A Workflow for the Synthesis of Irregular Memory Access Microbenchmarks

Kevin Sheridan†, Jered Dominguez-Trujillo, Galen Shipman
{kss,jereddt,gshipman}@lanl.gov
Los Alamos National Laboratory
Los Alamos, New Mexico, USA

Christopher Scott†, Agustin Vaca Valverde
{christopher.scott,jvalverde6}@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Patrick Lavin
prlavin@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Richard Vuduc, Jeffrey Young
{rvuduc,jyoung9}@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Codesign of hardware technologies and applications for sparse memory access dominated workloads can be challenging due to the complexity of the codes, restrictions on access to the codes, or both. To address this challenge we have developed a novel methodology and set of tools, GS Patterns, that analyze and then synthesize memory access patterns from applications of arbitrary complexity. The results of this are patterns which only contain the normalized sampled memory access addresses as an array of indirection indices organized as either gather (read) or scatter (write) operations. These patterns can then be used to generate memory traffic suitable for hardware optimization and design.

In this paper we present GS Patterns including a detailed description of the workflow and algorithms underlying it. The results of analysis and synthesis of access patterns in both proxy and real-world applications using GS Patterns are presented followed by evaluation of performance of these patterns on latest generation hardware technologies including AMD EPYC 9654P, Intel Xeon Max, NVIDIA Grace, and NVIDIA Hopper H100 and H200. Results of this evaluation clearly demonstrate performance differences across different hardware technologies that are not captured by and in many cases are contrary to the performance behavior of simpler memory microbenchmarks.

KEYWORDS

Memory systems, Benchmarking, Sparse Algorithms, Workload Analysis

1 INTRODUCTION

Sparse memory access patterns are common in a variety of applications spanning HPC [19], data intensive computing, and AI. These access patterns are often fundamental to the choice of algorithms and/or data structures necessitated by a variety of constraints such as required discretization approaches or space complexity of the problem being solved. Optimization of these sparse access patterns at the hardware [20], software, and compiler level can provide significant performance improvements but are often encumbered by the sheer complexity of the applications that exhibit sparse patterns making them intractable to a number of commonly used methods in hardware and software codesign such as system modeling and simulation.

Prior work on the Spatter [13] benchmark has helped narrow this gap between the complexity of a full scale application and the utility of microbenchmarks in assessment and codesign of systems for sparse memory access patterns. While a significant advance, this prior work did not address one of the major obstacles in bridging this gap, namely the ability to easily analyze full scale applications and capture the most important / critical memory access patterns in a form suitable for subsequent use in codesign and technology evaluation. Previous work with the Spatter benchmark used closed source tools and analyzed vector-based memory traffic to obtain sparse access patterns.

In this paper we present GS Patterns, a novel tool workflow that enables rapid analysis of sparse memory access patterns in applications on modern hardware and the synthesis of related microbenchmarks. GS Patterns has relatively few requirements beyond a modern program instrumentation tool such as Intel Pin [9, 21] or NVIDIA's NVbit [23], a C/C++ compiler, and the use of CMake for compilation. This workflow allows the user to execute a representative application and dataset on both CPU and GPU technologies to generate compact representations of sparse memory access patterns suitable that can then be used with the previously released Spatter [13] tool for subsequent evaluation on both current (using real hardware) and prospective future technologies (using modeling/simulation). These access patterns are exceedingly compact and are simple enough to enable their use in a variety of environments.

Significant contributions to the state-of-the-art include:

- a novel algorithm for capturing repeated memory access patterns that can be expressed as scatter or gather operations without relying upon a compiler successfully detecting and emitting vector gather/scatter operations.
- a robust open-source toolkit, GS Patterns, for dynamic runtime analysis of memory accesses on modern CPU and GPU architectures
- deep analysis of memory access patterns of applications spanning microbenchmarks to full-scale multi-physics applications exceeding 600K LOC.
- results of benchmarking these access patterns on a selection of the most recent CPU and GPU architectures including

AMD EPYC 9654P, Intel Xeon Max, NVIDIA Grace, and NVIDIA Hopper

The remainder of this paper is organized as follows: Section 2 describes the high-level workflow of GS Patterns. Section 2 provides an overview of the open source GS Patterns tools and how to use them. In section 2.4, changes required in the Spatter benchmark itself in order to enable generation of representative workloads based on sparse patterns captured by GS Patterns are discussed. Section 3 details benchmarking results with Spatter as well as a meta-analysis of patterns including detailed inspection of representative patterns and relation of these patterns back to specific code sections in real world applications followed by discussion in Section 4 and conclusions in Section 5.

2 WORKFLOW OVERVIEW

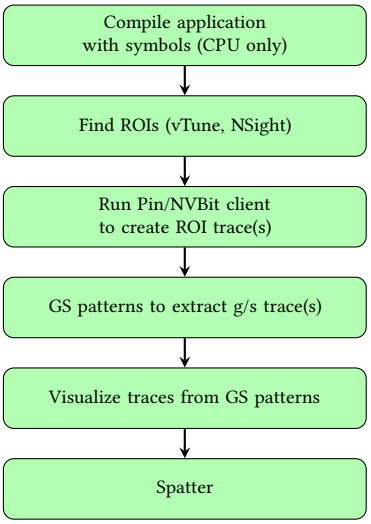


Figure 1: Workflow for extracting and benchmarking non-trivial gather/scatter memory access sequences.

The goal of the workflow (Figure-1) introduced here is to take as input an application we wish to do memory performance analysis on and output concise trace files containing the access sequences of the top memory instruction aggressors. The trace(s) can then be replayed by anyone using spatter on many types of systems to characterize the memory performance of the application's most challenging accesses. Application instrumentation by-hand is not needed, as prior knowledge of the code's algorithms are not required. There are two main benefits of using replay performance analysis. First, spatter is lightweight, runs fast and is fairly easy to compile. Second, some applications are not publicly available, but still need to have their performance measured and analyzed by a broader audience on various systems.

