for GPU platforms. The LFWS filters warps in the ready state to a ready queue and updates the queue in time according to changes in the status of the warp. The LFWS divides the warps in the ready queue into long and short operation groups based on the type of operations in their instruction buffers, and it gives higher priority to the long-operating warp in the ready queue. This can effectively use the long operations to hide some of the latency from each other and enhance the system's ability to hide the latency. To verify the effectiveness of the LFWS, we implement the LFWS algorithm on a simulation platform GPGPU-Sim. Experiments are conducted over various CUDA applications to evaluate the performance of LFWS algorithm, compared with other five warp scheduling algorithms. The results show that the LFWS algorithm achieves an average performance improvement of 8.01% and 5.09%, respectively, over three traditional and two novel warp scheduling algorithms, effectively improving computational resource utilization on GPU.

***key words:*** *GPU, warp scheduling, long operation latency, utilization*

## 1. Introduction

GPU is composed of thousands of simple SPs (Streaming Processors) and a few control units. Earlier GPUs were mainly used to accelerate graphics processing until NVIDIA introduced the general purpose parallel computing architecture called CUDA (Compute Unified Device Architecture) which expanded the scope of GPU applications and made GPU dominate in areas such as high performance computing, artificial intelligence computing, and parallel computing. For most parallel applications, GPUs are able to provide better performance than CPUs [1]. Modern GPU architectures are based on the Single Instruction Multiple Thread (SIMT) computing model, where 32 threads form a warp that executes the same instructions and processes different data at the same time. Warp is the smallest unit in the GPU that can be scheduled and executed in parallel. There will be pipeline stalls when executing long-latency operations. The GPU does not have a complex chaotic pipeline and branch predictor like the CPU, and therefore relies on fast switching of warps to hide pipeline stalls. The ability to hide long-latency operations affects the utilization of computational resources on the GPU [2]. The key issues affecting computational resource utilization on the GPU are shared resource contention, long operation latency, and branching operations [3].

Hardware control of warp scheduling is required in GPU. A warp scheduler selects a warp from the list of candidate warps and executes its instructions every cycle [4]. The scheduler determines the execution order and switching policy of warps, which determines the ability of the GPU to hide long latency operations and also affects access memory locality, and has a critical impact on the utilization of computational resources on the GPU [5].

Effective utilization of intra- and inter-warp data locality can improve on-chip cache hit rate, mitigate cache interference [6], reduce the number of costly off-chip accesses, and improve GPU performance [7]. The traditional LRR (Loose Round Robin) and GTO (Greedy Then Oldest) scheduling algorithms preserve inter-warp locality and intra-warp locality, respectively. In the literature [8], the authors dynamically choose a LRR or GTO scheduling policy suitable for a task based on the locality of task load. Oh et al. [9] proposed the adaptive anticipation and scheduling policy ARPES (Adaptive Prefetching and Scheduling), which divides the warps executing the same memory operation instructions into a group and prioritizes the execution of the group. Rogers et al. [10] prioritized the warps based on the degree of data locality within the warp and proposed a cache-aware warp scheduling algorithm CCWS (Cache-Conscious Wavefront Scheduling) which tracks the invalidation of the L1 data cache, adjusts the number of active warp in time, and reduces cache contention to preserve access locality. All these approaches reduce the numbers of long operations as much as possible but do not directly address the pipeline stalling problem caused by long operation delays [11]. In literatures [12]–[14], a series of studies have been conducted on how to better hide the latency of long operations on GPUs. The literatures [15]–[19] have tried to dynamically choose the best warp scheduling strategy for different applications.

In this paper, we propose a Long-Operation First Warp Scheduling (LFWS) algorithm for general-purpose GPU architectures, which enables long operations to hide part of the latency from each other and then uses short operations to hide the latency. This algorithm filters the ready warps to the ready queue, and then prioritizes the execution of the

Manuscript received July 29, 2022.
Manuscript revised January 1, 2023.
Manuscript publicized February 10, 2023.
†The authors are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, 710049 China.
a) E-mail: wgwu@xjtu.edu.cn
DOI: 10.1587/transfun.2022EAP1084

long operation warps in the ready queue to improve GPU's ability to hide latency. The LFWS scheduler is simulated and implemented in GPGPU-Sim [20] to test the execution performance of various GPU applications. The LFWS algorithm is compared with polling scheduling algorithm, greedy scheduling algorithm, two-level scheduling algorithm and two warp scheduling algorithms for hiding long latency problem. The experimental results show that LFWS can improve the ability of hiding long latency, especially in long operation intensive applications, LFWS exhibits much more significant performance improvement than other algorithms.

## 2. Long Operation First Warp Scheduling Algorithm

The warp scheduler determines the execution order of warps and the strategy of warp replacement, which has a critical impact on GPU performance. This section designs and implements the long-operation-first warp scheduling algorithm to improve the GPU's ability to hide latency and improve application runtime performance by using long operations to hide a portion of latency from each other.

