Divergence-Aware Warp Scheduling

Timothy G. Rogers
Department of Computer and Electrical Engineering
University of British Columbia
tgrogers@ece.ubc.ca

Mike O'Connor
NVIDIA Research
moconnor@nvidia.com

Tor M. Aamodt
Department of Computer and Electrical Engineering
University of British Columbia
aamodt@ece.ubc.ca

ABSTRACT

This paper uses hardware thread scheduling to improve the performance and energy efficiency of divergent applications on GPUs. We propose Divergence-Aware Warp Scheduling (DAWS), which introduces a divergence-based cache footprint predictor to estimate how much L1 data cache capacity is needed to capture intra-warp locality in loops. Predictor estimates are created from an online characterization of memory divergence and runtime information about the level of control flow divergence in warps. Unlike prior work on Cache-Conscious Wavefront Scheduling, which makes reactive scheduling decisions based on detected cache thrashing, DAWS makes proactive scheduling decisions based on cache usage predictions. DAWS uses these predictions to schedule warps such that data reused by active scalar threads is unlikely to exceed the capacity of the L1 data cache. DAWS attempts to shift the burden of locality management from software to hardware, increasing the performance of simpler and more portable code on the GPU. We compare the execution time of two Sparse Matrix Vector Multiply implementations and show that DAWS is able to run a simple, divergent version within 4% of a performance optimized version that has been rewritten to make use of the on-chip scratchpad and have less memory divergence. We show that DAWS achieves a harmonic mean 26% performance improvement over Cache-Conscious Wavefront Scheduling on a diverse selection of highly cache-sensitive applications, with minimal additional hardware.

Categories and Subject Descriptors

C.1.4 [**Computer System Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Design, Performance

Keywords

GPU, Caches, Scheduling, Divergence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MICRO-46, December 7-11, 2013, Davis, CA, USA.
Copyright 2013 ACM 978-1-4503-2638-4/13/12...$15.00.
http://dx.doi.org/10.1145/2540708.2540718

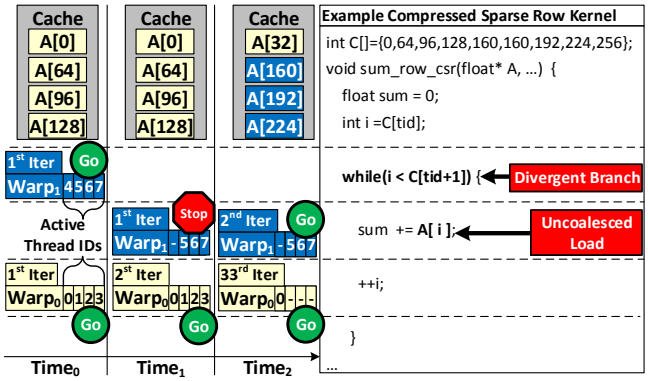

Figure 1: DAWS example. Cache: 4 entries, 128B lines, fully assoc. By $Time_0$, warp 0 has entered loop and loaded 4 lines into cache. By $Time_1$, warp 0 has captured spatial locality, DAWS measures footprint. Warp 1 is prevented from scheduling as DAWS predicts it will oversubscribe cache. By $Time_2$, warp 0 has accessed 4 lines for 32 iterations and loaded 1 new line. 3 lanes have exited loop, decreasing footprint. Warp 1 and warp 0 are allowed to capture spatial locality together.

1. INTRODUCTION

In the face of diminished voltage scaling [10], massively multithreaded programmable accelerators, like *Graphics Processing Units* (GPUs), can potentially make more efficient use of the available chip power budget. GPUs gain some of their efficiency by utilizing a *Single Instruction Multiple Thread* (SIMT) [27] execution model. SIMT execution groups a collection of threads into a warp (or wavefront) to run them on the hardware's *Single Instruction Multiple Data* (SIMD) cores. SIMT execution can be very efficient on highly regular code where control flow and memory accesses can be predicted by the programmer, who can restructure their algorithm to make it more efficient on a GPU, if necessary. However, restructuring highly irregular applications can be difficult or impossible without completely redesigning the software. Moreover, porting an existing piece of parallel code to an accelerator and having it run efficiently is a challenge [34]. Running irregular code on a GPU can cause both memory and control flow divergence. Memory divergence (or an uncoalesced memory access) occurs when threads in the same warp access different regions of memory in the same SIMT instruction. Control flow (or branch) divergence occurs when threads in the same warp execute different control flow paths. This work focuses on improving the performance of several such irregular applications through warp scheduling.

Figure 1 presents a small example of divergent code to illustrate how scheduling can be used can make effective use of on-chip cache capacity. The example code sums each row of a *Compressed*

Sparse Row (CSR) [5] data set. Each thread in the kernel sums one row using a loop. This code is divergent due to the data dependent nature of each sparse row's length and the position of each row's values in memory. This translates into branch divergence when threads within a warp travel through the loop a different number of times and memory divergence when threads access $A[i]$. This code has three key characteristics that can be leveraged to make effective use of cache capacity: (1) Each thread has spatial locality across loop iterations, since i is incremented by 1. (2) Each warp's load to $A[i]$ can access multiple cache lines. (3) The number of cache lines accessed when a warp loads $A[i]$ is dependent on the warp's active mask. Figure 1 also illustrates how our proposed *Divergence-Aware Warp Scheduling* (DAWS) technique takes these characteristics into account to maximize on-chip cache utilization. In the example, two warps (each with 4 threads) share a cache with 4 entries. Warp 0 enters the loop first and each of its threads loads its section of A into the cache. During warp 0's execution of the loop, Divergence-Aware Warp Scheduling learns that there is both locality and memory divergence in the code. At Time_1, warp 1 is ready to enter the loop body. Divergence-Aware Warp Scheduling uses the information gathered from warp 0 to predict that the data loaded by warp 1's active threads will evict data reused by warp 0 which is still in the loop. To avoid oversubscribing the cache, Divergence-Aware Warp Scheduling prevents warp 1 from entering the loop by de-scheduling it. Now warp 0 captures its spatial locality in isolation until its threads begin to diverge. By Time_2, warp 0 has only one thread active and its cache footprint has decreased. Divergence-Aware Warp Scheduling detects this divergence and allows warp 1 to proceed since the aggregate footprint of warp 0 and warp 1 fits in cache.

The code in Figure 1 contains intra-warp locality. Intra-warp locality occurs when data is loaded then re-referenced by the same warp [33]. The programmer may be able to re-write the code in Figure 1 to remove intra-warp locality. Hong et al. [16] perform such an optimization to *Breadth First Search* (BFS). However, this can require considerable programmer effort. Another option is to have the compiler restructure the code independent of the programmer, however static compiler techniques to re-arrange program behaviour are difficult in the presence of data dependant accesses [35]. One of this paper's goals is to enable the efficient execution of more workloads on accelerator architectures. We seek to decrease the programmer effort and knowledge required to use the hardware effectively, while adding little to the hardware's cost.

Previously proposed work on *Cache-Conscious Wavefront Scheduling* (CCWS) [33] uses a reactionary mechanism to scale back the number of warps sharing the cache when thrashing is detected. However, Figure 1 illustrates that cache footprints in loops can be predicted, allowing thread scheduling decisions to be made in a proactive manner. Our technique reacts to changes in thread activity without waiting for cache thrashing to occur. By taking advantage of dynamic thread activity information, Divergence-Aware Warp Scheduling is also able to outperform a scheduler that statically limits the number of warps run based on previous profiling runs of the same workload [33].

This paper makes the following contributions:

- It quantifies the relationship between data locality, branch divergence and memory divergence in GPUs on a set of economically important, highly cache-sensitive workloads.
- It demonstrates that code regions can be classified by both data locality and memory divergence.
- It demonstrates, with an example, that DAWS enables unoptimized GPU code written in a scalar fashion to attain 96% of the performance of optimized code that has been re-written for GPU acceleration.
- It proposes a novel *Divergence-Aware Warp Scheduling* (DAWS) mechanism which classifies static load instructions based on their memory usage. It uses this information, in combination with the control flow mask, to appropriately limit the number of scalar threads executing code regions with data locality. DAWS achieves a harmonic mean 26% speedup over Cache-Conscious Wavefront Scheduling [33] and 5% improvement over a profile-based warp limiting solution [33] with negligible area increase over CCWS and only 0.17% more area than simple warp schedulers.

This work focuses on a set of GPU accelerated workloads from server computing and high performance computing that are both economically important and whose performance is highly sensitive to *level one data* L1D cache capacity. These workloads encompass a number of applications from server computing such as Memcached [15], a key-value store application used by companies like Facebook and Twitter, and a sparse matrix vector multiply application [9] which is used in Big Data processing.

