# ELF: Maximizing Memory-level Parallelism for GPUs with Coordinated Warp and Fetch Scheduling

Jason Jong Kyu Park
University of Michigan
Ann Arbor, MI
jasonjk@umich.edu

Yongjun Park
Hongik University
Seoul, Korea
yongjun.park@hongik.ac.kr

Scott Mahlke
University of Michigan
Ann Arbor, MI
mahlke@umich.edu

## ABSTRACT

Graphics processing units (GPUs) are increasingly utilized as throughput engines in the modern computer systems. GPUs rely on fast context switching between thousands of threads to hide long latency operations, however, they still stall due to the memory operations. To minimize the stalls, memory operations should be overlapped with other operations as much as possible to maximize memory-level parallelism (MLP). In this paper, we propose Earliest Load First (*ELF*) warp scheduling, which maximizes the MLP by giving higher priority to the warps that have the fewest instructions to the next memory load. *ELF* utilizes the same warp priority for the fetch scheduling so that both are coordinated. We also show that *ELF* reveals its full benefits when there are fewer memory conflicts and fetch stalls. Evaluations show that *ELF* can improve the performance by 4.1% and achieve total improvement of 11.9% when used with other techniques over commonly-used greedy-then-oldest scheduling.

## CCS Concepts

**•Computer systems organization → Single instruction, multiple data; •Software and its engineering → Runtime environments;**

## Keywords

Graphics Processing Unit, Compiler, Memory-level Parallelism, Warp Scheduling

## 1. INTRODUCTION

The trend of using graphics processing units (GPUs) as throughput engines in the modern computer systems is constantly increasing as their computing capability and energy efficiency exceed the traditional processors (CPUs). New programming models such as OpenCL [11] or CUDA [20] enable programmers to develop data parallel kernels that can exploit a huge number of available compute resources on GPUs. These models launch thousands of threads together to the hundreds of processing units on the GPU. By context switching between the large number of threads quickly, GPUs can hide long latency operations and achieve high throughput.

As more diverse applications adopt GPUs to exploit their computing capability, many have reported the difficulty of achieving the peak performance [25, 26]. Many recent works attempted to tackle this problem [17, 23, 9, 10, 8, 28]. CCWS [23] modified warp scheduling to reduce L1 cache contention. DYNCTA [10] noted that reducing the number of concurrently running thread blocks can reduce memory contention. MRPB [8] showed the importance of prioritizing memory requests from the same warp. These works show that cache/memory contention is one of the main reasons why GPUs are not achieving peak performance.

Cache/memory contention often makes GPUs stall because memory operations have the highest latency. GPUs schedule a group of threads called a *warp* at the same time to leverage data-level parallelism. When a warp is blocked due to an operation that is dependent on a memory operation, the warp is swapped for another warp until the memory request comes back. If all the warps are blocked, GPUs can no longer hide the memory latencies. To reduce the impact of these unhidden memory latencies, it is strongly suggested that memory operations are overlapped with each other as much as possible. In other words, memory-level parallelism (MLP) has to be maximized.

In this paper, we propose Earliest Load First (*ELF*) scheduling, which maximizes the MLP by issuing memory operations as soon as possible. To maximize the MLP, *ELF* gives higher priority to the warps with fewer remaining instructions to reach the next memory operation. The highest priority warp will continue to issue instructions until it issues the next memory operation. *ELF* leverages compiler techniques to identify program points that are needed to calculate the priority, and annotate their information in the binary to notify the hardware.

In order to ensure that the highest priority warp continuously has instructions to issue, the fetch unit also has to fetch according to the issue priorities. Otherwise, a warp scheduler has to suffer because a lower priority warp will have to issue. By using the coordinated warp priority between fetch and warp scheduling, the decision from warp scheduling becomes more effective as the fetch unit supplies instructions to the warps that are trying to issue with higher priority. Moreover, we employ instruction prefetch to reduce fetch stalls that can prohibit the highest priority warp from



*SC '15, November 15–20, 2015, Austin, TX, USA*



DOI: http://dx.doi.org/10.1145/2807591.2807598

making progress.

*ELF* can be limited by memory resource saturation when trying to issue a memory request. However, with multiple independent warps concurrently running on GPUs, the memory resource saturation from one warp does not necessarily mean that other warps will experience the memory resource saturation as well because other warps may see hit-under-miss or avoid associativity stalls. Without any solutions to allow *ELF* to issue memory requests even if previous memory request is blocked due to memory resource saturation, *ELF* may lose its effectiveness because it cannot further exploit the MLP.

This paper makes the following contributions:

- We propose *ELF*, a warp scheduling technique that maximizes the MLP by prioritizing warps with fewer remaining instructions to the next memory operation. Furthermore, we also show that the interplay between the warp scheduler and fetch unit is important in *ELF*.
- We introduce a compiler technique that analyzes the program and passes the necessary information for priority calculation to the GPU hardware. With hardware/software co-design, the overhead of priority calculation can be minimized.
- We propose to use *ELF* with other orthogonal techniques, which help to avoid situations when the actual scheduling cannot follow the expected scheduling from *ELF*. We discuss an extended version of cache access re-execution (NewCAR) to handle memory conflicts, and an instruction prefetch to reduce stalls from the fetch unit.

# 2. BACKGROUND AND MOTIVATION

In this section, we briefly introduce the GPU terminology and the GPU architecture. We also motivate the case for *ELF* scheduling.

## 2.1 Terminology

We use Nvidia's terminology throughout the paper. The GPU programming model is based on a single instruction multiple thread (SIMT) model to explicitly express the parallelism in the *kernel* code, which is the parallel code section that runs on the GPUs. In the SIMT model, a programmer writes a code for a *thread*. In the GPU hardware, the threads are executed in a group called a *warp*. A warp is not an exposed concept to the GPU programming model, but rather a micro-architectural decision to process threads efficiently. In Nvidia's GPU architecture, a warp consists of 32 threads. The programmer also specifies a group of threads called a *thread block*. The threads within the same thread block can be synchronized with an explicit barrier operation, and have access to a common, fast, on-chip scratch-pad memory called *shared memory*. Finally, a *grid*, which is a group of thread blocks, will be grouped to form a kernel.

## 2.2 GPU Architecture

Figure 1 illustrates a modern GPU architecture, which is similar to the Nvidia's Fermi architecture [19]. GPUs consist of multiple streaming multiprocessors (SMs), an interconnect, and multiple memory partitions, where each memory partition contains a shared L2 cache bank, a memory controller, and an off-chip DRAM channel. The detailed

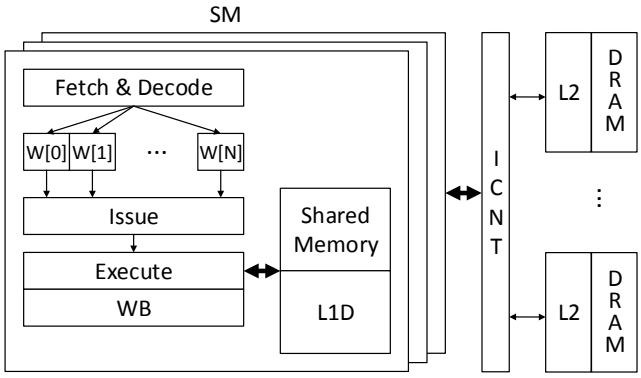



**Figure 1: An overview of GPU architecture. W[i] denotes a warp.**

execution pipeline of an SM is also shown in the figure. Instruction fetch, decode, issue, and execute are performed at a warp granularity. Note that the figure neglects a read-only texture cache and a read-only constant cache in each SM for simplicity.

