Atomic Vector Operations on Chip Multiprocessors

Sanjeev Kumar Daehyun Kim Mikhail Smelyanskiy Yen-Kuang Chen Jatin Chhugani
Christopher J. Hughes Changkyu Kim Victor W. Lee Anthony D. Nguyen

Microprocessor Technology Labs, Intel Corporation
sanjeev.kumar@intel.com

Abstract

The current trend is for processors to deliver dramatic improvements in parallel performance while only modestly improving serial performance. Parallel performance is harvested through vector/SIMD instructions as well as multithreading (through both multithreaded cores and chip multiprocessors). Vector parallelism can be more efficiently supported than multithreading, but is often harder for software to exploit. In particular, code with sparse data access patterns cannot easily utilize the vector/SIMD instructions of mainstream processors. Hardware to scatter and gather sparse data has previously been proposed to enable vector execution for these codes. However, on multithreaded architectures, a number of applications spend significant time on atomic operations (e.g., parallel reductions), which cannot be vectorized using previously proposed schemes.

This paper proposes architectural support for atomic vector operations (referred to as GLSC) that addresses this limitation. GLSC extends scatter-gather hardware to support atomic memory operations. Our experiments show that the GLSC provides an average performance improvement on a set of important RMS kernels of 54% for 4-wide SIMD.

1 Introduction

Future chip multiprocessors (CMPs) will use a combination of multiple cores, multiple hardware thread contexts per core, and single-instruction-multiple-data (SIMD) support to deliver high performance. Each of these features has different costs in terms of area, power, and design complexity, as well as provide different performance. Among these features, SIMD support for exploiting vector parallelism has the highest performance-to-cost ratio.

The best SIMD width (number of 32-bit data elements on which to operate simultaneously) for a processor is determined by a combination of two things: the amount of vector parallelism in the target applications, and the specific architectural support for SIMD in the processor. Many mainstream processors [1, 4, 16] already have support for 4-wide SIMD. The high data-level parallelism in graphics applications allows GPUs to use much wider SIMD (effective SIMD width of 32 on G80 [5]) to deliver high FLOPS.

Three main inefficiencies limit the effectiveness of SIMD operations. First, if the different SIMD elements take different control paths, only part of the computation performed by each SIMD instruction will be useful. Second, if the program has irregular data accesses, more powerful memory operations [2, 5, 16] that can read (gather) and write (scatter) data to non-contiguous memory locations are beneficial (Section 2.2). Third, if the program spends significant time in atomic operations on a set of sparse memory locations, these can become a bottleneck if the architecture lacks support for atomic vector[1] operations.

We examine a set of benchmarks from an emerging application domain, Recognition, Mining, and Synthesis (RMS) [9, 11, 23]. Many RMS applications have a lot of data-level parallelism and should benefit from SIMD support. However, we find that an important subset of these RMS applications spends significant time in atomic operations on a set of sparse locations and sees limited benefit from SIMD due to the lack of support for atomic vector operations in current architectures.

This paper, therefore, proposes novel architectural support for atomic vector operations (referred to as `GLSC`). To the best of our knowledge, our proposal is the first to provide support for fast atomic vector operations. Our contributions are as follows:

- `GLSC` extends scatter-gather and read-modify-write functionality to provide support for atomic vector operations. The proposed solution is easy for software to use and has low hardware complexity.
- We demonstrate that the proposed architectural support provides significant performance improvements. On a set of seven RMS benchmarks, `GLSC` provides an average performance improvement of 54% over existing hardware when using 4-wide SIMD with 16 software threads (four cores with four hardware threads per core).

[1] In this paper, we use the terms SIMD and vector interchangeably.

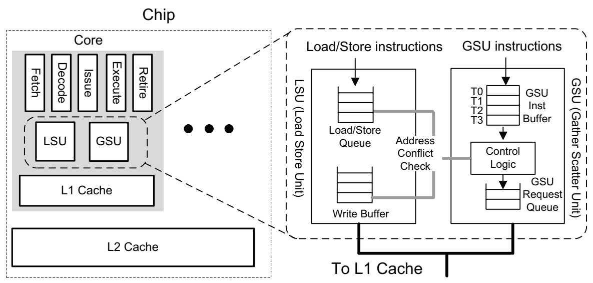

Figure 1. Base system architecture and internal organization of GSU(gather/scatter unit)

Further, we show that if SIMD widths increase in the future, the benefits from `GLSC` will grow substantially for applications with high SIMD efficiency.

2 Baseline Architecture

Many of today's processors take advantage of both thread- and data-level parallelism. They have both multiple cores, to exploit thread-level parallelism, and SIMD hardware, to exploit data-level parallelism.

The left-hand side of Figure 1 shows our base CMP. It has multiple cores, and each core has support for simultaneous multithreading. Each core has a single level of private cache, and all cores share an inclusive, physically distributed second-level cache. Cores and slices of the second-level cache are both attached to an on-die interconnect. The shared cache holds directory information for each cache line to maintain coherence amongst the private caches.

Our base system also includes SIMD support. In SIMD execution, a single instruction operates on multiple data elements simultaneously. This is typically implemented by extending the widths of registers and ALUs, allowing them to hold or operate on multiple data elements, respectively. The rest of the system is left untouched.

Adding and widening SIMD hardware is significantly cheaper than increasing the number of cores in a CMP. However, it is often more difficult to use SIMD efficiently than to use multiple threads. There are three common scenarios which traditionally limit the applicability of SIMD operations and, as a result, lower the SIMD efficiency of an application: (1) control flow, (2) irregular data accesses, and (3) atomic operations. Hardware support for SIMD in the first two situations has previously been proposed. Hardware support for atomic vector operations has received little attention and we address it in this paper. We now describe the three situations in more detail and explain how our base system deals with them.

2.1 Conditionals

A SIMD instruction simultaneously operates on all data elements packed into a SIMD register. However, sometimes programmers want to use conditional SIMD operations, where only a subset of the data elements are operated on. This is typically supported through the use of bit masks [2]. For each SIMD instruction, a bit mask is provided to specify which operands to ignore. Our base system includes support for masked SIMD instructions, where the masks are held in special mask registers.

2.2 Irregular Access Patterns

SIMD instructions operate on a set of densely packed data elements, which are typically aligned in the same cache line. As a result, conventional SIMD architectures require that the elements being read or written are stored contiguously in memory. However, many applications utilize data structures where elements are accessed indirectly (e.g., A[B[i]]) rather than contiguously. Efficiently utilizing SIMD in these applications often requires rearranging data, which can result in substantial overhead. To address this, hardware support for SIMD loading and storing non-contiguous data elements has been previously proposed [2, 5, 16]. This hardware performs what is commonly referred to as "gather/scatter" operations. Namely, a gather operation reads (gathers) multiple data elements from indirectly addressed locations, and packs them into a single SIMD register. Conversely, a scatter operation unpacks the elements in a SIMD register and writes (scatters) them into a set of indirectly addressed locations.