2. A PROGRAMMABILITY CASE STUDY

This section presents a case study using two implementations of Sparse Matrix Vector Multiply (SPMV) from the SHOC benchmark suite [9] [1]. This case study is chosen because it is a real example of code that has been ported to the GPU then optimized. Example 1 presents SPMV-Scalar which is written such that each scalar thread processes one row of the sparse matrix. This is similar to how the algorithm might be implemented on a multi-threaded CPU. The bold code in SPMV-Scalar highlights its divergence issues. Example 2 shows SPMV-Vector which has been optimized for performance on the GPU. Both pieces of code generate the same result and employ the same data structure. The bold code in SPMV-Vector highlights the added complexity introduced by GPU-specific optimizations.

One goal of this work is to enable less optimized code such as Example 1 to achieve performance similar to the optimized code in Example 2. In SPMV-Scalar, the accesses to $cols[j]$ and $val[j]$ will have significant memory divergence and the data-dependent loop bounds will create branch divergence. Like the code in Figure 1, SPMV-Scalar has spatial locality within each thread since j is incremented by one each iteration. Divergence-Aware Warp Scheduling seeks to capture this locality.

In the SPMV-Vector version each warp processes one row of the sparse matrix. Restructuring the code in this way removes much of the memory divergence present in the scalar version since the accesses to $cols[j]$ and $val[j]$ will have spatial locality across each SIMT instruction. However, this version of the code forces the programmer to reason about warp length, the size of on-chip shared memory, and it requires a parallel reduction of partial sums to be performed for each warp. In addition to writing and debugging the additional code required for SPMV-Vector, the programmer must tune thread block sizes based on which machine the code is run on. Even if the programmer performed all these optimizations correctly, there is no guarantee that SPMV-Vector will outperform SPMV-Scalar since the shape and size of the input matrix may render the optimizations ineffective. Previous work has shown that sparse matrices with less non-zero elements per row than the GPU's

[1]For brevity, some keywords in the original version of Examples 1 and 2 were removed. All of our experiments are run without modifying the original kernel code.

Example 1 Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * blockDim.x
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Loads
            int col = cols[j];
            t += val[j] * vecTexReader(col);

        }
        out[myRow] = t;
    }
}
```

Example 2 GPU-Optimized SPMV-Vector Kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;
    int myRow = (blockIdx.x * warpsPerBlock)
              + (t / warpSize);
    texReader vecTexReader;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

    if (myRow < dim)
    {
        int warpStart = rowDelimiters[myRow];
        int warpEnd = rowDelimiters[myRow+1];
        float mySum = 0;
        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
        if (id <  2)
            partialSums[t] += partialSums[t+ 2];
        if (id <  1)
            partialSums[t] += partialSums[t+ 1];

        // Write result
        if (id == 0)
        {
            out[myRow] = partialSums[t];
        }
    }
}
```

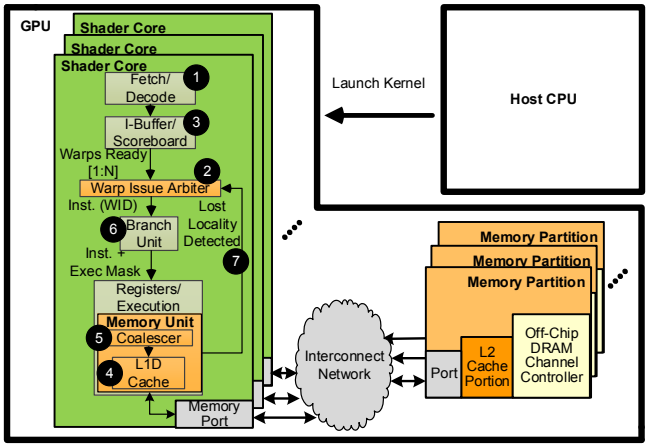

Figure 2: Overview of our baseline GPGPU-Sim pipeline.

warp width do not take advantage of the potential increase in coalesced accesses offered by SPMV-Vector [6].

This reliance on per-machine tuning and the unpredictability of manual optimization techniques can make programming GPUs difficult. In Section 7.2 we demonstrate that Divergence-Aware Warp Scheduling allows the programmer to write the simpler, more portable SPMV-Scalar while still capturing almost all of the performance benefit of SPMV-Vector.

This case study should not be construed to suggest that Divergence-Aware Warp Scheduling can replicate the performance of any hand tuned optimization or generally solve the performance issues surrounding divergence on GPUs. The study is presented as one real world example of optimized GPU code to demonstrate how intelligent warp scheduling can capture almost as much locality as this particular hand tuned implementation.

3. BASELINE ARCHITECTURE

Figure 2 presents the GPU-like accelerator architecture we study in this paper. The applications we study are written in CUDA [1] or OpenCL [23]. These programs begin their execution on a host CPU which assigns kernels of work to the GPU. Scalar threads are grouped together by the application developer into thread blocks. Thread blocks can communicate via a shared on-chip scratchpad memory. Thread blocks are subdivided by the hardware into warps. Threads within the same warp execute in lock-step. Our baseline architecture assigns thread blocks to each shader core until the core's static resources (shared memory scratchpad, register space) are exhausted. Our shader cores are able to schedule warps from multiple thread blocks concurrently.

3.1 Core Microarchitecture and Issue Arbitration

Figure 2 also illustrates the microarchitecture of our baseline. The pipeline has decoupled fetch and issue stages. The fetch (❶ in Figure 2) and issue (❷) stages communicate via an instruction buffer (❸). Each warp has dedicated slots in the instruction buffer. The fetch stage updates a valid bit for each instruction, which indicates if the instruction has been filled. Each buffer entry also contains a ready bit which is updated by an in-order scoreboard and informs the *Warp Issue Arbiter* (WIA) if this instruction is ready to issue. The purpose of the WIA is to select which of the multiple ready warps is issued next. Our paper focuses on this decision.

3.2 Memory Stage and Branch Unit

There are several memory space classes in GPUs. This work focuses on global and local memory which is cached in the core by the L1D cache (❹). Issuing a memory instruction from a single warp can generate up to W data cache accesses where W is the number of threads per warp. Our baseline attempts to reduce the number of memory accesses generated from each warp by coalescing (❺) the accesses of its lanes. Coalescing reduces the number of memory requests by merging accesses from multiple lanes into cache line sized chunks when there is spatial locality across the warp [1]. These coalesced requests are sent to the L1D cache. Memory instructions whose lanes all touch the same cache line can generate as little as one memory access. Memory divergence occurs when coalescing fails to reduce the number of memory requests generated by an instruction to two or less. The GPU's L1D caches are not coherent, but they evict global data on writes and reserve cache lines on misses.

Our work uses information from the branch unit (❻) to make warp scheduling decisions. When scalar threads within the same warp take different control flow paths, only one path can be executed at a time due to the SIMT nature of the hardware. Our baseline handles control flow divergence by clearing the execution mask of threads not executing at the current *Program Counter* (PC) and handles re-convergence with a *post dominator* (PDOM) re-convergence stack described by Fung et al. [12].

The GPGPU-Sim Manual [2] describes the pipeline and all the components of the baseline architecture in more detail.

3.3 Previously Proposed Warp Limiting Techniques

In this paper, we compare Divergence-Aware Warp Scheduling against *Cache Conscious Wavefront Scheduling* (CCWS) [33]. CCWS is a dynamic warp throttling mechanism that reacts to feedback from the memory system (❼). CCWS limits the number of warps allowed to issue memory instructions when it detects the Warp Issue Arbiter is issuing loads from so many warps that locality private to one warp is not being captured by the cache. This loss of intra-warp locality is detected using L1D cache victim tags private to each warp. These victim tags are used to determine if a cache miss might have been a hit had the warp that missed been given more exclusive access to the L1D cache. Warps losing the most locality are prioritized by de-scheduling warps that have lost the least locality. CCWS backs off warp exclusivity over time, as long as no lost locality is detected.

We also compare against a *Static Warp Limiting* (SWL) [33] mechanism that limits the number of warps interleaved to a static value set when the kernel is launched. Specifically we compare against Best-SWL which uses profiling to tailor the warp limit to the best value for a given workload.

4. DIVERGENCE, LOCALITY AND SCHEDULING

A key observation of our work is that a program's memory divergence, control flow divergence and locality can be profiled, predicted and used by the warp scheduler to improve cache utilization. This section is devoted to describing this observation in detail and is divided into two parts. Section 4.1 explores where locality occurs in our highly cache-sensitive benchmarks and Section 4.2 classifies the locality in ways that are useful for our warp scheduler.

