Leveraging Memory Level Parallelism Using Dynamic Warp Subdivision

Jiayuan Meng, David Tarjan, Kevin Skadron
Univ. of Virginia Dept. of Comp. Sci. Tech Report CS-2009-02
Apr. 2009

Abstract

SIMD organizations have shown to allow high throughput for data-parallel applications. They can operate on multiple datapaths under the same instruction sequencer, with its set of operations happening in lockstep sometimes referred to as warps *and a single lane referred to as a* thread*. However, ability of SIMD to gather from disparate addresses instead of aligned vectors means that a single long latency memory access will suspend the entire warp until it completes. This under-utilizes the computation resources and sacrifices memory level parallelism because threads that hit are not able to proceed and issue more memory requests. Eventually, the pipeline may stall and performance is penalized. Therefore, we propose warp subdividing techniques that dynamically construct run-ahead "warp-splits" from threads that hit the cache so that they can run ahead and prefetch cache lines that may be used by others that fall behind. Several optimization strategies are investigated and we evaluate the techniques over two types of memory systems: a bulk-synchronous cache organization and a coherent cache hierarchy. The former has private caches communicating with the main memory with coherence taken care of by global barriers; the latter has private caches coherently sharing an inclusive, on-chip last level cache (LLC). Experiments with eight data-parallel benchmarks show our technique improves performance on average by 15% on the bulk-synchronous cache organization with a maximum speedup of 1.6X, and 17% on a coherent cache hierarchy with a maximum speedup of 1.9X. This can be achieved with an area overhead of less than 2%.*

1 Introduction

SIMD organizations use a single instruction sequencer to control multiple datapaths (hence the term SIMD), and they go back to the Solomon/ILLIAC project [16]. SIMD is generally more efficient than MIMD organizations in exploiting data parallelism, because it allows greater throughput within a given area and power budget by amortizing the cost of the instruction sequencing over the multiple datapaths. This observation is becoming important, both because data parallelism is common across a wide range of applications in the fields of scientific computing, media processing, machine learning and data mining; and because data-parallel throughput is increasingly important for high performance as single-thread performance slows.

SIMD operations can operate on multiple datapaths in the form of a vector, or on a set of scalar datapaths with independent, scalar register files, sometimes referred to as Single Instruction, Multiple Threads (SIMT). For purposes of generality in this paper, we will refer to the set of operations happening in lockstep as a *warp* and the application of an instruction sequence to a single lane as a *thread*. We refer to a set of hardware units under SIMD control as a *warp processing unit* or WPU[1]. SIMD organizations can also be multithreaded to hide pipeline or memory latencies. This requires the WPU to time-multiplex among multiple concurrent warps, each with their own PCs and registers.

SIMD organizations are now prevalant in microprocessors, including the SSE and other multimedia, short-vector instruction sets in commodity CPUs; Cell BE [13], and various signal processors; and longer vector instructions in future Intel instruction sets, including Intel's Larrabee [19]. SIMT organizations are more common in architectures for high performance computing, exemplified today in Clearspeed [15] and various other accelerators. Graphics processors (GPUs) are also SIMT organizations today [8, 2] and are increasingly used for general-purpose computing.

The chief drawback with SIMD organizations is that the single instruction sequencer *limits* throughput when SIMD lanes exhibit divergent behavior. This divergence may arise due to different branching or memory latencies. In the case of conditional branches, a WPU can only execute one path of the branch at a time, with threads from a given warp masked off if they took the branch in the alternate direction. In the case where threads from a single warp experience different memory-reference latencies, a similar problem arises. In this case, the entire warp must currently wait until the last thread has its reference satisfied. This can occur in vector as well as SIMT organizations, if the vector instruction set allows *gather* operations (loading a vector from disparate addresses).

Divergent memory latencies are possible when threads access different DRAM banks or in the presence of caching. Although caching is uncommon in contemporary SIMT organizations, it is common in vector organizations, and is likely to grow in importance to reduce off-chip bandwidth and to avoid redundant memory references in irregular data structures [11]. Otherwise, compiler-controlled local stores end up being used to implement software caches, at much higher cost than a proper hardware cache. Even graphics processors use caching, albeit only for read-only data, and this is important in general-purpose as well as graphics workloads [4, 5].

This paper focuses on reducing the penalties of divergent cache latencies. Characterizations show that with 16 KB D-caches, WPUs with four warps and a SIMD width of eight diverge on up to 60% of SIMD memory instructions that incur at least one miss, and WPUs may spend 53% of the time waiting for memory accesses. Although the WPU can switch to another warp and continue execution, its latency-hiding capability is constrained by the limited number of warps and the fact that each warp may be incurring the same divergent behavior. Furthermore, adding more warps to hide latency will multiply the number of thread contexts, register-file overhead, and cache contention.

Specifically, this paper proposes to *subdivide* warps in response to divergent cache accesses. This allows threads that hit to run ahead and prefetch cache lines that may be needed

[1]We invent a new term here to distinguish it from "cores" or "PEs" which may refer to individual scalar pipelines that constitutes the WPU. In commodity GPUs, this is also referred to as shader processors, which is more specific to the graphics pipeline.

by the remaining threads that fall behind. The run-ahead threads are likely to stall on future cache accesses, allowing the split warps to eventually rejoin. The challenge is to manage this process in a way that exploits greater memory level parallelism (MLP) without reducing overall throughput due to reduced warp utilization — aggressive subdivision may result in performance degradation because it may lead to a large number of narrow warp-splits that only exploit a fraction of the computation resources. A dynamic mechanism is needed because the divergence pattern varies across applications, phases of execution, and even inputs.

We evaluate several strategies for dynamic warp subdivision based upon eight distinct data-parallel benchmarks and study them on two types of cache organizations: a bulk-synchronous cache organization that has private caches communicating through main memory with global barrier synchronizations (a possible direction for future SIMT organizations), and a coherent, two-level cache hierarchy that has private L1 caches sharing an inclusive, on-chip L2 (representative of many of today's vector organizations, including the upcoming Larrabee processor). Experiments show that our technique improves the performance on average by 15% on the bulk-synchronous cache organization with a maximum speedup of 1.6X, and 17% on the coherent cache hierarchy with a maximum speedup of 1.9X. Our technique has less than 2% area overhead, it is also robust and shows no performance degradation in any case.

2 Related Work

Fung et al. studied dynamic warp formation for efficient *control flow* management in SIMD processors [12]. Following the occurrence of diverging branch outcomes, their hardware implementation is able to dynamically regroup threads that take the same program path into new warps, thereby improving the utilization of SIMD pipelines. Our techniques leverages MLP and it addresses another significant source of low pipeline utilization — long latency memory accesses. We demonstrate that when SIMD hardware uses caches, dynamically subdividing warps upon divergent cache accesses can further improve performance. Our technique is especially useful for memory intensive, data parallel applications, no matter whether they subject to intensive divergent control flows or not.