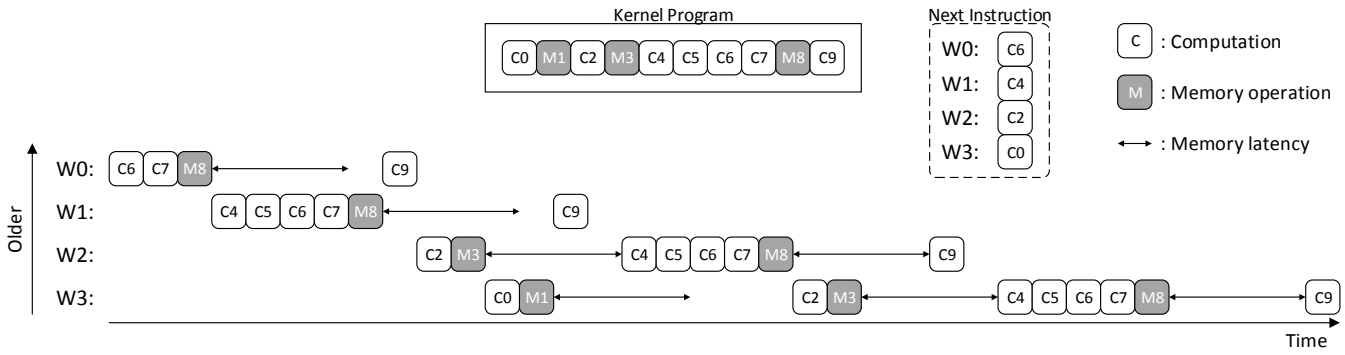
In an SM, fetch and issue pipelines are shared by all the warps. Because there is no dependency between the execution of the warps except for the synchronization within thread blocks, deciding which warp should fetch or issue an instruction among ready warps is an important problem in GPUs. Although the issue problem (or warp scheduling) has been studied extensively in the past [17, 23, 24, 9, 28], fetch scheduling has received less attention [12]. Note that in the Fermi architecture, there are multiple warp schedulers, which issue from independent subsets of warps.

## 2.3 Maximizing Memory-level Parallelism

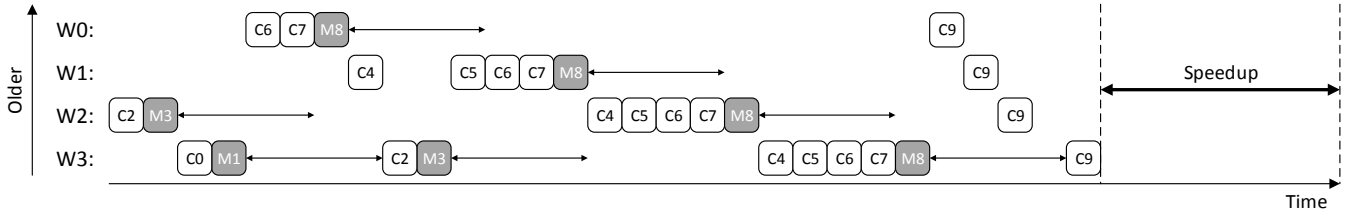
Figure 2 illustrates an example execution timeline when greedy-then-oldest (GTO) scheduling [23] and *ELF* scheduling are applied. For simplicity, we assume that all the computations have a single cycle latency, and all memory operations have a four cycle latency. We also assume that each instruction is dependent on the previous instruction. The top box denotes a kernel program with 10 instructions. The top right box shows the next instruction to be executed for each warp. Unless strict round-robin scheduling is used, it is likely that warps are executing different instructions.

In Figure 2 (a), the GTO scheduler selects an instruction from the warp which was issued in the previous cycle. If the warp cannot progress because of a long latency memory operation, GTO selects the oldest warp among the ready warps. Because GTO prioritizes older warps, it cannot tolerate long memory latencies from younger warps. On the other hand, *ELF* in Figure 2 (b) tries to issue memory instructions as early as possible. Because memory instructions are issued quickly, their long latencies are overlapped more with themselves or other computations. As a result, *ELF* can achieve a better aggregate throughput with higher MLP.

The intuition behind *ELF* scheduling is simple. GPUs become idle when there are no ready instructions, which typically happens when warps are waiting for long latency memory operations to finish. If these memory operations can be scheduled earlier than the computations from other warps, their latency can be overlapped with these computations and other memory operations. For example, W2




(a) An example of GTO warp scheduling




(b) An example of ELF warp scheduling

**Figure 2: Execution timeline of (a) greedy-then-oldest (GTO) warp scheduling, and (b) ELF warp scheduling for the given kernel program, where warps are at different execution points.**

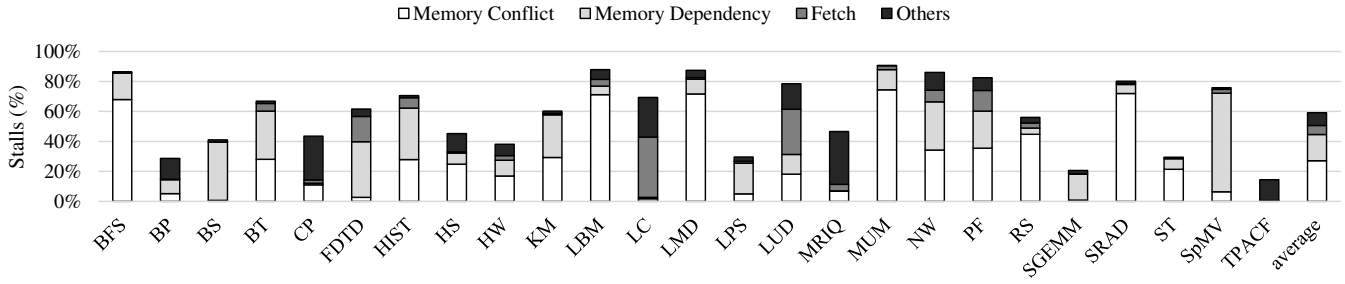



**Figure 3: Distribution of stalls in the GTO scheduler. Memory conflict stalls occur when memory resources are saturated. Memory dependency stalls occur when instructions are waiting for the memory requests to come back. Fetch stalls happen when warps are waiting for instruction fetch. Other stalls include conflict and dependency stalls in other functional units.**

and W3 send out their memory requests earlier in Figure 2 (b), which makes them ready for their compute-intensive phase (four consecutive computations) earlier. The illustrative example in Figure 2 shows the importance of issuing long latency memory operations as early as possible.

## 2.4 Memory Conflicts and Fetch Stalls

If GPUs have infinite memory resources, issuing memory operations as soon as possible will always maximize the MLP. In reality, GPUs have limited memory resources, which can result in memory conflicts that prohibit the MLP to be maximized.