4.1 Application Locality

Figure 3 presents the hits and misses for all the static load instruction addresses (PCs) in our highly cache-sensitive benchmarks (described in Section 6). Each hit is classified as either an intra-warp hit (when data is loaded then re-referenced by the same warp) or an inter-warp hit (when one warp loads data that is hit on by another). This data was collected using Cache-Conscious Wavefront Scheduling. The loops in each program are highlighted by dashed boxes. This figure demonstrates that the bulk of the locality in our programs is intra-warp and comes from a few static load instructions. These load instructions are concentrated in the loops of the program.

To understand the locality in these loops, Figure 4 presents a classification of intra-warp hits from loads within the loops of each application. Loads are classified as *Accessed-This-Trip* if the cache line was accessed by another load on this loop iteration. If the value in cache was not *Accessed-This-Trip*, then we test if it was accessed on the previous loop trip. If so, it is classified as *Accessed-Last-Trip*. If the line was not accessed on either loop trip, it is classified as *Other*, indicating that the line was accessed outside the loop or in a loop trip less recent than the last one. This data demonstrates that the majority of data reuse in these applications is *Accessed-Last-Tip*. If the scheduler can keep the data loaded by a warp on one loop iteration in cache long enough to be hit on in the next loop iteration, most of the locality in these applications can be captured.

To illustrate the source of this locality in the code, consider the code for SPMV-Scalar in Example 1. Figure 4 indicates that all of the intra-warp locality within the loop of this code is *Accessed-Last-Trip*. This comes from the loading $cols[j]$ and $val[j]$. When inside this loop, each thread walks the arrays in 4 byte strides since j is incremented by one each iteration.

Based on these observations, we design our scheduling system to ensure that when intra-warp locality occurs in a loop, much of the data loaded by a particular warp in one iteration remains in the cache for the next iteration. We attempt to ensure this happens by creating a cache footprint prediction for warps executing in loops. The prediction is created from information about the loads inside the loop and the current level of control flow divergence in a warp on its current loop iteration.

4.2 Static Load Classification

To predict the amount of data each warp will access on each iteration of the loop, we start by classifying the static load instructions inside the loop. We classify each static load instruction based on two criteria, memory divergence (detailed in Section 4.2.1) and loop trip repetition (Section 4.2.2).

4.2.1 Memory Divergence

If the number of memory accesses generated by a load equals the number of lanes active in the warp that issues it, then the load is completely diverged. Loads that generate one or two accesses no matter how many threads are active are completely converged. Anything in between is somewhat diverged. To understand the relationship between memory divergence and static instructions, consider Figure 5. Figure 5 plots the number of threads active and accesses generated for every dynamic load instruction in BFS, grouped by the load instruction's PC. This figure illustrates that memory divergence behaviour can be characterized on a per-PC basis. Some PCs are always converged (328, 400 and 408 in Figure 5), some are almost always completely diverged (272) and others are consistently somewhat diverged (240). This result is consistent across all the highly cache-sensitive applications we studied. For simplicity, DAWS classifies each static load instruction that is not consistently converged as diverged.

This figure also demonstrates that there is a significant amount of control flow divergence in this application. This control flow diver-

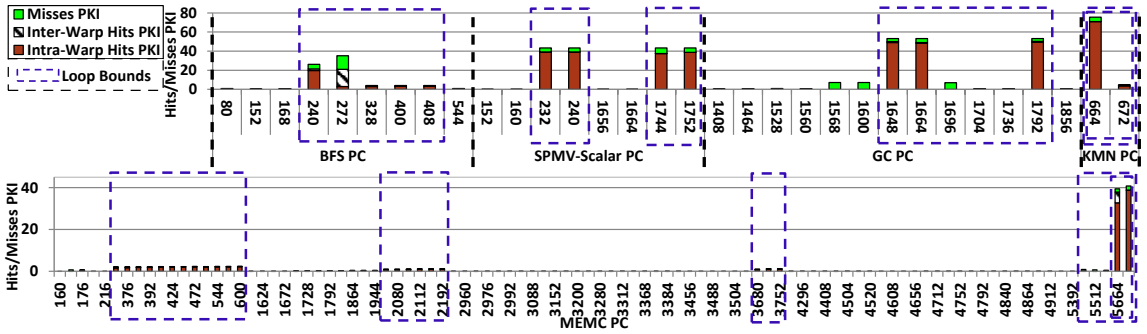

Figure 3: Intra-warp hits, inter-warps hits and misses *per thousand instructions* (PKI) for all the static load instructions in each of our highly cache-sensitive benchmarks, identified by PC. The PCs contained in loops are highlighted in dashed boxes.

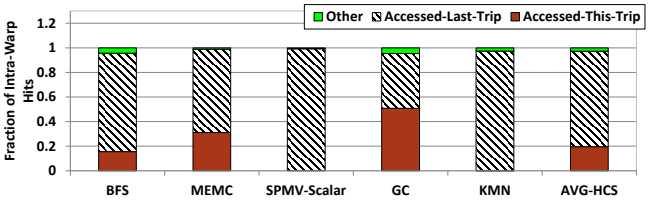

Figure 4: Classification of intra-warp hits within loops using an 8M L1D cache. Accessed-This-Trip=hit on data already accessed this loop iteration. Accessed-Last-Trip=hit on data accessed in immediately-previous loop iteration.

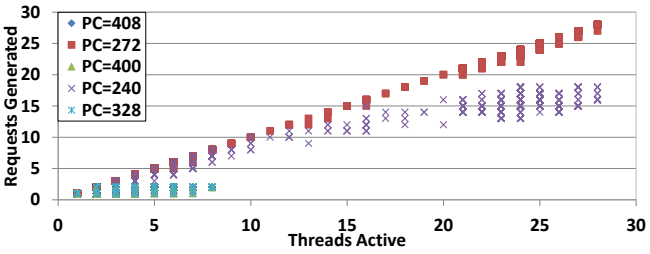

Figure 5: Number of threads active and number memory accesses generated for each dynamic load in BFS's loop. Accesses are grouped by PC.

gence makes a solution that statically limits the number of warps when the kernel is launched [33] suboptimal, since it does not adapt to thread activity as the program executes. Some of the static loads in Figure 5 never have more than 8 threads active (for example, PC 328). These loads occur inside a branch within the loop and are only generated in BFS when a thread is processing a node with an unexplored edge.

Additionally, all 32 threads are never active in this loop due to branch divergence occurring prior to loop execution. The loop is only executed if a node is on the program's exploration frontier, which can be relatively sparsely distributed across threads. This illustrates that there is an opportunity to improve the estimated cache footprint for a loop by taking advantage of branch prediction. However, for the cache footprint prediction generated by DAWS, we assume the worst possible case (i.e., all of the loads in the loop get uncovered by all threads active on this loop iteration). Exploring branch prediction is beyond the scope of this work.

4.2.2 Loop Trip Repetition

Multiple static loads within one loop-trip may reference the same cache line. The *Accessed-This-Trip* values in Figure 4 demonstrate this can be significant. These loads do not increase the cache footprint because the data they access has already been accounted for by another load. We introduce the concept of a repetition ID to filter them out. All loads predicted to reference the same cache line are assigned the same repetition ID. When predicting the cache footprint of a loop, only one load from each repetition ID is counted. Classification the repetition ID is done either by the compiler (predicting that small offsets from the same pointer are in the same line) or by hardware (described in Section 5.2.2).

5. DIVERGENCE-AWARE WARP SCHEDULING (DAWS)

The goal of DAWS is to keep data in cache that is reused by warps executing in loops so that accesses from successive loop iterations will hit. DAWS does this by first creating a cache-footprint prediction for each warp. Then, DAWS only allows load instructions to be issued from warps whose aggregate cache footprints are predicted to be captured by the L1D cache.

Figure 6 illustrates how DAWS works at a high level. A prediction of the cache footprint for each warp is created. These predictions are summed to create a total cache footprint. At time T_0, all warps have no predicted footprint. Warps that enter loops with locality are assigned a prediction and consume a portion of the estimated available cache. When a warp exits the loop its predicted footprint is cleared. When the addition of a warp's prediction to the total cache footprint exceeds the effective cache size, that warp is prevented from issuing loads. The value of the effective cache size is discussed later in Section 5.1. To illustrate DAWS in operation, consider what happens at each time-step in Figure 6. Between time T_0 and T_1, warp 0 enters a loop. From a previous code characterization, DAWS has predicted that this loop has intra-warp locality and one divergent load. Sections 5.1 and 5.2 present two variations of DAWS that perform this code characterization in different ways. Warp 0's active mask is used to predict that warp 0 will access 32 cache lines (one for each active lane) in this iteration of the loop. The value of the footprint prediction for more complex loops is discussed in detail in Section 5.1.1. Between time T_1 and T_2, warp 1 enters the loop with only 16 active threads and receives a smaller predicted footprint of 16. Between T_2 and T_3, warp 2 reaches the loop. The addition of Warp 2's predicted cache footprint to the current total cache footprint exceeds the effective cache size, therefore warp 2 is prevented from issuing any loads. Between T_3 and T_4, 16 of warp 0's 32 threads have left the loop (causing control flow divergence) which frees some predicted cache capacity, allowing warp 2 to issue loads again.