### 2.1 Typical Warp Scheduling Algorithms

The warp scheduling algorithms commonly used on GPUs are polling scheduling algorithms and greedy scheduling algorithms. The polling algorithm is divided into SRR (Strict Round Robin) and LRR, while the greedy scheduling algorithm is divided into GTO and GTRR (Greedy Then Round Robin).

SRR scheduling algorithm strictly follows the principle that all warps have the same priority. If the currently scheduled warp is blocked and hung, the pipeline will wait idle until the warp can be executed. In the LRR scheduling algorithm, all warps have the same priority, but if the current warp is hung, it switches to the next warp in time. In the GTO algorithm, a warp switching occurs only when the current warp is blocked, and it is to switch to the earliest arriving warp, i.e., the warp with the smallest unique identifier (ID). The difference between the GTRR and the GTO is that the GTRR selects the warp to switch to according to the rules of the polling method when a warp switch occurs.

Two-level warp scheduling algorithm (2-level) divides all warps in the warp pool into two groups, i.e., the active group and the pending group. The warps in the active group are in the ready state, while the warps in the pending group are in the blocking state. The active group is dynamically changing, and the scheduler will promptly call out the ineligible warps to the pending group, and when the number of warps in the active group is less than the set value, it will promptly call in the warps in the pending group to replenish them. The warps in the active group are scheduled according to the LRR algorithm. To a certain extent, the two-level scheduling algorithm embodies a better ability to hide the delay.

Besides, some scheduling algorithms are specifically designed for the long operation latency problem on GPUs, such as the LPI [13] and the Long-Latency Operation-Based Scheduling (LLOS) [14] algorithms.

The LPI scheduling algorithm inserts waiting warps into active warps to form the queue of warps traversed by the scheduler, and the insertion is done in a time-ordered manner as much as possible. The main procedure of the LPI algorithm is as follows. The warps are first divided into active and blocking warp groups. The active and blocking warp groups are arranged according to the IDs of the warps from small to large. Then, the ordered blocking warps are inserted into the active warps at intervals, and each blocking warp is inserted into two active warps adjacent to its ID as much as possible. Finally, the scheduler traverses the active warp groups after the above process. This scheduling algorithm uses interval execution between active and blocking warps to hide the blocking warp latency, and thus improving application runtime performance. However, the LPI algorithm does not schedule the long operations within warps, and a small number of active warps are difficult to fully hide the latency of the blocking warps, so the performance improvement of this algorithm is very limited.

LLOS scheduling algorithm uses the core idea of scheduling long operation warp first. The main procedure of LLOS algorithm is as follows. First, the warp that will perform the read operation and the scoreboard gives an invalid signal is put into the guiding queue, and the rest warps are put into the filling queue. Then, the scheduler first traverses the guiding queue, and if there is no warp in the guiding queue that successfully executes, it schedules the warp in the filling queue. Both the filling queue and the guiding queue are scheduled according to the polling method. Although the LLOS algorithm adopts the idea of scheduling long operation warp first, but it does not consider whether the warp is in the ready state, and only the warp that scoreboard gives an invalid signal can be determined as a long operation warp, therefore, the long operation warp cannot be executed in the current clock cycle, leading to insufficient utilization of GPU resources.

### 2.2 Design of LFWS

There are two main types of instruction operations in GPU, i.e., long-latency operations and short-latency operations. Long-latency operations mainly refer to the access operations to global memory, local memory or texture memory. Short latency operations are mainly some arithmetic logic operations, integer and floating point operations. Usually, short operations can be used to hide the latency of long operations and avoid pipeline stalls. However, when there are few short operations, it is difficult to hide the latency sufficiently. This section designs a long-operation-first warp scheduling algorithm LFWS to improve the GPU's ability to hide the latency of long operations.

The LFWS algorithm is mainly divided into two steps, firstly, filtering the ready warps into the ready queue and adjusting the priority of the warps in the ready queue, and secondly, iteratively scheduling the warps in the ready queue.

The filtering process selects the ready warps from the pool of warps and puts them into the ready queue, and then adjusts the priority of the ready queue according to the type of instruction operation to be performed by the threads in the ready warps and the warp ID. The core idea of this algorithm is to prioritize the warps in the ready warp queue, which are about to perform long operations, i.e., long-operation warps. In this way, long operations can be used to hide some of the latency from each other.