Figure 3 illustrates the distribution of stalls with GTO, which is percentage of cycles when a warp scheduler could not issue an instruction. We categorize the stalls into four classes: memory conflict stalls, memory dependency stalls, fetch stalls, and other stalls. Memory conflict stalls occur when a warp scheduler could not issue an instruction because of memory resource conflicts. These stalls occur when the load/store unit cannot accept any more instructions. Memory dependency stalls occur when a warp scheduler could not issue an instruction because at least one of the source operands in the instruction are waiting for the memory requests to come back. Fetch stalls occur when a warp scheduler is waiting for instruction fetch. Other stalls include conflict and dependency stalls from other functional units. For example, a dependency stall can occur for special function units (SFUs) because they can take multiple cycles.

As shown in the figure, there are 27.2% memory conflict stalls, 17.5% memory dependency stalls, 6.0% fetch stalls, and 8.4% other stalls on average. The results show that memory conflict stalls are already a dominant source of stalls, and can be a limiting factor when we are trying to maximize the MLP as in Figure 2 (b). Therefore, a strategy of maximizing the MLP should be accompanied by a technique, which can reduce the memory conflicts. Another important observation is that fetch stalls can take a large portion of stalls for benchmarks such as FDTD, LC, and LUD. In such benchmarks, a strategy of maximizing the MLP is again limited by the fetch unit. Therefore, a tech-

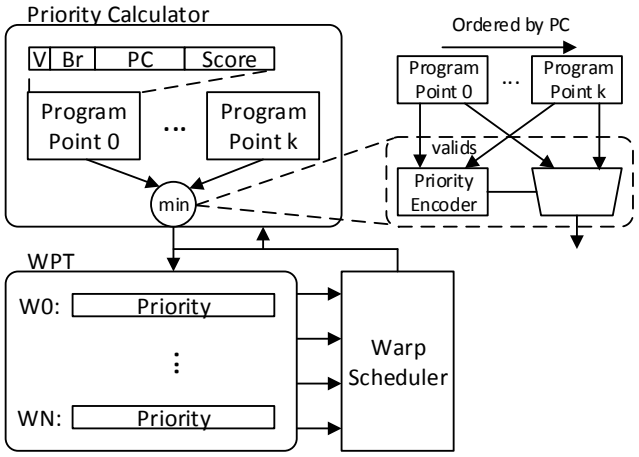



**Figure 4: Overall architecture for *ELF*. There are two components in *ELF*: a priority calculator, and a warp priority table (WPT). The priority calculator calculates the priority of a warp using the program points generated by the compiler. The warp priority table stores the results from the priority calculator, and directs the warp scheduler to issue accordingly.**

nique that can reduce fetch stalls should also be coupled with maximizing the MLP.

# 3. ARCHITECTURE

*ELF* is a warp scheduling technique that utilizes both compiler and hardware to maximize MLP by prioritizing a warp that has the earliest memory load. *ELF* relies on the program points that are necessary to compute the priority. A *program point* is defined as an instruction that can change the distance to the next memory load. By definition, memory loads are program points. Branch instructions are also program points because they alter the control flow of the program. Remaining instructions are not program points.

Figure 4 illustrates the overall architecture for *ELF*. At the top, a priority calculator is shown, where program points and their related values are computed from the compiler side, and conveyed to the hardware through binary. On the right side of the top box, an implementation of minimum functionality with a priority encoder is shown, which will be discussed in detail in Section 3.2. At the bottom, we show the warp priority table (WPT), which keeps the priority for each warp. In *ELF*, a priority equals to the number of remaining instructions to the next memory load hence lower value in the priority means higher priority to issue an instruction. The priorities from the WPT are referenced by the warp scheduler when it issues.

## 3.1 Finding Program Points

*ELF* generates the list of program points from the compiler. In *ELF*, each program point is either a memory load operation or a branch. Among memory loads, we do not consider shared memory loads, constant memory loads, and parameter loads because they are likely to be cache hit, which will have similar latency as ordinary computations. In *ELF*, we only consider backward branches to reduce the number of total program points, however, forward branches can also be considered. These program points are notified to the GPU hardware, and further used by priority calculator to compute the priority for each warp. Two hardware parameters are given to the compiler beforehand: the maximum number of program points is restricted by the number of available program point slots in the GPU, and the maximum score is limited by the bitwidth of the score field in the program point slot.

Each program point in *ELF* is associated with a data called score. For a branch program point, it stores the minimum distance to the next memory load considering all the possible paths after the branch. For a load program point, score is meaningless and zero by default. *ELF* utilizes the score field of memory loads to merge program points as described by Algorithm 2 when the kernel has more program points than the available program points in the hardware.

Algorithm 1 shows how the program points are generated by the compiler. The algorithm starts by finding memory loads and branches to form initial program points (lines 1-9). Iteratively, *ELF* computes the score of branches by taking the minimum of the scores from the taken path and the fall-through path (lines 10-24). The score of each path is the addition of the number of instructions to the next program point in each path and the next program point's score (lines 36-45). If next program point does not exist, the maximum score is returned (line 46). After the program points are resolved, meaningless program points are first removed (lines 25-31). A branch program point is meaningless if it has the maximum score or the score of the fall-through path. Lastly, the program points are merged if they exceed the available slots in the GPU (lines 32-34).

Algorithm 2 shows how to merge program points. First, two program points are found, where the distance between the two is the minimum (lines 1-15). When merging program points, the program point that merges its previous program point has to be a memory load (line 3). Then, the score field of the later program point is updated with the distance to the previous program point (line 16). Note that we subtract the merged program point's score if it is a branch (line 11). Finally, the merged program point is removed from the program points, which reduces the number of program points by 1 (line 17).

## 3.2 Priority Calculator

The priority calculator loads the program points embedded in the binary when a kernel is launched onto an SM. As shown by the top box in Figure 4, each program point has a valid bit, a branch bit, program counter (PC), and a score. Using the PC and the score of a program point with the PC of an instruction, a distance to a memory load can be calculated. A priority, which is the minimum distance to the next memory load instruction, can be calculated by taking the minimum of the distances to all the memory loads. As shown in Figure 4, a priority encoder can practically implement the minimum functionality when program points are ordered by PC because it is guaranteed that the closest program point will give the minimum.