The DAWS warp throttling mechanism is somewhat similar to the lost locality scoring system presented in CCWS [33], however there are several key differences. In CCWS, scores are assigned based on detected lost locality. Warps losing the most locality are given more exclusive cache access by preventing warps losing the least locality from issuing loads. CCWS is a *reactive* system that has to lose locality before trying to preserve it. DAWS is a *proactive*

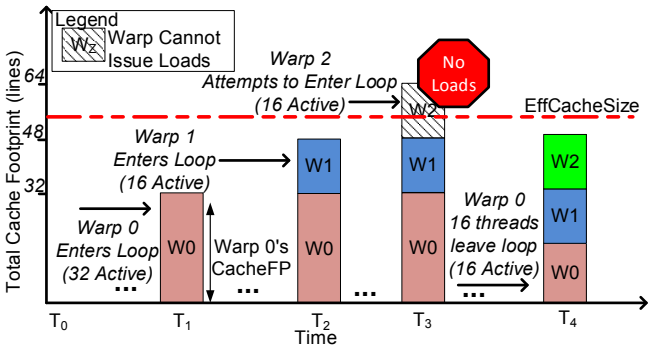

Figure 6: High level view of how DAWS's cache footprint prediction mechanism dynamically throttles the number of threads sharing the cache. *CacheFP=Cache Footprint*

system that tries to prevent lost locality before it happens. DAWS is also proactive in decreasing the level of thread throttling. As threads within warps progress through a loop a different number of times, the data accessed by their divergent loads is reduced causing DAWS to decrease their predicted cache footprint. DAWS takes this control flow divergence into account immediately and scales up the number of warps allowed to issue load instructions as appropriate. In contrast, CCWS scales back thread throttling by a constant factor each cycle, unless more lost locality is detected. In CCWS, when warps with the most exclusive cache access stop losing locality, their exclusivity is lost and they have to start missing again to get it back. DAWS ensures that all warps in loops with intra-warp locality do not lose their cache exclusivity until they exit the loop.

Figure 7 presents the microarchitecture required to implement our two proposed Divergence-Aware Scheduling Techniques. Section 5.1 details *Profiled Divergence-Aware Warp Scheduling* (Profiled-DAWS), which uses off-line profiling to characterize memory divergence and locality. Section 5.2 presents *Detected Divergence-Aware Warp Scheduling* (Detected-DAWS), which detects both locality and memory divergence as the program executes. Both techniques make use of feedback from the branch unit (Ⓐ in Figure 7) which tells the Warp Issue Arbiter the number of active lanes for any given warp. Detected-DAWS is implemented on top of Profiled-DAWS. In Detected-DAWS, locality and memory divergence information is detected as the program runs based on feedback from the memory system (Ⓑ). This feedback allows Detected-DAWS to classify static load instructions based on dynamic information about how much locality each instruction has and how many memory accesses it generates.

5.1 Profiled Divergence-Aware Warp Scheduling (Profiled-DAWS)

Figure 7 presents the microarchitecture for both Profiled- and Detected-DAWS. Both versions of DAWS are implemented as an extension to the WIA's baseline warp prioritization logic. The output of the scheduler is a *Can Issue* bit vector that prevents warps from issuing. The task of the scheduler is to determine this bit vector. As described in Section 5, this is driven by cache footprint predictions.

To create the cache footprint prediction for each warp, DAWS must classify the behaviour of static load instructions in loops. One method to predict the behaviour of static load instructions is to do a profiling pass of the application. To provide a bound on the potential of an online solution, we propose Profiled-DAWS. We classify each static load instruction using the two criteria presented in Section 4.2: (1) Is the load converged or diverged? (2) Does the load contribute to the footprint for this iteration of the loop (i.e.,

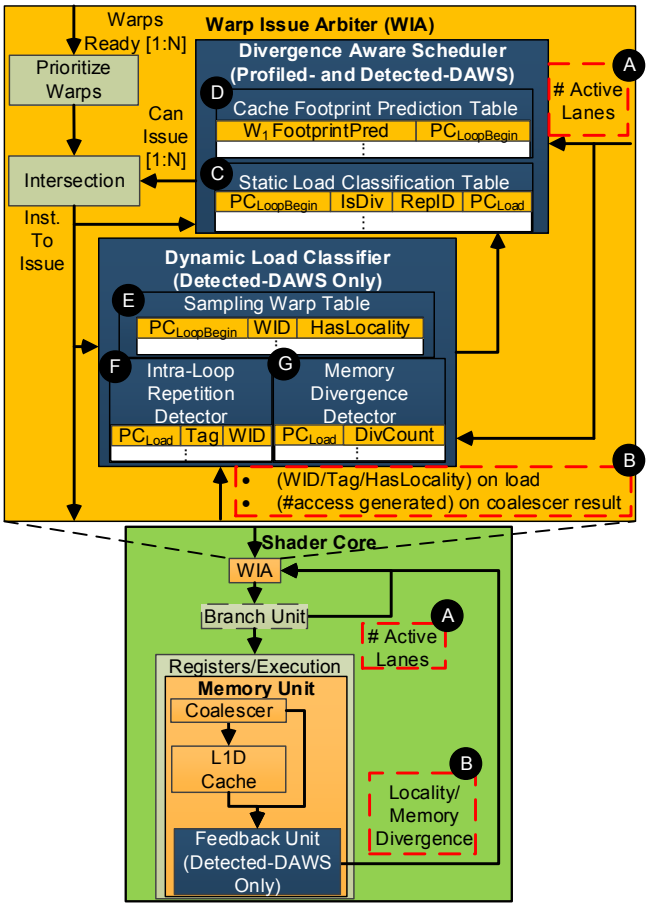

Figure 7: Detailed core model used for our DAWS solutions. N is the number of warp issue slots on the core.

The load's repetition ID)? To collect this information for Profiled-DAWS, we perform an analysis of compiled assembly code and use runtime information gathered from a profiling pass of each application. Determining if a load is converged or diverged is done by profiling all the accesses of each load, similar to the analysis done on BFS in Section 4.2.1. To determine intra-loop repetition we do not use profile information. Instead, we examine the assembly and assume that all loads using the same base address register whose displacement values are within one cache line are repeated in a loop iteration. Profiling similar to the analysis in Section 3 is performed to determine which loops in the code have locality.

From a microarchitectural perspective, the classification information for all static load instructions in loops is stored in a static load classification table (Ⓒ). Each entry in the table contains the PC of the first instruction in the loop where the load is located ($PC_{LoopBegin}$), a flag indicating if it is a diverged load (IsDiv) and a repetition ID (RepID) that is used to indicate the intra-loop repetition ID of the load. Although only necessary for Detected-DAWS, the PC of the load instruction PC_{Load} is also stored here. Profiled-DAWS populates this table when a kernel is launched. These values are based on profiling information from previous runs of the kernel. The table is finite in size and can be spilled to memory, however our applications have at most 26 static load instructions within loops.

The cache footprint prediction for each warp is stored in the cache footprint prediction table (Ⓓ). This table has one entry for each warp issue slot on the core. In our baseline this is 32 entries. Each entry of the table contains the value of the predicted footprint (in cache lines) and the PC identifying the loop ($PC_{LoopBegin}$). The scheduler checks instructions as they are issued, looking for loop begin/end points. To identify the loop bounds, we require that

the compiler adds markers to the first and last instruction of each loop. This can be implemented by using two previously unused bits in the opcode (one bit for loop start, one bit for loop end), or by adding an additional instruction to indicate loop start/end. The current CUDA compiler already outputs the loop bounds in the form of comments. We anticipate that our small addition of loop bound markers would have a minor impact. NVIDIA GPUs use a virtual ISA, which has made it easier to modify the hardware ISA in each of the last 3 architecture iterations.

When the scheduler detects that a warp has issued the first instruction of a loop, it uses the number of active lanes in the warp (Ⓐ) to create the warp's prediction for this loop iteration. This value is written to the warp's entry in the cache footprint prediction table. Section 5.1.1 details how the cache footprint prediction is computed. The update logic also writes the PC of the first instruction in the loop to the table ($\text{PC}_{LoopBegin}$). When the warp leaves the loop, the prediction table entry for the warp is cleared. To prevent deadlock, predicted footprints are also cleared while a warp waits at a barrier.

