Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding

Keunsoo Kim* Sangpil Lee* Myung Kuk Yoon*
Gunjae Koo† Won Woo Ro* Murali Annavaram†

*School of Electrical and Electronic Engineering, Yonsei University
{keunsoo.kim,madfish,myungkuk.yoon,wro}@yonsei.ac.kr

†Ming Hsieh Department of Electrical Engineering, University of Southern California
{gunjae.koo, annavara}@usc.edu

ABSTRACT

This paper presents a pre-execution approach for improving GPU performance, called P-mode (pre-execution mode). GPUs utilize a number of concurrent threads for hiding processing delay of operations. However, certain long-latency operations such as off-chip memory accesses often take hundreds of cycles and hence leads to stalls even in the presence of thread concurrency and fast thread switching capability. It is unclear if adding more threads can improve latency tolerance due to increased memory contention. Further, adding more threads increases on-chip storage demands. Instead we propose that when a warp is stalled on a long-latency operation it enters P-mode. In P-mode, a warp continues to fetch and decode successive instructions to identify any independent instruction that is not on the long latency dependence chain. These independent instructions are then pre-executed. To tackle write-after-write and write-after-read hazards, during P-mode output values are written to renamed physical registers. We exploit the register file underutilization to re-purpose a few unused registers to store the P-mode results. When a warp is switched from P-mode to normal execution mode it reuses pre-executed results by reading the renamed registers. Any global load operation in P-mode is transformed into a pre-load which fetches data into the L1 cache to reduce future memory access penalties. Our evaluation results show 23% performance improvement for memory intensive applications, without negatively impacting other application categories.

1. INTRODUCTION

GPUs use single instruction multiple thread (SIMT) execution model to execute a group of threads concurrently. The grouping of threads is called a warp or a wavefront. Threads within a warp execute the same instruction but on different input operands using multiple execution lanes. Each execution lane uses in-order processing pipelines. Thus, a shared front-end but with multiple in-order execution lanes enables warp execution to be power efficient. To hide instruction processing delay GPUs rely on warp interleaving. Typically tens of warps are interleaved, which is sufficient for hiding typical ALU latency of tens of cycles [1]. However, in the presence of long latency memory access operations interleaving of tens of warps alone is insufficient to hide the latency. Recent studies measured average access latency to off-chip DRAM to be 400 to 600 cycles [1, 2]. The first instruction that depends on the long latency instruction causes the warp to stall. When multiple warps hit this stall bottleneck the number of warps that are ready for issue steadily decreases. As the number of ready warps decreases the ability to hide even a typical ALU latency is compromised. Eventually the number of ready warps may dwindle to zero. Our empirical evaluations (more details in Section 3) showed that across 28 different GPU applications 23% of the execution cycles are wasted due to a warp's dependence on a long latency memory operation. The problem is even more acute when we consider memory intensive applications only. Of the 14 memory intensive applications, 40% of the cycles are wasted.

Prior studies have proposed specialized warp scheduling policies to tackle a variety of concerns, including memory access latency. Some techniques showed energy efficiency improvements [3, 4], while others improved performance through improved latency hiding abilities [5, 6, 7, 8]. However, even after optimized warp scheduling policies were applied, these prior schemes showed that memory access latency is not hidden entirely. Increasing the number of in-flight warps may seem like one possible solution. However, increasing the number of concurrent warps requires increasing many of the microarchitectural structure sizes proportionally. For instance, more warps require more architected registers, and more scratchpad memory. In some cases more warps also increases memory resource contention.

In this paper, we explore a different approach to improve latency hiding abilities of GPUs. We design and evaluate warp pre-execution which continues to issue independent instructions that may appear after the first stalled instruction. Out-of-order CPUs have the ability to issue independent instructions. CPUs rely on a host of additional features such as reservation stations, reorder buffers, and CAMing logic to achieve this goal. Replicating these structures for multiple concurrent warps in a GPU is power inefficient. Instead we propose *warp pre-execution mode* (P-mode for short), a concept that is inspired by non-blocking pipelines [9, 10, 11,

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

12]. When a warp is blocked due to a long-latency operation, the warp is switched to P-mode. In P-mode, all the instructions that are independent of the long latency operation are pre-executed, while those instructions that are on the dependent chain of the long latency operation are skipped.

The key challenge with the P-mode operation is to maintain sequential program semantics as independent but younger instructions may have executed earlier than older dependent instructions. When two instructions are executed the instruction that appears first in the program order is referred to as older instruction and the instruction that occurs later in the program order is referred to as younger instruction. Sequential semantics require solving register and memory hazards. By definition P-mode does not have any read-after-write (RAW) dependencies, as instructions with RAW dependencies are skipped. However, write-after-read (WAR), and write-after-write (WAW) hazards can occur. If a younger independent instruction overwrites a register that may be read by an older instruction, then WAR hazard may occur. Similarly, if a younger independent instruction overwrites a register that may be overwritten by an older instruction, then WAW hazard may occur. P-mode does not generate any memory hazards. If an independent global load instruction is encountered in P-mode, it is transformed into a pre-load operation which prefetches data into the L1 cache without modifying the register file. Stores are not allowed to execute in P-mode.

To solve register WAW and WAR hazards, we will rely on a register renaming scheme. Recent studies on GPU register files showed that they are underutilized [13, 3, 14, 15]. They exploit the underutilization to reduce leakage and dynamic power. In this work, we exploit the underutilized register file to re-purpose a few unused registers to store P-mode results. During the P-mode execution a warp writes values into a separate P-mode register set through renaming its destination register. During P-mode if the input operand register of an instruction is not defined by any prior P-mode instructions then that register is read from the architected register file. If the input operand register was defined by a prior P-mode instructions, the renamed register is consumed.

When the long latency instruction completes, the warp switches from P-mode to normal mode of execution (N-mode). In N-mode, the warp restarts fetching instructions from the first blocked instruction. Any instruction that is dependent on the long latency operation would be executed normally. However, any independent instruction that may have been already executed in P-mode reused whenever possible. If the stored pre-executed values are available in the renamed registers the instruction completes immediately without re-executing and simply renames the P-mode register to be the new architected register.

This approach has multiple benefits. First, we enable independent instructions to be pre-executed under the shadow of long latency operation. Second, by repurposing unused registers to save the P-mode register state we do not need additional registers to support pre-execution. In the worst case, if there are no available registers for renaming P-mode is halted and the system falls back to the default baseline execution mode. We improve the performance of all 28 benchmarks by 13% on average. If we apply the technique to memory intensive applications only the performance improvements are 23%.

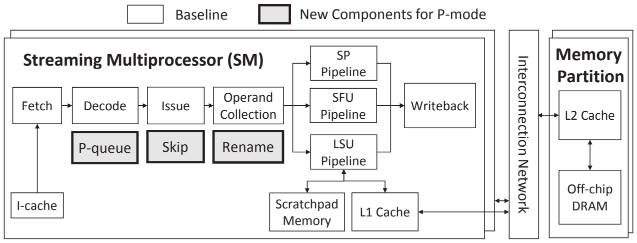


Figure 1: The proposed GPU pipeline

We summarize the contributions of this paper as follows.

- We present the motivation for warp pre-execution on GPUs to improve their ability to hide long latency operations. We then describe the architectural support necessary to support the P-mode execution.

- We present a thorough design space exploration of P-mode. P-mode improves the performance of latency sensitive applications by 23% by reducing long-latency stalls from 40% to 24% of total cycles.

The rest of this paper is organized as follows. Section 2 presents the baseline GPU architecture used in this paper. In Section 3, we present motivational data in support of warp pre-execution. Section 4 discusses the microarchitecture of P-mode execution with detailed design considerations. Section 6 presents the evaluation methodology and results. Section 7 discusses related work, and we conclude in Section 8.

2. BASELINE GPU MODEL

A GPU has multiple streaming multiprocessors (SM). Figure 1 shows the high-level organization of one SM. The additional architectural blocks used in the P-mode execution are shaded in the figure. We will describe the purpose of each of these additional blocks shortly. Each SM supports 48 concurrent warps that are divided into two groups of 24 warps each. Each cycle at most two instructions are fetched, one from each warp group. If a conditional branch instruction is found, the fetch logic will not fetch from the warp until the direction of the branch is known. Decoded instructions are stored in a per-warp instruction buffer which holds the next instruction to be issued in program order.