Given the instruction and a memory program point, it is possible to compute the number of instructions between the instruction and the memory load using PC. Given the instruction and a branch program point, the distance to the branch is first calculated using PC, and then the score field is added to give the distance to the memory load after the branch. Note that a memory program point may also have

**Algorithm 1** Finding Program Points

```
    findProgramPoints(Kernel):
    Output: ProgramPoints[1..MaxProgramPoints]
 1: for each inst in Kernel do
 2:     if inst is MemoryLoad then
 3:         inst.score = 0     ▷ Lower score is higher priority
 4:         ProgramPoints.push(inst)
 5:     else if inst is Branch then
 6:         inst.score = MaxScore
 7:         ProgramPoints.push(inst)
 8:     end if
 9: end for
10: changed = true
11: while changed do
12:     changed = false
13:     for each inst in ProgramPoints do
14:         if inst is Branch then
15:             nextScore = getScore(inst.nextInst) + 1
16:             targetScore = getScore(inst.targetInst) + 1
17:             minScore = min(nextScore, targetScore)
18:             if minScore <inst.score then
19:                 inst.score = minScore
20:                 changed = true
21:             end if
22:         end if
23:     end for
24: end while
25: for each inst in ProgramPoints do
26:     if inst is Branch then
27:         if inst.score == MaxScore or inst.score ==
    getScore(inst.nextInst) + 1 then
28:             ProgramPoints.remove(inst)
29:         end if
30:     end if
31: end for
32: while ProgramPoints.size() >MaxProgramPoints do
33:     ProgramPoints.merge()
34: end while
35: return ProgramPoints[1..MaxProgramPoints]

    getScore(inst):
    Output: score
36: currInst = inst
37: score = 0
38: while currInst.valid do
39:     if currInst in ProgramPoints then
40:         score = min(score + currInst.score, MaxScore)
41:         return score
42:     end if
43:     currInst = currInst.nextInst
44:     score += 1
45: end while
46: return MaxScore
```

**Algorithm 2** Merging Program Points

```
    merge(ProgramPoints):
 1: minDistance = MaxScore
 2: for each inst in ProgramPoints do
 3:     if inst is MemoryLoad then
 4:         prevPoint = inst.getPrevProgramPoint()
 5:         currDistance = getScore(prevPoint.nextInst) + 1
 6:         if currDistance <minDistance then
 7:             minDistance = currDistance
 8:             memInst = inst
 9:             prevInst = prevPoint
10:             if prevPoint is Branch then
11:                 minDistance -= prevPoint.score
12:             end if
13:         end if
14:     end if
15: end for
16: memInst.score = minDistance
17: ProgramPoints.remove(prevInst)
```

non-zero score if other program points have been merged to that program point. In such a case, the calculated priority is subtracted by the score if it is larger than the score to account for the merged program point.

Naively computing priority every time when a warp issues an instruction can have significant overhead. Because priority is the number of remaining instructions to the next memory load operation, it can be decreased by one most of the time. To exploit this property, the priority calculator is triggered only in two cases: when a memory load operation is issued, or when a branch has altered the priority. For the second case, *ELF* extends the instruction format of each branch with a recalculation bit. The recalculation bit indicates whether the priority is altered when the branch is taken. If it is set, a taken branch triggers the priority calculator. Otherwise, a branch triggers the priority calculator if it is not taken. The compiler sets the recalculation bit when the taken path has fewer remaining instructions to the next program point than the fall-through path. Priorities of not issued warps are not changed hence no computation is required.

## 3.3 Fetch Scheduling in *ELF*

Fetch scheduling can play an important role as fetch unit is shared by the warps as discussed in Section 2.2. We propose to use the same priority as the warp schedulers to prioritize instruction fetch. In Fermi, there are two warp schedulers, where each of them schedules among an independent subset of the warps, while a single fetch unit exists. To match the issue width, the fetch unit requests two instructions for one warp in a cycle. To handle two distinct priority orders from the warp schedulers without starving one warp scheduler, *ELF* constructs an unified priority order by interleaving the priority orders from each warp schedulers.

## 3.4 *ELF* with Cache Access Re-execution

*ELF* may not be able to maximize the MLP when memory conflicts occur. The problem of memory conflicts can be more severe in GPU execution model, where warps share the load-store unit (LSU), because not only the current memory request is blocked but also other memory requests from other warps that may be totally independent can be blocked as a result.

Prior works have proposed solutions that can mitigate the problem of memory conflicts. For example, MRPB [8] reported that GPUs can have an associativity stall because of the allocate-on-miss policy on the L1 cache. In such a case, MSHRs and miss queues cannot be utilized even though they are available. MRPB avoids such problem with bypassing. Mascar [28] reported the opportunity of hit-under-miss, and exploited the opportunity with the cache access re-execution (CAR). Another problem that was not mentioned exists because the LSU is shared among the constant, texture, and L1D cache. For example, when the LSU is blocked by L1D cache, memory requests that can be served by constant or texture cache are also blocked. Although the mentioned problems seem different, they can be solved at the same time with any of the previous solutions.

Instead of devising a completely new way to overcome the

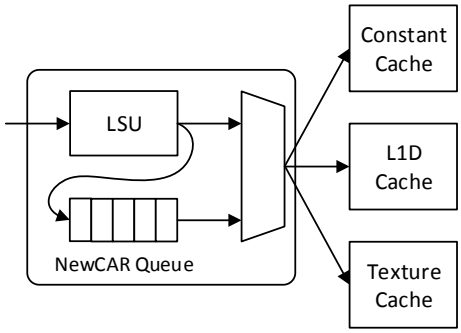



**Figure 5: Extending LSU with NewCAR. Structurally, NewCAR is almost identical to CAR [28]. The key difference is more relaxed conditions on when and how the NewCAR queue is controlled.**

problems, *ELF* adopts the CAR with 32 entry re-execution queue from Mascar with extensions. The NewCAR is structurally similar to the CAR as shown in Figure 5. However, NewCAR operates with more relaxed conditions compared to CAR, which can reduce more memory conflict stalls. First, multiple memory requests per warp are allowed in NewCAR. This is intra-warp optimization, which allows a warp to progress more even with the memory conflicts. Second, memory requests in the queue can be processed out-of-order when the weak memory consistency semantics are preserved. This is both intra and inter-warp optimization, which allows more freedom compared to CAR.

Figure 5 illustrates the extension of LSU with NewCAR. A NewCAR queue is attached to the LSU, where a memory request can enter the queue from the LSU. A memory request is inserted into the queue if one of the two conditions is met: 1) if a memory request was sent to one of the caches and not accepted, or 2) if a memory request cannot bypass the queue due to the memory consistency semantics. NewCAR always gives priority to the memory request from the LSU. A memory request from the queue is processed if one of the two conditions is met: 1) if the queue is full, or 2) if LSU is idle. Note that when the queue is full, the LSU is prohibited from issuing a new memory request until the queue has an empty slot.

Figure 6 shows which memory request reorderings are allowed when there is a prior memory request from a warp in the NewCAR queue. Because memory requests are independent between the warps, memory requests can always bypass another warp's memory requests. If a warp already has a load in the queue, it can issue a new load request ahead of the prior load but not a store request. If a warp has a store in the queue, it cannot issue any new memory request before the store has been processed. Whenever the memory request is not allowed to bypass the queue due to the violation of memory consistency semantics, it goes straight into the NewCAR queue.