The advantage of the LFWS algorithm in hiding the latency is analyzed in the following example. Suppose there are 6 warps, numbered 1 to 6, where the order of instructions to be executed in warps 1, 3, 5, and 6 is $long \rightarrow long \rightarrow short \rightarrow long$, and the order of instructions to be executed in warps 2 and 4 is $short \rightarrow short \rightarrow long \rightarrow long$, where $long$ means long operation which is assumed to take 10 clock cycles to complete, and $short$ means short operation which is assumed to take 1 cycle to complete. In fact, the execution time of long and short operations on the GPU can differ by tens of times. The scheduler needs to schedule the 6 warps for execution, and assumes that only one instruction is fired per clock cycle per warp. The execution of the warps using different scheduling strategies is shown in Fig. 1. The horizontal axis represents the execution time in cycles, w1, w2, . . . , and w6 respectively represent the execution process of each warp, the L block represents the long operation, the S block represents the short operation, and the solid black line segment represents the delay of the long operation.

The cost of warp switching on the GPU is very low, so when a warp is blocked, the scheduler immediately switches to the next warp. Different scheduling algorithms take advantage of this feature to different degrees. By looking at Fig. 1(a), using the SRR scheduling strategy, it takes a total of 45 clock cycles to complete the execution of the 6 warps, generating 21 pipeline stops. The SRR scheduling algorithm is the least effective because it keeps idle waiting for the blocking warp. The LRR algorithm, on the other hand, is able to switch to other ready warps in a timely manner, basically solving the SRR problem. Figure 1(b) shows the execution with the LRR scheduling strategy, consuming a total of 39 clock cycles, of which the pipeline is idle for 15 cycles. In summary, the polling scheduling algorithms ensure the fairness of scheduling as much as possible, but tend to cause a larger number of warps to fall into a blocking state at similar clock cycles. Figure 1(c) shows the process of executing the 6 warps using the GTO scheduling strategy, which takes a total of 41 clock cycles and generates 17 pauses. The greedy scheduling algorithm spreads out the time when the warps fall into blocking compared to the SRR scheduling algorithm, but destroys the inter-warp locality. All these three algorithms, with a limited number of short operations, show unmaskable delays.

To solve all above mentioned problems, we introduce the LFWS algorithm. The LFWS algorithm reduces pipeline stalls by using long operations to hide some of the latency from each other. The core idea of the LFWS algorithm is to prioritize the scheduling of warps that are to perform long operations. As shown in Fig. 1(d), in the first clock cycle, warps 1, 3, 5, and 6 will perform long operations, so these 4 warps are executed sequentially to hide some of the latency from each other, and then the short operations of warps 2 and 4 are used to hide the latency. At each cycle, the priority of the warps is adjusted according to the operations to be performed by each warp. When there are no long operation warps available for execution, the short operation warps are executed. Besides, LFWS will filter out the ready warp according to the state of the warp, therefore, the operations in the ready warp can be executed continuously. In this way, the GPU's ability to hide the latency of long operations is improved, while reducing the application completion time and improving the system computational resource utilization.

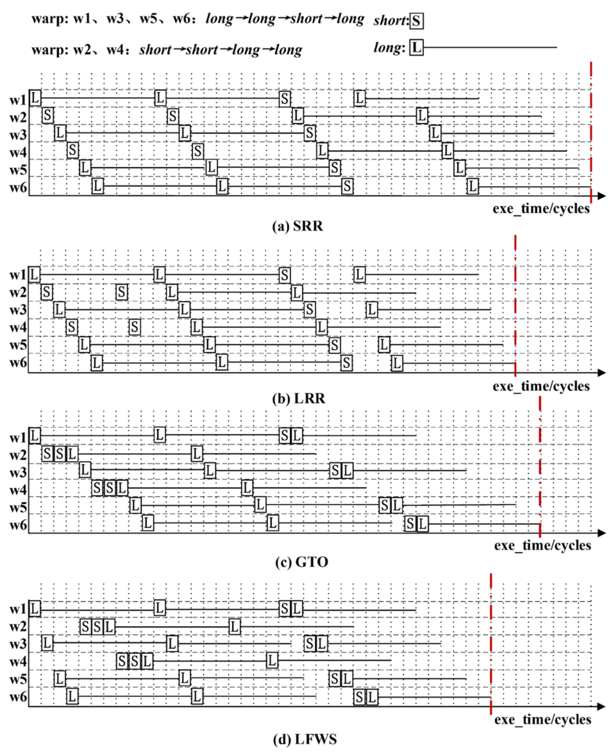
warp: w1、w3、w5、w6: long→long→short→long
short: S
warp: w2、w4: short→short→long→long
long: L
w1
w2
w3
w4
w5
w6
exe_time/cycles
(a) SRR
(b) LRR
(c) GTO
(d) LFWS

**Fig. 1** The ability of different warp scheduling algorithms to hide long operation latencies.

To execute all instructions of the 6 warps, the LFWS algorithm requires a total of 37 clock cycles and generates only 13 pipeline pauses. Compared to SRR, LRR and GTO, the completion time with LFWS is reduced by 8, 2 and 4 clock cycles, respectively, in this illustrated example.