Our base system includes hardware and instruction support for gather and scatter operations. The right-hand figure in Figure 1 illustrates the organization of the gather/scatter unit (GSU) and the related load/store unit (LSU). A single instruction initiates a gather or scatter request. The instructions are blocking; that is, until all elements are read or written, the thread cannot proceed. The GSU instruction buffer has one gather/scatter instruction entry per SMT thread (four in our case). Once a gather/scatter instruction is inserted in the GSU instruction buffer, the control logic generates the sequence of corresponding addresses to gather or scatter. Each address is compared with the addresses in the load/store queue and the write buffer. To preserve memory ordering correctly in the presence of a conflict, a conflicting request waits in the GSU until corresponding requests in the LSU and write buffer have been sent to the L1 cache. The GSU shares the L1 cache ports with the LSU. In order to reduce port contention and improve GSU throughput, all memory accesses from a single gather/scatter instruction which fall onto the same cache line are combined. For example, if all elements are on the same cache line, the instruction will require only a single cache access.

2.3 Atomic Operations

Shared memory multiprocessor architectures usually include hardware support for performing scalar atomic read-modify-write operations on memory. This is commonly

```
1:  for( i = 0; i < numPixels; i++) {
2:      // determine the bin to increment
3:      bin = Minput[i] % numBins;
4:      do {
5:          ll Rtmp, &Mbins[bin];
6:          Rtmp++;
7:          sc Rsuccess, &Mbins[bin], Rtmp;
8:      } while (!Rsuccess); // Retry if sc failed
9:  }
```

Figure 2. Pseudo-code to perform parallel histogram using traditional support (load-linked and store-conditional).

used to implement lock acquires and parallel reductions. Our base system includes support for the well known load-linked (`ll`) and store-conditional (`sc`) read-modify-write primitives [19], which we explain through an example.

Figure 2 shows pseudo-code that computes a histogram in parallel. Histogram calculations are common operations in many image processing applications. A histogram is an array of bins, where each bin contains the number of elements that map into it. In multi-threaded execution, multiple threads may update the same bin simultaneously. Figure 2 includes an example usage of the `ll` and `sc` primitives to guarantee the atomicity of each update. The bin index, calculated in line 3 is used to perform the `ll` to load the current bin value into the register Rtmp, and set a hardware reservation bit. Next, the register value is incremented and the `sc` attempts to store the incremented value back into the bin. If the reservation bit has been cleared since the last `ll`, indicating an intervening conflicting write from another thread, `sc` fails and the entire `ll-sc` sequence repeats.

Atomic operations account for a significant fraction of time in some applications. This fraction is even larger in applications where the rest of the code can utilize SIMD. These applications can significantly benefit from SIMD execution of atomic operations. The hardware required for this must be capable of simultaneously performing multiple independent atomic read-modify-write operations. To our knowledge, no existing processor has such support. Transactional memory schemes have been proposed to allow atomic execution of *dependent* operations in critical sections [18]. While transactional memory has some similarities with our proposal, there are fundamental differences between the two that are discussed in detail in Section 6.

3 Architectural Support for Atomic Vector Operations

There are many possible models for supporting atomic vector operations. Supporting atomic vector operations on a set of contiguous memory locations requires only a relatively small extension to existing hardware. In contrast, providing more general support (i.e., for sparse locations) is significantly more challenging since a single atomic vector operation may touch multiple cache lines or even pages. We tackle the more general case because many of the applications we have examined require this support. We therefore leverage existing gather and scatter operations since they already have the ability to handle irregular memory access patterns.

We also choose to enforce a best-effort model, where a subset of the SIMD elements involved in an atomic operation are allowed to succeed rather than requiring that all elements either succeed or fail. A best-effort model simplifies a number of hardware issues, and allows a wide range of hardware implementations. For example, a very cheap but slow implementation would only attempt the atomic operation on one element at a time, whereas an aggressive one would attempt to operate on all elements simultaneously, regardless of issues such as the number of pages the elements are located on. The semantics of the already existing scalar load-linked and store-conditional instructions match this best-effort model. Load-linked sets a reservation for a location, and the corresponding store-conditional is expected to succeed unless the reservation has been invalidated by another thread writing to the location. However, an implementation is correct as long as it is conservative enough—it is acceptable to have reservations invalidated for other reasons, such as cache line evictions.

Therefore, to provide atomic vector operations, we extend load-linked and scatter-conditional to a SIMD context, and add support for gathering and scattering. We call the resulting instructions *gather-linked* and *scatter-conditional* (referred to as `GLSC`). These instructions are similar to conventional gather and scatter instructions, except that gather-linked obtains reservations for the locations being gathered, and scatter-conditional will only scatter values to elements whose corresponding reservations are still held. Since a scatter-conditional may only succeed for a subset of the elements (or for none at all), the instruction has an output mask which indicates success or failure, analogous to the output of a store-conditional. We also support an output mask for the gather-linked instruction; while this is not analogous to load-linked, it allows more flexibility in hardware implementations.

Another key difference between scatter-conditional and conventional scatter operations is in the handling of element aliasing, where a single SIMD operation attempts to write multiple values to the same location. Conventional scatters have undefined behavior in this situation because aliasing tends to be rare in general. However, we find that in many applications atomic operations have the potential for aliasing. Therefore, scatter-conditional's behavior in the presence of aliasing is well-defined—only one of the aliased element updates will succeed, indicated by the output mask. Since both gather-linked and scatter-conditional have output masks, the alias detection and resolution can be implemented as part of either instruction.

In Section 3.1, we detail the two new instructions. We discuss some of the implications of our proposed ISA additions in Section 3.2. Then, we propose a specific hardware implementation of the instructions in Section 3.3, and provide a detailed example of the proposed scheme in Section 3.4.

3.1 ISA Extensions

We propose two new instructions for gather-linked and scatter-conditional:

```
1.  vgatherlink   Fdst, Vdst, base, Vindx, Fsrc
2.  vscattercond  Fdst, Vsrc, base, Vindx, Fsrc
```

Instruction vgatherlink gathers up to SIMD-width data elements under mask Fsrc from SIMD-width (not necessarily contiguous or unique) memory locations, base[Vindx[0]], ..., base[Vindx[SIMD_WIDTH-1]], and stores them into destination register Vdst. Analogous to its load-linked counterpart, this instruction also reserves the memory locations of the gathered data elements. It may succeed for only a subset of the memory locations; it sets the bits of the output mask, Fdst, corresponding to these elements. It clears the bits of Fdst corresponding to the failed elements or the elements whose Fsrc bits are not set.

Instruction vscattercond scatters up to SIMD-width contiguous data elements under mask Fsrc from source register Vsrc, into SIMD-width (not necessarily contiguous or unique) memory locations base[Vindx[0]], ..., base[Vindx[SIMD_WIDTH-1]]. Analogous to store-conditional, vscattercond stores only to memory locations retaining their reservation from the most recent vgatherlink instruction. It may therefore succeed for only a subset of elements, for which it sets the corresponding bits in the output mask, Fdst, and clears all other bits in the output mask.