## 3.5 *ELF* with Instruction Prefetch

Fetch stalls can stop high priority warps from issuing instructions in *ELF* because they need to wait for an L1I cache miss as discussed in Section 2.4. To reduce such waits, we employ a simple next line prefetcher [29] for the L1I. However, naively using a prefetcher on GPUs can increase

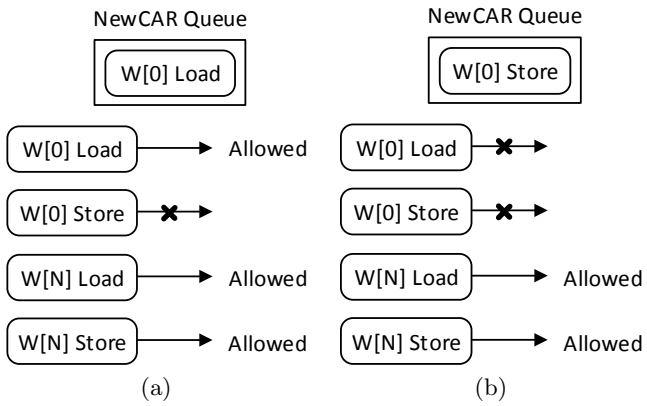



**Figure 6: Allowed reordering of memory requests when a warp has (a) a load, and (b) a store waiting in the NewCAR queue. W[0] denotes an example warp that has a memory request in the NewCAR queue, and W[N] denotes any other warp. Memory requests from other warps can always bypass the memory request from the example warp. Loads from the example warp can bypass the loads from itself, but not the stores. Stores from the example warp cannot bypass any loads or stores from itself.**

the memory contention, which can negatively impact the GPU performance as studied by prior work on data prefetchers [13, 27]. To avoid this problem, the occupancy of MSHRs can be monitored to determine whether there is memory contention. The next line prefetcher for the L1I only issues a prefetch request when the total number of occupied MSHRs in both L1I and L1D is less than a threshold. As shown in Section 4.1, we only issue an instruction prefetch when the total number of occupied MSHRs is less than 16.

# 4. RESULTS

We use the GPGPU-Sim v3.2.2 [1] to evaluate *ELF*. GPGPU-Sim only models the GPU, where the host code and the overhead of data transfers between the CPU and the GPU do not affect the simulation results. We model a Fermi [19] architecture, which is similar to the Nvidia GTX480. The detailed configuration is listed in Table 1. For *ELF*, we assumed there are 32 available program point slots, and the score field is 8-bit, which can store up to a distance of 256 instructions. We implemented the compiler part of *ELF* as a part of the run-time system, which performs all the necessary analysis before the kernel is launched and passes the generated information to the GPGPU-Sim.

We evaluate a wide range of GPGPU applications from Nvidia SDK [18], GPGPU-Sim [1], Rodinia v2.4 [2], and Parboil [31] benchmark suite. Table 2 lists all the evaluated benchmarks, their labels, and kernels with the number of program points before merge. We left out trivial kernels from SDK, and few benchmarks from other suites that took too much time to simulate even with the smallest input.

We first explore the individual performance improvements *ELF* and other two orthogonal techniques over the baseline greedy-then-oldest (GTO) scheduling. We also evaluate whether the improvements are additive or have synergy when used together, which is called *ELF++*.

We compare *ELF++* with three prior warp scheduling

| System | Parameters |
|---|---|
| SM | 15 SMs, 1400 MHz, 32 SIMT width<br>32768 registers per SM<br>1536 maximum threads per SM<br>8 maximum thread blocks per SM<br>48 kB shared memory |
| Memory Subsystem | 2 kB/4-way/128B L1I per SM<br>8 MSHRs per L1I<br>16 kB/4-way/128B L1D per SM<br>32 MSHRs per L1D<br>768kB/16-way/128B L2<br>32 MSHRs per L2 partition<br>6 memory partitions<br>FR-FCFS DRAM scheduler<br>177.4 GB/s bandwidth |

**Table 1: System configuration**

policies: 2-LV [17], CCWS [23], and DYNCTA [10]. For 2-LV, we used the version provided with the GPGPU-Sim v3.2.2. For CCWS, we used publicly available version, which is based on a prior GPGPU-Sim version than the one that evaluates *ELF*. We modified the GPU configuration to match our baseline Fermi architecture that resembles the GTX480. To be fair, we used the GTO in CCWS version as the baseline of CCWS. We implemented DYNCTA, and verified the results with the prior published work.

## 4.1 *ELF* Performance

Figure 7 shows the individual performance improvement of NewCAR, instruction prefetch, and *ELF* over the baseline GTO as well as *ELF++*, which uses the three techniques together. On average, NewCAR, instruction prefetch, *ELF* and *ELF++* improve the performance by 2.5%, 4.9%, 4.1%, and 11.9%, respectively. While NewCAR and instruction prefetch are very effective for few benchmarks, they do not provide consistent benefit across all the benchmarks. On the other hand, *ELF* is broadly effective among the benchmarks. Also, NewCAR may degrade the performance as in the case of BS and LBM because the locality may be lost in the cache when there are many requests waiting in the re-execution queue. FDTD and LC, which show large improvements from instruction prefetch, correspond to the benchmarks with high fetch stalls from Figure 3. The result illustrates that the fetch scheduling can be as important as the warp scheduling.

As shown in the figure, *ELF* performs better than the baseline GTO because the MLP is maximized in warp scheduling as well as the fetch scheduling. Note that *ELF* improves performance for not only the memory-intensive benchmarks like BFS and KM [23] but also the compute-intensive benchmarks like CP and HS [10].

When *ELF* is combined with NewCAR and instruction prefetch, they compensate each other. As shown by the results, *ELF++* shows additive improvements from the three techniques, and a synergy of 0.4% additional improvement on average. The synergy mostly comes from *ELF* and NewCAR, where *ELF++* can exploit more MLP with NewCAR because NewCAR reduces memory conflict stalls.

Figure 8 illustrates the percentage of reduction in the number of priority recalculation in *ELF* compared to the naive priority recalculation. The number of priority recalculation can be reduced by 91.4% on average, by only invoking the recalculation when necessary. Because a warp

| Benchmark | Label | Kernel | PP |
|---|---|---|---|
| Breadth First Search [2] | BFS | Kernel | 9 |
| | | Kernel2 | 1 |
| Back Propagation [2] | BP | bpnn_layerforward | 2 |
| | | bpnn_adjust_weights | 12 |
| BlackScholes [18] | BS | BlackScholesGPU | 4 |
| B+ Tree [2] | BT | findRangeK | 22 |
| | | findK | 13 |
| Coulombic Potential [31] | CP | cenergy | 3 |
| 3D Finite-Difference Time-Domain [18] | FDTD | FiniteDifferences | 60 |
| Histogram [31] | HIST | prescan_kernel | 7 |
| | | intermediates_kernel | 16 |
| | | main_kernel | 15 |
| | | final_kernel | 20 |
| HotSpot [2] | HS | calculate_temp | 2 |
| Heart Wall [2] | HW | kernel | 94 |
| Kmeans [2] | KM | invert_mapping | 2 |
| | | kmeansPoint | 3 |
| Laplace-Boltzmann Method [31] | LBM | performStrideCollide | 20 |
| Leukocyte Tracking [2] | LC | GICOV_kernel | 4 |
| | | dilate_kernel | 3 |
| | | IMGVF_kernel | 10 |
| LavaMD [2] | LMD | kernel_gpu_cuda | 23 |
| 3D Laplace [1] | LPS | GPU_laplace3d | 5 |
| LU Decomposition [2] | LUD | lud_diagonal | 16 |
| | | lud_perimeter | 48 |
| | | lud_internal | 3 |
| Magnetic Resonance Imaging [31] | MRIQ | ComputePhiMag | 2 |
| | | ComputeQ | 5 |
| MUMmerGPU [2] | MUM | mummergpuKernel | 9 |
| | | printKernel | 12 |
| Needleman Wunsch [2] | NW | needle_cuda_shared_1 | 19 |
| | | needle_cuda_shared_2 | 19 |
| Particle Filter [2] | PF | likelihood | 22 |
| | | sum | 2 |
| | | normalize_weight | 8 |
| | | find_index | 5 |
| RadixSort [18] | RS | radixSortBlocks | 2 |
| | | findRadixOffsets | 1 |
| | | reorderData | 4 |
| Matrix Multiply [31] | SGEMM | mysgemmNT | 20 |
| Speckle Reducing Anisotropic Diffusion [2] | SRAD | srad_cuda_1 | 9 |
| | | srad_cuda_2 | 10 |
| Stencil [31] | ST | hybrid_coarsen_x | 13 |
| Sparse Matrix Vector Multiply [31] | SpMV | spmv_jds | 11 |
| Two Point Angular Corr. Function [31] | TPACF | gen_hists | 8 |