The high-level view of the workflow is divided into six steps, though some steps may be skipped depending on the use cases. First, compile the application with symbol debugging on (not required for CUDA code). Symbols help with identifying source code lines in profiling, but also is a requirement for GS patterns if non-symbol filtering is turned on (CPU only). The second step is to use your favorite profiler to identify functions that are dominating runtime and/or are cache missing excessively. Next, run a memory tracing

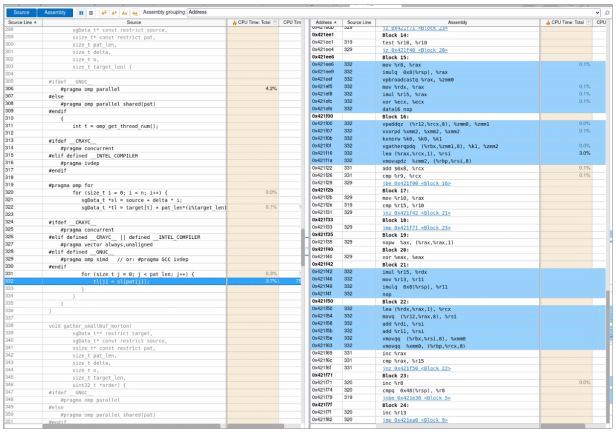


Figure 2: Example of vTune output showing AVX-512 gather/scatter instructions.

tool that only traces the functions of interest from the profiling. Intel Pin tool and NVBit clients were developed for this purpose and are available for download at https://github.com/lanl/gs_patterns [12]. For smaller applications, it is possible to create one monolith trace for the entire application execution (main). Optionally, instruction count ranges can be traced as well. After creating the trace(s) we run GS patterns which extracts memory index sequences of the likely top memory instruction aggressors. The next section goes into detail about how GS patterns works. For step five we developed plotting tools to help visualize each sequence of gather/scatter accesses. Visualizing the memory accesses can help identify patterns in the application and see expected or unexpected memory behavior. Finally, spatter is run with the GS patterns traces as input. Spatter replays the gather/scatter access sequences and produces memory bandwidth metrics.

GS patterns is an open source tool to automatically extract gather and scatter access sequences from arbitrary applications. This tool was developed out of necessity to properly identify and characterize memory accesses likely to hurt application performance. The basic approach of GS patterns is to look for repeating instruction addresses (loops) with load/store operations, record the load/store access sequences from each memory instruction and apply filters to remove entire sequences with only trivial or close to trivial load-/store access sequences. If there are multiple loads/stores in one instruction, each is treated as their own access sequence. The final results are the likely top memory instruction aggressors with their respective normalized (lowest array index set to 0) index access sequences. Scalar and vector instructions are both analyzed with each treated as a sequence of memory accesses. The input to GS patterns are DynamoRio formatted traces which must first be created using DynamoRio, Intel pin or Nvidia NVBit tool clients provided in the GS patterns github repository [12]. Multiple process elements with multi-threading is supported in the pin clients, but only thread 0 on process element 0 is traced by default with other options being configurable. This feature is desirable as we can run large problems on many nodes while still getting a concise trace of what is needed. If there are widely different code paths and/or data patterns among the process elements or threads then multiple traces may be needed.

Table 1: Memory access sequence removal filters.

Filter #	Description
1	Index distances are only -1, 0, and/or 1
2	No symbol
3	Not in top 10 window appearance counts
4	Less than 1024 instances
5	Less than 6 unique index distances and less than 50% out of bounds distances*

*Filter rules apply to each full memory access sequence. Sub-sequences are not removed. *Default out of bounds index distances in* $(-\infty, -513]$ *or* $[513, \infty)$.

$$\textbf{GatherWindow} = \begin{array}{c} \begin{matrix} \text{maddr1} & \text{maddr2} & \dots & \text{maddrN} & \textit{iaddr} \end{matrix} \\ \begin{bmatrix} 0x0080 & 0x0088 & \dots & 0x1888 \\ 0x1280 & 0x1388 & \dots & 0x0883 \\ \dots & \dots & \dots & \dots \\ 0x9080 & 0x5088 & \dots & 0x3881 \end{bmatrix} \begin{matrix} \text{0x0000} \\ \text{0x0002} \\ \dots \\ \text{0x1018} \end{matrix} \end{array} \tag{1}$$

$$\textbf{ScatterWindow} = \begin{array}{c} \begin{matrix} \text{maddr1} & \text{maddr2} & \dots & \text{maddrN} & \textit{iaddr} \end{matrix} \\ \begin{bmatrix} 0x0480 & 0x0488 & \dots & 0x1488 \\ 0x1780 & 0x1788 & \dots & 0x0783 \\ \dots & \dots & \dots & \dots \\ 0x9580 & 0x5588 & \dots & 0x3581 \end{bmatrix} \begin{matrix} \text{0x0001} \\ \text{0x0003} \\ \dots \\ \text{0x1019} \end{matrix} \end{array} \tag{2}$$

2.1 GS Patterns Core Implementation

The implementation of GS patterns required some optimizations that have little or no effect on the accurate discovery of the result access sequences. A pair of moving windows , one for loads and the other for stores, are used to record memory instruction addresses with their respective access sequences efficiently. Examples (1) and (2) show the layout of the windows with truncated addresses for illustration purposes. By default, the windows size is 1024 memory instruction addresses. This can be changed but 1024 entries was determined to be sufficient to discover all or nearly all non-trivial gather/scatter sequences for our applications. The pair of windows are considered full and then emptied if the number of unique memory instruction addresses in either window exceeds 1024 or the accumulated size in any memory instruction address sequence exceeds 64 bytes. 64-bytes as the max accumulated load-/store size per window row works well as the max vector width in most current CPU hardware is 512-bits. 512-bits is also suited well for most CPU compilers as they usually don't expect vector widths that exceed 512-bits. GS patterns makes the max size value configurable to allow for more experimentation with large/smaller vector widths, for example 2048-bits for Nvidia GPUs. Unrolling in the application's compilation is not necessarily recommended because the compiler may split memory instructions into multiple instruction addresses, though vectorization may be facilitated. GS patterns considers memory instruction addresses independent, each with their own memory sequence.

There are two passes in GS patterns. The first pass finds all unique memory instruction addresses (iaddrs) and puts them in the master list. The second pass records the memory address (maddr) access sequences for each memory instruction address. The top non-trivial gather / scatters stay in the master list if they make it through a series of filters throughout the passes. The defined filters used in Table-1 work well for finding non-trivial access sequences in our applications. Though, the filter parameters are easily configurable if less or more stringent filtering is needed. For example, disabling the no symbol filter (Filter 2) may be desirable if non-symbol library accesses are of interest. The full GS patterns algorithm flowchart is illustrated in Figure-3.

