Exposing Memory Access Patterns to Improve Instruction and Memory Efficiency in GPUs

NEAL C. CRAGO, MARK STEPHENSON, and STEPHEN W. KECKLER, NVIDIA

Modern computing workloads often have high memory intensity, requiring high bandwidth access to memory. The memory request patterns of these workloads vary and include regular strided accesses and indirect (pointer-based) accesses. Such applications require a large number of address generation instructions and a high degree of memory-level parallelism. This article proposes new memory instructions that exploit strided and indirect memory request patterns and improve efficiency in GPU architectures. The new instructions reduce address calculation instructions by offloading addressing to dedicated hardware, and reduce destructive memory request interference by grouping related requests together. Our results show that we can eliminate 33% of dynamic instructions across 16 GPU benchmarks. These improvements result in an overall runtime improvement of 26%, an energy reduction of 18%, and a reduction in energy-delay product of 32%.

CCS Concepts: • **Computer systems organization → Parallel architectures**;

Additional Key Words and Phrases: GPU architecture, vector instruction sets, vector memory instructions

ACM Reference format:
Neal C. Crago, Mark Stephenson, and Stephen W. Keckler. 2018. Exposing Memory Access Patterns to Improve Instruction and Memory Efficiency in GPUs. *ACM Trans. Archit. Code Optim.* 15, 4, Article 45 (October 2018), 23 pages.
https://doi.org/10.1145/3280851

1 INTRODUCTION

GPUs continue to be widely used in throughput computing, leveraging high compute density and energy efficiency. While traditionally used for high-performance computing, GPUs are now also being used widely for data center applications and the emerging field of machine learning. Each of these fields continue to demand more compute to solve larger and more complex problems, requiring computer architects to design subsequent generations of GPUs with more capabilities while remaining at the same or lower power budget. While this challenge can be partially met by leveraging smaller manufacturing processes and scaling existing architectures to fully utilize larger transistor budgets, exploiting application behavior remains a large opportunity.

One such opportunity is to leverage GPU application memory access patterns to improve performance and reduce energy. While GPU applications often have well-defined memory access patterns, modern GPUs currently have limited support for exploiting those patterns. The most prominent existing support for patterns lies in memory access coalescing across a group of threads

New Paper, Not an Extension of a Conference Paper.
Authors' addresses: N. C. Crago, M. Stephenson, and S. W. Keckler, NVIDIA Corporation, 2788 San Tomas Expressway Santa Clara, CA 95051.

ACM 1544-3566/2018/10-ART45
https://doi.org/10.1145/3280851

known as a *warp*. When a memory instruction executes and reaches the L1 cache, the set of addresses across the warp is used to access the cache. To maximize performance, the programmer attempts to write the application such that the set of addresses across the warp is contiguous and aligned to a cache line. Such an organization of addresses within a warp promotes the best possible L1 cache performance, allowing cache hits to access a single cache line from the L1 cache while also allowing cache misses to be coalesced into a single miss request. However, coalescing techniques are limited in scope and there are other remaining memory access patterns that can be accelerated.

This article proposes new ISA instructions targeted at common memory access patterns found across a variety of GPU applications. Given the strength of GPU memory access coalescing *across* threads in a warp, the new instructions focus on patterns *within* a single thread. Specifically, we identify strided and indirect memory access patterns that are prevalent in many data-parallel algorithms and architect new vector instructions that exploit those patterns. We implement the instructions in hardware with modifications to the load/store functional unit and warp scheduler, and enable substantial energy savings and performance improvement.

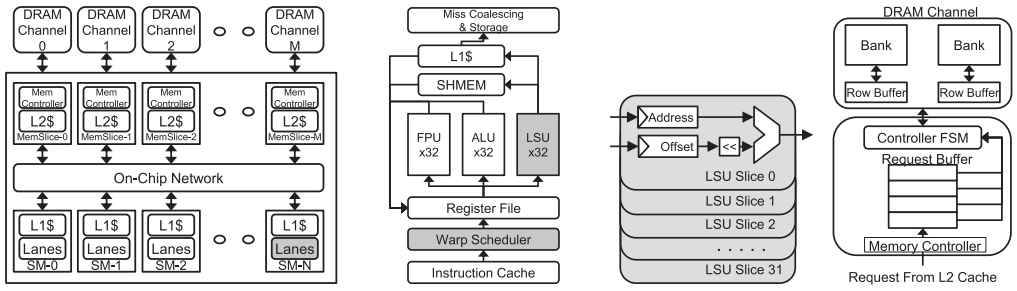
We identify two significant benefits to exploiting memory access patterns using the new GPU instructions: reduced address generation instructions for more efficient execution, and reduced memory interference for better exploitation of memory locality. Both address generation overhead and memory request interleaving are problems unique to GPUs due to the *single-instruction multiple-thread* execution model and throughput-oriented design not found in CPUs. Currently, GPU instructions dedicated to address generation make up a substantial portion of total instructions, up to 50% in some memory-intensive workloads. These instructions are a costly overhead for using the memory subsystem, with each instruction demanding scheduling and fetch resources, reading and writing the register file, and executing in the functional units within the compute pipeline. Similarly, the massive number of warps executing concurrently on GPUs can also lead to missed opportunities to exploit memory locality. While a memory access stream for a single warp may have inherent data locality in the L1 cache, L2 cache, and DRAM row buffers, this locality can be lost as multiple warps arbitrate for shared resources and become interleaved. Leveraging memory access patterns to group together related memory requests can help prevent interference, preserving memory locality and providing substantial runtime and energy consumption improvements.

2 GPU ARCHITECTURE OVERVIEW

Figure 1(a) presents the high-level view of a modern GPU compute architecture. The GPU chip is divided into three parts: the *Streaming Multiprocessors* (SM) made up of compute lanes and L1 cache, the *memslice* banks consisting of L2 cache banks and memory controllers connected to DRAM channels, and the on-chip network.

The SM as depicted in Figure 1(b) consists of instruction fetch and scheduling resources, pipelined functional units, a large register file, an L1 data cache, a miss buffer and coalescing hardware, and a shared memory scratchpad. The SM operates as a *single-instruction multiple-thread* (SIMT) processor, with a single instruction being fetched and executed across a group of 32 threads (known as a *warp*). Each thread executes in lockstep with the rest of the threads in the warp, performing computation and generating memory requests. Warps can cooperatively work together on shared data in a software entity known as a *cooperative thread array* (CTA). Many warps execute concurrently on each SM, competing for shared functional units, L1 cache, and shared memory scratchpad. The SM's *warp scheduler* manages the execution of the collection of warps, interleaving instructions from different warps to tolerate functional unit and memory latency.

The load/store unit generates the memory addresses when a warp issues a load or store instruction on the SM (Figure 1(c)). Depending on the instruction type, either the L1 cache or the



(a) System-Level View (b) Streaming Multiprocessor (SM) (c) Load/Store Functional Unit (d) Memory Controller and DRAM

Fig. 1. GPU system diagram, including detail of the Streaming Multiprocessor (b), SM's load/store functional unit (c), and DRAM memory interface (d). Shaded areas are modified by this work.

shared memory scratchpad receive the addresses. For addresses sent to the L1 cache, cache misses coalesce into the minimum number of cache lines required to service the instruction, and the SM sends the resulting L1 cache miss requests to the on-chip network.

Memory requests from the SMs arbitrate at the on-chip network for service by the memslice banks. Requests that can be immediately serviced by the L2 cache banks return to the SMs, while L2 cache misses enter the memory controller and are stored in the request buffer for DRAM scheduling (depicted in Figure 1(d)). DRAM channels consist of row buffers which act as a cache for the most recently accessed row in a memory bank. The DRAM memory controller prioritizes memory requests that access data currently found in a row buffer (i.e., row buffer hits) using a policy such as *first-ready first-come/first-serve* (Rixner et al. 2000) to maximize memory bandwidth.

3 MEMORY REQUEST OVERHEADS AND INEFFICIENCIES

This section discusses two limitations of current GPUs: the address generation instruction overhead required to make memory requests, and destructive memory request interference at the SM and on-chip network levels. To generate a single load or store request, multiple integer instructions are often required to calculate the address, resulting in additional activity and energy consumption in the SM. Memory request interference occurs at the SM and on-chip network due to arbitration policies promoting fairness among the many concurrently executing warps. These two limitations are a major cause of overhead in GPU application slowdown.

3.1 Address Generation Overheads

Address generation is an expensive side effect of making a memory request. Multiple instructions may be required to generate a single request, each of which requires access to the L1 instruction cache, reads and writes to the main register file, and functional unit activity, which can include expensive logic such as integer multiplication. For GPUs, each thread in a warp must separately generate addresses for its own memory accesses, compounding the overhead significantly. The duplication of address calculation across threads of a warp is a unique challenge for SIMT GPUs as traditional single-instruction multiple data (SIMD) processors such as CPUs only require a single address for vector memory instructions. Therefore, the instructions required for address generation significantly affect SM energy consumption and instruction scheduling slots.