Existing techniques that address long latency memory accesses in the context of simultaneous multithreading (SMT) do not help in the case of SIMD because of the intrinsically different datapaths. In both cases, the concern is reduced throughput. But with SMT, the problem is reduced instruction level parallelism (ILP) as a stalled thread fails to use idle issue slots and also occupies expensive issue queues, rename-register, and reorder-buffer entries, which limits ILP discovery for the other thread. Techniques for SMT resource distribution [6, 23] do not apply to SIMD. SIMD datapaths are usually in-order organizations, because simpler datapaths provide greater area efficiency in the presence of sufficient parallelism [10]. Stalled warps, however, require other warps to keep the hardware occupied and maintain throughput. Each additional warp that a WPU hosts incurs extra costs in terms of wide, SIMD register state. Instead, the main problem raised by long latency memory accesses is the risk of pipeline stall due to divergent cache accesses, and the lack of warps to hide the latency and exploit more MLP. The less time that warps spend stalled, the fewer warps needed to maintain high throughput.

Another extensively investigated technique that leverages MLP in the context of SMT or multicore architecture is pre-computation using speculative threads [7]. These approaches target at improving *single-threaded* performance and they extract future instruction streams and execute them on additional computing resources to prefetch data. More recently, runahead threads are proposed that allow a thread to continue speculatively despite long latency memory accesses, prefetching data for the corresponding thread while releasing resources for other SMT threads [18]. These speculative approaches, however, require run-time dependency analysis among instructions as well as the ability for out-of-order execution and commit. These requirements are usually not met with simple, in-order SIMD hardware. Besides, SIMD cores can switch among many different warps to hide long latency memory accesses which the above SMT techniques typically do not consider. We provide a solution designed specifically for SIMD hardware that allows it to exploit more MLP without speculative execution.

Finally, the dynamic warp subdivision technique described in this paper is independent of the specific organization of the memory hierarchy — the decision of subdividing warps is solely based on pipeline utilization. It is therefore complementary to MLP-aware cache replacement policies [17].

3 SIMD Architecture

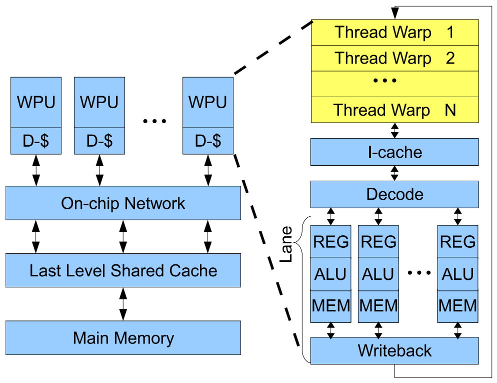


Figure 1: The baseline SIMT architecture groups scalar threads into warps and executes them using the same instruction sequencer. A thread operates over a scalar pipeline or lane that consists of register files, ALUs, and memory units.

We demonstrate warp subdivision using SIMT architectures while the same technique can be extended to vector machines, as we discuss in Section **??**. We focus on SIMT architectures due to its generality; SIMT supports single program, multiple data (SPMD), and can achieve the same computation bandwidth as vectors without the complexity and overhead of supporting instructions such as permutation [21].

A baseline architecture is demonstrated in Figure 1. In this case, we illustrate a two level coherent cache hierarchy that has private I- and D-caches which interact with an on-chip shared cache before looking up the off-chip memory. The use of shared caches is exploited by Larrabee [19] as an example of general-purpose chip multiprocessors (CMP) with SIMD support.

We also investigate another type of memory system for bulk-synchronous models (not shown in Figure 1), where there is only one level of per-WPU caches and they communicate directly with the main memory. Data is moved from one WPU to another by using global barriers that synchronize threads after the updated data is written to the main memory and before they are loaded remotely. It resembles the memory system in Cell BE [13] and Tesla [8] where private read-only caches or software-controlled local stores interact directly with the device memory.

3.1 Warp Processor

A WPU groups scalar threads into warps and these parallel threads in the same warp simultaneously execute the same instruction. The register files are highly banked so that multiple threads can access their operands at the same time. The D-cache is also highly banked to cater to the high bandwidth demand of memory accesses. Memory requests are forwarded from the lanes to the D-cache banks through a crossbar. Upon bank conflicts, requests are queued and processed sequentially. The banked register files, together with the execute units, are formed into lanes. A thread has its own registers reside in one of the lanes and it is executed by the corresponding scalar pipeline.

Upon conditional branches, threads within the same warp may end up with divergent control flow. The WPU chooses one of the control paths and execute threads that fall through the corresponding path, suspending others until later. This can be handled by post-dominator based reconvergence [12] described in more details in Section 5.4.

3.2 Latency Hiding and Its Limitation

If a thread misses in the D-cache, other threads belonging to the same warp must also wait for the memory request to complete before they can proceed in a SIMD manner. Unlike out-of-order processors, WPUs' pipelines are mostly in-order and they can rarely identify and continue with future independent instructions. Therefore, it may result in significant performance penalty if the pipeline stalls for every D-cache misses.

WPUs hide this latency by having multiple warps interleave their instruction sequences. When a warp is stalled by D-cache accesses, the WPU removes the warp from the pool of ready warps and switches to another warp in a round-robin fashion. The suspended warp reenters the ready queue when memory requests from all its threads complete. In this way, WPU mitigates the under-utilization of pipelines. In addition, MLP is improved since warps that take over the WPU can issue more memory requests in parallel with the previous long latency misses.

With limited number of warps, however, the pipeline may stall eventually if all the warps are suspended due to D-cache misses. Adding more warps multiplies the number of hardware thread contexts or register files. A large number of warps is therefore impractical due to its area cost, and the optimal number of warps may vary for different applications. In addition, more active thread contexts lead to more cache contention, and it may penalize performance, as we will show in Section 7.3.

4 Characterization

The throughput-oriented WPUs are targeted mostly at data-parallel applications with large input data sets. We simulate a set of parallel benchmarks shown in Table 1. They are selected from several benchmark suites and cover the application domains of scientific computing, media processing, machine learning and data mining. They represent data-parallel applications with varied data access and communication patterns. We adjust the input sizes such that our benchmarks exhibit sufficient parallelism for evaluation while maintaining manageable simulation times.

	Benchmark Description
FFT	Fast Fourier Transform (Splash2 [20]) Spectral methods. Butterfly computation Input: a 1-D array of 32,768 (2^{15}) numbers
Filter	Edge Detection of an Input Image Convolution. Gathering a 3-by-3 neighborhood Input: a gray scale image of size 500×500
HotSpot	Thermal Simulation (Rodinia [5]) Iterative partial differential equation solver Input: a 300×300 2-D grid, 100 iterations
LU	LU Decomposition (Splash2 [20]). Dense linear algebra Alternating row-major and column-major computation Input: a 300×300 matrix
Merge	Merge Sort. Element aggregation and reordering Input: a 1-D array of 300,000 integers
Short	Winning Path Search for Chess. Dynamic programming. Neighborhood calculation based on the the previous row Input: 6 steps each with 150,000 choices
KMeans	Unsupervised Classification (MineBench [14]). Map-Reduce. Distance aggregation. Input: 10,000 points in a 20-D space
SVM	Supervised Learning (MineBench [14]) Support vector machine's kernel computation. Input: 1,024 vectors with a 20-D space

Table 1: Simulated benchmarks with descriptions and input sizes.