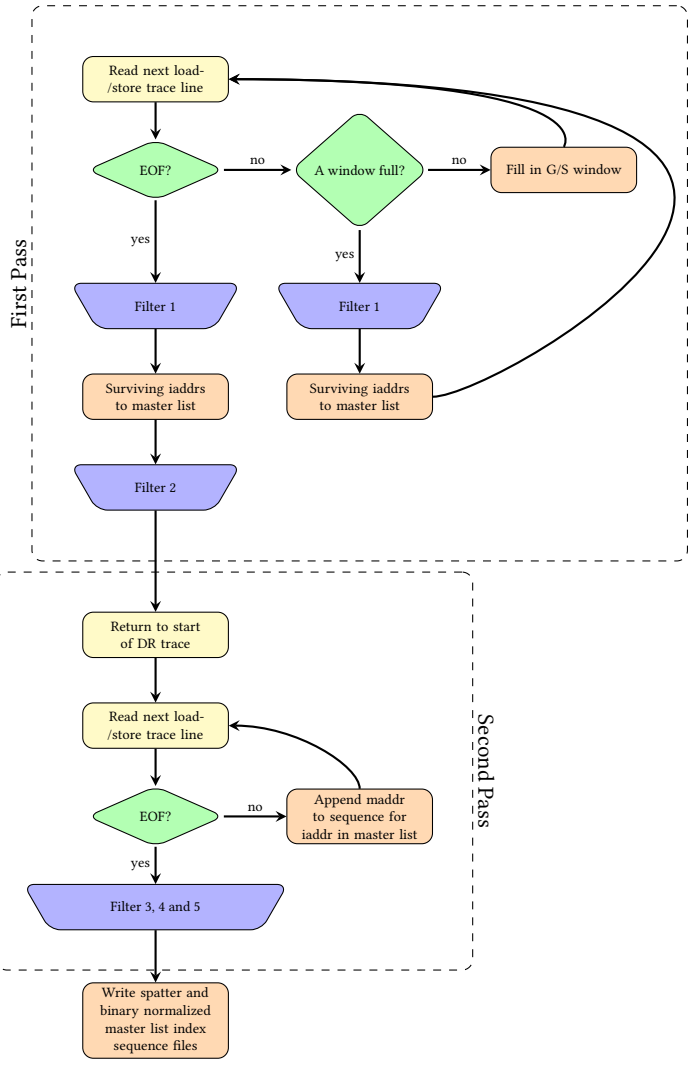


Figure 3: Core GS patterns algorithm.

2.2 GPU Features for GS Patterns

Increasingly GPUs represent a significant source of computational (and memory access) concentration within many scientific and high-performance applications, where SIMD (Single Instruction Multiple Data) processing is used to parallelize execution of specific portions of these applications. Such applications typically employ NVIDIA's CUDA framework and are structured around several kernel functions which execute directly on the GPU. It is therefore desirable to develop a way to trace CUDA based application kernels and to enhance GS patterns to support the associated traces.

To support GPU based memory patterns we identified 3 logical yet interrelated areas of work.

(1) Utilization of NVIDIA's NVBit [23] – to provide trace data for memory accesses of CUDA kernels.
(2) Restructuring of GS patterns to separate bucketing and pattern generation logic, from trace file reading and memory address handling.
(3) Organizing GS patterns into 3 components; a core shared library which provides the pattern bucketing and trace handling functionally (`libgs_patterns_core`), a binary runner for command line execution (`gs_patterns`) and an NVBit tool shared library suitable for instrumenting CUDA kernels using NVBit (`gsnv_trace`).

2.2.1 Memory Traces via NVBit. NVBit is a framework created by NVIDIA for instrumenting CUDA kernels, it provides an API which gives direct access to the raw SASS (an architecture dependent assembly created for NVIDIA GPUs) instructions performed by the GPU. These APIs can be used to build an NVBit tool (similar to a Pin tool) which is loaded at runtime by a CUDA application. The NVBit tool itself is a shared library loaded via `LD_PRELOAD` which intercepts calls made by CUDA based kernels. The typical implementation inserts instructions before and/or after each kernel instruction allowing the tool to inspect and/or modify those instructions at runtime. This approach is used to insert instrumentation code which writes the details of instructions of interest (in our case memory access instructions) over a channel to be handled by GS Patterns.