As explained earlier, if element aliasing exists within a group of SIMD-width elements, vscattercond detects the aliasing and clears the corresponding output mask bits for all but one of the affected elements. An implementation where vgatherlink performs alias detection and resolution is equally valid.

Figure 3(A) shows our previous histogram example (Figure 2) using vgatherlink and vscattercond.

We load SIMD-width elements from the input array Minput into vector register Vinput (line 3), which we use to compute the index vector, Vbins, into a global histogram array Mbins (line 5). We initialize the mask register FtoDo to all 1's (line 6) to indicate that the SIMD reduction should be performed on all SIMD-width vector elements.

We then loop (lines 7-15) until we have completed the histogram update on all SIMD-width elements. On line 10, we use vgatherlink to gather data elements from Mbins into the vector register Vtmp, and set mask Ftmp for the successfully gathered and linked elements. We increment the

A: Using GLSC to perform the reduction directly

```
 1:  for( i = 0; i < numPixels; i += SIMD_WIDTH) {
 2:    // Load the next SIMD_WIDTH inputs into Vinput
 3:    vload Vinput, &Minput[i];
 4:    // Compute the bins
 5:    vmod Vbins, Vinput, numBins;
 6:    FtoDo = ALL_ONES;
 7:    do {
 8:      // Do remaining elements specified by FtoDo
 9:      Ftmp = FtoDo;
10:      vgatherlink Ftmp, Vtmp, Mbins, Vbins, Ftmp;
11:      vinc Vtmp, Vtmp, Ftmp; // Increment bins
12:      vscattercond Ftmp, Vtmp, Mbins, Vbins, Ftmp;
13:      // Record elements that processed successfully
14:      FtoDo ^= Ftmp;
15:    } while (FtoDo != 0);
16:  }
```

B: Using GLSC to implement locks for critical sections

```
 1:  Vzero = { 0, 0, 0, ... };
 2:  Vone  = { 1, 1, 1, ... };
 3:
 4:  // Implements test-and-set lock (0 => available)
 5:  #define VLOCK(MlockArray, Vindex, F)  {
 6:    // Gather-linked locks indicated by F
 7:    vgatherlink Ftmp1, Vtmp, MlockArray, Vindex, F;
 8:    // Determine which locks are avaiable
 9:    vcompareequal Ftmp2, Vzero, Vtmp, Ftmp1;
10:    // Attempt to obtain available locks
11:    vscattercond F, Vone, MlockArray, Vindex, Ftmp2;
12:    // F now indicates locks acquired successfully
13:  }
14:
15:  #define VUNLOCK(MlockArray, Vindex, F) {
16:    // Free the locks indicated by F
17:    vscatter Vzero, MlockArray, Vindex, F;
18:  }
19:
20:  for( i = 0; i < numPixels; i += SIMD_WIDTH) {
21:    // Load the next SIMD_WIDTH inputs into Vinput
22:    vload Vinput, &Minput[i];
23:    // Compute the bins
24:    vmod Vbins, Vinput, numBins;
25:    FtoDo = ALL_ONES;
26:    do {
27:      // Do remaining elements specified by FtoDo
28:      Ftmp = FtoDo;
29:      VLOCK( MlockArray, Vbins, Ftmp)
30:      // Call function that increments the specified
31:      // bins using SIMD instructions
32:      updateFn( &Mbins[0], Vbins, Ftmp);
33:      VUNLOCK( MlockArray, Vbins, Ftmp)
34:      // Record elements that processed successfully
35:      FtoDo ^= Ftmp;
36:    } while (FtoDo != 0);
37:  }
```

Naming Conventions: Names starting with M refer to memory locations, those starting with F refer to mask registers (Section 2.1), and those starting with V refer to vector registers. The mask register is a bitmask with SIMD_WIDTH bits. ALL_ONES is a binary immediate value containing SIMD_WIDTH ones.

Figure 3. Pseudo-code to perform parallel histogram using GLSC.

elements of Vtmp on line 11. On line 12, we use vscattercond to scatter the updated Vtmp back to the shared Mbins data structure. We do this under mask to ensure that we only attempt this operation for elements that were successfully gathered and linked, and that we have not updated in a previous iteration. Vscattercond clears Ftmp for the elements it failed to scatter. The final xor operation (line 14) updates the mask FtoDo so that it only holds 1's corresponding to elements that have not yet been updated.

Besides enabling SIMD-friendly parallel reductions, vgatherlink and vscattercond can be used for acquiring locks in SIMD. We demonstrate this with the same histogram example. In this example (Figure 3(B)), we use fine-grained locking so that each bin is protected by a separate lock variable. In each iteration of the inner loop (lines 26-36), we acquire a subset of the locks corresponding to a set of SIMD-width elements, update the elements for which we acquire the lock, and release the locks. It is possible that due to lock contention with other threads no locks will be acquired in a given iteration of the while loop. This will result in the updateFn (line 32) acting as a NOP because the mask will have all 0's.

3.2 ISA Implications

Our definition of vgatherlink and vscattercond is flexible. For example, in the above example, we prevent deadlock by proceeding with execution on a subset of elements rather than waiting until we hold all SIMD-width locks simultaneously. However, we could choose to acquire all SIMD-width locks before performing the updates, and prevent deadlock another way.

In addition, our definition also provides design freedom for trading-off between various hardware implementations. In contrast to scalar load-linked, which always succeeds, vgatherlink may fail on a subset of elements. A hardware designer may choose to have vgatherlink fail on an element in a number of situations in order to simplify the design or to deliver better performance. These include: (a) Another thread has already linked a cache line containing one of the elements, (b) Bringing one of the elements into the cache will evict an already linked line (e.g., due to limited associativity), and (c) The latency for accessing the element is higher than others in the same set (e.g., it is a cache miss).

Allowing vgatherlink to fail in some cases can reduce the amount of contention. In particular, if we never allow vgatherlink to fail, we may need to hold the reservations for some elements for a long time. This increases the chance that another thread will invalidate those elements' reservations. Proceeding with only the subset of elements that we can quickly gather reduces the chances for contention on those elements.

Our handling of element aliasing simplifies vectorization. We could require that the software guarantee uniqueness for all sets of SIMD-width elements. However, this forces a programmer to partition elements (or locks) into subsets of unique elements prior to entering the main computation loop, which may be expensive. For example, when multiple elements are simultaneously inserted into a tree, the programmer may not know at compile time which elements are unique. Instead, with our definition of how element aliasing is handled, this overhead can be avoided, and we make programming easier.