A warp scheduler is associated with each warp group. Every cycle the scheduler selects one warp instruction from its own group to be issued to the execution pipeline. The scheduler tracks dependencies between instructions using a per-warp scoreboard, which is implemented as a 63-bit vector. Note that each warp can use at most 63 architected registers in our baseline. Each bit in the vector corresponds to one architectural register number. When the scheduler selects an instruction for issue, the bit corresponding to the destination register number of that instruction is set to indicate that register is waiting for new data, and it is cleared after the destination register value is written back to the register file. Before issuing an instruction, the scoreboard bits associated

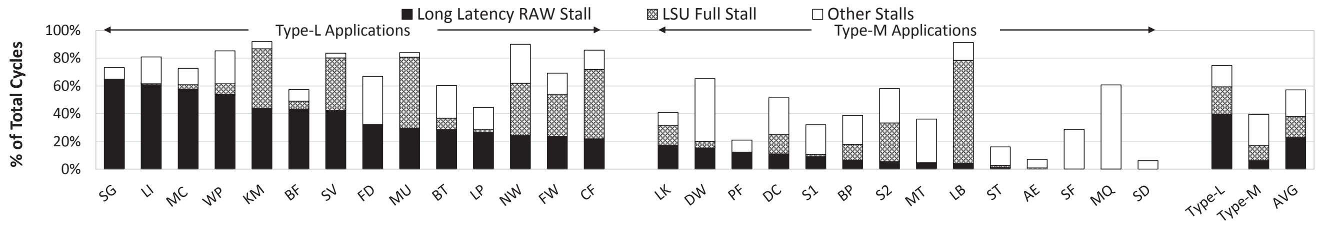


Figure 2: Breakdown of warp issue stall cycles

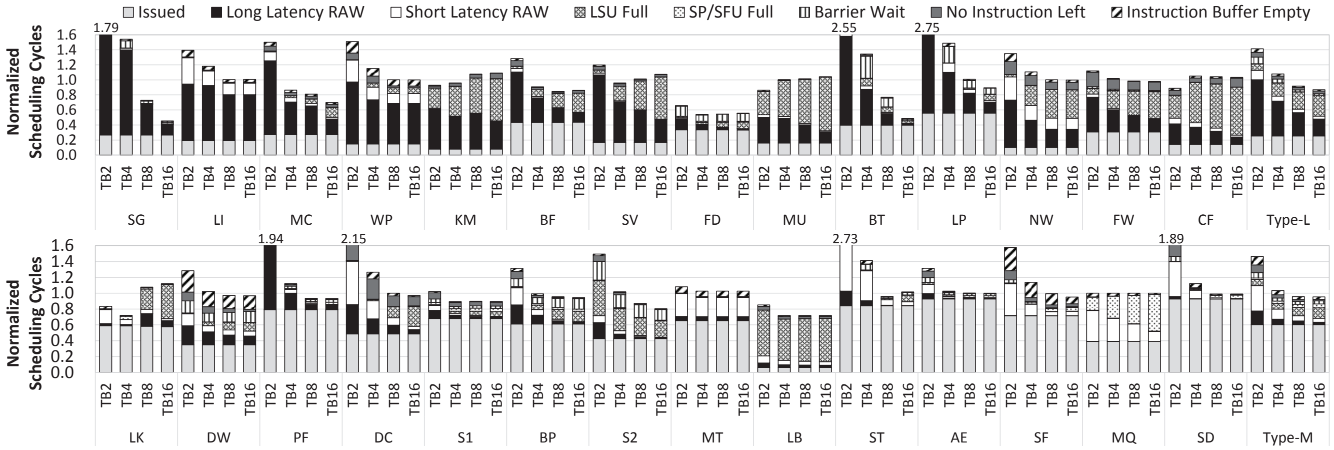


Figure 3: Breakdown of cycles consumed by warp schedulers (normalized to baseline GPU)

with source register numbers of the instruction are examined. If any bit is set, the instruction is not issued pending on source operands.

After an instruction is issued, the source register values are read from a multi-banked register file. The input operands are buffered in a collector unit (CU). When all the source operands are read into the collector unit the corresponding instruction is dispatched to functional units (FUs).

3. MOTIVATION FOR PRE-EXECUTION

In this section, we first classify the reasons for warp stalls. One reason for a warp stall is if the next instruction is waiting for the result of preceding instructions. We call this a RAW stall. The dependence could be due to a long latency operation (greater than 100 cycles) or due to a short latency. Although some ALU operations may take hundreds of cycles, such instructions are exceedingly rare in our benchmarks. Hence, global memory access instructions are primarily responsible for long latency operations. RAW stalls due to a dependence on long latency memory instruction is categorized as *Long-latency RAW*, otherwise it is categorized as *Short-latency RAW*. Second, a warp could be stalled if the corresponding functional unit is unavailable. For instance, when memory subsystem is saturated due to unavailable cache tags the LSU pipeline will not accept any more load instructions [16]. We classify such a stall as *LSU Full*. Similarly if the integer, floating point ALUs (collectively called SPs) or special functional units (SFUs) are busy to accept a new instruction then we classify it as *SP/SFU Full* stall. When the instruction buffer is empty, a warp is in the *Instruction Buffer Empty* state. A warp is in *Barrier Wait* state when the warp reaches a barrier and is waiting for other warps. Finally when a warp completes execution the warp cannot be released until the execution all warps in the thread block are completed. In this case, the warp is in the *No Instruction Left* state.

We first categorize warps using the two most common stalls encountered in our benchmarks as shown in Figure 2. Detailed description of benchmarks and simulation tools are described in Section 6.1. Long-latency RAW stalls as a fraction of the total execution cycles are plotted at the bottom of the stacked bar in Figure 2. The second major reason for a warp stall is the LSU full stall and these stalls as a fraction of the total execution cycles are plotted as the second component of the stacked bar. Finally, all other stalls are shown in the top component of the stacked bar. We use this data to categorize an application as latency sensitive (Type-L) if majority of stalls are due to RAW stalls on a long latency operation. Otherwise, we classify that application as Type-M (mixed) application. In Type-L applications 40% of execution cycles are wasted due to RAW stalls.

As stated in the prior section, GPU use thread level parallelism (TLP) to hide latency. To see the effectiveness of TLP in reducing the stall time we show the detailed stall breakdown with increasing number of thread blocks (also called cooperative thread arrays) per each SM in Figure 3. The labels for various stall categories are described earlier in this section. We use the notation *TBx* where *x* indicates the number of thread blocks assigned to an SM. The number of thread blocks that can be assigned to an SM varies depend-

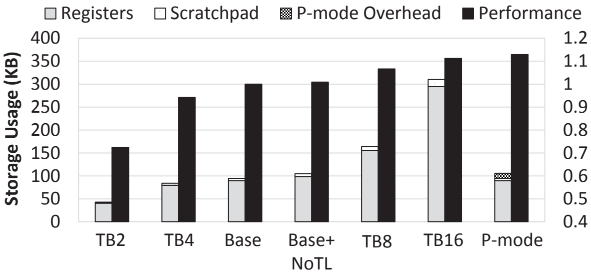
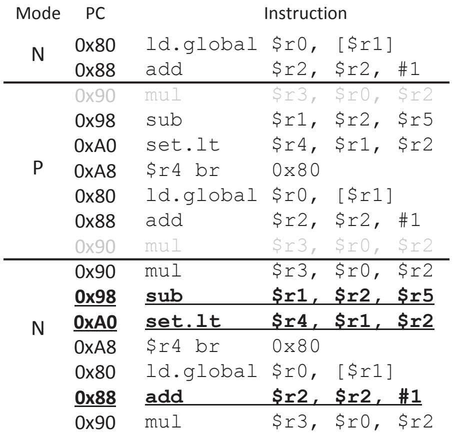


Figure 4: Effect of increased TLP

ing on the amount of scratchpad memory and register file demand per each thread block. But for the purpose of generating this motivational data we assume as much scratchpad memory and register file space as needed are available to execute the given number of thread blocks. Figure 3 shows detailed breakdown of warp scheduling cycles for each configuration. The height of the stacked bars indicates total scheduling cycles normalized to the baseline GPU which accommodates as many thread blocks as possible that fit within the baseline GPU limitations.