Mem Trace (`mem_trace`), an NVBit tool provided with the NVBit distribution was used as the basis of our trace efforts. Mem Trace utilized the most performant way of extracting and passing back traces to the GS Patterns code running on the CPU where there was greater flexibility in how traces are handled. Mem trace also already provided the instruction opcode and importantly the vector of memory addresses being requested by the GPU threads. The `mem_trace` tool was therefore modified to instantiate a new GSPatternsForNV class when loaded. GSPatternsForNV handles adapting GPU memory accesses to GS Patterns internal abstractions and contains methods which are invoked on every memory instruction so that the addresses being referenced can be bucketed by GS Patterns. Additional information has also been added to the instrumentation code to extract the source lines where the memory instruction occurred as well as passing back information describing whether the memory access was a load, a store, and the memory short opcode used (useful for classifying the type of memory access). We also pass back mapping information for these pieces of information which are included in the header of the NVBit trace

```
GSNV_TRACE: CTX  - grid_launch_id 1 - CTA 0,0,0 - warp 11 - LDG.E.64 - shortOpcode: 0 isLoad: 1 isStore: 0 size: 8 -
0x00007f640ff81b00 0x00007f640ff81b18 0x00007f640ff81b30 0x00007f640ff81b48 0x00007f640ff81b60 0x00007f640ff81b78 0x00007f640ff81b90
0x00007f640ff81ba8 0x00007f640ff81bc0 0x00007f640ff81bd8 0x00007f640ff81bf0 0x00007f640ff81c08 0x00007f640ff81c20 0x00007f640ff81c38
0x00007f640ff81c50 0x00007f640ff81c68 0x00007f640ff81c80 0x00007f640ff81c98 0x00007f640ff81cb0 0x00007f640ff81cc8 0x00007f640ff81ce0
0x00007f640ff81cf8 0x00007f640ff81d10 0x00007f640ff81d28 0x00007f640ff81d40 0x00007f640ff81d58 0x00007f640ff81d70 0x00007f640ff81d88
0x00007f640ff81da0 0x00007f640ff81db8 0x00007f640ff81dd0 0x00007f640ff81de8
GSNV_TRACE: CTX  - grid_launch_id 1 - CTA 0,0,0 - warp 15 - LDG.E.64 - shortOpcode: 0 isLoad: 1 isStore: 0 size: 8 -
0x00007f640ff82700 0x00007f640ff82718 0x00007f640ff82730 0x00007f640ff82748 0x00007f640ff82760 0x00007f640ff82778 0x00007f640ff82790
0x00007f640ff827a8 0x00007f640ff827c0 0x00007f640ff827d8 0x00007f640ff827f0 0x00007f640ff82808 0x00007f640ff82820 0x00007f640ff82838
0x00007f640ff82850 0x00007f640ff82868 0x00007f640ff82880 0x00007f640ff82898 0x00007f640ff828b0 0x00007f640ff828c8 0x00007f640ff828e0
0x00007f640ff828f8 0x00007f640ff82910 0x00007f640ff82928 0x00007f640ff82940 0x00007f640ff82958 0x00007f640ff82970 0x00007f640ff82988
0x00007f640ff829a0 0x00007f640ff829b8 0x00007f640ff829d0 0x00007f640ff829e8
GSNV_TRACE: CTX  - grid_launch_id 1 - CTA 1,0,0 - warp 5 - LDG.E.64 - shortOpcode: 0 isLoad: 1 isStore: 0 size: 8 -
0x00007f640ff83c00 0x00007f640ff83c18 0x00007f640ff83c30 0x00007f640ff83c48 0x00007f640ff83c60 0x00007f640ff83c78 0x00007f640ff83c90
0x00007f640ff83ca8 0x00007f640ff83cc0 0x00007f640ff83cd8 0x00007f640ff83cf0 0x00007f640ff83d08 0x00007f640ff83d20 0x00007f640ff83d38
0x00007f640ff83d50 0x00007f640ff83d68 0x00007f640ff83d80 0x00007f640ff83d98 0x00007f640ff83db0 0x00007f640ff83dc8 0x00007f640ff83de0
0x00007f640ff83df8 0x00007f640ff83e10 0x00007f640ff83e28 0x00007f640ff83e40 0x00007f640ff83e58 0x00007f640ff83e70 0x00007f640ff83e88
0x00007f640ff83ea0 0x00007f640ff83eb8 0x00007f640ff83ed0 0x00007f640ff83ee8
```

Figure 4: Example of warp-based memory traces showing 64bit load instructions executed by warps of 32 GPU threads, and the associated source addresses.

so that integer representations can be used to reduce trace file size and improve performance. The enhanced Mem Trace NVBit tool was called GSNV Trace (`gsnv_trace`).

2.2.2 Creating a plugin-based Infrastructure. In addition to passing back memory addresses to GS Patterns, GS Patterns can handle the differences between CUDA memory accesses which are provided per warp (e.g. 32 addresses at a time) vs CPU addresses which are provided one at a time. More details on warps and how they relate to GPU threads are provided in section 2.2.3, and a warp-based memory trace example is shown in Figure-4. As alluded to in the previous section these differences are handled by separate classes allowing the same bucketing logic within GS Patterns to be used regardless of whether the memory access was CPU or GPU based. Other differences between the CPU and GPU memory accesses include filtering out contiguous memory accesses which we defined as a warp memory access where the minimum difference between any 2 addresses in the warp is $\leq$ the size of the memory request (usually 8 bytes). In GPUs these are typically coalesced to reduce bank conflicts and improve performance, so filtering them out allows us to focus on sparse accesses and further reduces noise. We also filtered out memory access from divergent threads, identified as when less than all 32 threads in a warp have issued the memory access instruction. We believe these are infrequent enough in a well written CUDA kernel that discarding them won't significantly impact the resulting memory patterns.

The `gs_patterns_core` library was developed to be agnostic regarding the source of the memory access (CPU or GPU based) being passed to it, while preserving the existing CPU pattern behavior. To allow for this and further extensions in the future, GS Patterns was migrated to C++ and the implementation details were exposed through various virtual functions which act as interfaces. The restructured Pin and NVBit based implementations, GSPatternsForPin and GSPatternsForNV, respectively, implement these differences in behavior and provide the appropriate details to the bucketing code within `gs_patterns_core` thru standard interfaces.

In the NVBit based implementation, NBVit provides a direct API for retrieving the source line for each instruction which is available as long as the kernel was compiled with "–generate-line-info". This alleviated the need to call `addr2line` as well as the need to have the original binary available in order to generate patterns with source line details. This not only improved performance but also allowed the target program to be built with full optimization levels and without debug symbols which may not always be available.

2.2.3 Gather-Scatter on GPU Approach. As mentioned in the previous section, adapting GS Patterns to use GPU based memory addresses required us to take a warp's memory access of 32 threads

and normalize that into 32 memory accesses. However, there are other differences which are important to mention.

Generally speaking, GPUs utilize Cooperative Thread Arrays (CTAs). This is essentially a block of threads executed in groups called warps according to the hardware capabilities but typically 32 at a time. Each warp executes an instruction in lock step, where these threads have access to global memory, a per block shared memory region, a read-only texture memory region, their own stack, registers, and thread local storage. Each memory access that is provided by `gsnv_trace` allows us to determine the type of memory access. We use this information derived from the opcode used, to filter this down to just global and shared memory accesses (both are currently handled). We also use the address of the kernel function plus offset of the respective memory instruction as the instruction address (iaddr) which is required by GS Patterns. Also, a warp memory access is considered to be a vector memory access of sorts, in that each thread is in effect using an index derived from its own thread ID (and potentially other information) into the appropriate vector. This indexing is similar to what Intel's AVX-512 instruction does but with CTAs each thread is responsible for a single index. Thus, rather than 512bits (64bits * 8 accesses) of memory being read or written with AVX-512, a GPU can do 2048bits (64bits * 32 accesses) with each warp's memory access. Converting this to bytes AVX-512 can do 64 machine words on a 64bit machine while CTAs can do 256. These adaptions allow us to map GPU memory accesses to patterns using the same bucketing code within `gs_patterns_core`.