To determine the aggregate cache footprint, a prefix sum of each warp's cache footprint is performed, starting with the oldest warps. All of the warps whose prefix sum is less than our calculated effective cache size (defined in Equation 1) are eligible for issuing. Warps whose prefix sum is greater than the effective cache size are prevented from issuing load instructions.

$$EffCacheSize = kAssocFactor \cdot TotalNumLines \quad (1)$$

To decide how many cache lines DAWS should assume are available in the L1D cache (i.e., determining our $EffCacheSize$ value), we need to take the associativity of the cache into account. If we had a fully associative cache, we could assume that an LRU replacement policy would allow us to take advantage of every line in the cache. Since the L1D caches we study are not fully associative (our baseline is 8-way) our technique multiplies the number of lines in the cache by the $kAssocFactor$. The value of $kAssocFactor$ is determined experimentally and explored in more detail in section 7.3.

If the working set of one warp is predicted to exceed the L1D cache capacity, then no warps are de-scheduled and scheduling proceeds in an unthrottled fashion inside this loop. Doing no descheduling inside loops that load more data than is predicted to fit in cache reverts the system to hiding latency via multithreading again. We did not observe these large predictions in our workloads.

The prediction update logic is run each time the first instruction in a loop is issued. This way the prediction is reflective of threads leaving the loop because of differing loop trip counts across the warp.

5.1.1 Warp-Based Cache Footprint Prediction

This section explains how the number of cache lines accessed for a given warp in a single iteration of a loop with significant intra-warp locality is predicted. In a single threaded system, predicting the number of cache lines accessed in a loop iteration could be achieved by summing all the static load instructions predicted to be issued in the loop, while accounting for repetition caused by multiple static loads accessing the same data. However, to create a prediction of the data accessed by a warp in one loop iteration, both memory and control flow divergence must be taken into account. We first find which loop the warp in question is executing within by looking at the $\text{PC}_{LoopBegin}$ for this warp in the prediction table. Next, we query the static load classification table for all the entries with this $\text{PC}_{LoopBegin}$ (i.e., entries for all of the loads in this loop). It sums all the entries returned as follows. If the entry indicates that the load is diverged (i.e., the IsDiv bit is set), then this entry contributes as many cache lines as there are active threads. If the entry is converged (and there is more than one thread active), then this entry contributes two cache lines to the prediction. All entries with one active thread contribute one cache line. During the summation, each intra-loop repetition group (identified by RepID) is only counted once. If there are different divergence characteristics within the same repetition ID, then we count it as diverged. In our applications, we did not observe a diverged load accessing data loaded by a converged load (or vice-versa) in the same loop iteration. The result of this summation is written to this warp's entry in the cache footprint prediction table.

5.1.2 Predicted Footprint of Warps Outside Loops

In the previous sections, we only considered de-scheduling warps within loops because this is where the bulk of the application's memory accesses are. However, some applications may load a significant amount of data outside of loops. Figure 3 shows that PCs 1568 and 1600 from the GC benchmark both occur outside of the program's loop and access a significant amount of data, which can interfere with the accesses of warps within the loop. For this reason, if there are warps executing inside a loop, warps outside of loops can be de-scheduled. If any of the entries in the cache footprint prediction table is non-zero (i.e., at least one warp is in a loop), loads issued by warps outside of loops have their predictions updated as if they are executing their closest loop. Ideally a warp's closest loop is the next loop they will execute. For our purposes, we define a warp's closest loop as the next loop in program order. Since warps may skip loops, this may not always be the case, but in our applications this approximation is usually true.

5.1.3 Dealing with Inner Loops

DAWS also detects when a warp has entered an inner loop. When a warp issuing a new loop begin instruction already has a $\text{PC}_{LoopBegin}$ value in the cache footprint prediction table that is less than the PC of the new instruction, then we assume the warp has entered an inner loop. When this happens, the footprint prediction table entry for the warp is updated normally, giving the warp the prediction of the inner loop. However, when the warp leaves the inner loop, it does not clear either the prediction value or the $\text{PC}_{LoopBegin}$. When the outer loop begins its next iteration, it detects it is an outer loop (because the PC entry in the table is greater than the outer loop's beginning PC) and it recomputes the predicted footprint based on the inner loop's loads. This effectively limits the warps that can enter the outermost loops based on the predicted footprint of the innermost loop. We made this design decision because we observed that the majority of data reuse came from the innermost loop and there is significant data reuse between successive runs of the innermost loop. If we do not limit the number of warps entering the outer loop based on the inner loop, then there is the potential for multiple warps to interleave their inner loop runs, which can evict data repeatedly used by the inner loop. This can be applied to any arbitrary loop depth, but none of our applications had a loop depth greater than two.

5.2 Detected Divergence-Aware Warp Scheduling (Detected-DAWS)

Profiled-Divergence-Aware Warp Scheduling (Profiled-DAWS) relies on two key pieces of profile information. First, it requires that loops with intra-warp locality be known in advance of running the kernel. Second, it requires that all the global and local memory loads in those loops are characterized as converged or diverged and that all the intra-loop-trip repetition between those loads is

known. *Detected-Divergence-Aware Warp Scheduling* (Detected-DAWS) requires no profile information. The only requirement for Detected-DAWS is that the compiler mark the beginning and ending of the program's loops. Detected-DAWS *detects* both memory divergence and intra-loop-trip repetition at runtime and populates the static load classification table (Ⓒ in Figure 7) dynamically using the dynamic load classifier. Detected-DAWS operates by following the execution of a sampling warp through a loop. The first warp with more than two active threads that enters a loop is set as the sampling warp for the loop. The sampling warp id (WID) and ($PC_{LoopBegin}$) for each loop being sampled are stored in the sampling warp table (Ⓔ). When the sampling warp leaves the loop, the next warp to enter with two or more active threads becomes the new sampling warp for the loop. At any given time, multiple loops can be sampled but only one warp can sample each loop. The sampling warp table also stores a locality counter (HasLocality) that is used to indicate if loads for this loop should be entered into the static load classification table. Like the static load classification table, the sampling warp table is finite in size. Each of our applications has at most five loops. The dynamic load classifier interprets memory system feedback about loads issued from sampling warps.

It is worth noting that, other than the addition of PC_{Load} to each static load classification table entry, nothing about the divergence aware scheduler used in Profiled-DAWS changes. The scheduler just operates with incomplete information about the loops until the dynamic load classifier has filled the static load classification table.

The following subsections describe how the dynamic load classifier uses the memory system feedback to populate the static load classification table.

5.2.1 Finding Loops with Locality

This section describes how Detected-DAWS determines which loops have intra-warp locality. Memory system feedback (Ⓑ) informs the scheduler when loops have intra-warp locality. The feedback unit sends signals to the dynamic load classifier on each load issued signifying if the load has intra-warp locality. The feedback unit reports both captured and lost intra-warp locality. To report this locality, cache lines in the L1D cache are appended with the WID of the instruction that initially requested them. Lost intra-warp locality is detected through the warp ID filtered victim tags mechanism described in CCWS [33]. Hits in the L1D cache on data that one warp loads and re-references are reported as captured intra-warp locality. If a load has neither lost nor captured intra-warp locality then the feedback unit informs the dynamic load classifier that the load has no intra-warp locality. Whenever the classifier is informed that a load from a sampling warp has taken place, it modifies that loop's locality counter in the sampling warp table. If the load was an instance of intra-warp locality, the counter is incremented otherwise the counter is decremented. DAWS creates cache footprint predictions for loops with positive locality counters.

5.2.2 Dynamically Classifying Static Loads in Hardware

Once a loop is marked as having intra-warp locality, the dynamic load classifier starts generating static load classification table entries for the loop. To avoid having more than one entry for each static load in the static load classification table, Detected-DAWS requires the PC of the load be stored (PC_{Load}). Before inserting a new entry into the table, the dynamic load classifier must ensure that this PC_{Load} does not already exist in the table. If the entry does exist, the dynamic classifier updates the existing entry. The classifier consists of two components, an intra-loop repetition detector (Ⓕ) and a memory divergence detector (Ⓖ).

Table 1: GPGPU-Sim Configuration

# Compute Units	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc. LRU
L2 Unified Cache	128k/Memory Channel, 128B line, 8-way assoc. LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [12]
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

Memory Divergence Detector: The memory divergence detector is used to classify static load instructions as convergent or divergent. It receives information about load coalescing from the memory feedback unit. After an instruction passes through the memory coalescer, the resulting number of memory accesses is sent to the dynamic load classifier. The classifier reads this value in combination with the active thread count of the load instruction. If more than two threads in the instruction were active when the load was issued, the number of accesses generated is tested. If the number of accesses generated is greater than two, the divergence counter for this PC is incremented. If two or less accesses are generated, the counter is decremented. If the divergence counter is greater than one, this load is considered diverged, otherwise it is considered converged.