These benchmarks are programmed in an OpenMP-style API implemented on our simulation platform based on M5, a cycle-accurate, event-driven simulator originally designed for a network of processors [3]. They are cross-compiled to the Alpha ISA using gcc 4.1.0. We do not use the original Pthread [1] or OpenMP [9] programs because M5 does not simulate the run-time support for these applications in system emulation mode. By instrumenting the code and using our own primitives to signal parallel, nested `for` loops, the simulated run-time library is able to tile the data-parallel tasks and execute them on available hardware thread contexts.

	Non-loop cond. branches	Divergent cond. branches	Cache misses in SIMD	Divergent cache-access
FFT	1399	45	41851	15716
Filter	0	0	53428	45142
HotSpot	26948	169	54669	36145
LU	0	0	40310	24257
Merge	19785	6829	17033	8655
Short	24240	5555	63573	45906
Kmeans	4764	1495	23924	7101
SVM	0	0	29332	6290

Table 2: Average numbers of non-loop conditional branches (i.e. conditional branches that does not delimits loop iterations), divergent conditional branches, SIMD cache misses (i.e. SIMD memory accesses that result in at least one cache miss), and divergent cache accesses within an interval of 1 M cycles.

We characterize these benchmarks with four WPUs that each have 8 thread contexts formed into eight warps. The characterization involves per-WPU private caches and a shared last level cache (LLC) with configuration details same as that shown in Table 3. Except for Merge and Short, all other benchmarks have few divergent *conditional branches*. On the other hand, divergent *cache-accesses* occur much more often — all benchmarks show that a significant portion of the instructions that incur cache misses actually result in divergent cache-accesses. Due to the commonness of divergent cache-accesses, performance can be improved by allowing threads that hit to run ahead in order to exploit more MLP. We further characterize the *divergence degree* of cache-accesses defined as the number of threads that miss the cache upon a divergent cache-access. Figure 2 shows that the divergence degree varies across benchmarks or even phases. We therefore propose to adaptively subdivide warps at run-time.

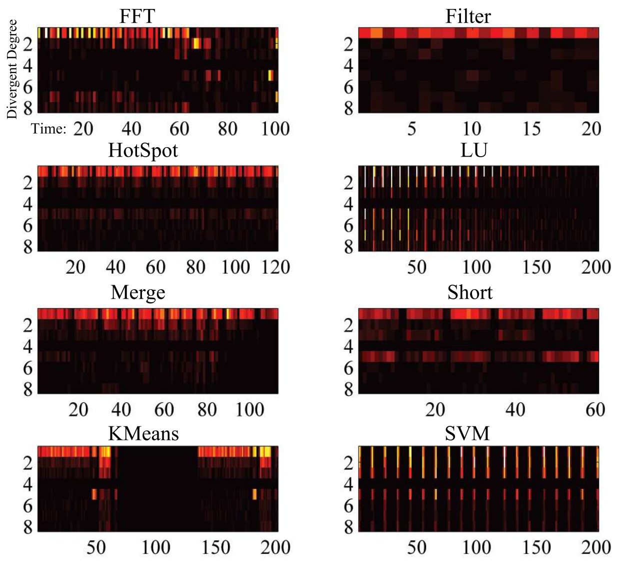


Figure 2: Characterizing the sampled time distribution of the divergence degree of cache accesses. The brightness of each cell reflects the intensity of divergent cache-accesses that end up with a particular divergence degree within a time interval of 1 M cycles.

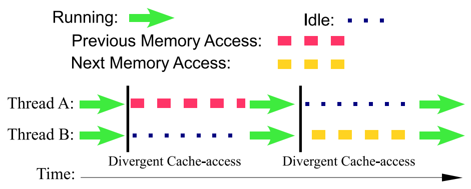
5 Dynamic Warp Subdivision

To mitigate the penalty of long latency memory accesses with limited number of warps, we propose dynamic warp subdivision that allows threads that hit during divergent cache accesses to continue and exploit more MLP. We name this technique as *MLP-aware warp subdivision* (MAWS). The split warps after the subdivision are referred to as *warp-splits*. Threads that continue after they hit the cache are named as *run-ahead threads* and they form a *run-ahead warp-split*. Threads that stall due to cache misses are called *fall-behind threads* and they form a *fall-behind warp-split*. The original warp is regarded as the *root warp-split* and it can be subdivided recursively.

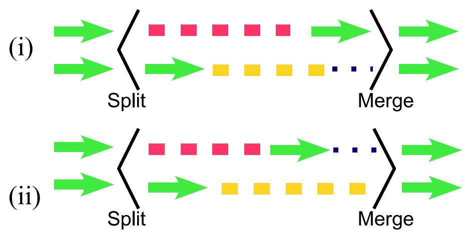
5.1 Exploiting MLP

Figure 3 compares conventional SIMT execution with MAWS upon divergent cache-accesses. Consider a warp that has two memory access instructions which would both incur diverged cache misses in the conventional execution model. Assuming all other warps have been suspended already, MAWS can avoid stalling (Figure 3(b)(i)) or reduce the stalling cycles (Figure 3(b)(ii)) in two scenarios:

- With conventional SIMT execution, the fall-behind threads would hit the cache upon the latter instruction anyway, and it is the run-ahead threads that now miss the cache. In this case, MAWS allows run-ahead threads to issue their memory requests earlier.



(a) Conventional SIMT Execution



(b) Warp Split Upon Diverged Memory Access

Figure 3: Comparing (a) conventional SIMT execution with (b) MAWS using a simplified WPU model with all its warps stalled due to cache misses except for one. For illustration purpose, the SIMD width is shown as two but similar scenario exists for wider SIMTs as well. MAWS allows threads that hit to proceed and issue more memory requests in parallel. As a result, threads that missed the cache previously may not have to stall (i) or only have to stall for a much shorter period (ii) upon the next memory request which would otherwise incur long latency in the conventional implementation.

- With conventional SIMT execution, the fall-behind threads would miss the cache upon the latter instruction, and they request the same cache block as some of the run-ahead threads. In this case, the run-ahead warp-split plays the role of prefetch threads for the fall-behind warp-slit. Different from speculative precomputation [7] or run-ahead simultaneous threads [18] in the context of SMT, the run-ahead warp-split always perform *useful* computation and threads' states are saved right away, requiring no ROB or dependency analysis that would otherwise complicate the design of the simple, in-order WPU.