**Table 2: Benchmark specification. PP refers to the number of program points before the merge.**

progresses by one instruction most of the time, which reduces the distance to the next memory load by one, most of the priority recalculation can be avoided. The reduction is bounded by the ratio of memory loads and branches in the total executed instructions as they are the only source of priority recalculation. For example, BT has less reduction because it has more fraction of memory loads than the other benchmarks.

Figure 9 (a) depicts the sensitivity of *ELF++* to the number of available program points on a GPU. On average, performance is improved by 11.9%, 11.9%, 11.9%, and 12.1% when the number of program points is 16, 32, 48, and infinite, respectively. As expected, the performance is dropped when the number of program points changes from infinite to a finite number. Although performance on average slightly improves as the number of program points is decreased, however, individual benchmark shows random trend. For example, HW, which has the largest number of program points, shows the peak performance when the number of program

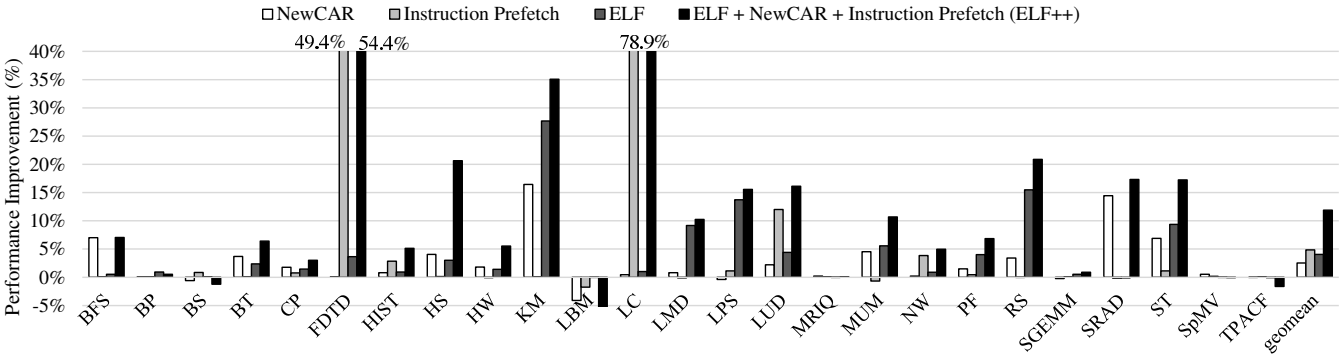



**Figure 7: Performance improvement of NewCAR, instruction prefetch, *ELF*, and *ELF++* over the baseline GTO. *ELF++* combines *ELF* with NewCAR and instruction prefetch.**

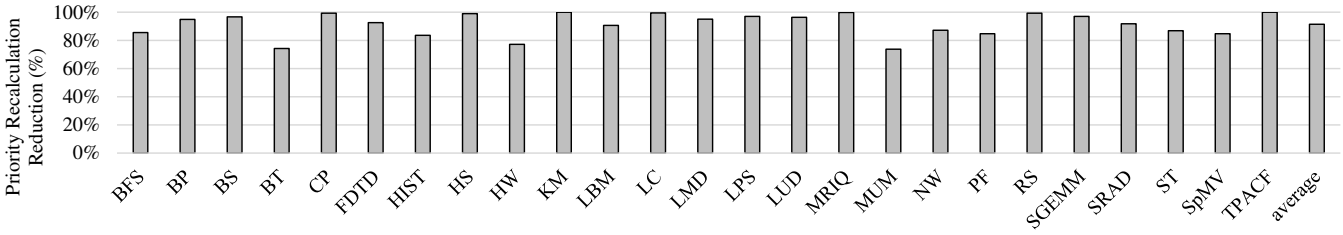



**Figure 8: Reduction of priority recalculation in *ELF* compared to the naive priority recalculation.**

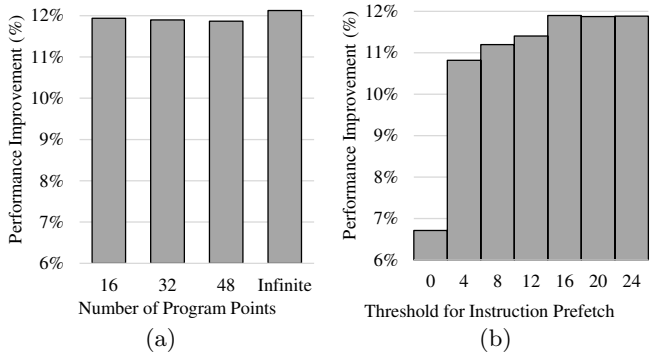



**Figure 9: Sensitivity of *ELF++* to (a) the number of available program points, and (b) the threshold for instruction prefetch. A geometric mean of IPC improvement for all the benchmarks is presented.**

points is 32. *ELF* chooses to merge the two program points with the least distance between them, but removed program point may be an important load that should be considered for priority calculation. We believe that profiling can help to identify the critical program points, but we leave it as a future work because the overall difference is small.

Figure 9 (b) shows the sensitivity of *ELF++* to the threshold for instruction prefetch, where an instruction is prefetched only when the number of occupied MSHRs in both L1I and L1D is below the threshold. Instruction prefetch is disabled when the threshold is zero. Performance is improved by an average of 6.7%, 10.8%, 11.2%, 11.4%, 11.9%, 11.9%, and 11.9% when the threshold is 0, 4, 8, 12, 16, 20, and 24, respectively. As discussed in Section 3.3, blindly issuing instruction prefetch can increase the memory contention,

| Structure | Storage per Entry | # Entries | Total |
|---|---|---|---|
| Priority Calculator | 42-bit | 32 | 0.16 kB |
| Warp Priority Table | 8-bit | 48 | 0.05 kB |
| NewCAR | 301-bit | 32 | 1.18 kB |

**Table 3: Hardware overhead per SM from additional structures.**

which can degrade the overall performance. Therefore, performance is expected to improve until a certain threshold, where instruction prefetch provides benefit without congesting the memory too much. In the figure, the curve has the maximum improvement at threshold of 16 although the performance degradation is small afterwards. Looking at the performance of individual benchmarks, we found out that a subset of benchmarks like MUM, which lose performance with a larger threshold, only shows small performance degradation. Benchmarks like FDTD, LC, and LUD, which obtain the most benefit from instruction prefetch, were not negatively impacted by the increased threshold. Nevertheless, *ELF++* chooses to use 16 as the threshold in our experiments because the performance improvement is maximized at the point although the difference is small.