## 2.3 Implementation of LFWS

This subsection specifies the implementation of the LFWS algorithm described in Algorithm 1. The LFWS algorithm first divides the warps into ready warps and pending warps based on the current state of warp and whether the instruction buffer is empty. Warps have two states, pending or ready. There are four situations when a warp is in a pending state, waiting to be initialized, waiting for another synchronized

**Algorithm 1: Determine whether a warp is in the pending state**

```
Input: $warp$
Output: **True or False**
if $waiting\ to\ be\ initialized$ then
    return True;
end if
if $waiting\ for\ other\ warps\ within\ a\ block$ then
    return True ;
end if
if $waiting\ for\ memory\ barrier$ then
    return True ;
end if
if $waiting\ for\ atomic\ operation$ then
    return True ;
end if
return False ;
```

**Algorithm 2: Determine the ready warp group**

```
Input: $pending\_warps$
Output: $ready\_warps$
for $warp$ in $pending\_warps$ do
    if $isPending(warp)\&\&ibuffer$ is not $empty$ then
        $ready\_warps.push\_back(warp)$;
        $pending\_warps.erase(warp)$;
    end if
end for
initialize $long\_warps$, $short\_warps$, $update\_warps$ to $null$;
for $warp$ in $ready\_warps$ do
    $inst \leftarrow inst.ibuffer\_next\_inst()$;
    $mem\_type \leftarrow inst.space\_type()$;
    if $inst$ is $load$ or $store$ && $mem\_type$ is $global$, $local$ or $tex$ $space$ then
        $long\_warps.push\_back(warp)$;
    else
        $short\_warps.push\_back(warp)$;
    end if
end for
if ! $long\_warps.empty()$
    $sort\_by\_greedy\_then\_warpID\_asc(long\_warps)$;
    $update\_warps.\_insert(long\_warps)$;
end if
if ! $short\_warps.empty()$
    $sort\_by\_greedy\_then\_warpID\_asc(short\_warps)$;
    $update\_warps.\_insert(short\_warps)$;
end if
$ready\_warps \leftarrow update\_warps$;
return $ready\_warps$;
```

warp, waiting for the current memory operation to complete, and waiting for an atomic operation in the warp to complete. If a warp is in a pending state or the instruction buffer of the warp is empty, the warp is classified into the pending group, otherwise it is classified into the ready group. Algorithm 1 describes the $isPending()$ function, determining whether a warp is in the pending state.

At each clock cycle, the warp adjusts its group according to its current state. After determining the ready group, the warps in the ready group are prioritized, with long operations taking precedence over short operations, and the warps containing the same operation type are then scheduled according to the GTO. In other words, if there is a warp in the long operation group that has been executed in the previous cycle, the warp is put in the first place, and the rest of the warps are arranged according to the warp ID from small to large. And the same is ture for the short operation group. The specific determination process of long and short operations is to take an instruction from the instruction buffer corresponding to a warp, and if the instruction meets the long operation determination condition, i.e., the operation is an access operation to the off-chip memory, then the warp is added to the long operation warp group, otherwise it is put into the short operation warp group. Algorithm 2 describes the $getReadyWarps(pending\ warps)$ function to determine the ready warp group. The time complexity of Algorithm 2 is $O(n \log_2 n)$, where $n$ is number of warps. Although the warp classification process is executed throughout the whole scheduling process, the time cost of this process is trival compared with the warp execution time, and the classification process will bring significant performance improvement to the scheduling algorithm.

After obtaining the queue of ready warps, the LFWS algorithm iterates through the warps in the ready queue at each clock cycle. If the instruction of the current warp meets the execution conditions, the instruction is sent to the corresponding pipeline unit for execution. And if it does not meet the execution conditions, it switches to the next warp. At the same time, if a program branch occurs, the instruction in the instruction cache of current warp is updated according to the branch stack to handle the branch.

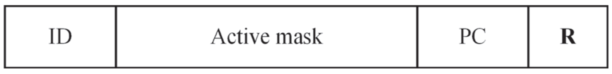

**Fig. 2** The structure of warp.

## 2.4 Microarchitecture of SM

The SM microarchitecture based on LFWS scheduler differs from the traditional SM architecture. The traditional warp scheduler directly iterates through all warps in turn from the pool of warps, detects the scoreboard of a warp, and executes the warp if it is given a valid signal, otherwise, it hangs the warp to select next warp. The scheduler designed in this paper requires an additional flag bit R for each warp to store whether the current warp is in the ready state, R=1 means ready and R=0 means waiting, as shown in Fig. 2. Active mask has 32 bits and each bit indicates whether a thread in that warp is active or not. The PC (Program Counter) is a program counter that stores the address of the next instruction.