To demonstrate the prevalence of address generation instructions, Listings 1 and 2 present a simple example using vector addition. Listing 1 shows the CUDA kernel code for vector addition,

```
__global__ void
vecadd(float* A, float* B, float* C, int Size)
{
  int idx = blockIdx.x*blockDim.x+threadIdx.x;
  int stride = gridDim.x*blockDim.x;
  for (int i = idx; i < Size; i+= stride)
  {
    C[i] = A[i] + B[i];
  }
}
```

Listing 1. Vector addition kernel in CUDA.

```
...   ...   ...
/*0080*/ IMAD R4.CC, R0, R11, c[0x0][0x140];
/*0088*/ IMAD.HI.X R5, R0, R11, c[0x0][0x144];
/*0090*/ IMAD R8.CC, R0, R11, c[0x0][0x148];
/*0098*/ LD.E R4, [R4];
/*00a0*/ IMAD.HI.X R9, R0, R11, c[0x0][0x14c];
/*00a8*/ IMAD R6.CC, R0, R11, c[0x0][0x150];
/*00b0*/ LD.E R2, [R8];
/*00b8*/ IMAD.HI.X R7, R0, R11, c[0x0][0x154];
/*00c0*/ IADD R0, R3, R0;
/*00c8*/ ISETP P0,PT,R0,c[0x0][0x158],PT;
/*00d0*/ FADD R2, R4, R2;
/*00d8*/ ST.E [R6], R2;
/*00e0*/ @P0 BRA 0x80;
...   ...   ...
```

Listing 2. Vector addition kernel inner loop in NVIDIA SASS assembly.

where the arrays are of size *Size*, and each thread executes a loop across a number of elements. Listing 2 shows the NVIDIA SASS assembly code for the loop. For vector addition, multiple 32-bit integer multiply add instructions (IMAD) are primarily used for generating the 64-bit addresses within the loop as highlighted in **bold**. In total, 46% of the loop body's instructions contribute to address generation.

While non-unrolled vector addition is a simple example, address generation also makes up a significant amount of total dynamic instructions across complex and optimized code. To further illustrate this concept, we instrumented a set of CUDA benchmarks (Section 7.3) using SASSI (Stephenson et al. 2015) to generate dynamic instruction execution histograms by instruction PC. In order to allocate integer instructions into the address generation (Agen), control (Control), and compute (Compute_Int, Compute_FP) categories, we further performed a backtrace using the source registers of relevant instructions. For example, we determined which integer instructions contributed to address generation by inspecting the backtrace of load and store instructions.

Figure 2 presents a breakdown of dynamic instructions by address generation, memory, control, and compute categories. While more compute-intensive benchmarks such as *mriq* and *sgemm* are less affected by address generation, address generation instructions in memory-intensive benchmarks such as *kmeans* and *spmv* can exceed 50% of all dynamic instructions. Overall, the instructions dedicated to address generation dominate, with an average of 42% of all instructions dedicated to address generation across the benchmarks. The vast majority of these address instructions are integer addition and integer multiplication instructions, many of which require at least two reads and one write to the register file.

Given the prevalence of address generation instructions, reducing this instruction overhead is highly desirable from an energy-efficiency perspective. A main contribution of this work is leveraging memory access patterns to significantly reduce this address generation instruction overhead.

3.2 Memory Request Interference

One of the biggest strengths of current GPUs is massive parallelism via many highly threaded compute cores. By supporting many warp contexts within and across SMs, long chip latencies resulting from L1 cache misses can often be tolerated without much impact in functional unit utilization. However, one side effect of having so much parallelism on-chip is that memory requests are often interleaved among warps. Interleaving can happen at both the SM and on-chip network levels, which can cause the loss of L1 cache, L2 cache, and DRAM memory locality. While similar interleaving can occur in CPUs, GPUs can have parallelism orders of magnitude larger leading to a much greater impact on memory locality.

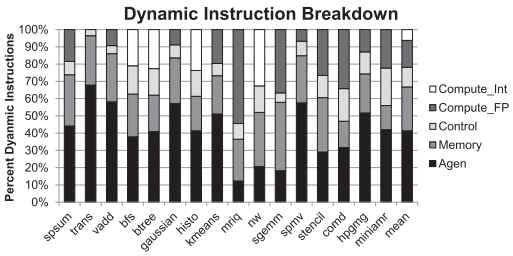


Fig. 2. Overhead of address generation instructions on CUDA benchmarks.

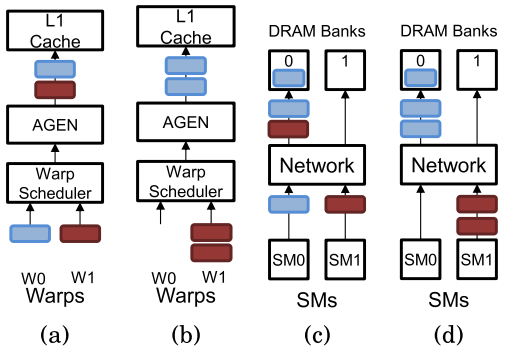


Fig. 3. Memory interference from request interleaving at the SM ((a) and (b)) and the on-chip network ((c) and (d)). Similarly colored messages represent different requests bound for the same DRAM row.

Figure 3 provides high-level examples of how interleaving can result in memory interference and loss of locality. In the figure, memory requests with the same color are different L1 cache line requests that map into the same DRAM row, while memory requests with different colors map to different DRAM rows. To more easily illustrate the phenomenon of memory request interleaving, in these examples we assume a single DRAM channel, a single DRAM row buffer, and that DRAM memory requests are serviced in FIFO order. This simplistic model makes the DRAM channel extremely sensitive to memory request orderings.

Concurrently executing warps in the SM compete for scheduling and execution resources, and the warp scheduler attempts to fairly allocate resources among warps. Figure 3(a) and (b) present the scenario where fairness results in destructive memory request interference. In the scenario, two warps are being actively scheduled on the SM, with Warp 0 having two instructions that generate L1 cache misses that map to the blue DRAM row, and Warp 1 having two instructions that generate L1 cache misses that map to the red DRAM row. Figure 3(a) presents the scheduler arbitrating fairly among the two warps, which results in interference as the two warp requests eventually reach the DRAM row buffer and thrash with one another. Figure 3(b) shows the ideal scenario, where all requests from Warp 0 are scheduled together and allowed to reach DRAM unencumbered, enabling DRAM row buffer locality to be exploited for better bandwidth and energy.

Much like the interleaving of memory requests at the SM level, interleaving at the on-chip network level can create less than optimal memory request orderings with respect to DRAM row buffer locality. Figure 3(c) and (d) depict a two-SM, two-DRAM bank system with each DRAM bank having a single row buffer and bank 0 currently holding the blue row. Figure 3(c) shows that fairly arbitrating between requests from the SMs results in a memory request ordering where a red request from SM 1 would cause the blue row to be evicted from the DRAM row buffer, resulting in a loss in locality when the second blue request from SM 0 later reaches DRAM bank 0. Figure 3(d) shows the ideal case, with the two requests from SM 0 being arbitrated together by the network, enabling them to reach DRAM bank 0 and fully exploit row buffer locality.

As illustrated by the examples, keeping similar memory accesses closer together in time is beneficial to better exploit data locality. Grouping memory requests together not only provides benefits for DRAM but also for the rest of the memory hierarchy, including the L1 cache, L1 miss coalescing, and L2 cache banks. If interleavings that destroy memory locality can be prevented, both GPU energy consumption and performance can be improved. We propose new memory instructions that

avoid SM-level interleaving by issuing multiple related memory requests with a single instruction, effectively grouping the accesses closer together in time.

4 MEMORY ACCESS PATTERNS AND PROPOSED NEW INSTRUCTIONS

Data parallel applications, such as those written in CUDA, often have well-defined memory access patterns. In this work, we exploit these patterns using new memory instructions to gain significant efficiency.

4.1 GPU Memory Access Patterns

We identify two strongly defined patterns found prominently in GPU applications: strided and indirect memory access patterns.

The **strided** pattern is a regular sequence of requests consisting of a starting address, stride or address distance between requests, and number of requests in the sequence. An example of the strided access pattern is a sequence that can be expressed in the form $A[i]$, where i is a loop induction variable.

The **indirect** pattern is a sequence where multiple memory requests are required to access each element. While *irregular* indirect sequences such as those found in some graph traversals are not well defined, some *regular* indirect sequences are well defined, such as those used in complex data structures or in gather/scatter operations. Such regular indirect sequences first access a memory array with a list of indices, then use each of those indices to access elements of a second memory array. An example of the regular indirect access pattern are sequences that can be expressed in the form $A[B[i]]$, where i is a loop induction variable.

Figure 4 presents a pattern categorization of memory requests within a single thread of a warp using the benchmarks found in Section 7.3. This data is gathered by detecting strided and indirect patterns in the memory instruction stream for each thread using a SASSI instrumentation handler (Stephenson et al. 2015). Instructions are only considered strided or indirect accesses if those instructions are found within the innermost loop structure, which makes them the most realistic targets for hardware acceleration. For strided access patterns, detection is performed by inspecting the memory address stream of a memory instruction executed by a thread and checking for a consistent stride across all accesses. Regular indirect access patterns are detected across multiple memory requests, by looking at the backslice of instructions contributing to the address. A memory instruction whose address is generated using a prior load instruction is considered regular indirect, whereas a memory instruction with a cyclic dependency on itself (e.g., pointer chasing) is irregular indirect and is not considered for this categorization.