Intra-Loop Repetition Detector: The *Intra-Loop Repetition Detector* (ILRD) dynamically determines which static load instructions access the same cache line in the same loop iteration. It is responsible for populating the RepID field of the static load classification table. Each entry in the detector contains a tag, PC_{Load} and WID. On each load executed by a sampling warp, the ILRD is probed based on the tag of the load. If the tag is not found, the tag and PC/warp id for the instruction that issued the load are written to the table. If the tag is found, then both the PC issuing the new load and the PC in the table are marked as intra-loop repeated and assigned the same repetition ID. When the sampling warp branches back to the start of the loop, all the values in the ILRD for this warp are cleared. Without the WID, multiple loops could not be characterized concurrently because the sampling warp for one loop could clear the entries for another. The ILRD is modeled as a set associative tag array, with an LRU replacement policy.

6. EXPERIMENTAL METHODOLOGY

We model Profiled-DAWS and Detected-DAWS as described in Section 5.1 and 5.2 in GPGPU-Sim [3] (version 3.1.0) using the configuration in Table 1. Loop begin and end points are inserted manually in the assembly.

The highly cache-sensitive and cache-insensitive workloads we study are listed in Table 2, four of which come from the CCWS infrastructure available online [32]. The SPMV-Scalar benchmark comes from the SHOC benchmark suite [9].

Our benchmarks are run to completion which takes between 14 million and 1 billion instructions.

7. EXPERIMENTAL RESULTS

Table 2: GPU Compute Benchmarks (CUDA and OpenCL)

Highly Cache Sensitive			
Name	**Abbr.**	**Name**	**Abbr.**
BFS Graph Traversal [7]	BFS	Kmeans [7]	KMN
Memcached [15]	MEMC	Garbage Collection [4, 36]	GC
Sparse Matrix Vector Multiply (Scalar) [9]	SPMV-Scalar		
Cache Insensitive (CI)			
Name	**Abbr.**	**Name**	**Abbr.**
Needleman-Wunsch [7]	NDL	Back Propagation [3]	BACKP
Hot Spot [7]	HOTSP	LU Decomposition [7]	LUD
Speckle Red. Anisotropic Diff. [7]	SRAD		

Table 3: Configurations for Best-SWL and CCWS.

Best-SWL		CCWS Config	
Benchmark	**Warps Actively Scheduled**	**Name**	**Value**
BFS	5	$K_{THROTTLE}$	8
MEMC	7	Victim Tag Array	8-way
SPMV-Scalar	2		(512 total entries)
GC	2		16 entries per warp
KMN	4	Warp Base Score	100
All Others	32		

This section is organized as follows, Section 7.1 examines the performance of our workloads using Profiled-DAWS, Detected-DAWS and other warp schedulers. Section 7.2 presents the results of our programmability case study introduced in Section 2. The remainder of this section is devoted to analyzing varying aspects of our design and exploring its sensitivity.

7.1 Performance

All data was collected using GPGPU-Sim running the following scheduling mechanisms:

GTO A greedy-then-oldest scheduler [33]. GTO runs a single warp until it stalls then picks the oldest ready warp. Warp age is determined by the time the warp is assigned to the shader core. For warps that are assigned to a core at the same time (i.e., they are in the same thread block), warps with the smallest scalar threads IDs are prioritized. Other simple schedulers (such as oldest-first and loose-round-robin) were implemented and GTO scheduling performed the best.

Best-SWL Static Wavefront Limiting as described in [33]. Warp limitation values from 1 to 32 are attempted and the highest performing case is selected. A GTO policy is used to select between warps. The warp limiting value used for each application is shown in Table 3.

CCWS Cache-Conscious Wavefront Scheduling as described in [33]. The configuration parameters presented in Table 3 are used.

Profiled-DAWS Profiled Divergence-Aware Warp Scheduling as described in Section 5.1. Loop profiles were generated manually based on PC statistics collected in sampling application runs. The applications were profiled with input data different from the evaluation data. GTO prioritization logic is used.

Table 4: Configuration parameters used for DAWS

DAWS Config	
ILRD size	64 entries per core, 8-way set associative
Associativity Factor	0.6
Victim Tag Array	Same as CCWS in Table 3

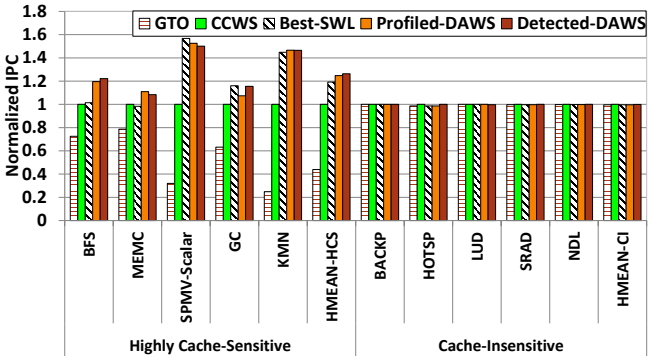

Figure 8: Performance of various scheduling techniques, normalized to CCWS.

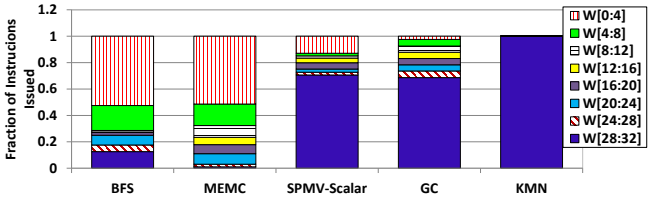

Figure 9: Breakdown of warp lane activity. Breakdown is presented as a fraction of total instructions executed. W[0:4] means 0 to 4 of an instruction's 32 lanes are active.

Detected-DAWS Detected Divergence-Aware Warp Scheduling as described in Section 5.2, with the configuration used in Table 4 GTO prioritization logic is used.

Figure 8 presents the *Instructions Per Cycle* (IPC) of our evaluated schedulers, normalized to CCWS. It illustrates that Profiled-DAWS and Detected-DAWS improve performance by a harmonic mean 25% and 26% respectively over CCWS on our highly cache-sensitive applications. In addition, they do not cause any performance degradation in the cache-insensitive applications. The cache-insensitive applications have no loops with detected intra-warp locality. Profiled-DAWS and Detected-DAWS are able to outperform Best-SWL by a harmonic mean 3% and 5% respectively. The performance of Profiled-DAWS and Detected-DAWS against Best-SWL is highly application dependent. Detected-DAWS is able to outperform Best-SWL on BFS by 20%, however it sees a 4% slowdown on SPMV-Scalar.

Figure 9 can help explain the skewed performance results against Best-SWL. It presents the control flow divergence in each of our highly cache-sensitive applications. It shows warp lane activity for all issued instructions. Bars at the bottom of each stack indicate less control flow divergence, as more lanes are active on each issued instruction. The two applications where DAWS improves performance relative to Best-SWL (BFS and MEMC) also have the most control flow divergence. The performance of Best-SWL is hampered most when there is significant control flow divergence. Selecting the same limiting value for every core over the course of the entire kernel is not optimal. This divergence occurs because of both loop-trip count variation across a warp and a discrepancy in the level of control flow divergence on each shader core. We also evaluated Detected-DAWS without control flow awareness by assuming all lanes were active on every loop iteration. Removing control flow awareness results in a 43% and 91% slowdown for BFS and MEMC respectively versus Detected-DAWS. Other applications showed no significant performance change.

Figure 10 presents the L1D cache misses, intra-warp hits and inter-warp hits *per thousand instructions* (PKI) for our highly cache-

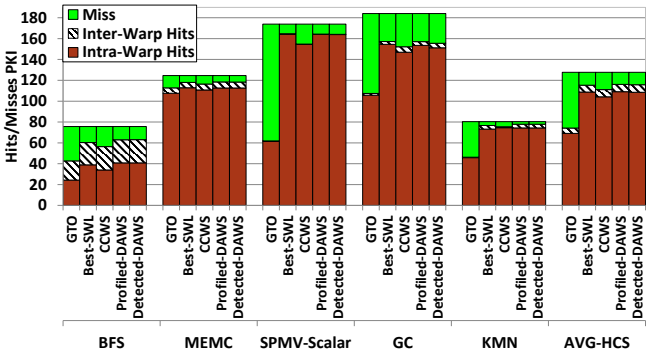

Figure 10: L1D intra-warp hits, inter-warp hits and misses *per thousand instructions* (PKI) of various schedulers.