One other way the patterns generated for GPU memory accesses differ from those of CPU memory accesses, is in terms of which threads are traced. For CPU based patterns GS Patterns selects the memory accesses of an exemplar thread, using that thread's activity to reflect the pattern intrinsic to the application's design. For GPU based patterns the new GS Patterns traces all threads in a warp as they all cooperate to perform a single memory instruction across many addresses and so the composition of all memory accesses performed by the kernel are reflected within the generated pattern. Pattern generation can optionally be limited to a single warp (warp 0) analogous to the single thread used in CPU based pattern generation. These differences makes it difficult to directly compare an application's CPU pattern with its GPU one (assuming the application supports execution in CPU and GPU modes). There is not yet an way to easily compact the GPU results to derive something which can be compared to the CPU pattern. However for the purpose of comparing GPU to GPU memory accesses within the Spatter workflow the current approach is sufficient.

2.2.4 Results. The GPU enabled features for GS patterns can handle either Pin or DynamoRIO trace files for CPU memory accesses and NVBit trace files (generated by `gsnv_trace`) for GPU memory accesses. The `gsnv_trace` NVBit tool can be used to extract memory accesses and memory patterns from one or more kernels within any CUDA application (as long as the application has been built with the NVCC `--generate-line-info` option).

We have used `gsnv_trace` to extract memory access traces from the UMT application ([16]) which has been built using CUDA 12.3 and using the `--use_cuda_sweep` runtime option which will invoke the SweepUCBxyzKernel CUDA kernel. Memory patterns which were generated from execution of UMT's benchmark 1 and 2 for both GPU and CPU based memory accesses are available on the Spatter Patterns GitHub repo [18].

2.2.5 Related tools for GPU based patterns. Tools such as Ocelot [6], GTPin [21] and Dyninst [22], provide similar functionality to NVBit and are worth noting in the context of GS Patterns. Ocelet which predates NVBit, operates at the PTX (Parallel Thread Execution) level; a virtual ISA, where it provides tools for mapping between data parallel execution and various models of threaded execution. It can be used along with approaches from [11] to discover memory patterns in applications and remap those patterns onto varying hardware architectures. Unlike Ocelot, NVBit works at a lower level e.g. on the raw SASS assembly executed on the GPU device, where it dynamically recompiles code before execution. GTPin an instrumentation framework created by Intel, can be used to instrument Intel based GPU's. It provides similar capabilities to NVBit in that it provides access to the low level instructions in this case EU assembly (including opcodes and their operands) executed on the GPU at runtime without access to source code. Dyninst an api for either static or dynamic instrumentation provides similar functionality to NVBit and supports various CPUs as well as AMD GPU's. In future versions of GS Patterns, support for Intel or AMD GPU backends could be added based on GTPin or Dyninst respectively.

2.3 Spatter Patterns

Spatter is a benchmark designed to evaluate the performance of sparse memory access on CPUs and GPUs [13]. Spatter takes as input a number of memory access patterns and outputs the performance of each in MB/s. These patterns can be specified individually on the command line or they can be generated from traces of applications in order to create sets of patterns that are more representative of application behavior than those generated by classical memory benchmarks such as STREAM or pointer chasing. The Spatter paper [13] used a proprietary simulator to generate Spatter inputs, and as such could not be shared with the community. GS Patterns, however, is an open source tool, meaning other researchers will be able to repeat the entire workflow in this paper.

Spatter is able to represent a large class of memory access patterns. A Spatter memory access pattern consists of four parts: (1) kernel, (2) offsets, (3) delta, and (4) count.

The *kernel* specifies if we will be doing gathers, scatters, or one of a number of new kernels, explained in Section 2.3.2. The *offsets* specify the data elements that will be read or written by the gather or scatter operations. The *delta* is how far we will move the base pointer in our data array between gather or scatter operations, and the *count* is how many gathers or scatters we will perform. With all of that in mind, let us take a look at some pseudocode.

Algorithm 1 Gather pseudocode

```
for i in 1..count do
  src = src + delta * i
  for j in 1..len(offset) do
    dst[j] = src[offset[j]]
  end for
end for
```

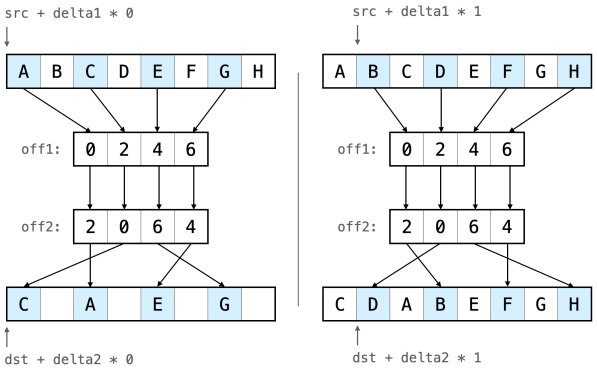


Figure 5: A visual representation of the first two iterations of the GatherScatter kernel. Both `delta1` and `delta2` are set to 1, and the offset buffers are shown in the figure.

2.3.1 Kernel pseudocode. The gather kernel is defined in 1. In words, the gather kernel performs the gather defined by the offset buffer by coping data from *src* to *dst*, then moves the base pointer by *delta*, and repeats this *count* times. The only difference between this and the scatter kernel is that the scatter kernel performs indirect writes instead of indirect reads, so the offset buffer is on the left of the assignment and the *dst* buffer is incremented, instead of *src*.

2.3.2 New Spatter kernels. There are three new kernels in Spatter since the original paper was published, *MultiGather*, *MultiScatter*, and *GatherScatter*. The *Multi* kernels utilize multiple levels of indirection, and the *GatherScatter* kernel will first gather data and then scatter it. See Algorithm 2 and Algorithm 3 for the pseudocode for these kernels. Notice that these patterns all require multiple *offset* buffers, which must be of equal length. Additionally, a depiction of the GatherScatter kernel is in Figure 5.