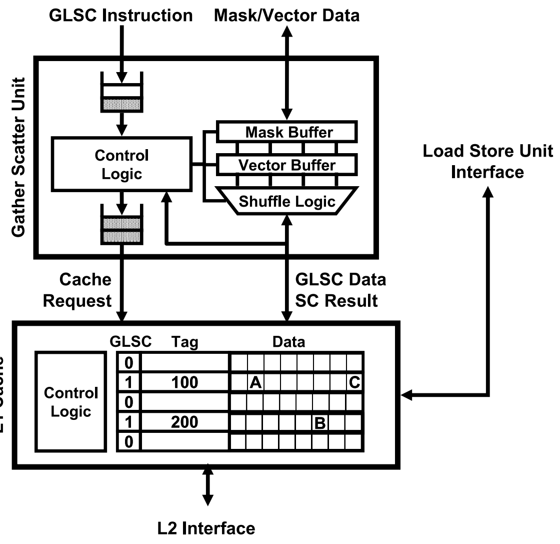

Figure 4. Gather-linked scatter-conditional microarchitecture

Additionally, supporting partial (or even complete) failures allows vgatherlink/vscattercond to gracefully handle exceptions such as page faults. The hardware designer can choose whether to deliver an exception when first encountered, or to group the exceptions together by initially clearing mask bits corresponding to the vector elements that encounter exceptions.

3.3 Gather-Linked Scatter-Conditional Architecture Design

We propose augmenting the gather/scatter unit (GSU) and L1 cache to implement the vgatherlink and vscattercond instructions as described in Section 3.1.

The GSU treats a gather-linked as a gather, except it sends load-linked requests to the L1 cache instead of normal loads. Similarly, the GSU sends store-conditional requests instead of normal stores. In addition, the GSU assembles and stores the output mask for these operations based on success or failure of the individual requests.

We extend the L1 cache tag structure with a single `GLSC` entry per cache line. A `GLSC` entry contains two fields: a valid bit and a hardware thread ID (to distiguish among the SMT threads on the same core). For gather-linked operations, some of the L1 requests may fail (for the reasons described in Section 3.1). For each request that succeeds, the cache updates the corresponding GLSC entry (the valid bit is set and the requester's thread ID is filled), the GSU sets the corresponding bit in the output mask, and the GSU places the data in the destination register. For scatter-conditional operations, an individual store-conditional request succeeds if the corresponding GLSC entry valid bit is set and the GLSC entry thread ID matches the requester's thread ID. In our implementation, this will be true if the corresponding cache line has not been modified by an intervening write or evicted since it was successfully linked by a matching load-linked request from a gather-linked. On store-conditional success, the cache clears the GLSC valid flag, modifies the data in the cache line, and the GSU sets the corresponding bit in the output mask.

The storage overhead for the GLSC entries is (1 + # of SMT threads) bits per cache line, or if we encode the thread ID, 1 + $\log_2$(# of SMT Threads) bits per cache line. For example, if the core supports 4-way SMT and the cache line size is 64 bytes, a GLSC entry is three bits per cache line, which is less than 1% of the data size.

An alternative implementation of the GLSC entries would be to hold them in a fully associative buffer. The buffer would additionally hold a tag for each GLSC entry. The number of entries in this buffer could vary from one to SIMD-width $\times$ # of SMT threads, and so could be made quite small. The buffer would need to be accessed whenever a GLSC entry is accessed (i.e., on load-linked and store-conditional requests, on cache line evictions, and on all normal stores since they might clear a GLSC valid bit).

3.4 Example

We now illustrate the behavior of the hardware when executing vgatherlink and vscattercond instructions on a processor with 4-wide SIMD (Figure 4). The input mask value to the vgatherlink instruction is 1011 (i.e., the second element is ignored), and the addresses of the first (A), third (B), and fourth (C) elements are 104, 220, and 128, respectively. During the execution of vgatherlink, the GSU sends a load-linked request for cache line 100 for the first element (A) and cache line 200 for the third element (B). For the fourth element (C), it does not send a load-linked request because it is located in the same cache line as A. The L1 cache sets the GLSC entry valid bits for the two lines (we do not show the thread IDs in the figure) and returns the lines to the GSU, which assembles a vector (A-BC) and writes it back into the destination register. The GSU also sets the output mask to 1011, indicating success for vgatherlink for A, B, and C. Figure 4 shows the state of the hardware just after executing the vgatherlink instruction.

Assume that the GLSC entry valid bit of cache line 200 is cleared by a write from another thread before the vscattercond executes. During the execution of vscattercond, the GSU sends a single store-conditional request to cache line 100 for elements A and C and another store-conditional request to cache line 200 for element B. The L1 cache checks the GLSC entry valid bits (and thread IDs) for lines 100 and 200. Since the bit is still set for line 100, it updates the contents of A and C and returns a notice of success to the GSU. However, the GLSC entry valid bit has been cleared for line 200, so the cache discards the new value of B and returns a notice of failure to the GSU. The GSU writes the output mask (1001). The application needs to retry this sequence until the update for element B succeeds.

4 Experimental Framework

4.1 System Modeled

We use a cycle-accurate, execution-driven CMP simulator for our experiments. Table 1 summarizes our system's configuration. Cores issue instructions in-order, and each core can simultaneously execute instructions from up to four threads. Each core has a private L1 data cache with a hardware stride prefetcher, and all cores share an inclusive L2 cache. Coherence is maintained between the L1s via a directory-based MSI protocol. Each L2 cache line also holds the directory information for the line. The L2 cache is broken into multiple slices and physically distributed across the chip.

To explore the future CMP design space, we vary SIMD width between one and sixteen, and vary the number of cores and hardware threads between one and four[2].

We model gather/scatter latency as follows. Once issued, gather and scatter instructions stall the subsequent instructions from the same thread until memory operations for all elements are complete. Address generation and cache accesses are pipelined to hide latency. The GSU generates at most one cache request per cycle. Hence, it takes SIMD-width cycles to generate all the requests for a single gather or scatter instruction. Multiple requests to the same cache line are combined to reduce contention for the L1 cache ports. The L1 cache arbitrates between the LSU and the GSU, giving the LSU higher priority. As it receives each data reply, the GSU assembles the corresponding parts of the result vector and output mask; thus, the latency for this is mostly hidden. Note that the minimum GLSC latency in Table 1 refers to the best case scenario, i.e., every access from the GSU hits in the L1 cache and has no L1 port contention from the LSU.

We evaluate the performance benefits of GLSC by comparing it to a baseline architecture (referred to as Base). Both configurations use same the simulation parameters presented in Table 1. The difference is in the mechanism

[2]For brevity, we use use the notation **mxn** to refer to a configuration that has **m** cores, **n** threads per core. In this configuration, the parallel benchmark is parallelized to exploit $m \times n$ software threads.

Processor Configuration		Memory Hierarchy		Memory Latency	
Number of Cores	1–4	Private L1 Cache	32KB, 4-way, 64B line	L1 Access Latency	3 cycles
Threads per Core	1–4	Shared L2 Cache	16MB, 8-way, 16 banks	Min L2 Access Latency	12 cycles
SIMD Width	1, 4, 16	GLSC Handling Rate	1 element/cycle	Main Memory Access	280 cycles
Core Issue Width	2	Min GLSC Latency	(4 + SIMD-width) cycles		

Table 1. Simulated system parameters.

used for atomic operations in the benchmarks; for `Base`, the benchmarks use the standard load-linked and store-conditional instructions (Section 2.3), while for `GLSC`, they use the new instructions proposed in this paper.