It is evident that for Type-L applications (the top chart) increasing thread blocks from TB2 to TB16 improves performance. The improvement is primarily due to reduced long-latency RAW stalls. However, even with 16 thread blocks several applications still suffer from long latency RAW stalls. Although more TLP reduces long-latency RAW stalls, in some applications such as KM, SV, MU, CF, and LK performance is even degraded due to increased memory system contention, as indicated by the large increase in the LSU stalls. This result has also been pointed out in recent studies [6, 17, 18], and the optimal performance can be achieved by reducing the number of active thread blocks in these benchmarks [17, 18]. Type-M applications are less sensitive to increasing TLP but the amount of stalls suffered by these applications is not as large as Type-L applications to begin with.

While increasing thread block count may reduce long-latency stall for some applications, this approach requires additional on-chip storage space for register file and scratchpad memory. The primary Y-axis on Figure 4 shows the total storage demand (register file and scratchpad memory size) averaged across all benchmarks, and the secondary Y-axis shows the performance improvement as we increase the thread block count. The X-axis labeled *Base* shows the storage needs and performance of our baseline architecture which allows up to 8 thread blocks per SM, with the additional constraint that the total number of threads assigned do not exceed 1536 (maximum of 48 warps), cumulative register demand of all the threads does not exceed 128 KB register file, and cumulative scratchpad demand does not exceed 48 KB. In some cases the additional constraints prevent the benchmarks from issuing the maximum 8 thread blocks per SM. The X-axis labeled *Base+NoTL* shows the performance and storage demand when using 128 KB register file and 48 KB scratchpad memory to the fullest extent possible assuming no limitations on thread count. In some cases more than eight thread blocks may be assigned in this case. The X-axis labeled *TB8* shows the performance and storage demands if 8 thread blocks are assigned per SM, ignoring the storage constraints imposed in the Base design. Clearly increasing thread block limit to 16 comes at the expense of very a large increase in the on-chip storage to over 300 KB, which is nearly double the baseline storage size. The X-axis labeled *P-mode* shows the performance and storage demands of our proposed approach. Warp pre-execution has negligible increase on the storage demands over baseline and yet provides even better performance than TB16. The P-mode implementation related storage overheads are also shown in the figure and these overheads will be discussed in the next section.



```
Mode  PC     Instruction
      0x80   ld.global $r0, [$r1]
 N    0x88   add       $r2, $r2, #1
------------------------------------
      0x90   mul       $r3, $r0, $r2
      0x98   sub       $r1, $r2, $r5
      0xA0   set.lt    $r4, $r1, $r2
 P    0xA8   $r4 br    0x80
      0x80   ld.global $r0, [$r1]
      0x88   add       $r2, $r2, #1
      0x90   mul       $r3, $r0, $r2
------------------------------------
      0x90   mul       $r3, $r0, $r2
      0x98   sub       $r1, $r2, $r5
      0xA0   set.lt    $r4, $r1, $r2
 N    0xA8   $r4 br    0x80
      0x80   ld.global $r0, [$r1]
      0x88   add       $r2, $r2, #1
      0x90   mul       $r3, $r0, $r2
```

Figure 5: Illustration of single-warp execution with P-mode

4. WARP PRE-EXECUTION

Pre-execution overview: To reduce RAW long latency stalls we rely on warp pre-execution. When a warp experiences a long latency stall, predominantly due to a cache miss, it is switched from normal execution mode to pre-execution mode (P-mode). In P-mode, all instructions dependent on the current blocking instruction are skipped and only instructions independent from the blocking instructions are issued. Hence, independent instructions are *pre-executed*. Once the long latency operation completes the warp switches back to normal mode of execution (N-mode). We propose to exploit pre-execution for two different purposes. First, during N-mode execution we propose to reuse computation results that were already performed during P-mode. The second benefit is that P-mode increases memory level parallelism by issuing future load instructions that are independent of the current stalls. In our approach all global load instructions in P-mode are converted into a pre-load instruction which only brings data into the L1 cache. Unlike prefetching, pre-load requests are in fact accurate and the loaded data is likely to be used soon.

Pre-execution illustration: We illustrate the concept of pre-execution in Figure 5 focusing only on a single warp execution progress. The instruction at PC 0x80 is a long latency instruction. Current GPUs can continue to issue any independent instruction that follows the long latency instruction.

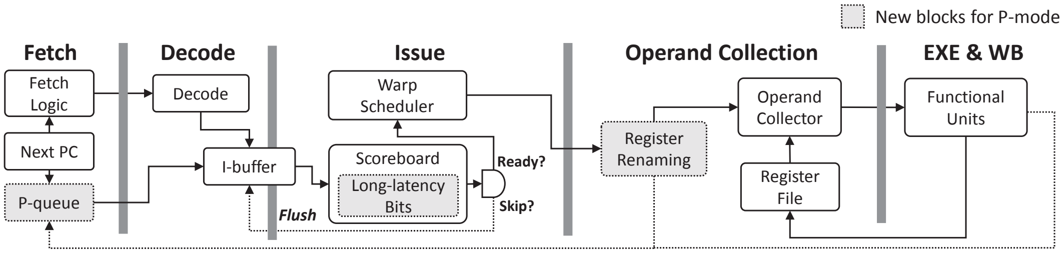


Figure 6: Microarchitecture of GPU with P-mode

The instruction at 0x88 which is independent of the long latency operation can be issued next. However, the instruction at 0x90 is the first instruction that is dependent on the long latency operation. Normally the warp is stalled at this time. However, in our proposed approach the warp enters into P-mode and starts to fetch and decode instructions to identify any independent instruction. Note that the warp schedulers give priority to N-mode warps before fetching from P-mode warps. The next five instructions in this example are guaranteed to be independent, and are then pre-executed. The second instance of global load at 0x80 is converted into a pre-load operation and is marked as a long latency operation. The second instance of *mul* instruction at 0x90 is again skipped as it depends on the second instance of *ld* instruction which is again a long latency operation.

Eventually the first instance of the long latency *ld* operation completes and the warp is switched back to the normal mode of operation (shown as N-mode in the figure). Then, the dependent *mul* instruction is re-fetched and re-decoded and is eventually issued. The following two instructions were already pre-executed and hence the results are simply read from the renamed register file, instead of re-executing those instructions in N-mode. Given that today's GPUs have deep execution pipelines (typically 8-10 cycle ALU pipeline) reusing the results allows the instructions that are dependent on short latency instructions to issue much faster during the N-mode operation. Furthermore, the second instance of the global load instruction that has already been issued during P-mode will hit the L1 cache or will be merged to the corresponding MSHR entry, both of which reduces its latency.

5. MICROARCHITECTURAL OPERATION WITH PRE-EXECUTION

There have been many recent studies that showed that register files in GPUs are underutilized [13, 3, 14, 15]. We analyzed a wide range of applications across different benchmark suites and only handful of benchmarks were able to use all the available registers. Figure 7 shows that across the 28 benchmarks in our evaluations on average only 78% of the register file is occupied, in other words 224 out of 1024 warp registers are unused. In P-mode implementation we use up to 128 of these unused registers (out of 1024 registers) to store pre-execution results.

In our baseline GPU there are a total of 1024 physical registers per each SM. Registers are banked into four banks of each 256 registers. Each register is logically 1024-bits wide (32×32-bits) so as to store all the thread registers within a warp into a single warp-wide register. The 1024-bit warp register is actually split into 8 banks and each bank provides data to four threads within a warp. Thus the register file is implemented using 32 register banks. Current GPU compilers allocate registers in sequential manner while concurrently avoiding bank conflicts. Hence, at most the last 128 registers (register numbers 895-1023) are considered available for pre-execution.