In both cases, the long latency memory request which would otherwise stall the pipeline is issued earlier than they would in the conventional execution model. In consequence, the pipeline stalls for fewer cycles and performance can be improved.

5.2 Pipeline Utilization Hazard

Despite the potential improvement in MLP, both run-ahead warp-splits and fall-behind warp-splits are *narrower* than the original warp (i.e. they have fewer threads that utilize only a fraction of the available lanes). If not merged in time, not only do they risk low pipeline utilization, they may also sacrifice MLP because the maximum number of outgoing memory requests issued per instruction decreases along with the reduced SIMD width. As a result, aggressively splitting warp-splits for every diverged cache-accesses is likely to result in a large number of narrow warp-splits each with only a few threads, which can otherwise run altogether in a wider SIMT group. Therefore, MAWS risks performance degradation if it subdivides warp without constrains. We investigate several ways to improve MLP while lowering the risk of performance degradation.

5.3 Handling Conditional Branches

In our implementation, warp-splits belonging to the same warp always execute the same instruction stream — upon conditional branches, the warp-splits have to wait for and eventually merge with others belonging to the same warp. Otherwise, the SIMT implementation may be complicated significantly since the structure of the reconvergence stack will not be preserved — a run-ahead warp-split with diverged control flows will push the reconvergence stack, and the new stack top will not mark threads in the fall-behind warp-splits as active. It may lead to long running, narrow warp-splits that underutilize the SIMT pipeline — the WPU has to suspend the fall-behind warp-split until the run-ahead warp-split finishes its divergent control paths and returns to the same layer in the reconvergence stack as the fall-behind warp-split. It may also defer the merging process of the warp-splits significantly.

5.4 Implementing Warp-splits

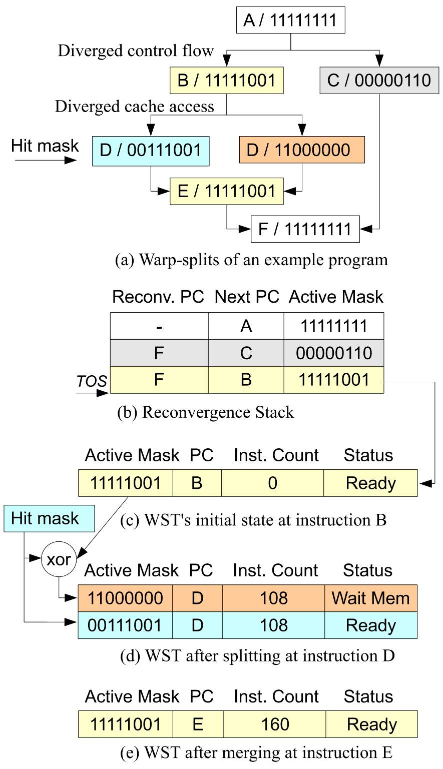


Figure 4: Warp splitting upon diverged cache accesses and its merging process.

Our hardware implementation uses a warp-split table (WST) for each warp to keep track of all its *existing* warp-splits. A WST entry records a warp-split's current PC, its instruction count, executing status, the priority, and the *active mask* with set bits denoting the belonging threads. Both the PC and the instruction count are used for the purpose of merging the warp-splits. The executing status and the priority are used for scheduling, and the active mask is used for selecting the threads in the corresponding warp-split to run in SIMT.

We adopt the post-dominator based reconvergence scheme to handle conditional branches [12]. Upon a divergent control branch, the WPU pushes the reconvergence stack with each level marking out threads that fall into the corresponding control path, as shown in Figure 4(b). Threads marked by the top of the stack are executed until they reach the post-dominator and pops the stack.

After a conditional branch, the active threads marked on the top of the warp's reconvergence stack is regarded as the initial warp-split (Figure 4(c)). Upon D-cache lookups that end up with divergent cache-accesses, a WPU constructs a bit mask with a set bit marking that the corresponding thread hits in the cache. This bit mask, referred to as the hit mask, becomes the active mask of the newly formed run-ahead warp-split. The active mask of the fall-behind warp-split is generated by an XOR between the hit mask and the active mask of its parent warp-split prior to the subdivision (Figure 4(d)).

To merge the warp-splits belonging to the same warp, the WPU first finds the warp-split that falls behind the most and increases its priority in the scheduling policy so that it can catch up with the run-ahead warp-splits. This is performed by finding the WST entry with the minimum instruction count recorded as the number of instructions that the warp-split has executed since the the root warp-split. Later on, when the WPU executes a load or a store and attempts to switch to another warp-split, it checks whether the PC of the warp-split matches that of another warp-split. If so, they will be merged into one (Figure 4(e)). Note that warp-splits from different warps may follow their merging processes independently.

The instruction counts always reflect the relative progresses among warp-splits in the same warp. It holds true even if the fall-behind warp-split goes beyond the run-ahead warp-split if the latter is subjected to future long latency cache misses and the former does not — in which case the fall-behind warp-split becomes the run-ahead warp-split. PCs of the warp-splits, however, cannot indicate their relative progresses because the run-ahead warp-split may unconditionally jump to some instruction addresses which can be lower or higher than that of the fall-behind warp-split.

5.4.1 Hardware Overhead

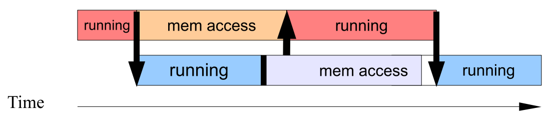
The maximum number of required per-warp WST entries equals the number of thread contexts within each warp, since the smallest warp-split is comprised of an individual thread. Note that after a subdivision, the entry of the obsolete warp-split is overwritten by the fall-behind warp-split so that a WST only records *existing* warp-splits. Overall, a WPU has the same number of WST entries as the number of hardware thread contexts.

We use Cacti [22] to estimate the area of WST entries as SRAMs and use an Opteron die photo to estimate the area of a WPU according to its ALUs and register files. Assuming a WPU operates over a 16 KB D-cache and 16 KB I-cache in a technology node of 65nm. If the WPU has four warps with a SIMD width of eight, it may have a maximum of 32 WST entries each requires 18 B. Overall, the WST entries consume less than 2% of the entire WPU.

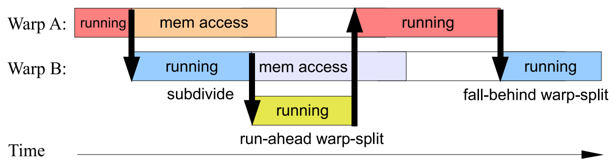
5.4.2 Lazy Split

As we discussed in Section 5.2, aggressively subdividing warps risks pipeline under-utilization and may even sacrifice MLP. In fact, upon divergent cache-accesses, a WPU can simply switch to an existing warp-split, and there is no need for subdividing warp-splits. Therefore, we investigate the option of subdividing warp-splits only when all others are suspended. This is referred to in the following discussion as *Lazy Split*. When an individual warp-split is suspended due to cache accesses, Lazy Split records the hit mask which is later used for subdividing warp-splits. On the other hand, the merging process begins as soon as a fall-behind warp-split completes its memory request, so that WPU reduces the risk of many long running, narrow warp-splits.