4.2 Benchmarks

We evaluate our proposal on benchmarks from a key emerging application domain: Recognition, Mining, and Synthesis [9, 11, 23]. All benchmarks were parallelized and vectorized within our group. The benchmarks and their datasets are shown in Table 3.

Most parallel architectures provide some support for scalar atomic operations which is necessary to implement various types of synchronization such as locks, barriers, and condition variables. In general, locks are used to implement critical sections. However, in a number of these benchmarks, critical sections perform only one atomic read-modify-write operation on a single memory location. Such operations can be more efficiently implemented by using instruction sequences that directly utilize atomic read-modify-write operations (referred to as reductions) instead of using locks. Using locks for these benchmarks would exaggerate the benefit of our proposal. Consequently, the atomic reductions are implemented directly using the optimal instruction sequences.

A wide variety of software techniques such as segmented scan [10], pre-hashing, and privatization can be employed to eliminate or to reduce the granularity of synchronization. These techniques involve additional preprocessing or post-processing computation. To ensure an optimal base case, we have used these techniques when they are beneficial. The parallel work is always split between threads to minimize contention on locks and reduction variables. HIP uses privatization. Privatization can accelerate a reduction by eliminating most of its synchronization operations. It involves making a private copy of the accumulators for each thread and combining them at the end of the computation. Updates to the private accumulators do not need synchronization. GPS reorders constraints to improve the efficiency of SIMD execution. Fine-grained synchronization works best for most benchmarks because, for low contention memory locations, atomic read-modify-write operations can be implemented very efficiently on CMPs.

Table 2 describes the benchmarks and Table 3 provides relevant characteristics of the benchmarks including type of critical section, and datasets used for evaluating our proposal.

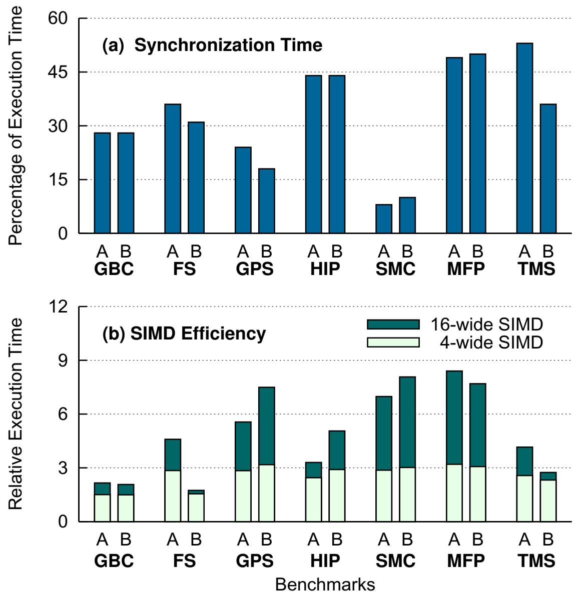

Figure 5. Benchmark behavior with `GLSC` in a 1x1 configuration. A and B bars show the data for the two datasets for that benchmark.

5 Evaluation

5.1 Benefit of GLSC on 4-wide SIMD

Figure 6 compares the performance of `GLSC` with `Base` for two datasets (A and B) for each benchmark. The figure shows the speedup with 4-wide SIMD for four configurations: 1 and 4 cores with 1 hardware thread per core (1x1 and 4x1, respectively) and 1 and 4 cores with 4 threads per core (1x4 and 4x4, respectively).

In most cases, `GLSC` delivers a significant improvement in performance over `Base`. `GLSC` is on average 76% and 54% faster than `Base` for the 1x1 and 4x4 configurations, respectively.

Benchmark characteristics. To better understand the performance benefits of `GLSC`, we first examine the relevant characteristics of the benchmarks. Figure 5(a) shows the percentage of execution time spent in synchronization operations for 1x1 using 1-wide SIMD and `GLSC`. Since 1-wide SIMD is effectively scalar, this is very similar to the amount of time spent in synchronization operations when using `Base`. These benchmarks spend a significant fraction

Benchmark	Atomic Operation	Datasets Description	
		Dataset A	Dataset B
GBC	Single Lock Critical Section	649 objects in 8191 grid cells	5649 objects in 65521 grid cells
FS	Floating-point Subtract	2171x5167 with 2.47% density	3136x9408 with 15.06% density
GPS	Multiple Lock Critical Section	625 objects	1600 objects
HIP	Integer Increment	480x480 image of cars	480x480 image of people
SMC	Floating-point Add	32K particles	256K particles
MFP	Multiple Lock Critical Section	1500 nodes and 6800 edges	3888 nodes and 18252 edges
TMS	Floating-point Add	21616x67841 with 0.87% density	209614x41177 with 0.01% density

Table 3. Benchmarks characteristics. Single vs. multiple lock critical sections refers to the number of locks acquired for each SIMD-element worth of work performed in a critical section.

of their time in synchronization operations. Without support such as GLSC for atomic vector operations, the synchronization portion of the code will remain non-vectorized, which fundamentally limits the SIMD efficiency. In contrast, GLSC allows the synchronization code to get faster with increased SIMD width. To quantify SIMD efficiency, Figure 5(b) shows the speedup due to 4-wide and 16-wide SIMD over 1-wide SIMD for each of the benchmarks. These benchmarks have significant SIMD parallelism and all of them can derive performance benefit from short vectors. With 4-wide SIMD, these benchmarks are on average 2.6x faster than with 1-wide SIMD. A few benchmarks can also get significant additional performance with 16-wide SIMD. On average, the benchmarks are 5x faster with 16-wide SIMD compared to 1-wide SIMD.

Detailed analysis. Table 4 presents statistics that help to explain the performance benefits from GLSC. The benefits are attributed to three main sources.

- As shown in the third column, GLSC results in a large reduction in the number of dynamic instructions compared to Base (33.8% on average).
- GLSC overlaps the memory requests from vgatherlink and vscattercond instructions. When two or more of these accesses miss in the L1 cache, their miss latencies will be overlapped. This can reduce the stalls due to cache misses. In contrast, with Base, atomic operations on memory locations are performed one at a time and do not overlap their L1 cache miss latencies. As shown in Table 4 GLSC incurs 23.4% fewer memory stalls than Base on average.
  Base could also overlap misses by explicitly issuing software prefetches, but this involves additional overheads. Not only does it require additional instructions to issue prefetches (and sometimes repeat address computations) but it also results in additional pressure on the L1 cache. For instance, we modified TMS to explicitly issue prefetches with Base, but performance decreased.
- GLSC reduces the number of L1 accesses. The GSU issues only one request for each distinct cache line specified by a vgatherlink or vscattercond instruction. Most benchmarks see a small reduction in the number of L1 accesses (2.5% on average), while FS and HIP see a significant reduction (16% and 17%, respectively).