Warps flow through the ready and pending queues based on flag bits. The warps in the ready queue are sorted according to the types of operations to be executed in the instruction cache, based on the principle of long operations first. If both the branch stack and the scoreboard give valid signals, the instructions in the warp are fired to the execution unit, otherwise, the next warp in the ready queue is switched. The SM microarchitecture based on the LFWS scheduler is shown in Fig. 3. The warp scheduler is the core optimization

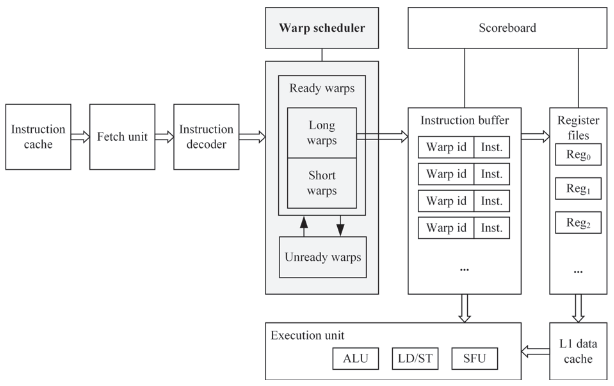

**Fig. 3** SM microarchitecture with LFWS scheduler.

component to implement the proposed scheduling algorithm in this paper, which selects a warp from the ready warp queue for execution every clock cycle. The queue of ready and non-ready warps dynamically changes according to the warp flag bits. The hollow arrows in Fig. 3 indicate the logical flow of instructions executed on the SM. First, the fetch unit fetches the instructions from the instruction cache and decodes them. Then, the warp scheduler selects a warp to emit from the ready queue. Finally, the instructions of the warp to be launched and the data in the register unit or data cache are fetched from the instruction cache and transferred to the corresponding functional unit for execution.

In a GPU with LFWS scheduler, each warp takes one more bit to store the current warp state. The total hardware overhead is the number of warps multiplied by 1 bit. If there are 2 SMs with 24 warps on each SM, the additional storage space required is 48 bits. The filtering and sorting process of warps has one more access to the instruction cache compared to the original scheduler. However, each warp is independent and the parallel execution is fast. The warp scheduling can be done directly based on the current ready queue, and the scheduling process does not conflict with the filtering process. Therefore, the overhead of this scheduler is fully acceptable. In addition, several comparators and logical AND gates are needed to complete the determination of long and short operations.

## 3. Experiments

### 3.1 Experimental Environment and Data Set

The LFWS warp scheduler was implemented on the simulation platform GPGPU-Sim 3.2.2, and several CUDA applications were tested to evaluate the execution performance of the proposed LFWS algorithm. The experiments were conducted using the default architecture of the simulator, namely the NVIDIA Fermi GTX480 architecture. Table 1 shows the specific configuration information.

In this paper, 20 CUDA applications from Rodinia [21], ISPASS [22], and NVIDIA SDK [23] are selected for experimental evaluation. The names of the selected applications, the input data sets, the total number of CUDA instructions at runtime, and the percentage of long operation instructions are given in Table 2. For each set of validation experiments, the performance statistics output from each application after execution on the simulator are recorded.

**Table 1** GPU configurations on GPGPU-Sim.

| Configuration parameter | Value |
|---|---|
| Number of SM | 15 |
| Number of threads in SM | 1536 |
| Maximum number of concurrent thread blocks per SM | 8 |
| Number of registers per SM | 32768 |
| Size of warp | 32 |
| Size of shared memory | 48 KB |
| Size of L1 data cache | 16 KB |
| Size of L1 inst cache | 2 KB |
| Size of L2 cache | 768 KB |
| DRAM | FR-FCFS, 6 channels,48B/cycle |

### 3.2 Experimental Results

We compared with LRR, GTO, 2-level, 2-level opt, LPI, and LLOS to evaluate the performance of LFWS. LRR, GTO, and 2-level are the classical scheduling algorithms. The “2-level opt” is the optimized 2-level algorithm that we applied the core idea of LFWS to the 2-level algorithm, and the “2-level opt” sorts the warps in the active warp group and prioritizes the long operation warp for scheduling. LPI and LLOS are the state-of-the-art algorithms for the long operation latency problem, which share the same goal with LFWS. All these algortihms are seperately implemented in GPGPU-Sim.

We use the instructions per cycle, IPC, to evaluate the performance of scheduling algorithms. All applications from Table 2 are executed with different algorithms. The number of instructions executed by the application and the total cycles of the application completion time were recorded to calculate the IPC of applications with each algorithm. The larger the IPC, the higher the application runtime computing resource utilization, and the better the corresponding scheduling algorithm performance. There are four kernels in application NN, and the thread block size of two kernels is 1. This will lead to a low utilization of GPU computing resources and affect the performance of the warp scheduling algorithm. Therefore, only the IPC values measured after the execution of the first two kernels are recorded for application NN.