However, performance may still degrade if Lazy Split generates a run-ahead warp-split which is not able to issue subsequent long latency memory requests in time (i.e. before an out-going requests completes and wakes up an existing warp-split), as illustrated in Figure 5. In this case, the run-ahead warp-split may occupy the pipeline, keeping other warp-splits from making progress while exploiting no more MLP. Afterwards, the same instruction stream will be executed again by the fall-behind warp-split, increasing the number of executed cycles.



(a) Example case for conventional SIMT execution



(b) Lazy Split may lead to performance degradation

Figure 5: Performance may degrade if the run-ahead warp-split does not generate long latency memory requests before any request completes. Such case may take place with Lazy Split. We demonstrate the case with two warps with arrows pointing to the warp-split that the WPU executes.

5.4.3 Latency-speculating Split

To make sure that subdividing a warp will result in beneficial run-ahead warp-splits that issue long-latency memory requests before any other warp-splits can resume their execution, we exploit heuristics that help identify warp-splits that would improve performance if subdivided. We name this as *Latency-speculating Split*.

We define the *miss-free cycles* (MFC) of a warp-split to be the number of cycles that it occupies the pipeline without incurring long-latency cache misses. Consider the scenario where an executing warp-split encounters a divergent cache-access (i.e. its remaining MFC is zero). Let $\sum(rMFC)$ be the sum of remaining miss-free cycles of all existing warp-splits, $MemLat$ be the memory latency in cycles which other warp-slits have to hide, and $rMFC'$ be the remaining miss-free cycles for the running warp-split's subset of threads that hit the cache (i.e. the run-ahead warp-split if subdivided). With the above information, the hypothetical fall-behind warp-split will resume its execution after $MemLat$ cycles, and the hypothetical run-ahead warp-split can successfully issue its next long-latency memory requests before that time only if $\sum(rMFC) + rMFC' < MemLat$. Therefore, we subdivide the warp only when this condition is met. Note that this is a necessary but not sufficient condition: the run-ahead warp-splits may still not be able to issue the next memory request in time if other suspended warp-splits resume their execution before the $MemLat$ cycles expire.

Nevertheless, applying this heuristic still requires pre-knowledge about $\sum(rMFC)$, $MemLat$, and $rMFC'$. We use hardware counters to dynamically profile the WPU's historical data to approximate their values. For each WPU, we add a hardware counter which estimates $\sum(rMFC)$, named as the *sum-rMFC counter*. It cooperates with warp-splits' *MFC counters* appended to each WST entry. An MFC counter records its corresponding warp-split's miss-free cycles so far. It is incremented every cycle in which the warp-slit occupies the SIMT pipeline and is stopped when it incurs a cache miss. It is used as an approximation for the remaining miss-free cycles ($rMFC'$) of the hypothetical run-ahead warp-split upon divergent cache-accesses. When the warp-split completes its missed memory requests, its MFC count is also used as a predictor of the miss-free cycles before its next cache miss, and it is added to the WPU's sum-rMFC counter. The MFC counter then clears itself and restarts counting. The sum-rMFC counter is decremented every cycle in which the pipeline executes until its value reaches zero. In addition, another field, the memory latency count, is appended to each WST entry and it stores the latency in cycles involved in the corresponding warp-split's last cache miss. It is used as a predictor of the memory latency for the warp-split's incoming cache miss ($MemLat$). Using these speculative values, the heuristic is tested upon every divergent cache-access to decide whether the running warp-split should be further subdivided.

5.5 Loop Bypassing

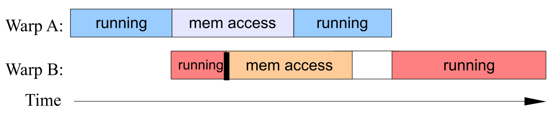
As we discussed in Section 5.3, MAWS preserves the conventional implementation of the reconvergence stack for simplicity. However, this keeps run-ahead warp-splits from proceeding beyond conditional branches and exploiting more MLP. A common source of conditional branches that holds the run-ahead warp-splits from proceeding comes from short loops. Fortunately, conditional branches that delimit loop iterations can be easily identified by the reconvergence stack, and we are able to allow a run-ahead warp-split to continue across iteration boundaries.

To detect loops upon conditional branches, a warp-split checks whether its reconverging PC matches that on top of the reconvergence stack. If so, the warp-split must be executing the same conditional branch as it previously did, and that the warp-split must be involved in a loop iteration. In this case, the warp-split may have divergent control flows. Those threads fall into the same path as they did before, marked active by the top of the reconvergence stack, will continue their execution as the run-ahead warp-split. Threads that take the other path will be suspended and they are marked active only in the the reconvergence stack's next-to-top position which corresponds to the other path. By doing so, the run-ahead warp-split continues and it is still guaranteed that it will execute the same control flow path as the fall-behind warp-splits.

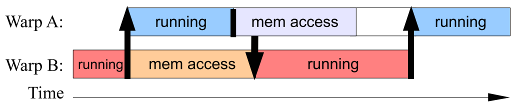
5.6 Scheduling Warp-splits

With MAWS, the scheduling entity becomes warp-splits instead of warps. Because a conventional round-robin (RR) scheduler prioritizes all ready entities equally, it is not able to identify warp-splits that are likely to miss in the near future. As a result, these warp-splits may not issue their long latency memory requests as soon as possible. As Figure 6 shows, this may end up with longer pipeline stalls due to fewer instructions are used to hide the memory latency.

We therefore propose a speculative scheduler that implements the *shallowest-warp-first* (SWF) policy, where the *shallowest warp-split* denotes the warp-split that has the fewest



(a) Execution trace if the scheduler is round-robin



(b) Execution trace if the scheduler is shallowest warp first

Figure 6: A Shallower-Warp-First scheduler can exploit more MLP than a round-robin scheduler.

remaining miss-free cycles (MFCs). The shallowest warp is identified speculatively using warp-splits' MFC counters which are described in Section 5.4.3. By storing the MFC value in another register and decrementing it every cycle in which the warp-split occupies the pipeline, the WPU is able to estimate the warp-split's remaining MFCs and identify the warp-split that is likely to miss the soonest. This shallowest warp is then given the highest priority in the scheduling policy.

5.7 Applicability to Vector Machines

While MAWS is demonstrated using SIMT architectures, it can be easily extended to vector machines with SIMD support that operate over cache structures. If a vector machine supports *gather loads* (i.e. load a vector from a vector of arbitrary addresses), the same problem of divergent cache-accesses would occur. To leverage MLP, vector components that hit the cache can continue their execution and issue more memory requests using the same principle.