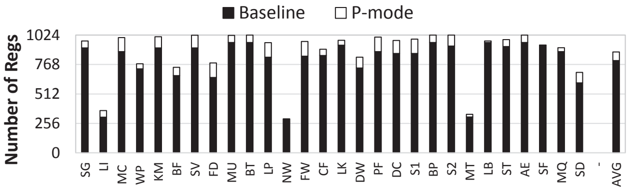


Figure 7: Warp register utilization of baseline GPU and P-mode. A warp register is a collection of 32 32-bit registers.

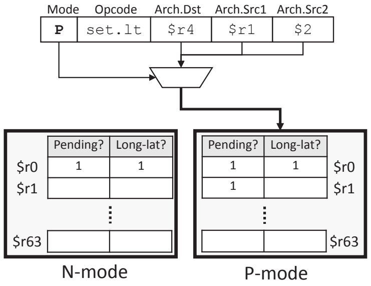


Figure 8: Organization of dual-mode augmented scoreboard

Initially all warps begin in N-mode. Mode switch is triggered by the scoreboard logic of an N-mode instruction. Note that the per-warp scoreboard tracks the register data availability of all the architected registers in that warp. At most 63 architected registers may be used in each warp in our baseline. As illustrated in Figure 8, the scoreboard is augmented with one additional bit per each register. Thus the scoreboard maintains two bits per each architected register (126 bits in total). One bit per architected register is needed for tracking if the register value is available or pending in the pipeline. The another bit, called long-latency bit, tracks if the register data is pending on a long latency operation. As mentioned earlier, in our benchmarks the predominant reason for long latency stalls are global load operations. Hence, when a global load operation is issued it sets the corresponding destination register bit to pending and also sets the long latency bit. When the scoreboard detects that an incoming instruction is waiting for the result of a preceding long-latency instruction, the warp is switched to P-mode. We use two scoreboards for independently tracking register pending state in N-mode and P-mode.

When a warp is switched to P-mode then a per-warp mode bit is set. The mode bit vector is 48-bits wide and stores the current execution mode of each of the 48 warps in the SM. Once the mode bit is set, the N-mode scoreboard is copied to a P-mode scoreboard. Similar to the scoreboard case we also need a new P-mode SIMT stack. On mode switch the current N-mode SIMT stack is copied to the P-mode SIMT stack. SIMT stack keeps track of branch re-convergence information and hence it is necessary to track a program's control flow [19, 20]. Even though the size of the stack is implementation dependent and not available in public, our applications require no more than 8 stack entries. Each stack entry is 96 bits wide and consists of two 32-bit entries for current and reconvergence PC values and one 32-bit active thread mask. Therefore, replicated SIMT stack requires 768 bits (96 bytes). We modeled the energy and latency costs for copying the scoreboard and SIMT stack in all our simulation results. The energy cost of copying the SIMT stack and scoreboard is less than half of a single register write operation as reported in [21]. Once a warp enters P-mode, the modified pipelined execution of the warp is described below.

5.1 P-mode Execution

Fetch/Decode stages: Instruction fetch and decode process in P-mode is the same as in N-mode. Decoded instructions are stored to instruction buffer even in P-mode. If later the decoded instruction is determined to be skipped, then the instruction buffer entry is simply invalidated and the next instruction is fetched to the buffer.

Issue stage: After decoding the instruction, the P-mode scoreboard is looked up with architected source and destination register numbers. In P-mode, when the scoreboard detects that any source register entry has a long-latency bit set, the instruction is skipped. At the same time the long-latency bit of the destination register is also set, so that subsequent instructions in the long-latency dependence chain are skipped. Some instructions such as branch instructions do not have a destination register. But they implicitly set the target PC and are handled as follows. Branch instructions are issued normally in P-mode if the source registers are not dependent on a long latency operation. On the other hand, if a branch instruction is dependent on the long latency operation, then P-mode execution of the warp is halted, because P-mode is unable to resolve the correct branch target.

Some instruction categories are skipped irrespective of whether the instruction is independent of a prior long latency operation.

- **Stores and Loads**: In P-mode all store instructions (global and scratchpad stores) are skipped to prevent memory hazards. Scratchpad loads are pre-executed and their data is brought into registers as long as there are no prior skipped stores. The reason for this unique restriction is that GPU's memory model has the following semantics. A scratchpad store instruction from one warp is only made visible to other warps in the thread block after a barrier instruction completes. Therefore, scratchpad loads can bring data into registers as long as there are no prior stores in that warp that were skipped. A prior store in the same warp could potentially cause a memory hazard and hence a skipped scratchpad store causes all future scratchpad loads to be skipped as well. Until the scratchpad store is encountered there is no reason to stop pre-executing the scratchpad loads to bring data into register file.

  Any global load instruction in P-mode is changed into a pre-load instruction that brings the data only into the L1 cache. Global loads are not pre-executed. Due to in-order execution needs, a warp must wait for the completion of the pre-executed load as soon as the warp reaches the first instruction that is dependent to the load. If any pre-executed load misses the L1 cache, then the load may complete later than the load which triggered P-mode. This means it is likely that the warp switches back to N-mode even before the pre-executed load is available. Hence, pre-loading is a better option.

- **Barriers**: Barrier instructions are always skipped in P-mode. But instructions past a barrier can be pre-executed. Any scratchpad load instruction past the barrier is skipped even if there are no skipped scratchpad store instructions. The reason for this more stringent requirement on scratchpad loads past the barrier is again due to the GPU memory model that allows stores from other warps in the thread block to be made visible to the current warp after a barrier.

Since warps in P-mode are essentially pre-executing the instructions they always get lower priority in warp scheduling. Hence, as long as there are N-mode instructions they are selected first by the warp scheduler. Only when there are no N-mode warps the scheduler selects P-mode warps. When multiple P-mode warps exists the scheduler uses round-robin scheduling.

Operand collection stage: Once an instruction is determined not to be skipped in P-mode, then it uses register renaming to rename its destination register after being issued. Since register renaming is restricted to at most 128 registers, a 128-bit physical register availability vector is added to track which of the unused registers are available for pre-execution. If the compiler already allocated some of these 128 registers to the application, then the register availability vector for those compiler used registers is set to prevent them from being used for P-mode. Note that P-mode warps are selected for issue only when renaming registers are available, to prevent a potential deadlock if no available renaming register is left for storing the pre-executed result.

The physical register availability vector is looked up to find the first unused register to store the destination result. To minimize potential bank conflict, we first search if an unused register available in the same bank as the original destination register. In that way, we can preserve the compiler generated bank allocation. A P-mode register renaming

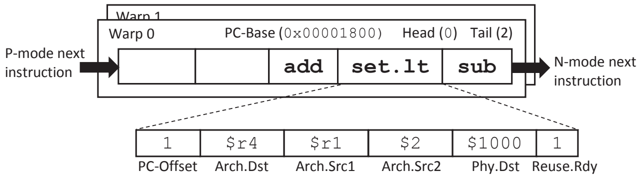


Figure 9: Organization of P-queue

table is used to point the architected register to the newly allocated physical register. The renaming table has 63 entries and each entry corresponds to one of the at most 63 architected registers that may be used in a warp. Each entry stores the 7-bit physical register ID since at most 128 registers are used for renaming, and 1-bit renamed bit for indicating the register is previously renamed. Source register is renamed to the reassigned physical register only when the associated renamed bit is set. Otherwise the value is fetched from the original register location.

Once an instruction is renamed in P-mode, then the instruction enters into the P-queue, which is a queue structure that holds the pre-execution state. The purpose of P-queue is twofold. First, P-queue stores the mapping between the architected register and physical register of the current warp instruction. P-mode renaming table itself is not sufficient since the renaming table may overwrite the architected to physical register mapping when a future warp instruction defines the same architected register. Each warp can use at at most 63 architected registers. Hence 6-bit destination architected register name is associated with a 7-bit renamed register number. The second benefit of P-queue is to reduce the need to re-fetch and re-decode instructions when the warp eventually switches back to N-mode. Hence the post-decode instruction is cached in P-queue, which is analogous to trace cache [22].