## 4.2 Hardware Overhead

Table 3 shows the hardware overhead of *ELF++* per SM. Priority calculator has 32 program points, where each program point requires 1-bit for the valid field, 1-bit for the branch field, 32-bit for the PC, and 8-bit for the score. The WPT requires 8-bit per warp for the priority. GTX480 can have up to 48 warps per SM. NewCAR requires the same overhead as the CAR [28]. In total, *ELF++* only consumes 1.39kB extra storage space per SM.

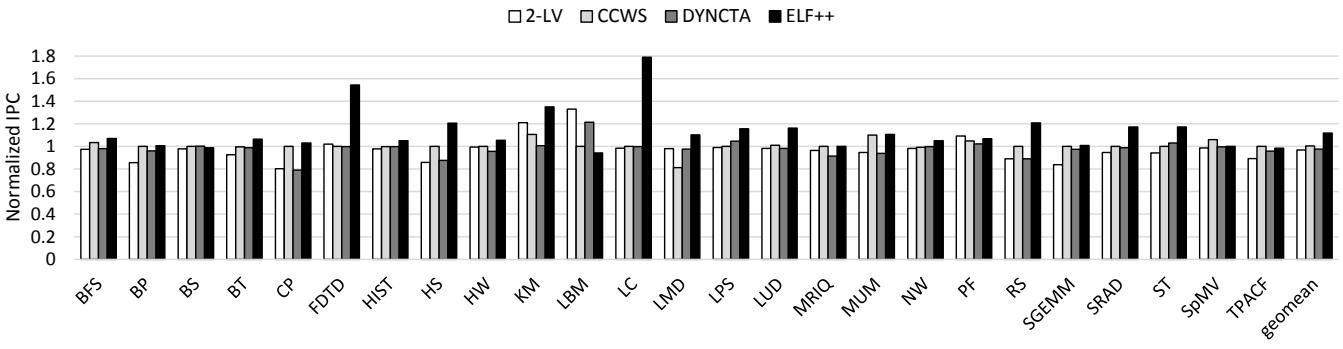



**Figure 10: Comparison of *ELF++* with prior works. GTO is the baseline.**

## 4.3 Comparison to Prior Works

Figure 10 compares the performance improvement of *ELF++* over GTO with three prior works: 2-LV [17], CCWS [23], and DYNCTA [10]. On average, 2-LV, CCWS, DYNCTA and *ELF ++* improves -3.2%, 0.1%, -2.3%, and 11.9% over the baseline GTO. With the wide range of applications, 2-LV and DYNCTA perform worse than the GTO. For example, BP, CP, HS, and RS show noticeable slowdown in 2-LV and DYNCTA. On the other hand, *ELF++* always shows either similar or better performance compared to the GTO.

2-LV and DYNCTA perform slightly worse than GTO on average. The only difference between 2-LV and GTO is that 2-LV continuously gives higher priority to the newer group until it issues all the ready instructions while GTO switches back to the older group (thread block) more quickly. Therefore, 2-LV and GTO provide similar performance on average. In DYNCTA, fewer thread blocks are scheduled to an SM to reduce the memory contention. Because warps are scheduled to an SM in a thread block granularity, GTO will schedule the warps in the oldest thread block more frequently than the other warps achieving similar effect to DYNCTA. As a result, DYNCTA performs similar to GTO.

CCWS reduces cache thrashing by limiting the number of warps that can send out new memory requests. Therefore, CCWS provides performance improvement over for cache-sensitive benchmark like KM, PF, and SpMV. However, in general, there are more benchmarks that are not cache-sensitive. As a result, CCWS provides similar performance to GTO on average.

In LBM, 2-LV and DYNCTA performs better than GTO while *ELF++* performs slightly worse than GTO. LBM has a lot of memory resource conflicts all the time because twenty memory loads are clustered back-to-back at the beginning of the program. These memory loads show row locality on the DRAM side when the warps within a thread block issue the same memory instruction. However, GTO and *ELF++* will prioritize one warp until it issues all the twenty memory loads, which takes more than half of the MSHRs. As a result, GTO and *ELF++* do not perform well as they lose the row locality on the DRAM side. In fact, any warp scheduler that round-robins within a thread block performs better than the warp schedulers that prioritize a warp in greedy fashion. For example, LRR scheduler achieves similar performance to 2-LV and DYNCTA for LBM.

Figure 11 compares *ELF++* with prior works when cache configuration is different. GTX480 can be adjusted to have either 16kB L1D or 48kB L1D. As shown by the figure,

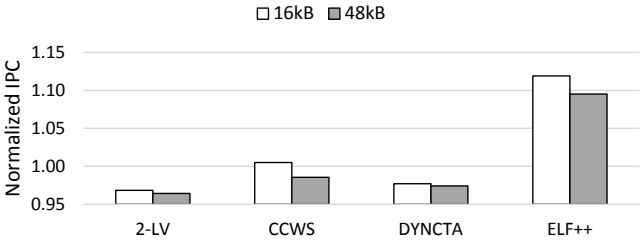



**Figure 11: Comparing *ELF++* to prior works when different cache configurations are used. GTO is the baseline.**

all the techniques have slightly less improvement compared to the baseline GTO with 48kB L1D. On average, normalized IPC equals to 0.964, 0.986, 0.974, and 1.095 for 2-LV, CCWS, DYNCTA, and *ELF++* with 48kB L1D. This is because memory requests are likely to have more L1D cache hits with a larger L1D as it can retain more data. Consequently, the average memory latency is reduced with the larger L1D. *ELF++* improves over GTO and other prior works in both cache configurations because it can issue memory requests faster than other schedulers, which can hide miss latencies better. As a result, *ELF++* overlaps more long latency memory operations with each other, which reduces the number of memory-related stalls more than other schedulers.

# 5. RELATED WORK

Since the emergence of GPUs as a general-purpose parallel computing platform, a significant number of studies have been proposed to improve GPU performance. We first review prior works that exploited memory-level parallelism in various platforms, and then summarize GPU-specific ideas that enhance GPU performance with alternative warp or thread block scheduling.

## 5.1 Memory-Level Parallelism

Memory-level parallelism (MLP) was recognized as one of the key objectives in computer architecture since CPUs' performance overwhelmed the memory performance [6]. Pai and Adve [21] applied code transformations to cluster more read misses within the same instruction window of an out-of-order processor. They also measured the improved MLP by measuring MSHR utilizations. Many other works [30, 15] also used the number of occupied MSHRs or stall cycles

due to full MSHRs to measure MLP. Zhou and Conte [32] used value prediction techniques to increase MLP by parallelizing loads that were sequentially dependent. MLP-aware replacement [22] notes that an out-of-order processor can exploit more MLP if the LLC misses occur in parallel. In that sense, it tries to remove isolated misses in the LLC. Chou et al. [3] showed that runahead execution [4, 16], effective instruction prefetching, and accurate branch prediction can improve MLP, which in turn enhances the overall performance. The challenge of achieving high MLP on GPUs is different from the CPUs because latency hiding on GPUs is done by fast context switch between warps, where inter-warp data locality is scarce.