Overall, the majority of the benchmarks have 80% or more of their memory requests fit into the two patterns. Strided access patterns are the most common and are found in requests to both shared memory and global memory. The most common stride lengths are 32 and factors of 32, corresponding to hardware warp width and naturally fitting existing memory coalescing hardware. Indirect memory requests are also an important pattern, being prominent in breadth-first search (*bfs*), b+tree traversal (*btree*), and sparse matrix dense vector multiply (*spmv*), as well as the microbenchmark sparse matrix sum (*spsum*). The remaining memory instructions that cannot be categorized into the two patterns either access data once without repetition or have long instruction distances between loop iterations such that the pattern cannot be realistically exploited. For example, the large CUDA HPC applications *comd*, *hpgmg*, and *miniamr* do not exhibit the strided or indirect patterns as strongly because a significant number of kernels have irregular control flow or large loop bodies where the time between inter-PC accesses is prohibitively large.

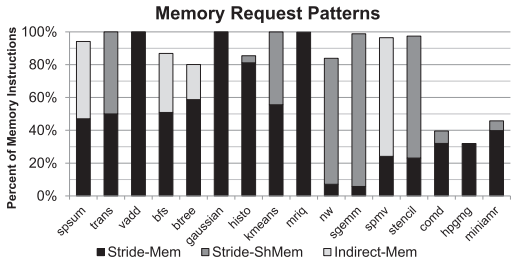


Fig. 4. GPU memory access pattern prevalence for global memory (-Mem) and Shared Memory (-ShMem).

Format - **opcode.**{**length**} *%dest* M[*%address*] *%srca %srcb*
ld.stride{**4,8,16**} %v_result, M[%start_address], %stride
st.stride{**4,8,16**} M[%start_address], %stride, %v_data
ld.indirect{**4,8,16**} %v_result, M[%start_address], %v_indices
st.indirect{**4,8,16**} M[%start_address], %v_indices, %v_data

Fig. 5. ISA specification for new memory instructions. "%" indicates a register, while "%v_" indicates a vector register (contiguous set of registers in the register file).

4.2 New Instructions and Instruction Formats

Motivated by the prevalent strided and indirect memory access patterns, we propose new vector memory instructions that can directly express these patterns. As opposed to SIMD vector memory instructions current found in CPUs, the main goal of these instructions is not to exploit instruction-level parallelism, but rather generate and service memory requests efficiently. We design the instructions to leverage existing SM hardware to minimize changes and overhead. Figure 5 presents the instruction format of the **ld.stride**, **st.stride**, **ld.indirect**, and **st.indirect** instructions. Effectively, the new instructions combine a group of address generation and memory instructions into a single instruction, reducing activity in SM scheduler, pipeline, and register file and reducing memory request interference. The instructions operate on 32-bit floating point and integer data, and each instruction has three possible vector lengths and can generate 4 to 16 memory requests per instruction. In general, the new instructions require a vector length for the instruction as specified by an opcode extension, a starting memory address, source registers for pattern metadata such as the stride between memory elements, and vector source and destination registers. The vector source and destination registers are mapped as contiguous regions in the register file, requiring that only the starting register index be provided. As a result, long sequence lengths can be supported without increasing the existing instruction encoding bitwidth. Similar to current GPU support for 128-bit loads and stores, the register index must be aligned to a 128-bit granularity.

The **ld.stride** and **st.stride** instructions implement a sequence of strided loads and stores and require the starting address of the data array and the stride between memory elements be provided as operands. The indirect pattern is captured by splitting the operation into two phases: the index fetch phase and the data access phase. The index fetch phase is implemented like a strided data fetch, and thus the **ld.stride** instruction can be used. Once the indices have been fetched and stored into the register file, the data access phase is implemented via the **ld.indirect** and **st.indirect** instructions. The **ld.indirect** and **st.indirect** instructions use the recently fetched indices to index into an array, and require the starting address of the data array and a vector of indices be provided as operands.

5 USING NEW INSTRUCTIONS IN CUDA

In this section, we explain how the instructions are used in practice using *Sparse-Matrix Vector Multiply (spmv)* (Stratton et al. 2012) as an example. The code generation method we present is the same method we use for instrumenting the new instructions into the benchmarks for evaluation.

5.1 SPMV Code Example

Listing 3 shows the inner loop of *spmv* in its original state in CUDA, unrolled so that each iteration of the loop operates on four data elements. In this implementation of *spmv*, each thread of a warp

```
for(i=0; (i+3) < bound; i+=4 ) {
    int  offset0 = d_ptr[i+0];
    int   index0 = d_index[offset0];
    float mdata0 = d_matrix[offset0];
    float vdata0 = d_vector[index0];
    sum += mdata0*vdata0;
    int  offset1 = d_ptr[i+1];
    int   index1 = d_index[offset1];
    float mdata1 = d_matrix[offset1];
    float vdata1 = d_vector[index1];
    sum += mdata1*vdata1;
    int  offset2 = d_ptr[i+2];
    int   index2 = d_index[offset2];
    float mdata2 = d_matrix[offset2];
    float vdata2 = d_vector[index2];
    sum += mdata2*vdata2;
    int  offset3 = d_ptr[i+3];
    int   index3 = d_index[offset3];
    float mdata3 = d_matrix[offset3];
    float vdata3 = d_vector[index3];
    sum += mdata3*vdata3;
}
```

Listing 3. Sparse-matrix vector multiply inner loop.

```
for(i=0; (i+3) < bound; i+=4 )
{
  int4 offset = __ldstride4(&d_ptr[i], 1);
  int4 index = __ldindirect4(d_index, offset);
  float4 mdata = __ldindirect4(d_matrix, offset);
  float4 vdata = __ldindirect4(d_vector, index);
  sum += mdata.x * vdata.x;
  sum += mdata.y * vdata.y;
  sum += mdata.z * vdata.z;
  sum += mdata.w * vdata.w;
}
```

Listing 4. Sparse-matrix vector multiply inner loop with new instructions.

is responsible for computing on a single row of non-zero values. In the inner loop, data elements from the sparse matrix and the dense vector are fetched and multiplied together. The result of the multiplication is then used to increment a running sum for each row, which becomes a unique element of the resultant vector at the end of the computation. Memory accesses to the sparse matrix and the vector require indirect memory addressing, as *d_matrix* depends on first accessing *d_ptr*, while *d_vector* requires accessing both *d_index* and *d_ptr*.

Listing 4 shows the inner loop of *spmv* using the new memory instructions. The four memory accesses to *d_ptr*, *d_index*, *d_matrix*, and *d_vector* are compacted into single instances of the new memory instructions of length four. The strided and indirect memory instructions are inserted by the programmer using intrinsics which are equivalent in functionality to existing CUDA intrinsics, such as *__syncthreads()* or *__ldg(float* address)*. In the code listing, *__ldstride4()* implements strided loads, while *__ldindirect4()* implements indirect loads. Four-wide integer and floating point data types (*int4* and *float4*) are used to interface with the new memory instructions, either as source or destination variables.

Using the new instructions significantly reduces address generation in this example, particularly for the indirect accesses. In Listing 3, the unique indices for each access of *d_index*, *d_matrix* and *d_vector* require separate integer addition and multiplication instructions to generate the addresses. In Listing 4, only the base addresses perform address generation, avoiding the extra integer instructions. The load/store unit handles the rest of the address generation for the strided and indirect instructions as described in Section 6. Similarly, the strided instruction used to access *d_ptr* also provides added efficiency.

5.2 Generating Code with New Instructions

Programmer Generation. It is generally straightforward for programmers to leverage these new instructions in real CUDA applications using the intrinsic functions. As a single strided or indirect instruction accesses multiple data elements, multiple related memory accesses are needed to utilize them in practice. Therefore, loop unrolling is the most common source-level code transformation to expose enough related memory accesses and leverage the new instructions.

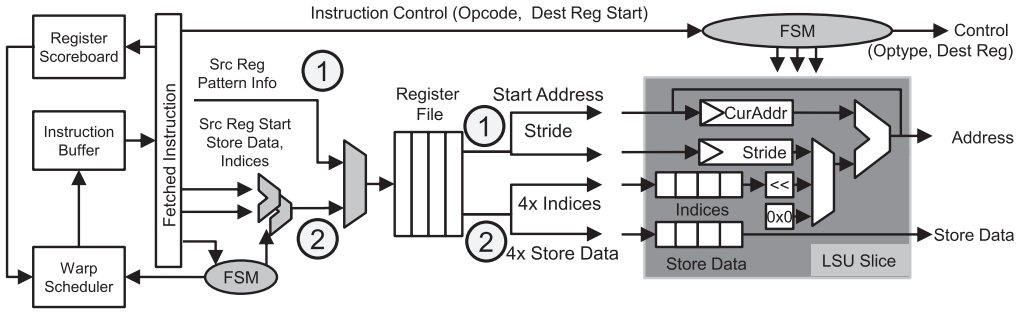


Fig. 6. SM support for new instructions, including an augmented warp scheduler and load/store functional unit.