Algorithm 2 MultiGather pseudocode

```
for i in 1..Count do
  src = src + delta * i
  for j in 1..len(off1) do
    dst[j] = src[off1[off2[j]]]
  end for
end for
```

Algorithm 3 GatherScatter pseudocode

```
for i in 1..Count do
  src = src + delta1 * i
  dst = dst + delta2 * i
  for j in 1..len(off1) do
    dst[off2[j]] = src[off1[j]]
  end for
end for
```

2.4 Spatter Improvements

Since the original Spatter publication, a number of improvements have been made to make it more suitable for evaluating HPC applications. These improvements include the ability for Spatter to ingest very long gather/scatter traces, MPI calls for weak-scaling and strong-scaling experiments, and the ability to truncate patterns for GPU throughput testing.

Figure 6 provides a detailed illustration of how we implemented each of these scaling and test workflows within the Spatter phase of Figure 1. The weak-scaling and strong-scaling tests indicate how the entire memory access pattern is passed to each rank for weak-scale testing while the pattern is partitioned in to continuous chunks for the strong-scaling tests. The GPU throughput test is implemented as a sweep along the number of accesses performed by the memory system across single-rank runs. As shown, this requires the ability to truncate and expand the access pattern to obtain the necessary number of accesses to saturate the GPUs SMs and memory subsystem.

2.4.1 Offset Buffer Length. The prior work with Spatter focused on very short gathers and scatters, typically of length 16. However, this limits the ability of a single pattern to capture application information. As such, Spatter was improved to support very long offset buffers. In this work, the application-derived patterns will be much longer.

2.4.2 MPI Support. While Spatter contained OpenMP support to perform multi-threaded runs on a shared buffer, MPI support for multi-process runs was needed to better represent a greater variety of application run configurations, such as MPI-only or MPI + OpenMP. Given the assumption that the indirect memory access pattern on each rank have similar characteristics during the application run, we can use the same pattern on each rank to implement MPI support.

Adding MPI support also required us to provide additional features and scripts to support scaling experiments. Many of these scripts [10] can be customized for weak-scaling, strong-scaling, or GPU throughput testing (See 2.4.3) on a variety of hardware and core counts with the use of command-line flags.

Weak-Scaling. Weak scaling support required the assumption that the indirect memory access patterns across ranks have similar characteristics. Therefore, each rank gets a copy of the entire pattern output from GS patterns. The gather/scatter kernel is repeated 10 times by default, with each run separated by an MPI Barrier before recording the performance. We then developed scaling scripts which sweep across the number of ranks and aggregate data from each run to perform a full weak-scaling experiment for a single pattern.

Strong-Scaling. Strong scaling support in Spatter required support for partitioning the indirect memory access pattern across ranks as evenly as possible (see Figure 6). To approximate the data decomposition as a problem is strong-scaled, each rank determines which contiguous chunk of the pattern it is responsible for prior to performing kernel runs separated by MPI barriers. The same set of scaling scripts can then be configured to run strong-scaling experiments on a single pattern.

2.4.3 GPU Throughput Testing. Performing throughput tests on a GPU requires the user to vary the amount of memory moved through the memory subsystem until the GPU is fully saturated. Supporting throughput testing with Spatter required the addition

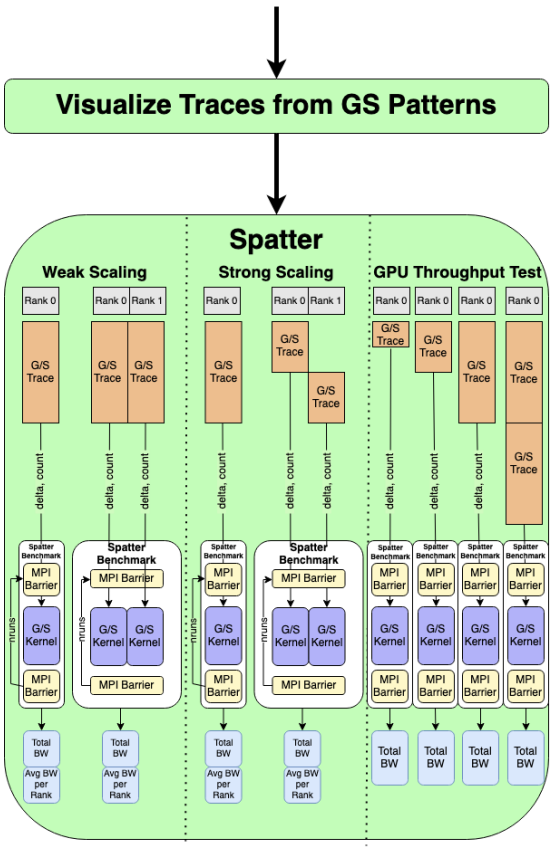


Figure 6: Spatter workflow for scaling and throughput testing.

of two additional features to Spatter: 1) the ability for Spatter to truncate a pattern to a certain length as specified by user input, and 2) the ability for Spatter to expand a pattern an arbitrary number of times. This allows the user to increase the effective size of the pattern and the amount of memory traffic generated in order to saturate the GPU with enough parallelism.

Specifically, looking at Algorithm 1, truncating the pattern would be as simple as decreasing the size of the *offset* array. This would have the effect of decreasing the size of the *dst* array, since *length(dst) = lenght(offest)*, while the size of the *src* array would decrease by a data dependent amount since the length of *src* is proportional to *max(offset)*.

Alternatively, expanding the pattern replayed by the Spatter benchmark once the entire memory access pattern has already been used to populate the *offset* array requires increasing the *count* parameter in Algorithm 1. The result is that the base of the *src* array is shifted by *delta* items (which is set to a default of 8), and the offset pattern is then repeated until the base of the *src* array has been shifted *count* times. This doesn't require any additional space for the *offset* array or the *dst* array, but requires the *src* array to expand to *length(src) = max(offset) + (count-1) * delta*. The *count* parameter is then used for a throughput test to increase the number of accesses performed by the benchmark. This is done for a given pattern of fixed length collected by GS Patterns and the workflow in Figure 1.

2.4.4 Atomics. The scatter operation is susceptible to race conditions when the offset array contains 2 equivalent offsets [17]. To guarantee correctness in multi-threaded configurations, we added an option to Spatter which enables atomic operations when using the OpenMP and CUDA backends during any kernel which contains a scatter.