Since the LRR is one of the most classical scheduling algorithms, we use the LRR as the baseline algorithm. The IPC of tested applications with each algorithm is normalized to that of the LRR to better show the performance of all comparison scheduling algorithms. Figure 4 shows the normalized IPC values of seven scheduling algorithms. The IPC value of LRR algorithm for each application is 1.0. To show the results more clearly, the normalized IPC value of the ordinate in Fig. 4 starts from 0.80.

Compared to the traditional scheduling algorithms, LRR, GTO, and 2-level, LFWS algorithm improves the average execution performance. As shown in Fig. 4, the LFWS

**Table 2** CUDA applications.

| Name of Applications | Abbreviation | Input data | Number of instructions | Number of long operation instructions | Percentage of long operation instructions in all instructions(%) |
|---|---|---|---|---|---|
| AES Cryptography | AES | 128 bit, 256 KB | 22 M | 131584 | 0.59 |
| Breadth First Search | BFS_1 | 65536 nodes | 4 M | 3722864 | 9.09 |
| Breadth First Search | BFS_2 | 1 M nodes | 724 M | 60791985 | 8.38 |
| Back Propgation | BP | 65536 nodes | 181 M | 15471724 | 8.50 |
| Coulombic Potential | CP | 200 atoms | 125 M | 131072 | 0.10 |
| EstimatePInlineP | EPP | 100000,float | 806 M | 160251896 | 19.88 |
| hotspot | HSP | 512*512 | 142 M | 1186944 | 0.83 |
| LIBOR Monte Carlo | LIB | 4096 | 900 M | 94453160 | 10.49 |
| 3D Laplace Solver | LPS | 100*100*100 | 82 M | 2568800 | 3.10 |
| MUMmerGPU | MUM | NC_003997.20k. | 87 M | 1731984 | 1.97 |
| Neural Network Digit Recognition | NN | 28 | 101M | 19108544 | 18.75 |
| nearestNeighbor | NNEI | 10691 | 1 M | 128292 | 10.31 |
| N-Queens Solver | NQU | 10 | 1 M | 319 | 0.02 |
| Needleman-Wunsch | NW | 1024*1024 | 169 M | 8929280 | 5.27 |
| PathFinder | PF | 100000*100*20 | 649 M | 11850200 | 1.82 |
| Ray Tracing | RAY | 32*32 | 744 K | 19935 | 2.67 |
| SingleAsianOptionP | SAOP | 100000, float | 1 G | 177851576 | 10.72 |
| StreamCluster | SC | 2048 | 6 G | 1324130304 | 19.76 |
| ScalarProduct | SP | 256*4096 | 22 M | 2097408 | 9.49 |
| StoreGPU | STO | 192 KB | 128 M | 835584 | 0.64 |