Compiler Generation. While we have not yet implemented automated compiler-directed transformations, prior research has shown that compilers can readily identify and exploit the types of memory access patterns we identify in this article. Software-controlled prefetching effectively identifies strided memory access patterns, and is implemented on GPUs (Pouchet et al. 2013; Wu 2002; Mowry et al. 1992; Gornish et al. 1990; Yang et al. 2010). Furthermore, automated vectorization identifies strided and gather/scatter patterns and has matured such that common compiler frameworks such as GCC and LLVM leverage the transformations (Kennedy and Allen 2002; Franchetti et al. 2005; Larsen and Amarasinghe 2000; Lattner and Adve 2004).

6 ARCHITECTURE MODIFICATIONS TO SUPPORT NEW INSTRUCTIONS

This section details the hardware support for these new memory instructions, which includes modifying the warp scheduler to stage operand reads from the register file and augmenting the load/store functional unit. Our general approach is to utilize existing pieces of the SM architecture and minimize the number of changes to GPU hardware. We also discuss how the new instructions interact with the SM's L1 cache and miss coalescing logic. Figure 6 presents the SM's warp scheduler, register file, and a single lane of the load/store functional unit.

6.1 Warp Scheduler and Operand Fetch

Current GPU hardware allows fetching of four contiguous registers in the register file per cycle as a single 128-bit source operand, provided that the registers are aligned to a 128-bit granularity. To avoid modifying the register file, we leverage this existing register file bandwidth to quickly provide source operands for our new instructions, similar to existing LD.128 and ST.128 instructions (NVIDIA 2018c; AMD 2016). Depending on the new instruction and the number of operands required, operand fetch is staged over several cycles. Therefore, the warp scheduling logic is modified to wait until all operands are fetched and sent to the load/store unit before allowing another instruction to issue.

Figure 6 presents the general flow of operand fetch for the new memory instructions. The first instruction issue cycle fetches the starting address and metadata such as the stride between elements (1). Strided loads can be completely issued in a single cycle as *ld.stride* instructions only require the starting address and stride. All other instructions require store data or index data and must fetch additional operands. In subsequent cycles, four operands are fetched per cycle leveraging the existing bandwidth of the register file (2). The total number of cycles to completely issue the instruction depends on the instruction type and length. *st.stride* and *ld.indirect* instructions need either store data or index data, resulting in two issue cycles for vector length 4, three issue cycles for vector length 8, and five issue cycles for vector length 16. For *st.indirect* instructions,

both store data and indices are needed, requiring three issue cycles for vector length 4, five issue cycles for vector length 8, and nine issue cycles for vector length 16. While the multiple cycles required to issue some instructions (particularly stores) may seem significant, those instructions still result in significant performance and energy benefits, as discussed in Section 8. In practice, the lower-overhead load instructions are much more commonly used across the benchmarks.

6.2 Load Store Unit Augmentation

We modify the load/store unit depicted in Figure 1(c) to support the new instructions. Whereas the baseline load/store unit has an integer adder and a shifter for handling offsets, our augmented functional unit adds additional latches, muxes, and a finite-state machine (FSM) to control multi-cycle instruction execution. The general strategy of the augmented load/store unit is to generate sequence addresses iteratively. For example, with new memory instructions of length four, four addresses will be generated and sent downstream to the L1 cache or scratchpad memory, one address per cycle. Depending on whether the instruction is a load or store, the generated address is either paired with corresponding destination register or store data element before being sent to the SM's memory subsystem.

After operands have been loaded, the FSM controls address generation depending on instruction type. For *ld.stride* and *st.stride* memory instructions, the starting address and stride are loaded into the two latches. The FSM controls iteration, selecting $0x0$ from the multiplexer to pass the starting address through the adder as the first address generated by the instruction. The FSM sends the result of the adder downstream and updates the latch storing the current address to prepare for the next iteration of address generation. Subsequent addresses are generated by the FSM controlling the multiplexer to add the stride to the current memory address. For indirect memory instructions, the FSM iterates over the index values one at a time, adding the index value (shifted for 32-bit data size) to the base address to generate the address for each memory request.

6.3 SM Memory Interface

No changes to the SM's memory subsystem are required for the new memory instructions, as the only change is the order in which addresses appear from the executing warps. Addresses generated by the augmented load/store unit are sent in warp-wide groups to access either the L1 cache or the shared memory scratchpad, similar to existing load and store instructions in the baseline architecture. When the addresses reach the L1 cache, any misses attempt to be coalesced into outstanding miss requests stored in the miss status handling registers (MSHRs).

The main memory benefit of the new instructions is the effect on memory request interleaving at the warp level. Each new instruction generates 4, 8, or 16 addresses per thread which are sent iteratively to the SM's memory subsystem, effectively grouping together multiple memory requests from a single warp in time. Grouping together memory requests from the same warp provides more opportunities to exploit data locality at the L1 cache, L2 cache, and DRAM, as long as the set of addresses are nearby each other spatially.

7 EXPERIMENTAL METHODOLOGY

7.1 Using New Instructions in CUDA with Modified SASSI Instrumentation Flow

SASSI is a compiler-based instrumentation tool that we use both for collecting instruction traces for GPU applications and for high-level application characterization (Stephenson et al. 2015). The general intent of SASSI is similar to popular CPU binary instrumentation tools such as PIN (Luk et al. 2005), although the instrumentation mechanism is different. SASSI is based on NVIDIA's production "backend" compiler, `ptxas`, that compiles from the PTX (NVIDIA 2018c) intermediate

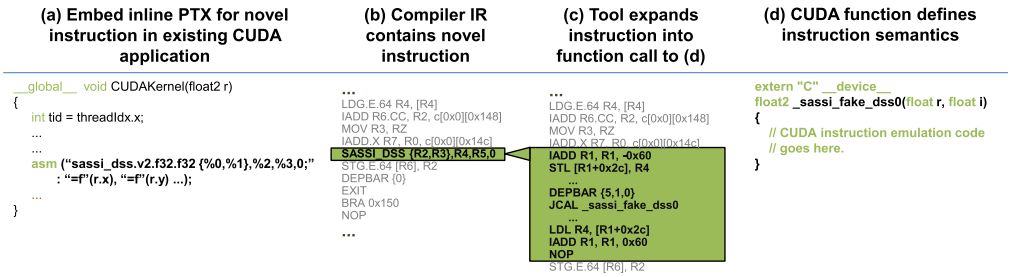


Fig. 7. Injecting new instructions in CUDA using modified NVCC and SASSI.

representation to NVIDIA's binary instruction format, SASS. SASSI is merely a compiler pass in `ptxas` that runs after all other compiler passes. When the compiler invokes the SASSI pass, users can selectively inject new functionality into the program without altering program semantics.

To support the evaluation of our new instructions, we modify the SASSI flow to allow insertion of our proposed memory instructions as native PTX assembly in each of the CUDA benchmarks we evaluate. With our modified SASSI flow, researchers can rapidly develop, encode, and execute novel instructions using generic instruction templates. Essentially, this novel SASSI flow removes the burden of modifying the compiler for each new proposed ISA execution by automating the process.

Figure 7(a) depicts our top-down approach, which starts at a high level. We modify our applications at the CUDA level by adding generic instructions using the standard *inline PTX* mechanism of CUDA (NVIDIA 2018b). Unlike the existing inline PTX mechanism, our tool allows users to specify instructions that do not exist. We have several templates for generic instructions, but common to all of them is that the templates themselves do not prescribe instruction semantics. Instead, the templates merely dictate how many destination and source operands the instruction contains, as well as the types of each operand (e.g., memory, 32-bit integer, floating point). For example, the generic instruction in (a) uses the template `sassi_dss` that has one destination operand ("d"), and two source operands ("ss").

As (b) shows, when the `ptxas` compiler lowers CUDA to the compiler's IR, it treats each generic inline PTX instruction as a single *atomic* SASS instruction. This approach has the benefit that it does not preclude many important compiler optimizations such as register allocation, predication, and loop unrolling, but at the same time acts as a scheduling barrier providing better control over how inline instructions are scheduled. Our generic instructions have no associated semantics, and therefore the compiler cannot freely schedule them safely.

The compiler finalizes the code and completes code generation as if there were a matching SASS ISA instruction and the associated hardware support for each of our generic instructions. All SASSI instrumentation passes happen immediately after this code generation, enabling measurement and instruction trace generation as if the instructions really existed.

The final SASS code is generated after the SASSI instrumentation passes. Since there are no matching ISA instructions or hardware support, we augment SASSI to convert our generic instructions into callbacks to emulation functions. The emulation functions enable the application to execute on the GPU with correct semantics for the new instructions. As (c) and (d) show, the tool passes the registers allocated to the instruction's sources as arguments to the callback function, so we can separately implement the semantics of the generic instructions in standard CUDA. Users are free to model arbitrarily complex instructions in CUDA, including those involving multiple lanes and persistent state.