Upon each divergent cache-access, a bit mask is introduced to mark out vector components that miss the cache. The information can be stored in a table with the PC upon which the divergent cache-access occurs. In this way, vector components that hit the cache can continue to be processed while those that miss can resume at the recorded PC afterwards.

6 Simulation Methodology

To simulate WPU, We extend the simple, in-order CPU model in M5 to have SIMT warps and a fetched instruction is executed for threads within the same warp simultaneously. Branch divergence is enabled by a stack based reconvergence mechanism [24] — using a bit mask, threads that do not fall into the current control path are not executed. Due to the lack of compiler support, we instrument the code with post-dominators that signals control flow reconvergence after branched control flow. Instructions-per-cycle (IPC) is assumed to be one except for memory references, which are modeled faithfully through the memory hierarchy (although we do not model memory controller reordering effects). A WPU switches warps in zero cycles upon every cache access. WPUs are simulated with up to 64 thread contexts and 64 lanes. Using the simulator, we study CMT systems with four such WPUs operating over cache hierarchies.

For the memory system, each WPU has a private I-cache. I-caches are not banked because only one instruction is fetched every cycle for all lanes. D-caches are always banked according to the number of lanes. We assume there is a perfect

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA, 64 hardware thread contexts 8 warps with a SIMD width of 8, in-order
I-Cache	16 KB, 4-way associative, 32 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back
D-Cache	32 KB, 4-way associative, 32 B line size MESI directory-based coherence 3 cycle hit latency, 16 MSHRs, LRU, write-back
LLC	2048 KB, 16-way associative, 128 B line size 30 cycle hit latency, 64 MSHRs, LRU, write-back
Crossbar	300 MHz, 57 Gbytes/s
Memory	300 cycles access latency

Table 3: Parameters for the two-level coherent cache hierarchy

Tech. Node	65 nm
WPU	1 GHz, 0.9 V Vdd, Alpha ISA, 32 hardware thread contexts 4 warps with a SIMD width of 8, in-order
I-Cache	16 KB, 4-way associative, 32 B line size 1 cycle hit latency, 4 MSHRs, LRU, write-back
D-Cache	32 KB, 4-way associative, 32 B line size 3 cycle hit latency, 16 MSHRs, LRU, write-back
Crossbar	300 MHz, 57 Gbytes/s
Memory	300 cycles access latency

Table 4: Parameters for bulk-synchronous one level cache organization.

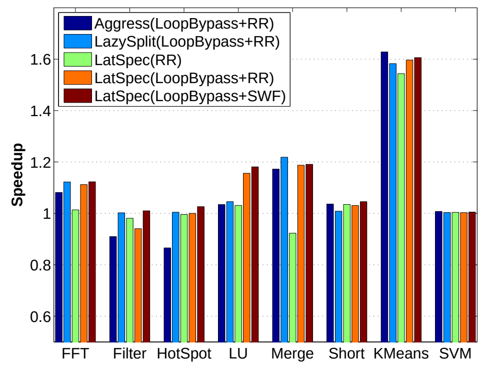
crossbar connecting the lanes with the D-cache banks. If bank conflicts occur, memory requests are serialized and a small queuing overhead (one cycle) is charged. The queuing overhead can be smaller than the hit latency because we assume requests can be pipelined. All caches are physically indexed and physically tagged with LRU replacement policy.

We investigate two types of cache organizations. In a two level coherent cache hierarchy, D-caches are private and together with the I-caches, they share an on-chip LLC through a crossbar. Table 3 summarizes its main parameters. Note that in Table 3, the aggregate LLC access latency is broken down into L1 lookup latency, crossbar latency, and the LLC lookup latency. The LLC then connects to the main memory through a 266 MHz memory bus with a bandwidth of 16 GB/s. The latency in accessing the main memory is assumed to be 300 cycles, and the memory controller is able to pipeline the requests.

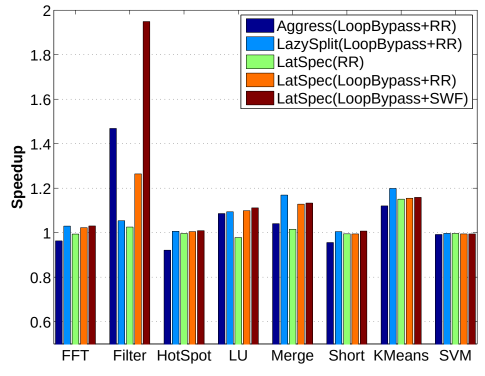
In the bulk-synchronous cache organization with one level of caches, the per-WPU D-caches interact directly with the main memory. A write to some data does not invalidate or update other copies immediately. Instead, WPUs can only communicate by flushing data to the main memory before a global barrier synchronization. Afterwards, WPUs reload data with values updated. Table 4 summarizes the main parameters for this type of cache organization.

7 Evaluation

We compare various strategies of dynamic warp subdivision by simulating benchmarks described in Section 4. *Conv* denotes conventional implementation of WPUs without MAWS. *Aggress* denotes naive MAWS implementations that subdivides warp-splits upon every divergent cache-accesses aggressively. *LazySplit* denotes MAWS implemented with the Lazy Split strategy, and *LatSpec* denotes MAWS implemented with the Latency-speculating Split strategy.



(a)



(b)

Figure 7: Speedup of various MLP optimizations on (a) a bulk-synchronous, one-level cache organization; and (b) a two-level coherent cache hierarchy.

7.1 Performance Improvement

As we will show in our sensitivity studies, LatSpec with LoopBypassing and SWF scheduling works consistently well across all applications without degradation. We compare speedups resulted from all combinations of the optimization techniques involved in MAWS. The combination includes subdivision strategies (Aggress, LazySplit, and LatSpec), loop bypassing, and scheduling policies (RR, SWF). The typical combinations are listed in Figure 7. For the bulk-synchronous cache organization, LatSpec(LoopBypass+SWF) outperforms LazySplit(LoopBypass+RR) significantly with LU and it leads to an average performance improvement of 15%, compared to LazySplit's performance gains of 12%. On the two-level cache hierarchy, LatSpec(LoopBypass+SWF) and LazySplit(LoopBypass+RR) achieve a performance improvement of 17% and 7%, respectively; no performance degradation is observed for both systems. On the other hand, although Aggress is able to achieve speedups for several applications, it leads to performance degradation up to 13% in the bulk-synchronous cache organization and 8% in the two-level cache hierarchy, which is caused by pipeline under-utilization due to narrow warp-splits that are not merged in time.

We observe that Filter benefits significantly from MAWS in the coherent cache hierarchy while FFT, LU, Merge, and KMeans benefit more in the bulk-synchronous cache organization. This phenomenon has to do with the cache miss latency, to which MAWS is sensitive and its impact is discussed in more detail in Section 7.5.