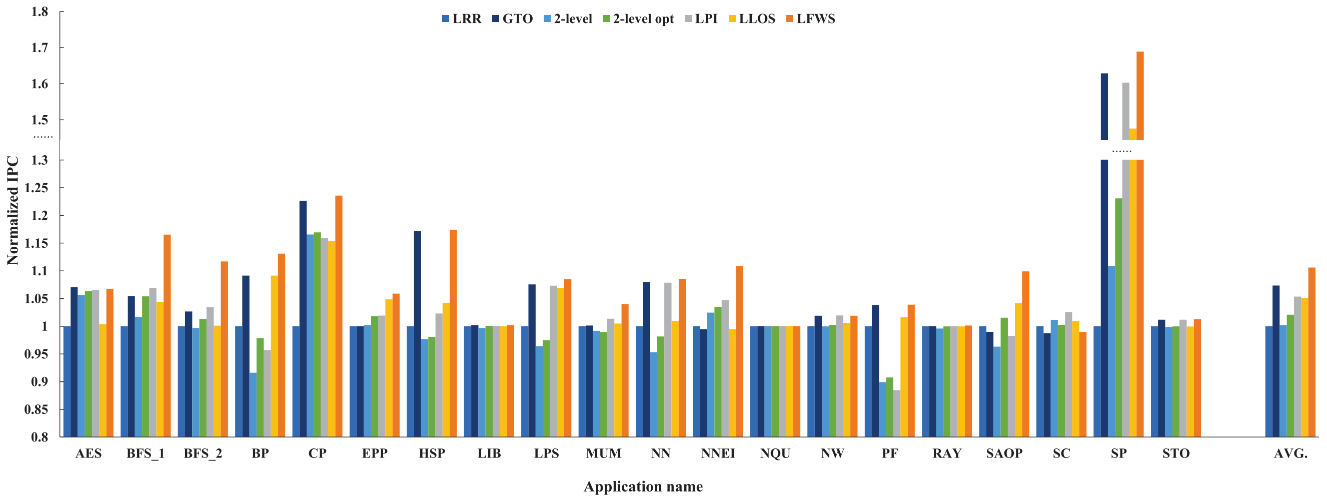

**Fig. 4** The IPC of LRR, GTO, 2-level, 2-level opt, LPI, LLOS, and LFWS, normalized to that of LRR.

achieves an averge IPC improvement of 10.60% over the baseline algorithm for all applications. For the applications BFS_1, BFS_2, BP, EPP, NN, NNEI, SAOP and SP, which have a relatively high proportion of long operations, the LFWS algorithm achieves an average IPC improvement of 18.17% compared to the LRR runtime. But for the application LIB and SC, which also have more long operations, the execution performance has almost no improvement using the LFWS algorithm. This is because these two applications have a small number of warps and different warp scheduling algorithms do not have a significant impact on the performance of these applications. In addition, the application HSP has only 0.84% long operations, but it achieves a performance improvement of 17.40% using the LFWS algorithm compared to LRR. This is because the long and short operation warp groups of the ready queue are scheduled according to GTO when with very few long operation warps. In this case, LFWS scheduling algorithm is equivalent to GTO, so this application shows almost no difference in performance under LFWS and GTO. Since the 2-level algorithm is designed to reduce the number of warps traversed by the scheduler per clock cycle and it performs warp scheduling based on LRR, the performance of applications using the 2-level algorithm has no improvement compared to that using LRR, and some applications have performance degradation.

When we applied the LFWS to the 2-level algorithm, the 2-level opt algorithm shows an average performance improvement of 1.88% over the orginal 2-level algorithm for all applications, as shown in Fig. 4. For the applications with a relatively high percentage of long operations, BFS_1, BFS_2, BP, EPP, NN, NNEI, SAOP, and SP, the 2-level opt achieves an average performance improvement of 4.27%. For the applications LIB and SC, the 2-level opt does not obtain performance improvement over the 2-level due to the small number of warps. Based on the results and analysis, it can be concluded that the scheduling rule of long operation priority for active warps can improve the runtime IPC performance of applications and increase the utilization of computational resources on GPUs. As mentioned before, the original 2-level algorithm is designed to save hardware resources, its performance has almost no improvement compared with the LRR algorithm. Therefore, the performance improvement of the 2-level opt is moderate.

LPI and LLOS are state-of-the-art warp scheduling algorithms for solving the long operation latency problem on GPUs. According to Fig. 4, the average performance improvements of LPI and LLOS over LRR are 5.34% and 5.05%, respectively. And the LFWS algorithm achieves an average performance improvement of 5.11% and 5.06% compared to LPI and LLOS, respectively, for all test applications. As described in section 2, the LPI algorithm uses interval execution between active and blocking warps to hide the blocking warp latency. However, a small number of active warps are not enough to hide the latency of blocking warps, so the performance improvement of the algorithm is limited, especially for the warps with more long operands, such as BP, EPP, SAOP, etc. For the LLOS algorithm, it does not determine whether the warp is in the ready state and cannot execute the determined long operation warp in current clock cycle, as explained in section 2. But the LFWS algorithm can schedule the long operation warp in time to improve the resource utilization on GPUs. For applications with more long operations, such as BFS_1, BFS_2, BP, EPP, NN, NNEI, SAOP, and SP, the LFWS algorithm achieves an average performance improvement of 8.72% compared to LLOS.

## 4. Conclusion

In this paper, we propose a long-operation-first warp scheduling algorithm LFWS for general-purpose GPU architectures. LFWS uses long operations to effectively hide part of the latency to each other, and thus improving the GPU's ability to hide latency. The algorithm filters the ready warps to a ready queue, then adjusts the order of warps in the ready queue. Warps in the ready queue are divided into long and short operation warp groups, and the long operation warp groups are prioritized for execution while warps within the long and short operation groups are schedules according to the GTO strategy. The long operation priority warp scheduler is simulated and implemented in GPGPU-Sim. Experiments show that LFWS achieves average performance improvements of over 8% and 5%, compared with three classic warp scheduling algorithms and two state-of-the-art algorithms, respectively. Results demonstrate that LFWS effectively hides latency to enhance the resource utilization of GPUs, especially for long-operation intensive applications.

## Acknowledgments

This paper was supported by National Natural Science Foundation of China (Grant No. 62002279) and Key Basic Research Projects of the Foundation Plan of China (Grant No. 2020-JCJQ-ZD-087).