Table 1. Performance Model Parameters

Chip	32 SM Partitions, 8 Memslice Partitions
SM Compute	32 lanes (Int/FPU/Agen), 256KB Register File Max 32 Concurrent Executing Warps Greedy-then-Oldest Warp Scheduler
SM LSU	4-,8-,16-length New Memory Instructions
SM L1	32KB, 4-way, 128B line, 256 MSHRs, 16 cycles
SM Batching Unit	4 entries, 16-entry store data buffer
On-chip Network	32×8 Crossbar, 16-entry per-input request buffer
MemSlice L2 Bank	256KB, 16-way, 128B line, 48-cycle
MemSlice Mem Controller	FR-FCFS, 128 entries
DRAM	8 channels, 16 banks/channel, 32-bit channels GDDR5: tCAS=12ns, tRCD=12ns, tRP = 12ns

7.2 Performance and Energy Modeling

We leverage SASSI to collect detailed instruction traces which can accurately be replayed in a performance model. We direct SASSI to add instrumentation code *before* all of the original SASS instructions, including the new memory instruction proposed in this work. For each instrumented instruction, our added instrumentation extracts the following detailed information about each instruction executed at the warp level:

(1) Instruction's opcode
(2) Warp's active mask
(3) Instruction's predication state
(4) Instruction's source and destination register names and data
(5) Memory addresses the instruction touches

Our instrumentation code subsequently "pushes" that information off of the GPU over an in-memory channel to a CPU thread that is responsible for writing the stream to a trace file.

We built a cycle-accurate trace-driven performance model that consumes the detailed instruction traces we generate with SASSI. Table 1 presents the parameters for our performance model, which models the architecture shown in Figure 1. We add the modified warp scheduler and augmented load/store unit to the performance model as outlined in Section 6. The model supports multiple SMs with instruction fetch, warp scheduling, register scoreboards, functional unit pipelines, banked shared memory, L1 cache, and miss coalescing. We also accurately model a crossbar, L2 cache banks, memory controller, and GDDR5 DRAM banks.

We model energy consumption by extracting per operation and static energy numbers from a combination of McPAT (Li et al. 2009) and GPUWattch (Leng et al. 2013) using the configuration in Table 1. These energy numbers are then combined with activity counts collected in the performance model to obtain energy consumed in the system. Such activity includes reads and writes to memory structures and active cycles for functional units.

7.3 Benchmarks

Table 2 presents the CUDA benchmarks used for evaluation, which represent a range of real applications, vary in memory access behavior and intensity, and include irregular control flow and complex data structures. Kernels that operate on a single data element are refactored to operate on multiple data elements and use looped control flow. All inner loops are unrolled by the factor enabling the best performance.

Table 2. CUDA Benchmarks for Evaluation

Benchmark Name	Abbr.	Mem Stride	ShMem Stride	Indirect	Inner Loop Runtime Contribution	Kernels Augmented {Num. Loops}
Sparse Matrix Sum	spsum	✓		✓	97.5%	neighborListUpdate{1}
Matrix Transpose (NVIDIA 2018a)	trans	✓	✓		98.2%	transposeNoBankConflicts {1}
Vector Addition (NVIDIA 2018a)	vadd	✓			99.7%	vectorAdd{1}
BFS Graph Traversal (Che et al. 2009)	bfs	✓		✓	82.4%	kernel{1}, kernel2{1}
B+Tree Graph Traversal (Che et al. 2009)	btree	✓		✓	81.6%	findK{2}, findRangeK{2}
Gaussian Elimination (Che et al. 2009)	gaussian	✓			93.2%	fan1{1}, fan2{1}
2D Histogramming (Stratton et al. 2012)	histo	✓			78.1%	histo_main_kernel{2}, histo_final_kernel{3}
K-means Clustering (Che et al. 2009)	kmeans	✓			86.2%	kmeansPoint{1}
MRI Reconstruction (Stratton et al. 2012)	mri-q	✓			99.7%	computeQ_GPU{1},
Needlman-Wunsch (Che et al. 2009)	nw	✓	✓		88.9%	needle_cuda_shared_1{2}, needle_cuda_shared_2{2}
Matrix Multiply (Stratton et al. 2012)	sgemm		✓		95.8%	mysgemmNT{2}
Sparse-Matrix Vector Mult. (Stratton et al. 2012)	spmv	✓		✓	92.4%	spmv_jds{1}
Stencil Computation (Stratton et al. 2012)	stencil	✓	✓		88.1%	block2D_hybrid_coarsen_x{1}
Molecular Dynamics (Heroux et al. 2009)	comd	✓			65.1%	eam_force_thread_atom{2}
Geometric Multigrid Method (HPGMG 2018)	hpgmg	✓			74.1%	cheby_smooth_kernel{1}, copy_block_kernel{1}
Adaptive Mesh Refinement (Heroux et al. 2009)	miniamr	✓	✓		71.1%	stencil_cache_separate_halos_flat{2}

Benchmarks are sourced from well-known suites such as Rodinia (Che et al. 2009) and Parboil (Stratton et al. 2012). We also leverage three benchmarks (*comd*, *hpgmg*, *miniamr*) adapted directly from large-scale HPC applications (Heroux et al. 2009; HPGMG 2018). These three benchmarks have many kernels, complex data structures, and range in code footprint size from 15,000 to 30,000 lines of source code. We also evaluate on two microbenchmarks from the CUDA SDK (NVIDIA 2018a) (*trans*, *vadd*), and we developed one microbenchmark (*spsum*) to help evaluate the new indirect memory instructions. *spsum* implements a kernel where each thread performs a summation of a single row of a sparse-matrix and stores the result in an array.

We augment each of these benchmarks with our new strided and indirect memory instructions by hand. As mentioned in Section 5.2, we leverage loop unrolling as the key source transformation to expose enough memory accesses to enable the use of the new instructions. To help assist in identifying source code locations to modify, we built a memory profiler using SASSI (Stephenson et al. 2015) which identifies strided and indirect memory patterns and the corresponding lines in CUDA source code where each pattern manifests as memory operations. The tool allows us to quickly find candidate locations in the source code for the new instructions. Table 2 shows which of the new instructions each benchmark utilizes, and what percent of total dynamic instructions are contributed by the inner loops targeted for the new instructions. The table also shows which kernels in and how many inner loops are augmented to utilize the new instructions for each of the new memory instructions.

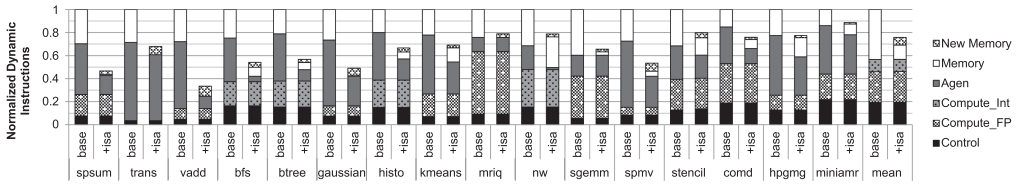


Fig. 8. New instruction effect on dynamic instructions.

8 EVALUATION OF NEW INSTRUCTIONS

We evaluate the new memory instructions given the GPU system parameters found in Table 1. Generally, for the new instruction data presented "*+isa*," we compare the best performing vector length (e.g., 4, 8, or 16) against the baseline. We first discuss the effect of the new instructions on address generation instructions and memory request interference, followed by a discussion of performance and energy impact across the benchmarks. We finish our analysis of the new instructions by discussing the performance effect of vector length, comparison to other work, and area overheads.

8.1 Address Generation Instruction Reduction

We used SASSI (Stephenson et al. 2015) to collect dynamic instruction counts for the benchmarks using the instrumentation handler described in Section 3.1. Figure 8 presents the effect of the new instructions on dynamic instruction counts, comparing benchmarks with a baseline implementation against an implementation with the new instructions. As expected, we are able to substantially reduce the total number of instructions, with the majority of the instruction reduction coming from fewer address generation (i.e., integer) and memory operations, while floating point and control remain the same.

The total instruction reduction is highly dependent on the instruction mix and type of benchmark. Highly memory-bound benchmarks such as *bfs*, *btree*, and *spmv* leverage both the strided and indirect memory instructions, seeing dynamic instruction reductions of nearly 45%, while memory-intensive benchmarks with more compute such as *histo*, *kmeans*, and *nw* see reductions of 20%–32%. Benchmarks that heavily rely on shared memory such as *sgemm* and *stencil* do not see any substantial reduction in integer instructions, as the compiler is able avoid extra address generation and use immediate values with the shared memory instructions. Similarly, compute-bound *mriq* has little address generation to reduce. However, the number of memory instructions is significantly reduced in those three benchmarks. Finally, the new instructions reduce dynamic instructions in the large HPC applications *comd*, *hpgmg*, and *miniamr* by 12%–24%.

Overall, across the 16 benchmarks, the geometric mean reduction in dynamic instructions is 33% when using our proposed memory instructions. These reductions correlate strongly to energy reduction in the SMs, as fewer instructions means less functional unit, pipeline, and register file activity.