Some prior works have studied data prefetching or direct memory access (DMA) on GPUs, which essentially improves MLP on GPUs. MT-prefetching [13] prefetches data for other threads rather than for itself. Also, it shows that throttling mechanism may be needed for prefetching when prefetching is harmful because of low accuracy. APOGEE [27] introduces timely prefetching by adjusting the distance of prefetching. APOGEE mainly focuses on improving energy efficiency by reducing the number of thread contexts to hide the memory latency with prefetching. $D^2MA$ [7] achieves MLP on GPUs by transferring data from the global memory to the shared memory behind the scenes. *ELF* also tries to maximize memory-level parallelism, but modifies the warp and fetch scheduler to achieve the goal, which includes instruction prefetching.

## 5.2 GPU Scheduling

Early works have noticed that it is better to prioritize a group of warps rather than giving equal priority to all the warps. Two-level warp scheduling [17] divides warps into groups, where the scheduling algorithm is re-structured into scheduling warps within a group and scheduling between the groups. Two-level warp scheduler only schedules from another group when all the warps within one group are blocked by long latency operations. Gebhart et al. [5] also utilize the two-level scheduling for energy-efficiency.

Some of the works have focused on utilizing warp scheduling to reduce cache contention. CCWS [23] observes that intra-warp locality is dominant in the L1 data cache for GPUs. By monitoring the L1 cache behavior, CCWS only allows a subset of active warps to issue memory requests to exploit the intra-warp locality in the L1 data cache. DAWS [24] also uses the idea of throttling a subset of warps from issuing requests, however, DAWS is also aware of memory divergence, which occurs when memory requests from threads within a warp cannot be coalesced. DAWS predicts a L1 cache footprint for each warp either by profiling or runtime sampling, and uses the information to find the number of warps to be throttled. OWL [9] proposes four schemes, which include thread block aware warp scheduling, locality aware scheduling, bank-level parallelism aware scheduling, and opportunistic prefetching.

Some other works have looked at controlling the number of scheduled thread blocks. DYNCTA [10] shows that issuing maximum number of thread blocks to an SM is not always beneficial because memory-intensive applications create cache contention. By looking at how many cycles are stalled by memory, DYNCTA tries to predict the optimal number of thread blocks to be scheduled. However, the choice of the thresholds is dependent on micro-architecture. LCS [14] notes that GPUs are designed to hide latencies by leveraging the thread-level parallelism (TLP). In that sense, the optimum number of thread blocks to be scheduled on an SM is the number of thread blocks that can hide latencies until one thread block finishes. LCS monitors the number of instructions issued for each thread block during an execution of a single thread block, and calculates the number of thread blocks to be scheduled.

More recent works have focused on prioritizing memory requests from one warp, and explored a possibility of doing useful works when the load/store unit is stalled. MRPB [8] prioritizes memory accesses from the same warp to preserve locality in the L1 data cache. MRPB also notes that an associativity stall may occur because of the allocate-on-miss policy on the L1 cache. Even if MSHRs and miss queues are available, the memory request cannot be accepted because there is no more way to allocate. In such cases, MRPB bypasses the L1 cache. Mascar [28] schedules the warps with outstanding memory requests first when memory resources are saturated. Mascar also utilizes cache access re-execution to enable hit-under-miss.

*ELF* shares the basic idea that the performance in GPUs can be improved by carefully scheduling warps. However, *ELF* primarily differs with prior works by giving higher priority to the warps with sooner memory operations. Also, *ELF* shows that reducing memory conflicts and fetch stalls can provide additional improvement.

# 6. CONCLUSION

In this paper, we presented *ELF*, a GPU scheduling mechanism that maximizes the MLP as fast as possible by scheduling a warp with an earliest memory load first. In order to achieve the goal, *ELF* utilizes both compiler and hardware to give higher priority to the warps that have fewer remaining instructions to the next memory load operation. We also showed that the interplay between the warp scheduler and the fetch scheduler is important for *ELF*: the fetch unit should prioritize according to the issue priorities. Moreover, we identified two cases when *ELF* could improve the performance further by explaining when *ELF* is not fully achieving its goal. First, memory conflicts can block new memory requests. We introduced an extended version of cache access re-execution (NewCAR) to reduce memory conflicts and thus increase the MLP. Second, fetch stalls can block higher priority warps from making progress because they are waiting for the instructions. We showed that a simple next line prefetcher for the L1I cache is sufficient to reduce most of the fetch stalls. Evaluations show that *ELF* can improve the performance by 4.1% over the greedy-then-oldest (GTO) scheduler with only 1.39kB extra storage per SM. When used with other techniques like NewCAR and instruction prefetching, *ELF* can achieve total speedup of 11.9% over the GTO.

# 7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers as well as the fellow members of the CCCP research group for their valuable comments and feedbacks. This work is supported in part by the National Science Foundation under grant SHF-1217917 and by the Defense Advanced Research Projects Agency (DARPA) under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program.

# 8. REFERENCES


[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[3] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 76–87, June 2004.

[4] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of the 1998 International Conference on Supercomputing*, pages 68–75, 1997.

[5] 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 *Proc. of the 38th Annual International Symposium on Computer Architecture*, pages 235–246, 2011.

[6] A. Glew. MLP yes! ILP no!, 1998. In ASPLOS Wild and Crazy Idea Session'98.

[7] D. A. Jamshidi, M. Samadi, and S. Mahlke. D$^2$MA: Accelerating coarse-grained data transfer for GPUs. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 431–442, 2014.

[8] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 272–283, Feb. 2014.

[9] A. Jog, O. Kayiran, N. C. 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 *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–406, Mar. 2013.

[10] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, 2013.

[11] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010.

[12] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on GPU performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, pages 1–10, 2010.

[13] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, pages 213–224, 2010.

[14] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 260–271, Feb. 2014.

[15] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 317–328, 2012.

[16] 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 *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, Feb. 2003.

[17] 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 *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 308–317, 2011.

[18] NVIDIA. GPU Computing SDK. http://developer.nvidia.com/gpu-computing-sdk.

[19] NVIDIA. Fermi: Nvidiaś next generation CUDA compute architecture, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[20] NVIDIA. *CUDA C Programming Guide*, May 2011.

[21] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 147–155, Nov. 1999.

[22] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 167–178, June 2006.

[23] T. G. Rogers, M. OĆonnor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 72–83, 2012.

[24] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 99–110, 2013.

[25] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 13–22, 2012.

[26] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 440–451, 2012.

[27] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, 2013.

[28] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proc. of the 21st International Symposium on High-Performance Computer Architecture*, pages 174–185, Feb. 2015.

[29] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, 1992.

[30] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ilp processors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 380–391, June 1998.

[31] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Mar. 2012.

[32] H. Zhou and T. M. Conte. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Transactions on Computers*, 54(7):897–912, 2005.