Figure 9 illustrates the organization of P-queue. Each P-queue stores a 10-bit PC offset which is the relative distance of the pre-executed instruction from the last stalled long latency operation which triggered the P-mode switch. As a result of this 10-bit limitation pre-execution is limited to executing instructions that are at most 512 bytes offset from the stalled long latency operation. This restriction is not a serious concern since GPU applications rarely jump to a target that is very far from the current PC. If an instruction does jump farther, then it is simply skipped. The second field stores three 6-bit architected register fields, and a 7-bit *Phy.Dst* field for physical register, and a 1-bit *Reuse.Rdy* field for indicating whether the result is ready for reuse and a 1-bit valid field. Hence each P-queue entry requires 37 bits. The P-queue also has two pointers that point to the head and tail entries. Entries are added always into the tail pointer and it is incremented on each entry insertion. We use an 8-entry P-queue as the default size. Hence, a 3-bit head and tail pointers are used to access the P-queue. Thus the total size of the P-queue is about 2 KB.

Note that while we advocate the use of P-queue, it is possible to enable pre-execution without P-queue. In the absence of P-queue pre-execution only improves memory level parallelism as it enables future load instructions to pre-load data into cache. Hence, without P-queue all P-mode instructions should be fetched/decoded/executed again later in N-mode. If P-queue is full (once the tail pointer reaches the P-queue size limit) no more instructions can be reused and P-mode is only useful for pre-loading cache lines to L1 cache. In Section 6.5 we present a detailed design space evaluation of the impact of P-queue size.

Writeback stage: When a P-mode instruction is completed in the functional unit, the output value is written to the allocated physical destination register. If the instruction is reusable, the renamed register is updated to the associated P-queue entry and the *Reuse.Rdy* bit is set to indicate the register value is now eligible for reuse. Note that branch instructions do not have a destination register. Hence, branch instructions that are selected for pre-execution would update the P-mode SIMT stack rather than the N-mode SIMT stack.

5.2 Reusing Pre-executed Instructions

Fetch/Decode stages: Once the long latency operation that initially triggered the P-mode switch completes, the warp switches to N-mode. This mode switch is triggered when the long latency instruction writes to its destination architected register. Once N-mode switch is triggered the warp points to the N-mode SIMT stack and N-mode scoreboard. Recall that both these structures were left unperturbed from the prior P-mode execution.

The first instruction that is dependent on the long latency operation is then released for execution. Then the warp starts the next instruction fetch in program order. Note that instructions are fetched in exactly the same order as in the P-mode when the warp switches back to N-mode. But in N-mode the warp has to check if each fetched instruction is pre-executed or skipped. To determine this outcome the warp in N-mode reads the head entry in the P-queue and gets the PC of the first pre-executed instruction. If the PC of the P-queue head matches with the PC of the instruction fetched next, then that instruction is already pre-executed. Since P-queue entries are inserted in the fetched order there is no need for N-mode warp to search all the P-queue entries. Instead it only has to check the head of the P-queue. If the fetched PC did not match the P-queue head entry then the instruction is deemed to have been skipped and it is executed normally. Note that when the P-queue head reaches the end of P-queue there is no further possibility of reuse and all instructions from that point in N-mode are treated as skipped instructions and are executed normally as well.

As illustrated in Figure 6, decoded instructions of reused instructions are directly moved from P-queue head to instruction buffer after bypassing fetch and decode stages. Instruction buffer carries *Reuse.Rdy* bit to mark the instruction is ready for reuse and *Phy.Dst* field that records the location of the reused result, as well as source and destination register IDs. The matched P-queue head entry is deleted at this time and the head pointer is moved to the next entry.

Issue and operand collection stages: Even though reused instructions do not re-execute, they must still update the scoreboard and should pass scoreboard checking to prevent WAW and WAR hazards before they are deemed completed. To reuse results, the N-mode warp renames its own destination register to point to the renamed physical register that is currently holding the pre-executed result. N-mode only needs

Table 1: GPU microarchitectural parameters

Parameter	Value
SM Clock Frequency	700 MHz
SMs / GPU	15
Warp Schedulers / SM	2
Warp Scheduling Policy	Greedy-Then-Oldest
SIMT lane width	32
Max # Warps / SM	48
Max # Threads / SM	1,536
Max. Registers / SM	32,768
Register File Size	128 KB, 32K 32-bit Registers
Scratchpad Memory Size	48 KB
L1 Cache	32 KB 64 Sets 4-way, 96 MSHR
NoC	Fully connected crossbar 32 B/direction/cycle @1.4GHz
L2 Cache	6 partitions, 128 KB 8-way, @1.4 GHz 200 cycles latency [16]
DRAM	32 entry scheduling queue, @924MHz 440 cycles latency [16]
P-queue entries	8 (varied)
Renaming Energy	2.94 pJ
P-mode Switch Energy	100 pJ
P-queue Access Energy	4.15 pJ

to maintain the mapping information between its destination and physical registers. We use a 63-entry N-mode renaming table to store this mapping information. The structure of N-mode rename table is identical to the P-mode rename table. Each entry is indexed by the architected destination register and the entry itself stores the corresponding physical register ID. Thus on a PC match the N-mode stores the *Phy.Dst* register number in the renaming table entry indexed by the architected register. Source registers of subsequent N-mode instructions will be also renamed for fetching values from the previously reused physical registers.

Note that it is possible to eliminate the use of renaming table in N-mode if the renamed physical register data is explicitly moved back to the original architected register space. But that process requires a register-to-register move operation. Since warp registers are quite wide the move operation is energy intensive. Instead we chose to use the renaming process to avoid register move operation.

5.3 Overhead

As described above, our proposed approach has the following storage overheads. First, it replicates the 8-entry SIMT stack with 96 bits per entry, requiring 768 bits per warp. It replicates the scoreboard which has 126 bits per warp. Each P-queue entry is 37 bits. We assume 8-entry P-queue requiring 334 bits per warp including 3-bit head/tail pointers and a base pointer. The renaming table is 63 entries and each entry stores a 7-bit physical register ID and a 1-bit renamed bit. Recall that renaming is limited only to the last 128 registers (7-bit register ID). We need two renaming tables for N-mode and P-mode. Hence, a total of 1008 bits per warp are needed for storing the renaming tables. Thus a total of 2236 bits per warp are needed. Register availability vector requires 128 bits for the entire SM. Assuming 48 warps per SM, the total overhead is 13.1 KB/SM. We estimate the size of an SM is $22mm^2$ based on the die photo of GF100 and according to the CACTI modeling the additional SRAM cells takes less than $0.5mm^2$, which is 2.3% of an SM.

Table 2: List of benchmark applications

Type-L			Type-M		
Name	Abbr.	Suite	Name	Abbr.	Suite
sgemm	SG	[23]	leukocyte	LK	[24]
LIBOR	LI	[25]	dwt2d	DW	[24]
MonteCarlo	MC	[26]	pathfinder	PF	[24]
WeatherPred	WP	[25]	dct8x8	DC	[26]
kmeans	KM	[24]	SRAD (v1)	S1	[24]
bfs	BF	[24]	backprop	BP	[24]
spmv	SV	[23]	SRAD (v2)	S2	[24]
FDTD3d	FD	[26]	MersenneTwist	MT	[26]
MummerGPU	MU	[25]	lbm	LB	[23]
b+tree	BT	[24]	stencil	ST	[23]
Laplace3D	LP	[25]	AES	AE	[25]
Needleman-Wunsch	NW	[24]	SobelFilter	SF	[26]
FastWalshTF	FW	[26]	mri-q	MQ	[23]
CFD Solver	CF	[24]	sad	SD	[23]

6. EVALUATION

6.1 Methodology

We used a simulator derived from GPGPU-Sim v3.2.2 [25]. The simulation parameters are listed in Table 1. Energy consumption is modeled using CACTI [27] and GPUWattch [28]. As shown in Table 2, we used 28 applications from Parboil [23], Rodinia [24], NVIDIA CUDA SDK [26], and ISPASS [25]. We found that some applications do not create sufficient number of threads with small input set. However, with large input set the simulation time takes excessively long. In this reason, we used the largest set available for each benchmark and executed up to 1 billion instructions [7]. All benchmarks are compiled with -O2 flag using NVCC 4.0. The PTXPlus feature of GPGPU-Sim is used to evaluate realistically optimized machine code.