8.2 Reduced Memory Request Interference

We demonstrate the effect of the new memory instructions on the GPU memory subsystem using our performance model. Figure 9 presents the L1 cache, L2 cache, and DRAM row buffer misses for the benchmarks augmented with the new memory instructions, normalized to the misses in the baseline benchmark version. Overall, the ability of the new memory instructions to reduce interleaving at the SM has a significant effect on reducing interference in the entire GPU memory

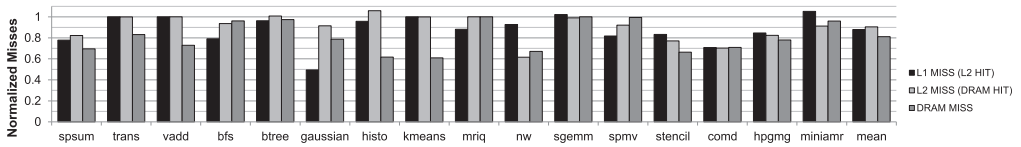


Fig. 9. Total L1 cache misses in the GPU memory system, broken down by location serviced.

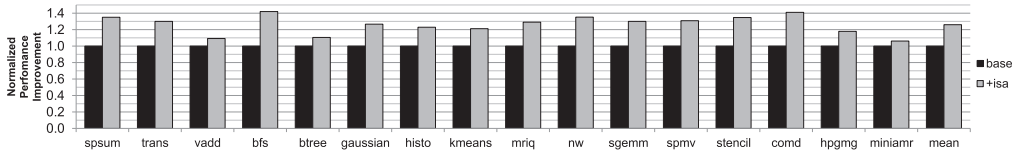


Fig. 10. Normalized performance improvement for baseline and new instruction augmented benchmarks.

subsystem. The new memory instructions reduce L1 cache misses by 12%, L2 misses by 10%, and DRAM misses by 19% on average across the benchmarks.

Similar to the effect on dynamic instructions, the effect on the GPU memory subsystem varies depending on the benchmark. DRAM-sensitive benchmarks such as *kmeans*, *spmv*, and *stencil* experience DRAM row buffer hit rate improvements of 5%, 4%, and 4%. Given the large penalty for missing in the DRAM row buffer, these hit rate improvements significantly improve runtime performance. Other memory-intensive benchmarks such as *bfs*, *nw*, *comd*, and *hpgmg* mainly benefit from L1 cache miss coalescing, which reduces the number of misses sent out from the SM. The ability to group memory requests together in time enables more opportunities to make use of data brought into the L1 cache before it is evicted. Memory-intensive *miniamr* sees an increase in L1 cache misses, but a 12% increase in L2 hit rate significantly reduces the number of requests sent to DRAM, providing a net benefit. Finally, compute-bound benchmarks such as *mriq* and *sgemm* see minimal improvement.

8.3 Performance Evaluation

Performance Improvement. Figure 10 presents the normalized performance improvement of the new memory instructions for each of the benchmarks. Overall, using the new instructions provides an improvement (i.e., application speedup) of 26% on average. This improvement comes from reduced dynamically executed instructions and reduced memory request interleaving at the SM level, which reduces memory interference and improves hit rates in the L1 cache, L2 cache, and DRAM row buffers as discussed in Section 8.2.

For memory-intensive benchmarks, the DRAM hit rate has the most influence over performance. Memory-bound benchmarks such as *vadd* and *btree* experience the lowest performance improvement at 9% and 10%. *vadd* is a DRAM-bound program more easily captured by the existing DRAM scheduling hardware, while for *btree* the memory requests are more irregular accesses to DRAM memory. Irregular memory-intensive benchmarks that have a lot of data reuse such as *bfs* and *spmv* see improvements of 42% and 31% due to reduced memory request interleaving which leads to fewer misses in the L1 and L2 caches. For memory-intensive benchmarks with regular access patterns such as *histo*, *nw*, *comd*, and *hpgmg*, the increased memory hit rates (particularly at DRAM row buffers) improve performance by 21% to 35%.

For benchmarks that are compute-bound or spend a lot of time in shared memory such as *sgemm*, *stencil*, and *mri*, the performance improvements are 30%–35%. The biggest effect on performance for these benchmarks is the reduction in dynamic instructions, which enables the compute operations to issue more frequently. Once the new memory instructions are issued, memory loads

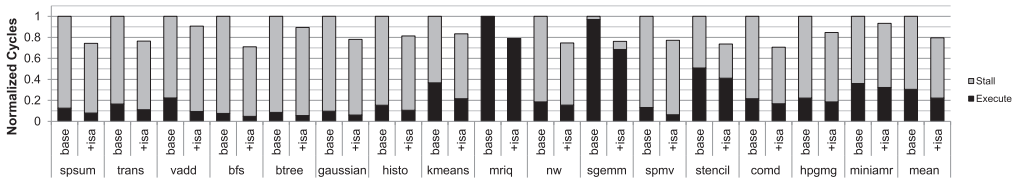


Fig. 11. Normalized runtime cycles spent for baseline and new instruction augmented benchmarks.

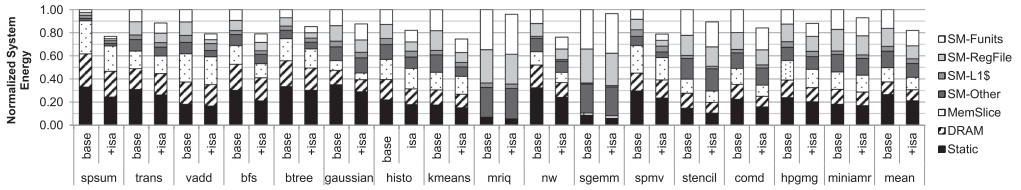


Fig. 12. System energy breakdown for baseline and new instruction augmented benchmarks.

and stores are handled directly from the augmented load/store unit, enabling the compute instructions to execute decoupled from memory access.

Runtime Cycle Analysis. Figure 11 presents the breakdown of SM runtime for each benchmark in two categories: cycles spent executing instructions (EXECUTE) and cycles spent stalled (STALL). SM stalls occur when there are no warps ready to issue, which can happen when instructions are waiting on memory requests to return or SM resources to be made available (e.g., MSHR slots). Looking at the data in Figure 11 helps further uncover whether the performance improvement from the new memory instructions is from reducing dynamically executed instructions (EXECUTE) or from reduced memory request interference to improve memory subsystem behavior (STALL). Most of the benchmarks are memory-intensive, and thus spend the majority of stall cycles waiting on memory requests to return. On average, both benefits of the new instructions contribute strongly to the overall performance improvement, as 41% of the improvement is due to reducing executed instructions, while the other 59% of the improvement comes from reducing memory request interference.

For compute-bound benchmarks such as *mriq* and *sgemm*, performance improvement is closely related to reducing dynamic instructions. Both algorithms have a substantial number of memory accesses, but those memory accesses either hit in the L1 cache (*mri*) or primarily utilize local scratchpad memory. We expect that other similar compute-bound algorithms would also benefit from the new instructions. For *vadd*, *kmeans*, and *miniamr* the vast majority of the performance improvement comes from reduced instructions executed, as the new instructions enable the benchmarks to spend less time generating addresses and more time servicing memory requests. For the other benchmarks, the time spent executing and time spent stalled are both reduced with the new memory instructions. Stall cycles in general see the most reduction, as they represent the reduced memory access latency through better memory hit rates.

8.4 Energy Efficiency Evaluation

System Energy Consumption. Figure 12 presents a breakdown by source of energy consumption for each benchmark. Overall, our new instructions are able to reduce total energy consumption in the system by an average of 18%. The energy benefits of the new memory instructions mainly affect energy consumption through reduced dynamic instructions and better memory subsystem behavior.

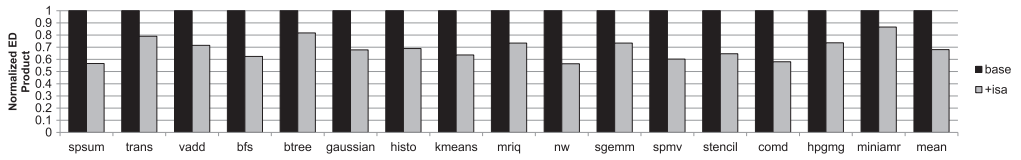


Fig. 13. Normalized energy-delay product for baseline and new instruction augmented benchmarks.

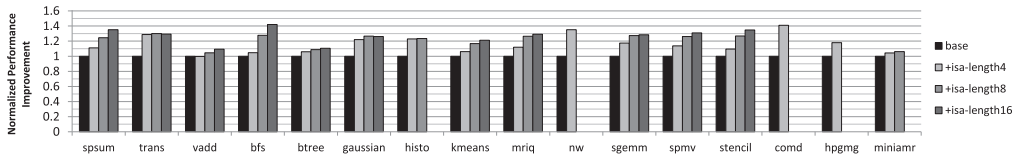


Fig. 14. Runtime performance, increasing the number of addresses generated per instruction (vector length).

Reduced address generation and memory instructions significantly affect the dynamic energy consumption of the SMs, which makes up more than 50% of all system energy consumption. Overall, the new memory instructions reduce SM energy consumption by 19% on average. On average, the new instructions reduce functional unit energy by 18%, register file energy by 28%, and all other SM energy (SM-Other: instruction fetch, warp scheduling, pipeline energy) by 15%. The effect varies benchmark to benchmark, depending on how many dynamic instructions are reduced. Benchmarks that reduce the most address generation instructions (e.g.,*btree*, *kmeans*, *spmv*) experience SM dynamic energy reduction up to 41%.