Applications that are subjected to intensive divergent cache-accesses and whose divergence degree remains relatively constant are likely to benefit more from MAWS. While SVM is subjected to intensive divergent cache-accesses as well, its divergence degree varies rapidly from one to five, as shown in Figure 2. The frequent variation in the divergence degree indicates that the optimal subdivision varies frequently and therefore a fall-behind warp-split is less likely to benefit from data prefetched by a run-ahead warp-split. Short and HotSpot, however, are subjected to frequent non-loop conditional branches (Table 2) which run-ahead warp-splits cannot bypass. It forces run-ahead warp-splits to wait and rejoin fall-behind warp-splits, penalizing their capability to prefetch more data.

Due to the commonness of short loops, all subdivision strategies, if implemented without loop bypassing, achieve little speedup except for KMeans. This is demonstrated in Figure 7 with measurement labeled as LatSpec(RR). In fact, performance may even degrade with LatSpec because its heuristic assumes warp-splits can continue to execute fluently as long as it has remaining miss-free cycles. It does not consider the effect of conditional branches that hold the run-ahead warp-splits. As a result, warps are more likely to be subdivided based on wrong decisions.

SWF scheduling is especially effective for Filter when LatSpec and LoopBypass are applied; it improves performance significantly in the coherent cache hierarchy and avoids performance degradation in the bulk-synchronous cache organization. However, the performance difference between RR and SWF scheduling is not apparent for most benchmarks. This is partly because that in our implementation, WPUs switch to another warp-split upon every cache accesses, rather than upon cache misses only. As a result, a shallower warp-split may still proceed to some extend — it needs not wait for other warp-splits to miss the cache before it can take over the pipeline.

In the following sensitivity studies, we will focus on the performance comparison of LatSpec and LazySplit. *As a default, LatSpec incorporates loop bypassing and SWF scheduling, and LazySplit incorporates loop bypassing and RR scheduling.*

7.2 Pipeline Utilization

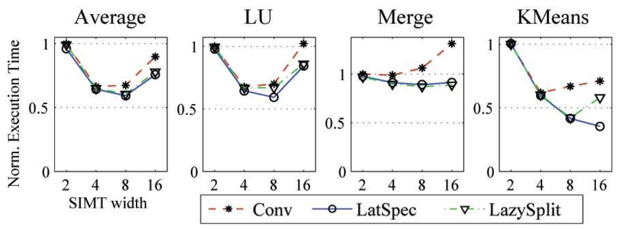


Figure 8: Normalized execution time vs. SIMD width on a bulk-synchronous, one-level cache organization. Each WPU has four warps. Performance is normalized to each benchmark's execution time under Conv at a SIMD width of two. The arithmetic mean for the normalized execution time of all benchmarks is shown as `Average`.

MAWS improves performance over conventional SIMT implementations especially for systems with more SIMD width, or lanes. This is because a wider SIMT pipeline executes more threads concurrently but it also increases the risk of divergent cache-accesses that would stall the pipeline in the conventional implementation. This risk can be reduced significantly using MAWS.

As Figure 8 shows, both LazySplit and LatSpec reach peak performance with a SIMD width of eight, while the performance of Conv saturates at a SIMD width of four. Similar trend is observed in the two level cache hierarchy as well. With a larger SIMD width, performance is improved first but eventually degrades as a result of cache contention due to more concurrent threads.

7.3 Latency Hiding with Fewer Warps

MAWS can improve the effectiveness of latency hiding in addition to having more warps. It may also achieve the same performance with fewer warps. To demonstrate this, we fix at a SIMD width of eight and vary the number of warps from one to eight. Figure 9 shows the performance scaling on the bulk-synchronous cache organization.

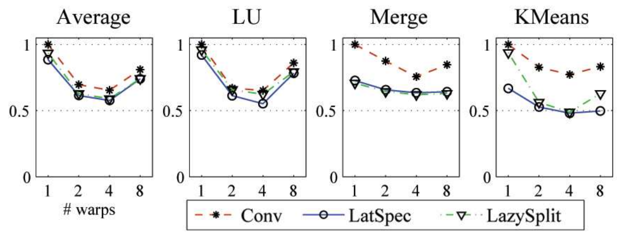


Figure 9: Normalized execution time vs. number of warps at a SIMD width of eight on a bulk-synchronous, one-level cache organization. Performance is normalized to each benchmark's execution time under Conv with WPUs that have one warp each. The arithmetic mean for the normalized execution time of all benchmarks is shown as `Average`.

In general, additional warps can hide latency and improve performance for all three configurations (Conv, LatSpec and LazySplit). MAWS can further hide latency in addition to having more warps. In fact, it achieves the same or better performance with two warps, compared to conventional implementations with four warps or more. Although the benefit of MAWS may decrease as the WPU incorporates more warps to hide latency, as can be observed from Merge, the benefit does not diminish. In fact, it may increase again in the form of graceful degradation. This is due to that cache contention increases the risk of divergent cache-accesses and MAWS can therefore mitigate its penalty. Finally, KMeans shows LatSpec demonstrates more robust benefit over LazySplit.

Given a certain budget of the number of hardware thread contexts, MAWS also helps dynamically balance the design tradeoffs between having more warps to hide memory latency and having more SIMD width for parallelism. In Figure 10, each WPU has 64 hardware thread contexts and we vary the SIMD width from two to 64 and the number of warps from 32 to one. Results show that the best tradeoff varies across benchmarks: while KMeans performs best with a single warp of width 64, Merge achieves its best performance with 16 warps and a SIMD width of four. In this case, MAWS is able to dynamically subdivide warps and adaptively translate more SIMD width to more warps. Using MAWS, performances of LU (not shown in Figure 10) and Merge degrade no more with larger SIMD width. Instead, performance may improve slightly.

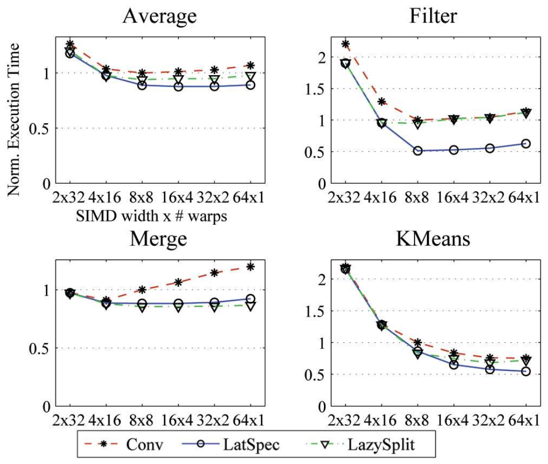


Figure 10: Normalized execution time vs. (SIMD width x number of warps) on a two-level coherent cache hierarchy. Each WPU has 64 hardware thread contexts. Performance is normalized to each benchmark's execution time under Conv with WPUs that have 8 warps and a SIMD width of 8. The arithmetic mean for the normalized execution time of all benchmarks is shown as `Average`.