For comparison purpose we also used a direct prefetching scheme inspired by the state-of-the-art GPU prefetching techniques [29, 30]. This prefetcher computes memory access stride per-warp and per-PC basis, which is similar to the many-thread aware prefetching (MTA) [29]. Each warp has its own memory access history table indexed by PC. When a load instruction is processed in the load/store unit, new access stride is calculated and updated to the table using the previously recorded access address generated from the instruction. For minimizing ineffective prefetch requests, our prefetcher also adopts a threshold-based throttling mechanism [30]. Each stride table entry has a confidence saturating counter, which is incremented when the new stride matches the previously computed stride value. Otherwise, the counter is decremented, and only when the confidence exceeds a predetermined threshold a new prefetching request is generated by adding the stride value to the current demand request address. We empirically determined the threshold value of 2, meaning at least two consecutive stride predictions should be correct. Prefetching requests are stored in the prefetch queue connected to the L1 cache, and they are injected to the memory system by accessing the cache when there is no demand requests to be processed in the load store unit.

6.2 ILP Opportunities with Pre-execution

Figure 10 shows the average number of instructions skipped or issued in P-mode. We found that on average 8.4 more in-

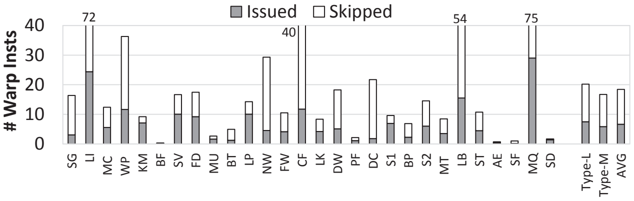


Figure 10: Average number of instructions processed per warp for one switch to P-mode

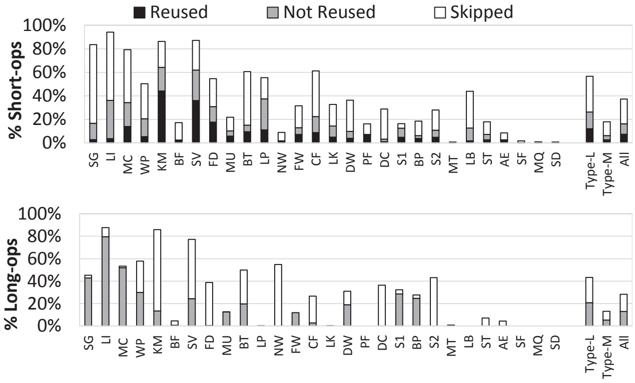


Figure 11: Percent of N-mode instructions covered by P-mode

structions per warp are pre-executed and 17.3 instructions are skipped when a warp is switched to P-mode.

We also measured the fraction of P-mode instructions that were reused in N-mode. Figure 11 shows the breakdown of N-mode dynamic instructions for short-latency (Top) and long-latency operations (Bottom). An instruction is *Reused* if the instruction is issued in P-mode and reused in N-mode. *Not Reused* instructions are at least executed once in P-mode but were not reused because the instruction is non-reusable type; note that all pre-load instructions are non-reusable, as those are used only for L1 cache warm up. Dependent instructions skipped in P-mode are marked as *Skipped*. We observe that in Type-L applications 26% of the total short-latency instructions are executed in P-mode (top chart in the figure), and 21% of long latency global memory loads are pre-loaded in P-mode (bottom chart in the figure). Type-M applications show lower coverage of 7% (short-ops) and 5% (long-ops), because long-latency RAW stall happens less frequently and warps are rarely switched to P-mode. Although BF suffers from long-latency RAW stall for 40% of time as shown in Figure 2, only 3% of short-latency instructions are covered. This small coverage is because BF has a branch which is dependent to the long-latency memory load, and this prohibits P-mode operation to continue past the branch.

6.3 Performance

We evaluate the performance (total execution time) of pre-execution and compare it with various alternatives. As shown in Figure 12 different approaches are compared, and all numbers are normalized to the performance of the baseline GPU (*Base*). *DirectPrefetch* indicates the prefetcher presented in Section 6.1. We also modeled the performance with the optimal number of thread blocks are assigned, which is marked as *OptTB* [6, 31, 17]. We determined the optimal thread block count for each benchmark based on the results presented in Figure 3.

Compared to the baseline, *P-mode* shows 23% speedup for Type-L applications. *P-mode+OptTB* achieves 36% and it surpasses the performance of all the prior scheme compared in the figure. P-mode does not affect Type-M applications that do not suffer as many stalls. As previously mentioned in these applications P-mode is rarely triggered.

Figure 13 shows the breakdown of scheduling cycles using the methodology presented in Section 3. Across Type-L applications long-latency RAW stall is reduced from 40% to 24% of total cycles. In SG, P-mode reduces 64% of long-latency RAW stalls to 6%. The combined impact of prefetching and reuse of P-mode was very effective in reducing the stall time. On the other hand, in LI only P-mode reuse capability effectively reduces RAW stalls but the prefetching benefits of P-mode are never triggered, and in KM, SV, and FD, performance degradation is observed when applying only prefetch but even in this case P-mode improves performance. In these workloads we observe *DirectPrefetch* increases the fraction of *LSU Full*, indicating that additional prefetching traffic increased memory system contention but pre-execution enabled us to overcome the negative impact of prefetching.

6.4 Impact on Memory System

We measured the impact on memory system with P-mode and the direct prefetching approach. Figure 14 shows the fraction of L1 accesses in N-mode either hits in the cache or merged to outstanding memory requests. In Type-L applications L1 cache hit rate is improved from 24% to 33%. In SG, although direct prefetching improves L1 cache hit rate 22% more than P-mode, the performance with P-mode is better. We observed *DirectPrefetch* consumes a significant portion of memory traffics for prefetching, as a result the round-trip latency of demand requests is increased, which offsets the effect of increased L1 hits.

Figure 15 summarizes the changes in on-chip traffic due to both a direct prefetching approach and with P-mode. This metric measures the total amount of data moved between L1 and L2 cache. Across Type-L applications prefetching increases data traffic by 10%, while P-mode reduces average traffics by 2%. Using prefetch alone in SV, DW, ST increases data traffic by 27%, 42%, and 34%, respectively. Such a large increase in prefetch traffic results in increased *LSU Full* by 15%, 10%, and 10%, which degrades performance. Hence, P-mode reduces data traffic demand while simultaneously improving performance.

Note that P-mode does not change the memory traffic because the pre-load cache lines are guaranteed to be accessed soon and in most of the cases the lines are likely to remain in the L1 cache. But in practice some of the lines that can be evicted and brought back into cache during P-mode execution, which potentially increase cache contention and memory traffics. SG, LI, MC show noticable traffic increase due to some contention. However, the negative impact of cache contention was overall negligible and P-mode improved performance at least 40% on these benchmarks.

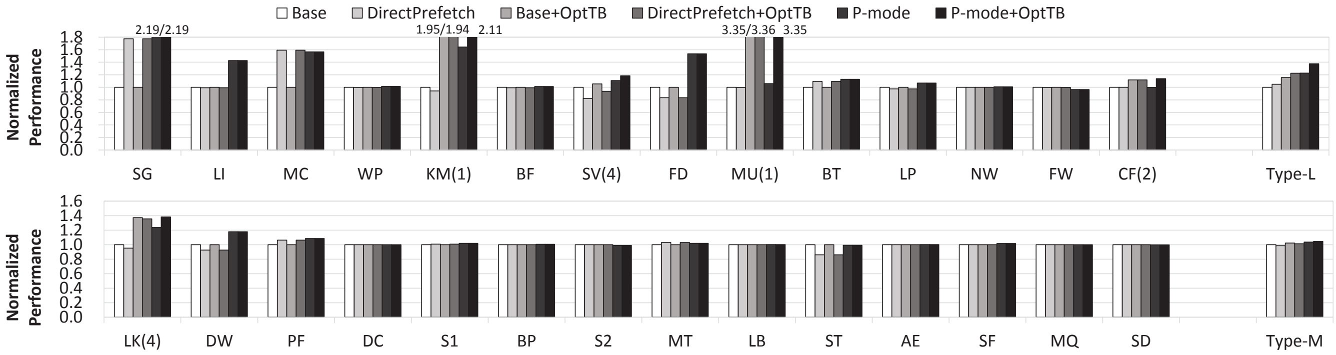


Figure 12: Performance impact of pre-execution and various techniques. Parentheses indicate optimal thread block count used for *OptTB*