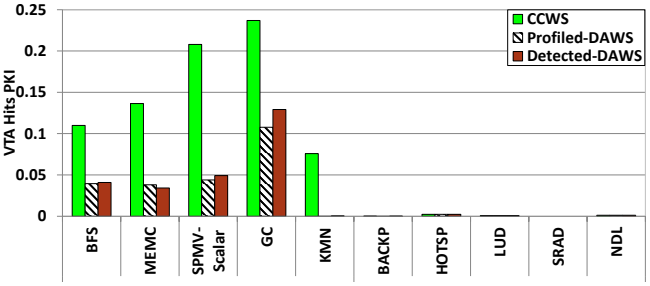

Figure 11: Victim tag array hits per thousand instructions (PKI) (indicating lost intra-warp locality).

sensitive benchmarks. It demonstrates that Profiled-DAWS and Detected-DAWS result in fewer cache misses than CCWS, which can account for a portion of the overall speedup. Since Profiled-DAWS and Detected-DAWS are able to predict the cache footprint of warps *before* they lose locality based on profile information created by other warps they can apply thread limiting before CCWS, removing the unnecessary cache misses. In addition, Profiled-DAWS and Detected-DAWS do not de-prioritize warps once they have entered a loop with locality. The scheduling point system in CCWS can potentially de-prioritize warps hitting often in cache when they stop producing accesses that hurt locality. We performed experiments and found that 46% of CCWS's lost locality occurs after a warp has been de-scheduled while in a loop. CCWS prioritizes warps based solely on detected lost locality. Warps may be de-scheduled inside a high-locality loop before they complete the loop, resulting in the eviction of their reused data. Once loops are properly classified, this type of lost locality never occurs using DAWS. DAWS ensures that once a warp enters a high-locality loop, it is not de-scheduled until the warp exits the loop or encounters a barrier. None of our highly cache-sensitive applications have barrier instructions. Figure 10 also demonstrates that the cache miss rate in Profiled-DAWS and Detected-DAWS is similar to that of Best-SWL. This suggests that the performance increase seen by Profiled-DAWS and Detected-DAWS over Best-SWL comes from decreasing the level of warp limiting when the aggregate footprint of threads scheduled can still be contained by the cache.

Figure 11 plots victim tag array hits, which indicate a loss of intra-warp locality. There is no victim tag array required to implement Profiled-DAWS, but for the purposes of this data, one is added. This figure illustrates that there is a large reduction in detected instances of lost locality when using the DAWS solutions. In addition, this figure shows a slight increase in detected lost locality in Detected-DAWS versus Profiled-DAWS. This is because Detected-DAWS requires some time to classify static load instructions before appropriate limiting is able to take effect.

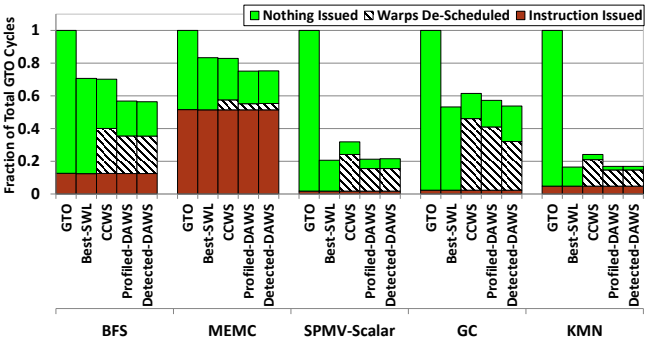

Figure 12: Breakdown of core activity normalized to GTO's total cycles for each application.

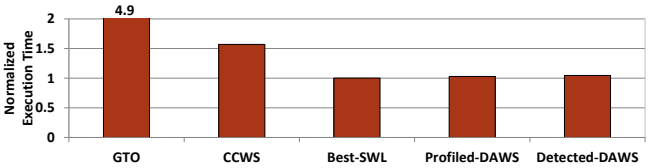

Figure 13: Execution time (lower values are faster) of SPMV-Scalar using various warp schedulers normalized to the best performing scheduler from SPMV-Vector.

Figure 12 breaks down core activity into cycles where an instruction issues, cycles where there are no instructions to issue (i.e., no warps are ready to be issued) and cycles where an instruction could have issued, if its warp had not been de-scheduled by the scheduling system. This is aggregate information collected over all the shader cores. This figure demonstrates that both Profiled-DAWS and Detected-DAWS reduce the number of cycles spent de-scheduling warps versus CCWS.

7.2 Programmability Case Study Results

In this section we examine the results of our case study presented in Section 2. To run these experiments, the size of on-chip scratchpad memory was increased to 48k, while leaving the L1D cache size constant. This was done so that shared memory usage would not be a limiting factor for SPMV-Vector and our results would not be biased towards SPMV-Scalar. The input sparse matrix is randomly generated by the SHOC framework. The matrix has 8k rows with an average of 82 non-zero elements per row. Figure 13 presents the execution time of SPMV-Scalar from Example 1 using our evaluated schedulers normalized to the GPU-optimized SPMV-Vector from Example 2 using its best performing scheduler. Like the other cache-insensitive applications we studied, the scheduler choice for SPMV-Vector makes little difference. There is $< 1\%$ performance variation between all the schedulers we evaluated. This figure demonstrates that SPMV-Scalar suffers significant performance loss when using previously proposed schedulers like GTO and CCWS. Best-SWL captures almost all the performance of SPMV-Vector, but requires the user to profile the application/input data combination with different limiting values before running. Detected-DAWS does not requiring any profiling information or additional programmer input and its execution time is within 4% of SPMV-Vector's.

Figure 14 compares several properties of SPMV-Scalar using Detected-DAWS to SPMV-Vector using its best performing scheduler. This graph shows that SPMV-Scalar has some advantages over SPMV-Vector, if Detected-DAWS is used. SPMV-Scalar executes 2.8x less dynamic instructions, decreasing the amount of dynamic power consumed on each core. SPMV-Scalar also requires 32x less warps, decreasing shader initialization overhead (which is not

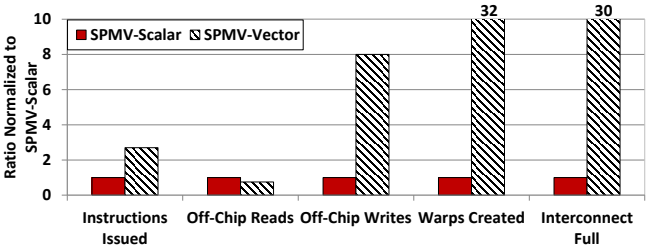

Figure 14: Ratio of various metrics for SPMV-Scalar using Detected-DAWS vs. SPMV-Vector using its best performing scheduler. *Interconnect Full*=instances where cores cannot access the interconnect due to contention.

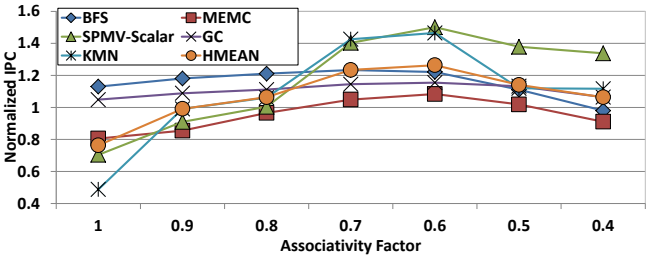

Figure 15: Detected-DAWS performance as the cache associativity factor is swept. Normalized to CCWS.

modeled in GPGPU-Sim) and the number of scheduling entities the GPU must deal with.

Since SPMV-Vector and SPMV-Scalar both perform the same computation on the same input, they fundamentally read and write the same data to and from memory. However, cache system performance and memory coalescing result in a discrepancy in the amount of off-chip traffic generated by each workload. Reads in SPMV-Vector are coalesced since lanes in each warp access consecutive values. However, since DAWS captures much of SPMV-Scalar's spatial locality in the L1D cache, there is only a 25% increase in read traffic. As a reference point, SPMV-Scalar using GTO produces $> 15\times$ more reads than SPMV-Vector. In addition, off-chip writes using SPMV-Vector are increased 8 fold. This happens because SPMV-Scalar is able to coalesce writes to the output vector since each warp attempts to write multiple output values in one SIMT instruction. SPMV-Vector must generate one write request for each row of the matrix and since the L1D caches evict global data on writes, all of these writes go to memory. The last metric compared indicates that contention for the interconnect is greatly increased using SPMV-Vector.