The effectiveness of GLSC is also determined by the fraction of SIMD elements that succeed for each dynamic instance of vgatherlink and vscattercond. For instance, with 4-wide SIMD, even if three of the four elements succeed, in most cases, the entire vgatherlink and vscattercond sequence needs to be repeated for the remaining element.

The last two columns in Table 4 show the GLSC element failure rate for the 1x1 and the 4x4 configurations. There are three main sources of element failures. First, if the number of cache requests which map into the same set exceeds the associativity of the cache, one or more of the requests will fail due to a set conflict. In our configuration, the SIMD width is equal to the cache associativity (4-way), hence such failures will not occur. Second, when there is element aliasing, only one of the aliased elements will succeed. In the 1x1 configuration, this is the only source of failure. Such conflicts are significant in only two of the benchmarks: GBC and HIP. In a number of other benchmarks, aliasing may be more common with other datasets. GLSC element failure can also occur when two threads running on the same or different cores try to perform atomic operations on the same memory location simultaneously. The difference between the element failure rate in 4x4 and the 1x1 configurations provides a rough estimate of element failures due to conflicting atomic operations by multiple threads.[3] In all benchmarks we study, this is fairly small (less than 0.1%).

Finally, HIP is an exception in that it performs slightly better with Base than with GLSC for one of the datasets (see Figure 6). This is due to a combination of two factors. First, HIP uses privatization (Section 4.2). Consequently, it does not need atomic operations when updating its private copy and so Base can use a simpler instruction sequence

[3]There are other possible reasons for element failures including cache line evictions. Their contributions are negligible in our experiments.

Benchmark	Dataset	Reduction with GLSC on 4x4					GLSC failure rate	
		Instructions	Memory Stalls	L1 Accesses			1x1	4x4
GBC	A	15.94 %	21.80 %	10.23%	of	2.17 %	31.08 %	31.18 %
	B	13.81 %	17.36 %	11.46%	of	1.94 %	34.24 %	34.29 %
FS	A	75.43 %	59.19 %	68.03%	of	24.67 %	0.00 %	0.05 %
	B	60.53 %	21.86 %	61.72%	of	10.30 %	0.00 %	0.33 %
GPS	A	18.79 %	-3.83 %	39.00%	of	1.91 %	0.00 %	0.02 %
	B	20.76 %	-5.46 %	49.09%	of	1.88 %	0.00 %	0.00 %
HIP	A	28.12 %	n/a %	34.39%	of	47.26 %	34.97 %	34.97 %
	B	39.85 %	n/a %	41.88%	of	47.15 %	19.62 %	19.62 %
SMC	A	29.96 %	45.48 %	67.83%	of	3.39 %	0.00 %	0.02 %
	B	33.59 %	49.93 %	68.34%	of	3.95 %	0.00 %	0.01 %
MFP	A	18.85 %	12.46 %	52.40%	of	2.67 %	0.00 %	0.04 %
	B	17.66 %	15.08 %	55.09%	of	2.47 %	0.00 %	0.01 %
TMS	A	47.10 %	56.02 %	21.47%	of	20.95 %	0.00 %	0.02 %
	B	52.28 %	37.62 %	33.63%	of	23.46 %	0.00 %	0.00 %

The **Instructions** column shows the reduction in the number of instructions for `GLSC` compared to `Base`. The **Memory Stalls** column shows the reduction in the number of stall cycles due to memory accesses for `GLSC` compared to `Base`. This number is *n/a* for HIP because its implementation with `GLSC` and `Base` are different. The **L1 Accesses** column shows two numbers for each row. The second number shows the percentage of L1 accesses due to atomic operations (i.e., `GLSC`). The first number shows the percentage reduction in the number of L1 accesses due to atomic operations because of cache line reuse in the gather-scatter unit. The last two columns show the percentage of atomic operations that fail due to aliasing or collisions between threads.

Table 4. Analysis of GLSC.

for this. The instruction sequence with `GLSC` is more complex since it needs to handle element aliasing (see Figure 3). For 1-wide SIMD, `GLSC` requires 28% more instructions than `Base`. Second, HIP experiences a high element failure rate since the elements map to a small number of locations. If the element failure rate was low, the 28% extra instructions can be offset with SIMD parallelism on 4-wide SIMD. In particular, we tested an input composed of random numbers. This has a much lower element failure rate than either of our experimental inputs, and `GLSC` is 26% faster on 4-wide SIMD than `Base`.

5.2 Microbenchmark

We use a microbenchmark to further analyze the three sources of performance benefits in `GLSC` discussed above. The microbenchmark consists of a loop with a fixed number of iterations, in which the threads operate on an array of counters. In each iteration, a thread selects an index randomly and atomically increments the corresponding counter. The array is chosen to be small enough to fit in the L1 and the cache is warmed up prior to taking measurements.

The sequence of indices is randomly generated, but has certain properties to allow us to study the benefits of `GLSC` in different cases. The indices are precomputed to avoid introducing artifacts during measurements. Note that even with microbenchmarks, it is difficult to completely isolate the three sources of performance improvements. We examine four scenarios.

Scenario A highlights the benefits from overlapping L1 misses with `GLSC`, although `GLSC` also benefits here from executing fewer instructions. For this scenario, each of the SIMD-width addresses is in a distinct cache line, i.e., there is no element aliasing and each SIMD operation needs SIMD-width cache lines. Further, when multiple threads are running, the needed cache lines are often in another core's L1 cache. Finally, the number of counters is large enough that two threads rarely try to update the same counter at the same time.

In the remaining scenarios, each thread operates on a disjoint subset of indices and each thread will always hit in its L1 cache. Thus, `GLSC` will no longer derive any benefits from overlapping L1 misses.

Scenario B highlights the benefits from reducing the number of instructions as well as accesses to the L1. To do this, the SIMD-width addresses are to different locations on the same cache line. This ensures two things: there is no element aliasing, and only one L1 access is needed.

Scenario C highlights the savings from the reduction in the number of instructions. To do this, Scenario B is modified so that each of the SIMD addresses is to a different cache line.

Scenario D highlights the case where there are no advantages available to `GLSC`. To do this, Scenario B is modified so that each of the SIMD addresses is the same, i.e., there is no SIMD parallelism available and `GLSC` has to serially process each of the SIMD elements.