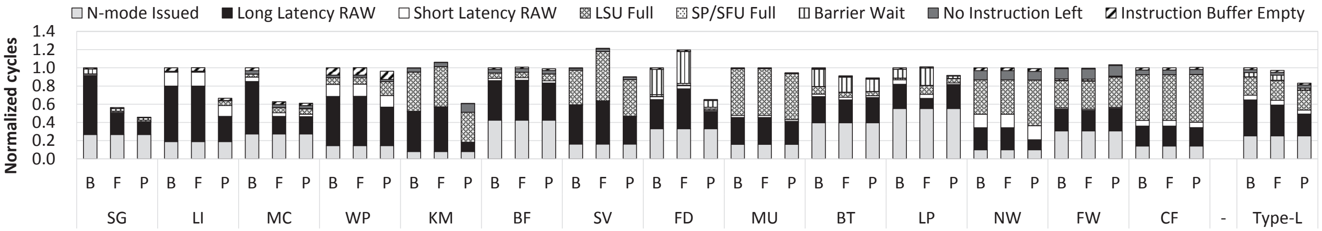


Figure 13: Scheduling cycle breakdown (B:Baseline, F:DirectPrefetch, P:P-mode)

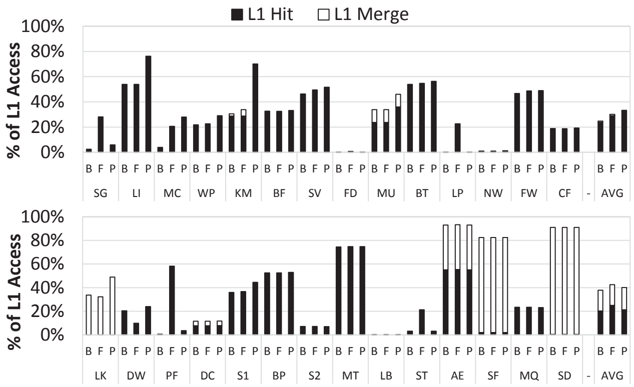


Figure 14: N-mode L1 cache hit rate (B:Baseline, F:DirectPrefetch, P:P-mode)

6.5 Energy Consumption

P-mode reduces leakage energy by reducing execution time. However, P-mode also increase fetch and decode demands and hence may increase the dynamic energy. Figure 16 shows the leakage and dynamic energy changes compared to baseline. We split the dynamic energy into two categories. *Dynamic* indicates the total dynamic energy including additional energy consumption for executing non-reused instructions multiple times. *Overhead* indicates increased energy consumption for supporting P-mode execution, including register renaming, P-queue access, and the overhead to switch between N-mode and P-mode. The net energy increase is 2.7%. Additional structures for P-mode increase energy consumption by less than 1%. Without P-queue, which is denoted as *NoPQ*, energy consumption is increased up to 5.1%, hence reusing improves performance and simultaneously minimizes energy increase with pre-execution. We also explored the energy impact with additional P-queue entries. Compared to the 8-entry case, additional energy saving with 16-entry P-queue is 0.3%.

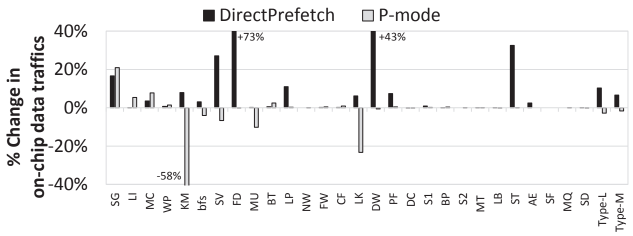


Figure 15: Impact on data traffics between L1 and L2

6.6 Impact of Microarchitectural Parameters

More concurrent threads: As discussed in Section 3, increasing thread parallelism provides higher latency tolerance. We compared the performance and storage demand of our technique with a hypothetical GPU having twice the in-flight threads in the scheduling pool, which is denoted as *2xTLP* on Figure 17. From the baseline GPU we doubled the size of register file, scratchpad memory, thread count limit, and thread block limit. Across the 28 benchmarks *2xTLP* requires 205 KB of storage, which is about twice of 105 KB on the baseline GPU. Due to increased latency hiding, *2xTLP* achieves 9% speedup across all benchmarks, while P-mode improves 13%. Additional storage for architectural support and increased register utilization with P-mode is about 13 KB. Hence, our technique provides higher performance with only about 12% of additional storage space.

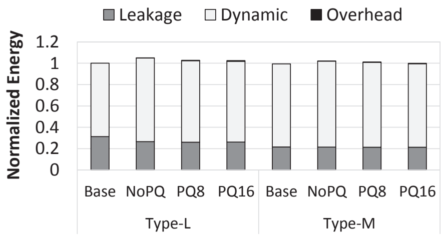


Figure 16: Energy consumption

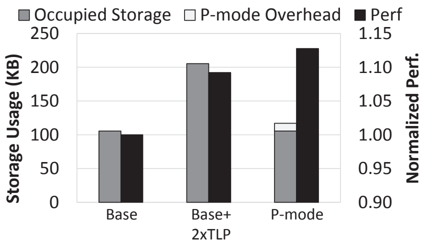


Figure 17: Storage usage

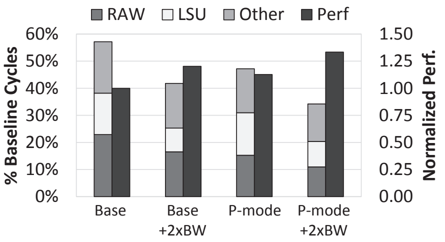


Figure 18: Impact of memory bandwidth

Higher memory bandwidth: We also explored the impact of doubling memory bandwidth. As shown the primary benefit with additional TLP is reduced long-latency RAW stalls incurred by long memory latency. We doubled operating frequency of on-chip crossbar, L2 cache, and off-chip memory to model memory system with 2x higher bandwidth. Figure 7 shows the performance and fraction of stall cycles. *Base* performance is improved by 20%, and 14 out of 28 benchmarks show more than 10% speedup. After doubling bandwidth, the fraction of LSU full stall is 12% and clearly RAW stall becomes new bottleneck of 20%. P-mode can again be applied to this new system to alleviate RAW bottleneck, and P-mode achieves 33% performance across all benchmarks.

7. RELATED WORK

SIMT model adopted in GPUs requires threads to be executed in a lock-step fashion. Thus, effective instruction throughput is proportional to the number of SIMT lanes activated per instruction [19, 32, 33, 20]. Hence, a number of techniques have been proposed for improving SIMT lane utilization when threads take divergent branching path on a conditional branch. Dynamic warp formation [32] and dynamic warp subdivision [33] dynamically regroup warps for improving lane utilization. While, dual-path [19] and multi-path model [20] interleaves instructions belong to multiple control paths for reducing pipeline stalls. These techniques require modifications of scoreboard logic to simultaneously keep track of multiple control flow paths. For instance, in the dual-path model [19], two scoreboards per warp are needed to interleave up to two path at the same time. Similarly, our technique uses an augmented scoreboard for independently tracking N-mode and P-mode control flow paths.

Warp scheduling policies have been extensively studied for improving both performance and energy efficiency [6, 5, 7, 6, 8, 34]. Greedy-then-oldest (GTO) [6] and two-level warp scheduling [5] mitigates inefficiency of the round-robin scheduling. GTO is used as our baseline scheduler. CTA-aware scheduling [7] reduces L1 cache contentions and improves long-latency hiding capability by exploiting locality among thread block and bank-level parallelsim of DRAM. Criticality-aware warp scheduling [8, 35] manage warps so that reducing the disparity in execution time between slowest and fastest warp in the same thread block, thereby mitigating warp stalls due to synchronization. These warp scheduling policies are based on the premise that all GPU applications have sufficient level of thread parallelism, while our technique addresses the case when no issueable warp is left in the warp scheduling pool due to long-latency RAW stalls.