3 EXPERIMENTS

The workflow illustrated in Figure 1 was utilized to collect indirect access patterns of 5 HPC applications from various scientific domains. This involved identifying regions of interest with vTUNE or NSight, running the PIN Tool and NVBit client to generate the required traces, using GS Patterns to identify and extract the gather/scatter patterns, and visualizing the gather/scatter patterns extracted from each application. The applications we collected patterns from are described in detail in Section 3.2, while the patterns are visualized in Section 3.3. Finally, the collected patterns were run through Spatter to conduct weak-scaling studies on 6 CPU hardware platforms and to perform throughput studies on 4 GPU hardware platforms. The hardware utilized for this step is described in Section 3.1, while the results are presented and discussed in Section 3.4.

3.1 Experimental Setup

A heterogeneous research testbed was used to collect the bandwidth results presented in this section across a variety of hardware platforms. These hardware platforms span 6 CPU node configurations and 4 GPU node configurations with a variety of memory subsystems including DDR4, DDR5, LPDDR5X, HBM2e, and HBM3. Details of the CPU node configurations and their memory subsystems can be found in Table 2, while the GPU node configurations and their memory subsystems can be found in Table 3.

First, the STREAM [14] benchmark was used to collect peak DRAM bandwidth as a function of thread count on the CPU platforms while the BabelSTREAM [5] benchmark was used on GPUs. The STREAM benchmark was built to iterate over 150 million elements on the Intel and NVidia CPUs, requiring 1.1 GiB per array and 3.4 GiB in total, while it was built to iterate over 800 million elements on the AMD CPU requiring 6.0 GiB per array, ensuring the arrays are much larger than the LLC. STREAM was run with *OMP_PROC_BIND=spread*. The BabelSTREAM benchmark used an array size of 268.4 MB for a total size of 805.3 MB. Figure 7 illustrates the results of the STREAM and BabelSTREAM benchmarks.

Next, weak-scaling studies were performed for the collected patterns. The weak-scaling tests were performed on all 6 CPU configurations for 99 patterns across the 5 applications in Section 3.2. The total bandwidth and average bandwidth per rank were measured for each pattern as the number of processes was swept from 1 up to the number of cores on the CPU platform. Hyperthreading was not used.

Finally, the GPU throughput tests were conducted across all patterns for each of the 4 GPU configurations. This was performed utilizing the scripts in the LANL Spatter repository [10] with the default setup which sweeps from a *count* of 1 to 512. These experiments all used the default *delta* of 8 and exercised the gather and scatter kernels of the Spatter benchmark. Additionally, these experiments required atomics to be enabled for any scatter patterns to

Table 2: CPU node configurations.

	Grace Superchip	9480 Max	Grace	9480L Platinum	EPYC 9654P	Gold 6152
CPU Cores / Socket	72	56	72	56	96	22
Sockets	2	2	1	2	1	2
L3 Cache	228 MB	225 MB	114 MB	215 MB	384 MB	30.25 MB
Memory Type	LPDDR5X	HBM2e	LPDDR5X	DDR5-4800	DDR5-4800	DDR4-2666
Memory Spec	8533 MT/s	3200 MT/s	8533 MT/s	4800 MT/s	4800 MT/s	2666 MT/s
Memory Bandwidth	1024 GB/s	1638.4 GB/s	512 GB/s	614.4 GB/s (16 channels)	460.8 GB/s (12 channels)	128.0 GB/s

Table 3: GPU node configurations.

	GH 200	H100-PCIe	A100-SXM4-40GB	V100-PCIe
SMs	132	114	108	80
L2 Cache	60 MiB	50 MiB	40 MiB	6 MiB
Memory Type	HBM3	HBM2e	HBM2e	HBM2
Total Memory Bandwidth	4.0 TB/s	2.0 TB/s	1.55 TB/s	900 GB/s

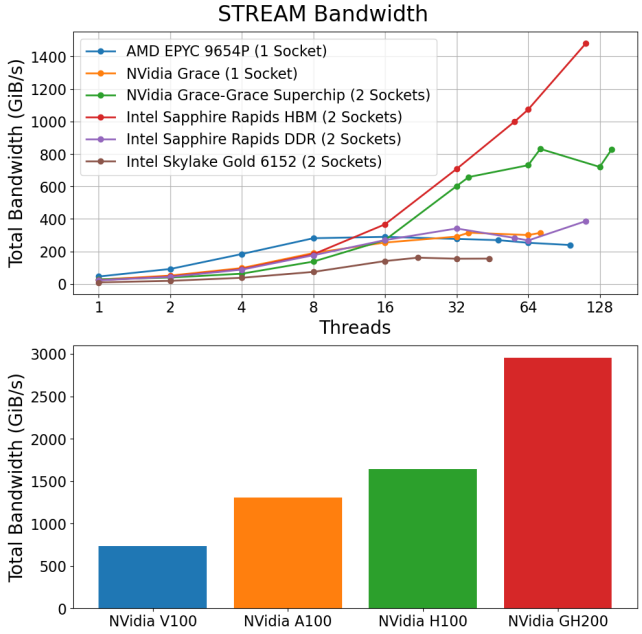


Figure 7: STREAM results on CPUs (Top) and GPUs (Bottom).

ensure correctness. The overhead of atomics has been measured to be between 10%-30% in previous experiments for the LANL ATS-5 benchmarking effort.

3.2 Applications Used

3.2.1 UMT. UMT [8] (Unstructured Mesh Transport) is a Lawrence Livermore National Laboratory (LLNL) Advanced Simulation and Computing (ASC) proxy application (mini-app) that solves a thermal radiative transport equation using discrete ordinates (Sn). It utilizes an upstream corner balance method to compute the solution to the Boltzmann transport equation on unstructured spatial grids. This class of problems is characterized by tens of thousands of unknowns per zone and upwards of millions of zones, thus requiring large, scalable, parallel computing platforms with tens of processors per node.

3.2.2 xRAGE. xRAGE [7, 15] is a 1D, 2D, and 3D, multi-material radiation transport hydrodynamics code developed by LANL. The Hydrodynamics are based on Euler's equations using cell-based Adaptive Mesh Refinement (AMR). xRAGE is able to simulate behavior that spans very large dynamical state and phase spaces. The choice of data structures to enable iteration over cells at the highest resolution and to access adjacent cells in xRAGE result in inherently sparse memory accesses (gather/scatter).