Figure 7 shows the results of the four scenarios. `GLSC` improves performance due to a combination of all three factors with fewer accesses to L1 being the smallest of the

Grid-based Collision Detection (**GBC**) is the broad phase of collision detection, which determines potentially colliding pairs of objects using a multi-resolution grid [12]. It maps each object into (potentially multiple) grid cells such that potentially colliding objects occupy the same grid cell. The objects in each cell are stored in a linked list. The objects are divided evenly amongst threads, and each thread processes multiple objects using SIMD. Insertion of objects into linked lists is protected with locks.
Forward Triangular Solve (**FS**) is an important step in a direct sparse linear solver [24], which solves a sparse lower triangular system of equations $Lx = y$. The matrix is divided into contiguous dense subblocks. Thread-level parallelism is exploited across the subblocks using a dependence graph, which specifies the order in which these subblocks are processed. SIMD parallelism is exploited within each subblock which involves dense matrix-vector multiplication and atomic reductions into a shared vector.
Game Physics Solver (**GPS**) iteratively solves a set of force equations in a game physics constraint solver [26]. These equations are represented as a set of constraints, each of which updates one or two distinct objects. The constraints are divided evenly amongst threads. Multiple constraints within a thread are updated using SIMD. Constraints must be updated atomically using locks. To avoid SIMD aliasing (for regular scatters), constraints within each thread are reordered into groups of independent constraints.
Histogram for Image Processing (**HIP**) generates a histogram of colors of pixels in an image for image-based retrieval [27]. The similarity of two images is defined by the similarity of the two color histograms. To parallelize, an image is row-wise partitioned among threads. Each thread updates its own local copy of the histogram using SIMD reductions and a global merge is performed at the end. Due to privatization, HIP does not utilize the atomicity feature of `GLSC`, but takes advantage of its alias detection.
Surface Extraction using Marching Cubes (**SMC**) extracts and renders a surface for fluid simulation using particles in a uniform 3D grid of nodes [22]. The nodes store density values which represent the 3D scalar field. Each particle updates the density of nodes in its local neighborhood and then extracts the fluid surface. We parallelize SMC by dividing the particles among threads. Multiple particles are processed using SIMD. Atomic SIMD reductions perform simultaneous updates of the nodes.
Maxflow Push (**MFP**) is a key computational kernel used in graph algorithms such as parallel push-relabel maximum flow [14]. The flow is repeatedly pushed from one node to another within the graph. Our parallel implementation evenly divides graph nodes among threads and pushes the flow within each partition using SIMD. The push operations must be performed atomically and use SIMD locks.
Transpose Matrix-Vector Multiply (**TMS**) is an important linear algebra kernel which performs the operation $y = A^T x$, where A^T is a transpose of sparse matrix A, and x and y are dense vectors. Algorithmically, each non-zero element A_{ij} is multiplied by x_i and the result is reduced into y_j. We parallelize TMS by evenly dividing nonzero elements of A amongst threads. Multiple elements are processed using SIMD and use atomic reductions to update the destination vector y.

Table 2. Description of Benchmarks.

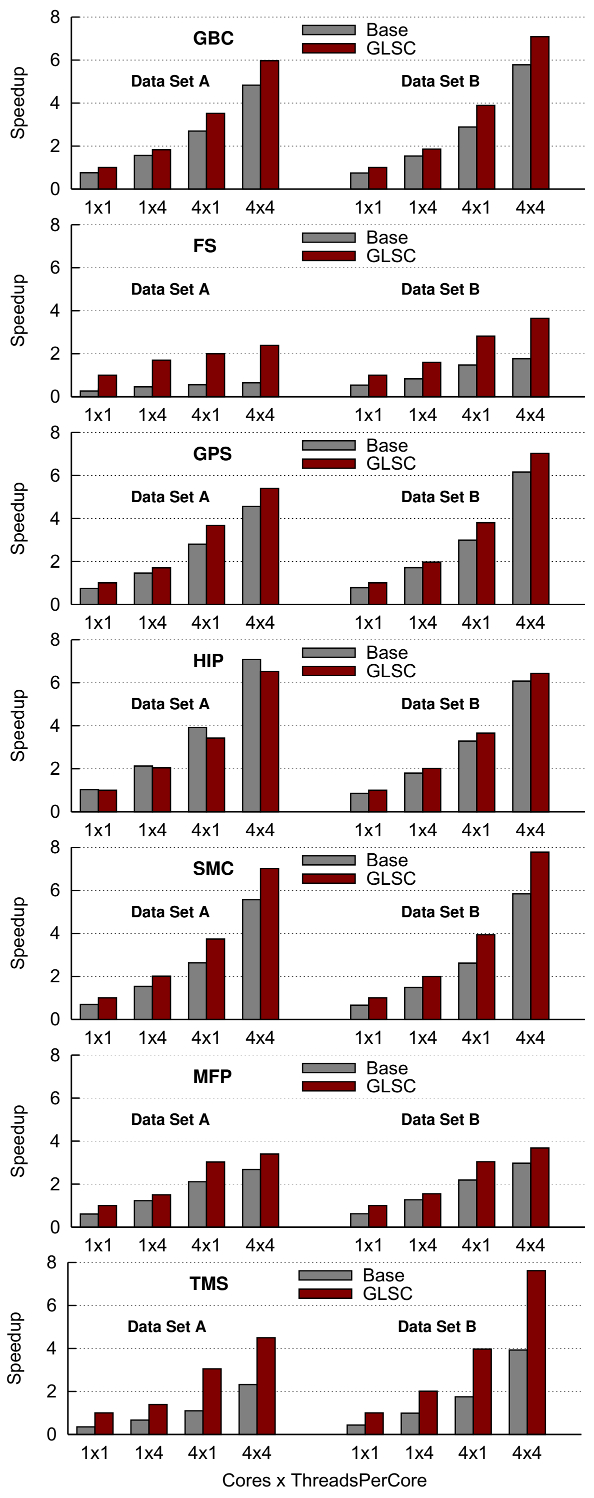

Figure 6. Normalized performance for 4-wide SIMD: In each graph, performance is normalized to execution time of the benchmark in the 1x1 GLSC configuration for that dataset.

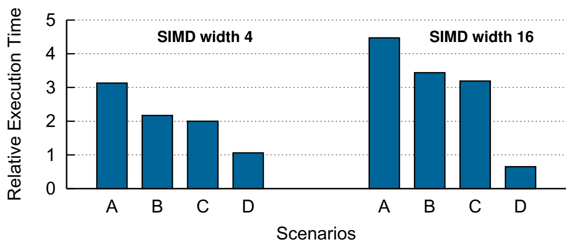

Figure 7. Benefit of GLSC on a microbenchmark for 4-wide and 16-wide SIMD in the 4x4 configuration. Each bar shows the ratio of the execution times of Base to GLSC.

three contributors. For 16-wide SIMD, GLSC is slower than Base for scenario D; although the instruction counts are roughly the same, GLSC instructions on average incur longer latency—even though the SIMD elements are the same, the GLSC hardware repeatedly generates and compares the addresses.

5.3 Sensitivity to SIMD width

Figure 8 compares the change in execution time with GLSC compared to Base for 1-wide SIMD and 16-wide SIMD on the 4x4 configuration.

1-wide SIMD. When the SIMD width is changed to one, GLSC does not provide any of the three sources of performance improvement discussed above. Thus, 1-wide SIMD exposes any overheads (additional instruction latency or extra instructions needed to manipulate the masks) that GLSC might introduce. On average, GLSC has the same performance as Base. The benchmarks for GLSC and Base use similar code sequences. For 1-wide SIMD, the dynamic instruction streams are therefore only slightly different. In some cases, the GLSC version of the benchmark uses fewer instructions per iteration than the Base version, and in some cases, a few more. The results indicate that GLSC will typically not have worse performance than Base even if each iteration only successfully completes an atomic operation on a single SIMD element.