## References

[1] N. Melab, J. Gmys, M. Mezmaz, and D. Tuyttens, "Many-core branch-and-bound for GPU accelerators and MIC coprocessors," High-Performance Simulation-Based Optimization, pp.275–291, 2020.
[2] C. Yu, Y. Bai, and R. Wang, "MIPSGPU: Minimizing pipeline stalls for GPUs with non-blocking execution," IEEE Trans. Comput., vol.70, no.11, pp.1804–1816, 2020.
[3] C. Fan, "Research on GPU warp scheduling algorithm optimization," Master's thesis, Nanjing University, 2018.
[4] J. Chen, X. Tao, Z. Yang, J.K. Peir, X. Li, and S.L. Lu, "Guided region-based GPU scheduling: Utilizing multi-thread parallelism to hide memory latency," 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp.441–451, May 2013.
[5] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu, and Y.N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp.308–317, 2011.
[6] J. Zhang, S. Gao, N.S. Kim, and M. Jung, "CIAO: Cache interference-aware throughput-oriented architecture and scheduling for GPUs," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp.149–159, 2018.
[7] T.G. Rogers, "Locality and scheduling in the massively multithreaded era," Ph.D. thesis, University of British Columbia, 2015.
[8] G.B. Kim, J.M. Kim, and C.H. Kim, "Dynamic selective warp scheduling for GPUs using L1 data cache locality information," International Conference on Parallel and Distributed Computing: Applications and Technologies, pp.230–239, 2018.
[9] Y. Oh, K. Kim, M.K. Yoon, J.H. Park, Y. Park, M. Annavaram, and W.W. Ro, "Adaptive cooperation of prefetching and warp scheduling on GPUs," IEEE Trans. Comput., vol.68, no.4, pp.609–616, 2019.
[10] T.G. Rogers, M. O'Connor, and T.M. Aamodt, "Cache-conscious wavefront scheduling," 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp.72–83, 2012.
[11] A. Jog, O. Kayiran, N.N. Chidambaram, A.K. Mishra, M.T. Kandemir, O. Mutlu, R. Iyer, and C.R. Das, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," ACM SIGPLAN Notices, vol.48, no.4, pp.395–406, 2013.
[12] M. Gebhart, G.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," 2011 38th Annual International Symposium on Computer Architecture (ISCA), pp.235–246, 2011.
[13] Y. Zhang, Z. Xing, C. Liu, C. Tang, and Q. Wang, "Locality based warp scheduling in GPGPUs," Future Generation Computer Systems, vol.82, pp.520–527, 2018.
[14] C.T. Do, H.J. Choi, S.W. Chung, and C.H. Kim, "A novel warp scheduling scheme considering long-latency operations for high-performance GPUs," The Journal of Supercomputing, vol.76, no.4, pp.3043–3062, 2020.
[15] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp.370–381, 2016.
[16] J.P. Anantpur, "Enhancing GPGPU performance through warp scheduling, divergence taming and runtime parallelizing transformations," Ph.D. thesis, Indian Institute of Science Bangalore, 2017.
[17] S.Y. Lee, A. Arunkumar, and C.J. Wu, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," ACM SIGARCH Computer Architecture News, vol.43, no.3S, pp.515–527, 2015.

[18] V.T. Vo and C.H. Kim, "KAWS: Coordinate kernel-aware warp scheduling and warp sharing mechanism for advanced GPUs," Journal of Information Processing Systems, vol.17, no.6, pp.1157–1169, 2021.
[19] J. Fang, Z. Wei, and H. Yang, "Locality-based cache management and warp scheduling for reducing cache contention in GPU," Micromachines, vol.12, no.10, p.1262, 2021.
[20] M. Khairy, Z. Shen, T.M. Aamodt, and T.G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp.473–486, 2020.
[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," IEEE International Symposium on Workload Characterization, pp.44–54, 2009.
[22] A. Bakhoda, G.L. Yuan, W.W. Fung, H. Wong, and T.M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp.163–174, 2009.
[23] NVIDA, CUDA SDK: http://developer.nvidia.com/gpu-computing-sdk

**Song Liu** received the B.S. degree in computer science and technology from the Northwestern Polytechnical University, China, in 2009. And he received Ph.D. degree in computer science and technology from the Xi'an Jiaotong University, China, in 2018. He is currently with the School of Computer Science and Technology at Xi'an Jiaotong University as an assistant professor. He is a member of the CCF. His research interests include parallel computing and code optimization.

**Jie Ma** received the M.E. degree in computer science and technology from the Xi'an Jiaotong University, China, in 2022.

**Chenyu Zhao** is currently pursuing the M.E. degree in computer science and technology in Xi'an Jiaotong University.

**Xinhe Wan** is currently pursuing the M.E. degree in computer science and technology in Xi'an Jiaotong University.

**Weiguo Wu** received the B.S., M.S. and Ph.D. degrees in computer science from the Xi'an Jiaotong University, China, in 1986, 1993 and 2006. He is currently with the School of Electronic Information and Engineering at Xi'an Jiaotong University as a professor. He is a senior member of the CCF. His research interests include high performance computer architecture, storage system, cloud computing, and embedded system.