7.3 Determining the Associativity Factor

Figure 15 plots the performance change of our highly cache-sensitive applications as the $kAssocFactor$ is swept. All the applications consistently peak at 0.6, except BFS which peaks shows a small performance gain at 0.7 versus 0.6. This is consistent with the assertion that $kAssocFactor$ should be mostly independent of the application. The slight performance improvement for BFS at 0.7 can be explained by the fact that it has branches inside its loop that cause some of the loads to be infrequently uncovered, as discussed in Section 4.2.1. Since DAWS overestimates by assuming all the loads in the loop are uncovered, a larger $kAssocFactor$ makes up for this per-warp overestimation by raising the effective cache size cutoff.

7.4 Area Estimation

The tables added for Profiled-DAWS (i.e., the cache footprint prediction table and the static load classification table) are each modeled with only 32 entries and are negligible in size. The additional area added by Detected-DAWS comes from a victim tag array, the other tables are 64 entries or less. A victim tag array is also used in CCWS, so there is negligible area difference between Detected-DAWS and CCWS. However, compared to Best-SWL or GTO schedulers both CCWS and Detected-DAWS have a CACTI [37] estimated area overhead of 0.17% which is discussed in more detail in [33].

7.5 Dynamic Energy Estimation

We investigated two energy models for GPUs to evaluate the effect DAWS has on energy, GPUSimPow [28] and GPUWattch [26]. Due to the recent release date of these simulators, we were unable to fully integrate our solution into their framework. However, we extracted the nJ per operation constants used in GPUWattch for DRAM reads, DRAM pre-charges and L2/L1D cache hits/misses, which are the metrics that dominate the overall energy consumed in the highly cache-sensitive applications and are the key metrics effected by DAWS. This calculation shows that DAWS consumes $2.4\times$ less and 23% less dynamic energy in the memory system than GTO and CCWS respectively. This power reduction is primarily due to an increase in the number of L1D cache hits, reducing power consumed in the memory system. This estimate does not include the dynamic energy required for Detected-DAWS or CCWS tables, victim tag array or logic. We anticipate this energy will be small in comparison to the energy used in the memory system.

8. RELATED WORK

There are a number of works that suggest throttling the number of threads in a system can benefit performance. Bakhoda et al. [3] demonstrate that limiting the number thread blocks assigned to a core can reduce contention for the memory system. Guz et al. [14] present an analytical model that quantifies the "performance valley" that exists when the number of threads sharing a cache is increased. Cheng et al. [8] also propose an analytical model. They quantify thread interference at the memory stage in a stream programming language.

In addition to the previously discussed CCWS and Best-SWL [33], there are a number of other papers evaluating warp scheduling policies on GPUs. Lakshminarayana and Kim [24] evaluate warp scheduling in a GPU without hardware managed L1D caches. Fung et al. [12, 11] examine the impact of thread scheduling when techniques aimed at reducing control flow divergence are performed. Gebhart and Johnson et al. [13] and Narasiman et al. [30] both propose two level scheduling techniques aimed at reducing power and improving performance respectively. In contrast to these techniques which statically determine active and inactive groups, DAWS dynamically selects the number of warps for active scheduling based on code locality predictions and dynamic control flow information. Jog et al. [20] and Kayiran et al. [22] propose locality aware thread block schedulers that seek to limit the number of thread blocks sharing the L1D cache. Their techniques apply warp limiting at a coarse grain. DAWS seeks to maximize cache usage using fine grain divergence information and code region characterization. Lee et al. [25] and Jog et al. [21] explore prefetching on the GPU, with the latter focusing on prefetching-aware scheduling. In contrast to prefetching, which cannot improve performance in bandwidth limited applications, DAWS makes more effective use of on-chip storage to reduce memory bandwidth. Meng et al. [29] present dynamic warp subdivision which splits warps, allowing threads that hit in cache to continue, even if their warp peers miss. Their technique attempts to improve performance by loading data into the cache more quickly, while DAWS attempts to limit the number of

threads sharing the cache at once.

There are a number of works that attempt to improve cache hit rate by improving the replacement policy (e.g., [18, 31] among others). Work on improving cache replacement can be considered orthogonal to work on scheduling. Jaleel et al. [17] use information from the last level cache to make OS level scheduling decisions about which threads should be assigned to which cores. Our work focuses on the low level thread scheduler in a GPU and uses fine grained information about per-PC locality to make predictions. Jia et al. [19] evaluate GPU L1D cache locality in a current GPU and use a compile time algorithm to detect loads with no locality that should bypass the cache. In contrast, our paper focuses on runtime scheduling decisions about what to do with loads that have locality.

9. CONCLUSION

This work quantifies the relationship between memory divergence, branch divergence and locality on a set of workloads commonly found in server computing. We demonstrate that divergence and locality characteristics of static load instructions can be accurately predicted based on previous behaviour. Divergence-Aware Warp Scheduling uses this predicted code behaviour in combination with live thread activity information to make more locality-aware scheduling decisions. Divergence-Aware Warp Scheduling is a novel technique that proactively uses predictions to prevent cache thrashing before it occurs and aggressively increases cache sharing between warps as their thread activity decreases.

Our simulated evaluations show that our fully dynamic technique (Detected-DAWS) results in a harmonic mean 26% performance improvement over Cache Conscious Wavefront Scheduling [33] and 5% improvement over the profile-based Best-SWL [33]. Performance relative to Best-SWL is improved as much as 20% when workloads have significant control flow divergence.

Our work increases the efficiency of several highly divergent, cache-sensitive workloads on a massively parallel accelerator. Our programmability case study demonstrates that Divergence-Aware Warp Scheduling can allow programmers to write simpler code without suffering a significant performance loss by effectively shifting the burden of locality management from software to hardware.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. We also thank Tayler Hetherington for his help with the GPUWattch power model. This work was supported by an NVIDIA Graduate Fellowship and the Natural Sciences and Engineering Research Council of Canada.

11. REFERENCES

[1] NVIDIA CUDA C Programming Guide v4.2, 2012.

[2] T. M. Aamodt et al. GPGPU-Sim 3.x Manual. `http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual`, 2012.

[3] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS 2009*, pages 163–174.

[4] K. Barabash and E. Petrank. Tracing Garbage Collection on Highly Parallel Platforms. In *ISMM 2010*, pages 1–10.

[5] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC 2009*.

[7] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC 2009*, pages 44–54.

[8] H.-Y. Cheng et al. Memory Latency Reduction via Thread Throttling. In *MICRO-43*, pages 53–64, 2010.

[9] A. Danalis et al. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU 2010*.

[10] H. Esmaeilzadeh et al. Dark Silicon and the End of Multicore Scaling. In *ISCA 2011*, pages 365–376.

[11] W. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *HPCA 2011*, pages 25 –36.

[12] W. W. L. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO-40*.

[13] M. Gebhart and D. R. Johnson et al. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *ISCA 2011*, pages 235–246.

[14] Z. Guz et al. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *Computer Architecture Letters*, pages 25 –28, jan. 2009.

[15] T. H. Hetherington et al. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. In *ISPASS 2012*, pages 88 –98.

[16] S. Hong et al. Accelerating CUDA Graph Algorithms at Maximum Warp. In *PPoPP 2011*, pages 267–276.

[17] A. Jaleel et al. CRUISE: Cache Replacement and Utility-Aware Scheduling. In *ASPLOS 2012*, pages 249–260.

[18] A. Jaleel et al. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *ISCA 2010*, pages 60–71.

[19] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the use of Demand-Fetched Caches in GPUs. In *ICS 2012*, pages 15–24.

[20] A. Jog et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS 2013*.

[21] A. Jog et al. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.

[22] O. Kayiran et al. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT 2013*.

[23] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`.

[24] 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*, 2010.

[25] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *MICRO-43*, pages 213–224, 2010.

[26] J. Leng et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA 2013*.

[27] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[28] M. Maas et al. How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator. In *ISPASS 2013*.

[29] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA 2010*, pages 235–246.

[30] V. Narasiman et al. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *MICRO-44*, pages 308–317, 2011.

[31] M. K. Qureshi et al. Adaptive Insertion Policies for High Performance Caching. In *ISCA 2007*, pages 381–391.

[32] T. G. Rogers. CCWS Simulation Infrastructure. `http://www.ece.ubc.ca/~tgrogers/ccws.html`, 2013.

[33] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO-45*, 2012.

[34] S. Rul et al. An Experimental Study on Performance Portability of OpenCL Kernels. In *Application Accelerators in High Performance Computing*, 2010.

[35] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA 1995*.

[36] D. Spoonhower, G. Blelloch, and R. Harper. Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection. In *Proc. of Int'l Conf. on Virtual Execution Environments (VEE 2005)*, pages 57–67.

[37] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.