7.4 Sensitivity of D-cache capacity

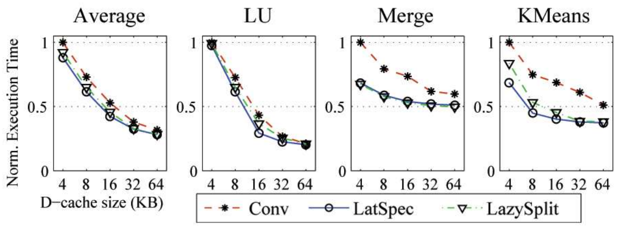


Figure 11: Normalize execution time vs. D-cache size on a bulk-synchronous, one-level cache organization. Performance is normalized to each benchmark's execution time under Conv with a D-cache size of 4 KB. The arithmetic mean for the normalized execution time of all benchmarks is shown as `Average`.

While MAWS leads to performance gains across a wide range of D-cache configurations, its benefit is most obvious with medium sized D-caches. For either very small or very large D-caches, the number of divergent cache-accesses decreases along with the increase in SIMT memory instructions that result in all cache misses or all cache hits, and therefore the benefit of MAWS may decreases as well.

We vary the D-cache size from 4 KB to 64 KB and Figure 11 illustrates their impact on MAWS for the bulk-synchronous cache organization. All benchmarks show that the speedup from MAWS decreases with larger D-cache sizes since divergent cache-accesses becomes rare. When performance is limited by small D-caches, MAWS is able to exploit the available memory bandwidth and improve cache throughput to some extent. However, as LU and KMeans show, the performance gains resulted from MAWS may be penalized by extremely small D-caches, in which case memory requests are mostly misses with fewer divergent cache-accesses.

7.5 Sensitivity of LLC Latency

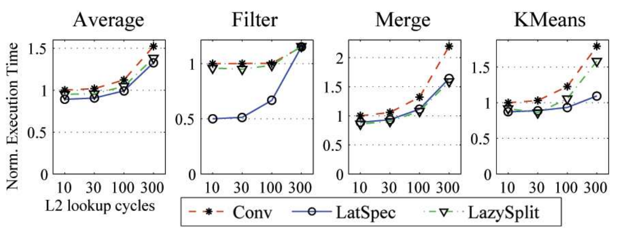


Figure 12: Normalize execution time vs. LLC lookup latency (in cycles) on the two-level coherent cache hierarchy. Performance is normalized to each benchmark's execution time under Conv with an L2 lookup latency of 10 cycles. The arithmetic mean for the normalized execution time of all benchmarks is shown as `Average`.

While MAWS persistently performs better than conventional SIMT implementations, it is usually more effective when the system is subjected to long latency memory accesses since it increases the need to have more warps to hide latency. This effect can be observed from Merge and KMeans in Figure 12 where we vary the lookup latency of the LLC from 10 cycles to 300 cycles in the two level cache hierarchy.

However, in the case of Filter, LatSpec exhibits an opposite response to longer LLC latency. This may be caused by the imperfect heuristic that we use in LatSpec; $MemLat$ estimates how much latency a WPU has to hide and it is approximated by the full memory access latency recorded previously, it does not account for warp-splits whose out-going memory requests are halfway in progress. In such scenario, the latency to hide can be much smaller than estimated, and subdividing the running warp-split may only under-utilize the pipeline. Such cases are more likely to occur with long latency LLCs. This is reflected by Filter's number of warp-splits which rapidly increases to 61 when the LLC latency is 300 cycles in the case of LatSpec. It also explains that Filter benefits less in the bulk-synchronous cache organization.

8 Conclusion

In this paper, we characterize the cache access behavior for several data-parallel applications running over SIMD pipelines and show they are subjected to a large number of divergent cache-accesses. As a result, warps are likely to suspend and those threads that hit the cache are not able to continue execution, leading to pipeline stalls in the case of limited number of warps available. To mitigate the penalty of divergent cache-accesses, we propose MAWS that leverages MLP by subdividing warps upon divergent cache-accesses and allow threads that hit the cache to run ahead and issue more memory requests. Several optimizations are proposed to lower the risk of pipeline utilization caused by narrow warp-splits. Lazy Split subdivides warps only when no more warps can proceed and exploit more MLP, and Latency-speculating Split reduces the number of unnecessary subdivisions when run-ahead warp-splits are not likely to be beneficial. Furthermore, loop bypassing improves the ability of run-ahead warp-splits to proceed across loop boundaries. On average, our technique improves the performance by 17% on the coherent cache hierarchy and 15% on the bulk-synchronous cache organization.

Future work include integrating MAWS with dynamic warp formation that handles divergent control flows. It may further improve performance by allowing the run-ahead warp-splits to proceed beyond not only loop boundaries, but also other conditional branches. Finally, we have noticed that the optimal number of hardware thread contexts vary in different applications. It will be helpful to adaptively restrict the number of active thread contexts in the case of severe cache contention.

9 Acknowledgements

This work was supported in part by SRC grant No. 1607, NSF grant nos. IIS-0612049 and CNS-0615277, a grant from Intel Research, and a professor partnership award from NVIDIA Research.

References

[1] R. A. Alfieri. An efficient kernel-based implementation of posix threads. In *USTC'94*, pages 5–5, Berkeley, CA, USA, 1994.

[2] ATI. Radeon 9700 Pro, 2002.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.

[4] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors. *IPDPS '09*, May 2009.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphisc processors using CUDA. *JPDC'08*, 2008.

[6] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA '06*, pages 239–251, Washington, DC, USA, 2006.

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO 34*, pages 306–317, Washington, DC, USA, 2001.

[8] NVIDIA Corporation. Geforce gtx 280 specifications. 2008.

[9] Leonardo Dagum. OpenMP: A proposed industry standard API for shared memory programming, October 1997.

[10] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP throughput with mediocre cores. In *PACT '05*, pages 51–62, Washington, DC, USA, 2005.

[11] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *ICS '07*, pages 93–104, New York, NY, USA, 2007.

[12] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO '07*, pages 407–420, Washington, DC, USA, 2007.

[13] M. Gschwind. Chip multiprocessing and the Cell Broadband Engine. In *CF'06*, New York, NY, USA, 2006.

[14] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. *WC'06*, pages 182–188, Oct. 2006.

[15] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. Performance improvement methodology for clearspeed's csx600. In *ICPP '07*, page 77, Washington, DC, USA, 2007.

[16] S. E. Orcutt. Implementation of permutation functions in illiac iv-type computers. *IEEE Trans. Comput.*, 25(9):929–936, 1976.

[17] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. *SIGARCH Comput. Archit. News*, 34(2):167–178, 2006.

[18] T. Ramirez, A. Pajuelo, O.J. Santana, and M. Valero. Runahead threads to improve smt performance. *HPCA '08*, pages 149–158, Feb. 2008.

[19] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[20] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[21] D. Talla and L. K. John. Cost-effective hardware acceleration of multimedia applications. In *ICCD '01*, pages 415–424, 2001.

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

[23] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO 34*, pages 318–327, Washington, DC, USA, 2001.

[24] S. Woop, J. Schmittler, and P. Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.