Reduced memory request interference leads to significantly better L1 cache, L2 cache, and DRAM row buffer hit rates, and which also reduces dynamic energy in the memory subsystem. Overall, dynamic energy in the GPU memory subsystem is reduced by 11% on average, including reductions in the L1 cache, L2 cache, and DRAM. Across the benchmarks, dynamic energy in the L1 cache is reduced by 5%, L2 Cache 11%, and DRAM 12%. The DRAM-sensitive benchmarks such as *histo*, *nw*, *stencil*, and *comd* see respective DRAM dynamic energy reductions of 22%, 35%, 28%, and 30% due to significantly improved hit rates throughout the GPU memory subsystem. Finally, reduction in runtime from better memory subsystem behavior reduces static energy consumption in the GPU and DRAM by an average of 16%.

Energy-Delay Product Improvements. Figure 13 combines the performance and energy results to compute the energy-delay product. Overall, the new memory instructions have a large effect on the system by reducing address generation instructions and memory request interference, reducing energy-delay product by 32% on average. The benchmark with the biggest improvement is *nw* with a 44% reduction in energy-delay product, while the benchmark with the smallest improvement is *btree* with an 18% reduction.

8.5 Vector Length Evaluation

Figure 14 presents the performance effect of the new memory vector lengths when generating 4, 8, and 16 addresses per instruction. Some of the configurations are not possible for certain benchmarks (e.g., *histo*, *nw*), as the benchmark's inner loop cannot be unrolled enough due to lack of iterations or due to register footprint and SM register limitations.

Generally, increasing the vector length improves runtime performance. This is especially true for *kmeans* and *stencil*, which see increased DRAM row buffer hit rates to improve performance. Memory-intensive *spmv* also sees improvement in both runtime and energy consumption, due to reduced interleaving which increases coalescing of L1 misses. Compute-bound benchmarks *mriq*

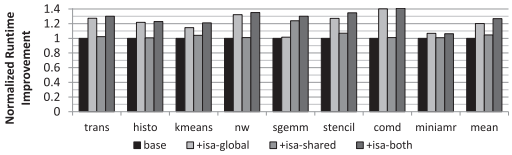


Fig. 15. Runtime performance for Global memory (-global) and Shared Memory (-shared) instructions.

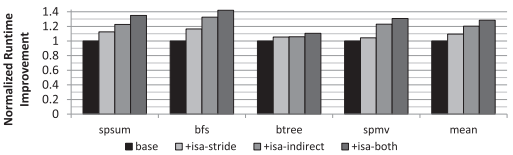


Fig. 16. Runtime performance for Strided (-global) and Indirect Memory (-indirect) memory instructions.

and *sgemm* also see increased performance due to a decrease in dynamic instructions as the vector length increases. As the vector length increases the additional performance benefit diminishes, which varies depending on the benchmark. While there is clearly a fundamental limit to the number of address generation instructions that can be removed from the dynamic instruction stream, there is also a limit to how much reduced memory request interleaving can affect performance. In theory, a single **ld.stride** instruction of length 16 can generate enough unique memory requests to fully utilize a DRAM row of size 2,048 bytes.

8.6 Instruction Type Evaluation

The new memory instructions target strided and indirect memory accesses to both global and shared memory as shown in Figure 4. We perform experiments to isolate the benefit of the new memory instructions on each pattern by evaluating runtime performance of the benchmarks that include a single instruction type.

Global and Shared Memory Instructions. We first investigate the benefits the new instructions targeting global memory and shared memory on the eight benchmarks that have both types of memory accesses. Figure 15 presents the runtime performance of the benchmarks augmented with strided memory instructions for global memory accesses, shared memory accesses, and both global and shared memory accesses.

In general, we find that for our collection of benchmarks, runtime performance is improved mostly by targeting global memory accesses. For memory-intensive benchmarks the performance bottleneck is in the memory hierarchy, as the GPU spends a significant amount of time stalled waiting on memory results. The new memory instructions targeting global memory are particularly useful for reducing stalls in the memory hierarchy by reducing memory request interference which enables better exploitation of data locality. On average, global memory instructions enable 78% of the total performance benefit on the eight benchmarks. Only more compute-intensive benchmarks such as *sgemm* and *stencil* see a significant improvement when only the strided shared memory instructions are utilized. *sgemm* is a benchmark that is especially compute-intensive, and nearly all memory accesses are shared memory accesses. As mentioned in Section 8.3, reducing the number of address generation and memory instructions issued enables the frontend of the pipeline to be more efficient and issue compute instructions more often.

Strided and Indirect Instructions. Figure 16 presents the runtime performance of benchmarks augmented with strided memory instructions, indirect memory instructions, and both strided and indirect memory instructions. The four benchmarks presented are *bfs*, *btree*, *spmv*, and *spsum* as they utilize both instruction types as found in Table 2.

Overall, the indirect memory instructions contribute 69% of the total runtime improvement across the four benchmarks. Most of the opportunity to improve performance in these benchmarks lies in exploiting spatial and temporal locality on data structures such as the vertices in *bfs* and *btree* and the vector data in *spmv*. These data structures experience data reuse in the form of

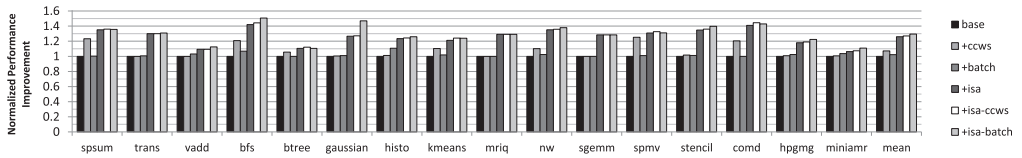


Fig. 17. Comparing performance improvement with Cache-Conscious Wavefront Scheduling and Request Batching.

multiple requesters accessing the same element, in contrast to elements in the index arrays accessed once per phase. While the accesses to these data structures are irregular in nature, we find the high amount of memory interference in the GPU memory hierarchy contributes significantly to poor memory behavior. The indirect instructions combat memory interference by grouping together related requests in time to provide a better opportunity to exploit data locality. For example, sparse matrices that are diagonal in shape are common across a number of applications, and when computing *spmv* on such a sparse matrix in compressed sparse-row (CSR) format, computation on neighboring matrix rows often requires similar data from the dense vector. Considering that work partitioning in the CSR format is often done by mapping matrix rows to threads, the new instructions can exploit spatial and temporal locality better by grouping together multiple requests across threads. We find that similar exploitation of data reuse is achieved in *bfs* and btree, with the success of exploiting vertex data reuse being more dependent on the vertex connectivity in the dataset.

8.7 Comparison to Related Work

We compare our new instructions to two related works: cache-conscious wavefront scheduling (CCWS) (Rogers et al. 2012) and alternative memory access scheduling which batch requests which map to the same DRAM row (BATCH) (Kim et al. 2011; Yuan et al. 2009). Though CCWS and BATCH do not affect the dynamic instruction stream of the application, both techniques can reduce memory request interference, similar to our new instructions. CCWS deactivates warps in the warp scheduler that cause contention in the L1 cache, enabling active warps to better utilize the cache hierarchy. Fewer active warps effectively helps eliminate a source of memory request interference at all levels of the memory hierarchy. As found in the original CCWS work, cache-sensitive algorithms are the only applications that benefit much from CCWS, though there is no real performance penalty for other applications. Our BATCH implementation batches L1 cache miss requests from a single SM and memory requests across SMs that map to the same DRAM row into network packets as in prior work. This prior work generally focuses on ways to simplify existing complex DRAM memory scheduler, but in practice BATCH architectures can be used to improve performance in a wide variety of systems by avoiding both inter- and intra-SM memory request interleavings. However, one limitation is that the techniques perform batching dynamically at runtime without knowledge of the application. As CCWS is designed only for L1 cache-sensitive applications and BATCH must attempt to find similar requests dynamically, we expect their applicability to our benchmarks to be limited.

Figure 17 presents the performance results, including systems that have both the new instructions and either CCWS or BATCH. As expected, CCWS performs well on several cache-sensitive benchmarks—*spsum*, *bfs*, *kmeans*, *spmv*, and *comd*—averaging a performance improvement of 18% across those kernels. While BATCH can group together requests across SMs, unlike our proposed memory instructions, it generally is not able to dynamically find enough DRAM row matches flowing through the memory hierarchy to outperform the reordering ability of the DRAM memory scheduler. However, BATCH still sees improvement on *bfs*, *histo*, and *miniamr*, averaging 6%

improvement on those kernels. While CCWS and BATCH improve performance on several benchmarks, the new instructions improve performance in all of our benchmarks. As shown in Figure 11, the new instructions reduce dynamically executed instructions which enables performance improvement; both CCWS and BATCH do not reduce execution instructions and cannot capitalize on this benefit. Exploiting memory access patterns in software using the instructions in general reduces memory interference more than both CCWS and BATCH. BATCH is limited in its ability to reduce memory request interference as it only has a small time window to batch together memory requests bound for the same DRAM row.