Excessive TLP may harm performance due to increased memory contention, and several thread throttling techniques have been proposed [6, 31, 17]. Cache-conscious wavefront scheduling [6] adjusts the number of scheduled warps dynamically based on cache locality. DYNCTA [31] and LCS [17] controls the number of thread blocks to reduce contention. P-mode can be applied independently with thread throttling and we have modeled and evaluated the impact of combined techniques in Section 6.

By definition, long-latency RAW stalls happen when all warps waiting for the result of memory operation. Hence, data prefetching [36, 37, 38, 29, 30] can be a solution for reducing effective memory access latency by bringing cache lines to L1 cache prior to demand request. Although existing studies have shown prefetching is also effective in GPUs, there are inherent challenges in data prefetching; accurate address prediction and timely prefetching. Existing prefetching schemes are designed on an observation that many GPU workloads have relatively regular access stride [29]. In our technique, regardless of access pattern if any independent future load is found it can be used to pre-load cache lines. Further, ineffective prefetching due to address misprediction may harm performance regarding GPUs have smaller cache and performance is often bandwidth limited. In Section 6, we have quantitatively compared prefetching with pre-execution.

Latency tolerance of conventional out-of-order processors is limited by the size of the instruction window [39]. Out-of-order execution in GPUs may achieve higher latency tolerance by executing more independent instructions per warp. Further, it will improve MLP by issuing more demand memory accesses and hence better utilize memory bandwidth. However, out-of-order issue logic is needed which is more complex and power hungry than in-order issue adopted in GPUs [40, 41, 42]. Alternatively, our pre-execution technique is inspired by non-blocking pipeline approaches [9, 10, 11, 12] which allows the processor to continue execution beyond any blocking instructions.

Our technique does not allow any speculation that potentially increase complexity and harm energy efficiency of P-mode [10, 43, 44]. Speculatively executed instructions are difficult to be reused by nature since reusing is possible only when the result speculation turns out to be correct later [10, 45]. Massive thread parallelism of GPUs allows continuing to issue from non-halted P-mode warps although some warps may be halted at long-latency dependent branch. Prohibiting speculation enables effective reuse of instructions in N-mode, thus improves performance and minimizes energy increase with P-mode. Unlike helper thread approaches, such as assisted warp [46], our approach is a purely hardware technique and additional software support is not necessary.

8. CONCLUSIONS

In this paper, we propose warped-preexecution, which executes instructions that are independent of a long latency operation in the shadow of the long latency. The pre-executed results can be reused later during normal operation. We presented the design and evaluated a wide range of design choices. Pre-execution improves performance of latency-sensitive applications by 23%. We also explored various design space choices to show that pre-execution is advantageous in a wide range of microarchitectural configurations.

9. ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01008281), and by the following grants: DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211, NSF-0834798.

10. REFERENCES

[1] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, ISPASS ’10, pp. 235–246, March 2010.

[2] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, “A hierarchical thread scheduler and register file for energy-efficient throughput processors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 30, pp. 8:1–8:38, Apr. 2012.

[3] M. Gebhart, D. 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,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA ’11, (New York, NY, USA), pp. 235–246, ACM, 2011.

[4] Q. Xu and M. Annavaram, “Pats: Pattern aware scheduling and power gating for gpgpus,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT ’14, (New York, NY, USA), pp. 225–236, ACM, 2014.

[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,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 308–317, ACM, 2011.

[6] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 72–83, IEEE Computer Society, 2012.

[7] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, 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,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’13, (New York, NY, USA), pp. 395–406, ACM, 2013.

[8] S.-Y. Lee and C.-J. Wu, “Caws: Criticality-aware warp scheduling for gpgpu workloads,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT ’14, (New York, NY, USA), pp. 175–186, ACM, 2014.

[9] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, “A large, fast instruction window for tolerating cache misses,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA ’02, (Washington, DC, USA), pp. 59–70, IEEE Computer Society, 2002.

[10] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, “Runahead execution: An alternative to very large instruction windows for out-of-order processors,” in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA ’03, (Washington, DC, USA), pp. 129–, IEEE Computer Society, 2003.

[11] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual flow pipelines,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, (New York, NY, USA), pp. 107–119, ACM, 2004.

[12] A. Hilton, S. Nagarakatte, and A. Roth, “icfp: Tolerating all-level cache misses in in-order processors,” in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 431–442, Feb 2009.

[13] M. Abdel-Majeed and M. Annavaram, “Warped register file: A power efficient register file for gpgpus,” in *Proceedings of the 2013 IEEE 19th International Symposium on High-Performance Computer Architecture*, HPCA ’13, 2013.

[14] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, “An energy-efficient and scalable edram-based register file architecture for gpgpu,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, (New York, NY, USA), pp. 344–355, ACM, 2013.

[15] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, “Gpu register file virtualization,” in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-48, 2015.

[16] A. Sethia, D. Jamshidi, and S. Mahlke, “Mascar: Speeding up gpu warps by reducing memory pitstops,” in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA ’15, 2015.

[17] M. Lee, S. Song, J. Moon, J.-H. Kim, W. Seo, Y. Cho, and S. Ryu, “Improving gpgpu resource utilization through alternative thread block scheduling,” in *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA ’14, 2014.

[18] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, “Managing gpu concurrency in heterogeneous architectures,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 114–126, IEEE Computer Society, 2014.

[19] M. Rhu and M. Erez, “The dual-path execution model for efficient gpu control flow,” in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA ’13, (Washington, DC, USA), pp. 591–602, IEEE Computer Society, 2013.

[20] A. ElTantawy, J. Ma, M. O’Connor, and T. Aamodt, “A scalable multi-path microarchitecture for efficient gpu control flow,” in *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA ’14, 2014.

[21] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, “Warped-compression: Enabling power efficient gpus through register compression,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA ’15, (New York, NY, USA), pp. 502–514, ACM, 2015.

[22] E. Rotenberg, S. Bennett, and J. E. Smith, “Trace cache: A low latency approach to high bandwidth instruction fetching,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-29, (Washington, DC, USA), pp. 24–35, IEEE Computer Society, 1996.

[23] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *Center for Reliable and High-Performance Computing*, 2012.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE International Symposium on Workload Characterization*, IISWC ’09, pp. 44–54, Oct 2009.

[25] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pp. 163–174, April 2009.

[26] NVIDIA, "CUDA C Programming Guide."

[27] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0," tech. rep., Technical Report HPL-2006-86, HP Laboratories Palo Alto, 2006.

[28] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 487–498, ACM, 2013.

[29] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, (Washington, DC, USA), pp. 213–224, IEEE Computer Society, 2010.

[30] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 73–82, IEEE Press, 2013.

[31] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 157–166, IEEE Press, 2013.

[32] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, (Washington, DC, USA), pp. 407–420, IEEE Computer Society, 2007.

[33] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 235–246, ACM, 2010.

[34] M. K. Yoon, Y. Oh, S. Lee, S. H. Kim, D. Kim, and W. W. Ro, "Draw: investigating benefits of adaptive fetch group size on gpu," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, ISPASS '15, pp. 183–192, March 2015.

[35] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 515–527, ACM, 2015.

[36] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, (New York, NY, USA), pp. 52–61, ACM, 2001.

[37] D. H. Woo and H.-H. S. Lee, "Compass: A programmable data prefetcher using idle gpu shaders," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 297–310, ACM, 2010.

[38] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Cpu-assisted gpgpu on fused cpu-gpu architectures," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[39] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, (Washington, DC, USA), pp. 423–, IEEE Computer Society, 2003.

[40] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 206–218, ACM, 1997.

[41] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, no. 2, pp. 56–69, 2010.

[42] NVIDIA, "Whitepaper: NVIDIA Fermi."

[43] W. Ro and J.-L. Gaudiot, "Spear: a hybrid model for speculative pre-execution," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, IPDPS '04, pp. 75–, April 2004.

[44] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, "Value prediction and speculative execution on gpu," *International Journal of Parallel Programming*, vol. 39, no. 5, pp. 533–552, 2011.

[45] O. Mutlu, H. Kim, J. Stark, and Y. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *Computer Architecture Letters*, vol. 4, pp. 2–2, January 2005.

[46] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 41–53, ACM, 2015.