16-wide SIMD. When the SIMD width is changed to 16, the benefit of GLSC increases as expected (103% on average). This is more pronounced for those benchmarks that have higher SIMD efficiency (Figure 5(b)). For benchmarks GPS, SMC, and MFP, GLSC is on average faster than Base by 55%, 115%, and 100%, respectively.

6 Related Work

Many microprocessors and research proposals have included hardware support for efficiently performing scalar atomic read-modify-write operations. IBM 370 included the atomic compare&swap instruction [3], while Intel x86 allows some instructions (like integer increment, exchange, compare&exchange, etc.) to have a "lock" prefix to make them atomic. MIPS [19] and PowerPC [6] offer hardware support for load-linked and store-conditional instructions. The NYU Ultracomputer [15] was the first to propose support for fetch-and-add operations in the memory controller. Such support is especially beneficial in the presence of contention. A number of other machines including SGI Origin [21] and Cray T3E [25] have provided such support. However, none of the above architectures provide hardware support for vector versions of atomic read-modify-write operations.

Scatter-add [8] extends the fetch-and-add mechanism to support parallel reductions on data parallel architectures. The Cray Black Widow [7] provides similar hardware support for atomic vector operations at the memory controller. Other proposals [13, 17] extend the fetch-and-add mechanism to streams of memory locations. In contrast to our proposal, these proposals require additional floating-point hardware in the memory controller and support only limited forms of reductions (typically only additions). They also require imprecise floating-point exception semantics. Finally, they do not address the implementation of locks.

Recently introduced NVIDIA GPUs (8500 and 8600) support concurrent atomic increments to shared device memory using a single SIMD instruction [5]. However, even in the presence of a single alias within a SIMD operation, our measurements show that its performance degrades dramatically (orders of magnitude). In addition, there is no efficient mechanism to implement vector locks.

There is a large body of work on transactional memory (TM) [18] ([20] discusses much of the work in this area). TM schemes have been proposed to allow atomic execution of a transaction, which is a sequence of instructions. While transactional memory provides an easy and intuitive programming model, it is overly restrictive for SIMD execution. SIMD execution of an atomic section is often parallel execution of multiple *independent* atomic sections (corresponding to each position in the SIMD vector). In contrast, a transaction in transactional memory treats all memory accesses as part of one atomic unit—if any of the memory accesses in a transaction conflict with another thread, the entire transaction aborts. Furthermore, current TM schemes provide no support to detect conflicts between the multiple atomic operations being executed in a SIMD fashion.

7 Conclusions

While vector parallelism can be efficiently supported in today's processors via SIMD hardware, SIMD efficiency is compromised in the presence of irregular data access patterns and atomic operations. We therefore introduce two new instructions, gather-linked and scatter-conditional (GLSC). GLSC delivers significant improve-

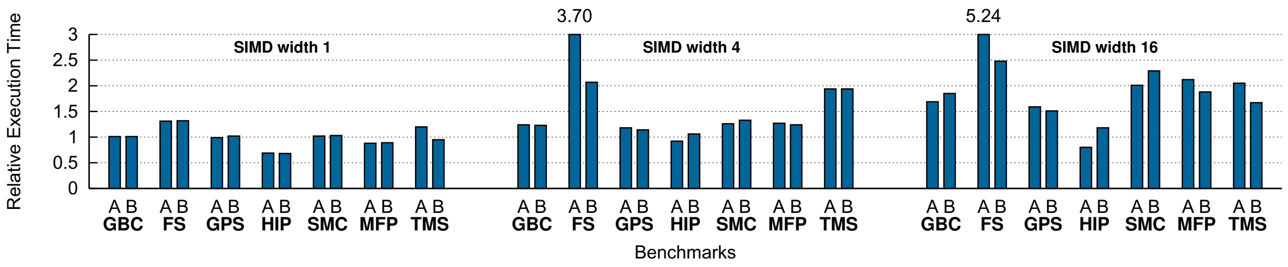

Figure 8. Benefit of GLSC for 1-wide, 4-wide, and 16-wide SIMD in the 4x4 configuration. Each bar shows the ratio of the execution times of `Base` to `GLSC`.

ment in performance over a baseline SIMD architecture with gather/scatter support. Using 4-wide SIMD, `GLSC` executes 76% faster on average for runs with one thread and 54% faster on average for runs with 4 cores and 4 hardware threads per core. The performance improvement comes from three key factors: (1) `GLSC` reduces the dynamic instruction count, (2) `GLSC` overlaps L1 misses for atomic operations, and (3) `GLSC` can reduce the number of L1 accesses. We find that the majority of the benefit comes from the first two factors. We also show that as SIMD widths increase in the future, the already significant benefits from `GLSC` will grow substantially for applications with high SIMD efficiency.

Acknowledgments

We thank Pradeep Dubey and the anonymous reviewers for their valuable feedback.

References

[1] AMD Opteron Processor Family. http://www.amd.com/.

[2] CRAY-2 Engineering Maintenance Manual. Cray Research Inc., Publication No. HM-2032, 1985.

[3] IBM Corporation. System/370 Principles of Operation. IBM Corporation, 1983.

[4] Intel Pentium/Core/Core 2 Processors. http://www.intel.com/.

[5] NVIDIA CUDA (Compute Unified Device Architecture). http://www.nvidia.com/, 2007.

[6] PowerPC User Instruction Set Architecture (Book I). 2003.

[7] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, M. Bye, and G. Schwoerer. The cray blackwidow: A highly scalable vector multiprocessor. In *Supercomputing*, 2007.

[8] J. Ahn, M. Erez, and W. J. Dally. Scatter-add in data parallel architectures. In *HPCA*, 2005.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Princeton University Technical Report TR-811-08, 2008.

[10] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing*, 1990.

[11] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Technology@Intel Magazine*, February 2005.

[12] C. Ericson. *Real-time Collision Detection*. Morgan-Kauffman, San Francisco, CA, USA, 2003.

[13] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker. Active memory operations. In *ICS*, 2007.

[14] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[15] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM TOPLAS*, 5(2):164–189, 1983.

[16] M. Gschwind. Chip multiprocessing and the Cell broadband engine. In *ACM Computing Frontiers*, pages 1–8, 2006.

[17] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. In *PACT*, 2007.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[19] G. Kane and J. Heirich. MIPS RISC Architecture: reference for the R2000, R3000, R6000 and the new R4000 instruction set computer architecture. Prentice-Hall, 1992.

[20] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[21] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, 1997.

[22] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, 1987.

[23] J. Rattner. Cool Codes for Hot Chips: A Quantitative Basis for Multi-Core Design. HotChips Keynote, 2006.

[24] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2005.

[25] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS*, 1996.

[26] R. Smith. Open dynamics engine v0.5 user guide. http://www.ode.org/ode-latest-userguide.html, 2006.

[27] J. Z. Wang. *Integrated Region-Based Image Retrieval*. Kluwer Academic Publishers, Boston, MA, USA, 2001.