CCWS and BATCH are somewhat orthogonal to the new memory instructions, and combining techniques can provide additive benefits. For example, adding CCWS and our new instructions provides an additional 2%–4% performance improvement on cache-sensitive benchmarks over the new instructions alone. In general, our new instructions help reduce the effect of L1 cache contention by keeping similar memory requests together in time, but further reducing contention for the L1 cache enables additional runtime improvements. Similarly, adding BATCH to our new instructions enables memory requests to be further grouped inter- and intra-SM, which in the case of benchmarks like *bfs*, *stencil*, and *hpgmg* can improve performance by another 6%. Nearly all of the extra performance benefit comes from the BATCH hardware grouping together similar memory requests from multiples warps in a single CTA executing on an SM.

8.8 Area Overheads

The extra chip area to implement the hardware required for the new instructions is dominated by storage elements. Therefore, we sum the storage requirements given the configuration in Table 1 to compare the overhead to existing structures in the SM. To support the full vector length of 16, the storage overhead for the functional units is 138 bytes for each lane to store a 48-bit address, 32-bit stride, sixteen 32-bit indices, and sixteen 32-bit store data elements. The total area overhead of our techniques is 4.3 kilobytes, or 1.7% of the area of the SM's 256-kilobyte register file. As the added area is dominated by the storage for the indices and store data elements, scaling down the vector length to a 4 or 8 will yield reduced area while still providing significant benefit as demonstrated in Figure 14.

9 RELATED WORK

Modern GPUs have instructions which can load or store 64 and 128 bits at a time, but require the memory accesses to be aligned to the requested bit width (NVIDIA 2018c; AMD 2016). Modern CPUs with SIMD vector units support a variety of gather/scatter instructions. For example, current Intel processors support both regular and irregular gather/scatter operations using the vgather, vscatter, vexpand, and vcompress instructions in the AVX-512 ISA (Intel 2016). ARM processors support vector load and store instruction with fixed stride lengths of two, three, and four (ARM 2013). Other processors with traditional vector instructions or SIMD functional units have proposed strided and gather/scatter loads and stores (Russell 1978; Dunigan et al. 2005; Espasa et al. 2002; Kozyrakis and Patterson 2002). Both the CPU SIMD and other vector architectures provide similar functionality to existing GPU SIMT warp execution, as each static-length vector instruction executes multiple operations in parallel on separate functional units. In contrast, our new memory instructions build upon the efficiency of GPU SIMT warp execution.

Other related work has investigated augmenting the GPU ISA with new instructions to improve efficiency. One such work leverages fused integer instructions, warp-shared scalar instructions, and SIMD integer datapaths (Gilani et al. 2013), while another focuses on leveraging scalar datapaths in GPUs (Liu et al. 2017). Another research proposal leverages affine value structure properties inherent in many SIMT applications to design a new GPU ISA with warp-wide scalar

instructions (Kim et al. 2013). In this prior research, instruction fusion and scalarization are the techniques most related to the new memory instructions proposed in this article. While the new memory instructions also focus on fusing instructions to reduce integer operations, we specifically focus on how address generation instruction overhead can be mitigated. Additionally, warp-wide scalar instructions is an orthogonal technique that we expect can apply to the new memory instructions we propose.

GPU warp scheduling techniques that improve memory subsystem behavior have been studied extensively. Techniques that prioritize warps to improve L1 cache and DRAM behavior such as cache-conscious wavefront scheduling (Rogers et al. 2012) and OWL (Jog et al. 2013) focus on managing warps that conflict with one another. Other GPU warp scheduling research focuses on predicting L1 cache footprint to regulate active warps (Rogers et al. 2013) and identifying and prioritizing critical warps dynamically (Lee and Wu 2014; Lee et al. 2015). In general, these proposals focus on memory-intensive GPU benchmarks, preventing thrashing in the memory subsystem. While the new memory instructions proposed in our work generate and group requests together for more effective warp scheduling, the new instructions require no changes to the warp scheduling policy. Furthermore, as warp scheduling research is orthogonal to our new instructions, we expect that the new instructions will benefit from future warp scheduling research.

DRAM memory request scheduling has been an active area of research for both CPUs and GPUs (Rixner et al. 2000; Mutlu and Moscibroda 2007, 2008). Memory controllers that implement these proposals attempt to find parallelism in the request stream in order to exploit DRAM row buffer locality and improve bandwidth and energy. Other related DRAM memory scheduling research propose methods to better schedule and prioritize requests from the SMs in order to avoid the effects of memory request interference and better exploit DRAM row buffer locality. One prior work suggests batching an SM's L1 cache miss requests by DRAM row into network packets (Kim et al. 2011). Another work focuses on exposing and prioritizing row buffer hits in the on-chip network (Yuan et al. 2009). While our instructions also prevent interleaving, our work focuses on improving scheduling within an SM, which does not preclude additional techniques that help reduce interleaving across SMs.

Strided memory access patterns have traditionally been detected and leveraged at runtime with hardware prefetchers (Baer and Chen 1995; Fu et al. 1992). Prefetchers have also recently been described for indirect memory requests (Yu et al. 2015). GPU implementations of hardware prefetching have focused on detecting strided patterns at the warp and CTA-level of granularity (Lee et al. 2010; Sethia et al. 2013). Despite the similar focus on memory access patterns, the goals of prefetching are different than the new memory instructions. First, the new instructions are demand memory requests and are not speculative in nature like prefetches, which removes any potential timeliness and accuracy concerns. Secondly, a major goal of the new instructions is to reduce address generation instructions, which is not a goal of prefetching. However, we believe the address stream generated by our new instructions may benefit from the memory latency improvement provided by prefetching.

10 CONCLUSION

Exploiting application patterns and behavior remains a large opportunity to improve compute capabilities on GPUs. In this article, we focus on common memory access patterns found in GPU data-parallel workloads and find that there are two major opportunities: reducing address generation instructions and preventing memory request interference. We propose and implement hardware for new ISA instructions to capture strided and indirect memory request patterns in order to group L1 cache requests and better exploit data locality in the memory subsystem. Our experimental results show that we can eliminate 33% of dynamic instructions across 16 benchmarks

and reduce memory request interference in the L1 cache, L2 cache, and DRAM row buffers. These improvements result in an overall runtime improvement of 26%, an energy reduction of 18%, and a reduction in energy delay of 32%.

REFERENCES

AMD. 2016. AMD GCN3 Instruction Set Architecture. Retrieved February 15, 2018 from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf.

ARM 2013. ARM: Coding for NEON - Part 1: Load and Stores. Retrieved February 15, 2018 from https://community.arm.com/processors/b/blog/posts/coding-for-neon---part-1-load-and-stores.

Jean-Loup Baer and Tien-Fu Chen. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (May 1995), 609–623.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC'09)*.

Thomas H. Dunigan, Jeffrey S. Vetter, James B. White, and Patrick H. Worley. 2005. Performance evaluation of the cray X1 distributed shared-memory architecture. *IEEE Micro* 25, 1 (2005), 30–40.

Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. 2002. Tarantula: A vector extension to the alpha architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*.

Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W. Ueberhuber. 2005. Efficient utilization of SIMD extensions. *Proc. IEEE* 93, 2 (2005), 409–425.

John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO'92)*.

Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. 2013. Power-efficient computing for compute-intensive GPGPU applications. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'13)*.

Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. 1990. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the International Conference on Supercomputing (ICS'90)*.

Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-Applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

HPGMG. 2018. High-Performance Geometric Multi-Grid Project. Retrieved February 15, 2018 from https://hpgmg.org/.

Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual - Instruction Set Reference. Retrieved February 15, 2018 from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS'13)*.

Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural mechanisms to exploit value structure in SIMT architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*.

Yonggon Kim, Hyunseok Lee, and John Kim. 2011. An alternative memory access scheduling in manycore accelerators. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*.

Christoforos Kozyrakis and David Patterson. 2002. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Proceedings of the International Symposium on Microarchitecture (MICRO'02)*.

Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*.

Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of the International Symposium on Microarchitecture (MICRO'10)*.

Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.

Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO'09)*.

Zhenhong Liu, Syed Gilani, Murali Annavaram, and Nam Sung Kim. 2017. G-scalar: Cost-effective generalized scalar execution architecture for power-efficient GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'17)*.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05)*.

Todd C. Mowry, Monica S. Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS'92)*.

Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO'07)*.

Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA'08)*.

NVIDIA. 2018a. CUDA Software Development Kit. Retrieved February 15, 2018 from https://developer.nvidia.com/cuda-toolkit.

NVIDIA. 2018b. Inline PTX Assembly: CUDA Toolkit Documentation. Retrieved February 15, 2018 from http://docs.nvidia.com/cuda/inline-ptx-assembly.

NVIDIA. 2018c. PTX ISA: CUDA Toolkit Documentation. Retrieved February 15, 2018 from http://docs.nvidia.com/cuda/parallel-thread-execution.

Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA.'13)*.

Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'00)*.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'12)*.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'13)*.

Richard M. Russell. 1978. The CRAY-1 computer system. *Commun. ACM* 21, 1 (Jan. 1978), 63–72.

Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.

John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, Urbana.

Youfeng Wu. 2002. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'02)*.

Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.* 45, 6 (June 2010), 86–97.

Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the International Symposium on Microarchitecture (MICRO'15)*.

George L. Yuan, Ali Bakhodai, and Tor M. Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO'09)*.

Received February 2018; revised August 2018; accepted September 2018