3.2.3 FLAG. FLAG [4] is an arbitrary Lagrangian-Eulerian (ALE) adaptive mesh refinement (AMR) multiphysics hydrodynamics code developed by LANL. To enable resolution to be concentrated where it is most needed, FLAG supports fully unstructured polytopal grids. FLAG has robust support for multiple materials with most operations taking place within material regions rather than across the global mesh. The data structures used to support fully unstructured grids and multiple materials result in inherently sparse memory accesses (gather/scatter).

3.2.4 Quicksilver. Quicksilver is an open source proxy application for the Mercury Monte Carlo particle transport code [3]. Quicksilver solves a time-dependent fixed source particle transport problem with significantly simplified and approximate physics, artificial multigroup cross sections, and a simple mesh-based geometric representation. The use of a dynamically *qs_vector* data structure results in several different variants of sparse accesses as the core vector structure is read and written by getter and setter methods as well as being referenced often in the *Collision_Event* function.

3.2.5 Branson. Branson is an proxy application for parallel Monte Carlo transport[2]. Branson implements a particle passing method for both replicated and domain decomposed meshes. In Branson, individual particles are transported through the mesh and often diverge significantly from the paths taken from previously adjacent particles. Then can result in sparse memory access into a cell array

representing the underlying mesh from one particle transport to another.

3.3 Pattern Examination

As discussed in Section 2, the GS Patterns workflow includes an optional step for visualizing the traces extracted from applications. This visualization uses a simple Python script to visualize the offsets in a pattern, the deltas between accesses, and histograms of the delta values found within a pattern. This visualization can be correlated with the specific code functions reported via the GS Patterns tool to investigate the sparsity characteristics of a specific gather or scatter pattern.

3.3.1 Quicksilver Patterns. Listing 1 and listing 2 both show code segments that result in sparse accesses that are captured by the GS Patterns workflow.

In the case of listing 1, the access to $\_crossSection[group]$ results in a gather operation that is accessing a vector datatype common to Quicksilver, qs_vector. The resultant pattern is shown in Fig. 8a with the zoomed in version shown in Fig. 8c. What is important to note here is that there this pattern demonstrates a relatively wide set of offsets that is repeated across the application's relative timescale. The zoomed in version also shows that each "set" of accesses has a somewhat sequential pattern, indicating some locality despite the large offsets.

Figure 8b and Fig. 8d visualize the pattern represented by the write to the x vector element in listing 2. Interestingly, this pattern is visualized as a sickle shape with a somewhat random looking set of offsets followed by a short monotonically increasing phase of scatter operations.

```
189 HOST_DEVICE
190 // Return the cross section for this energy group
191 double NuclearDataReaction::getCrossSection(unsigned int group)
192 {
193   qs_assert(group < _crossSection.size());
194   return _crossSection[group]; // maps to Gather0
195 }
196 HOST_DEVICE_END
```

Listing 1: Quicksilver Gather0

```
20  HOST_DEVICE_CUDA
21  MC_Vector& operator=( const MC_Vector&tmp )
22  {
23    if ( this == &tmp ) { return *this; }
24
25    x = tmp.x; // maps to Scatter1
26    y = tmp.y;
27    z = tmp.z;
28
29    return *this;
```

Listing 2: Quicksilver Scatter1

3.3.2 Branson Patterns. Listing 3 demonstrates another gather operation captured by GS Patterns that is accessing a data structure called *abs_groups*. The resultant pattern in Fig. 9a has fewer data points than the Quicksilver example gather, but it also demonstrate an increasing offset with some amount of "near-0" offset values, indicating reads from a small range of offsets. An exemplar Scatter from Branson (code listing not shown) illustrates a repeated set of writes to the same locations that are offset by a very large relative offset (approximately 9 trillion).

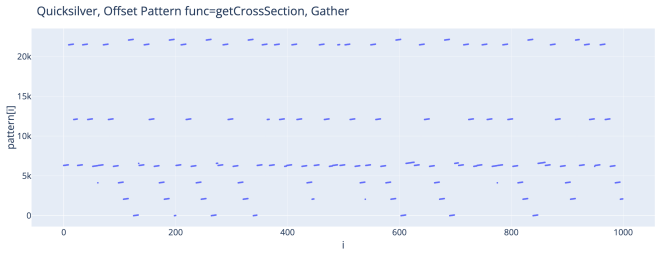
3.3.3 xRAGE Patterns. A deeper dive in to the visualization tools used in the workflow is illustrated by Figure 10. We have developed tools to understand how the offset array is distributed, allowing the temporal and spatial locality to be visually inspected.

Inspecting the full offset arrays of the Spatter5 gather pattern (Figures 10a) and the Spatter9 scatter pattern (Figure 10b) from the xRAGE application indicates several substructures and phases exist during these sparse access patterns. They are not random, but largely characterized by short monotonically increasing accesses in address space, followed by hard to predict jumps to the next monotonically increasing phase. Zooming in on to a single one of these monotonically increasing phases with 10,000 accesses (Figures 10c - 10d) indicates additional hard to predict sparsity even within these regions. Furthermore, we can visualize the evolution of the deltas/jumps in address space between subsequent accesses as the offset array is iterated through (Figures 10e - 10f) while inspecting histograms to identify the frequency of deltas of particular sizes. For these particular xRAGE patterns, many subsequent accesses have deltas of 10,000 to 30,000 items between them, providing a window in to the level and type of sparsity this pattern exhibits. While outside the scope of this paper, future analysis is planned to understand and model how different types of sparsity and sparse patterns affect the bandwidth performance of different systems.

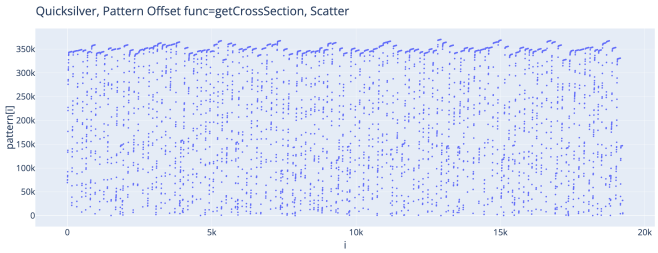
3.4 Spatter Results

Understanding the memory performance and bottlenecks of scientific applications is paramount to improving their performance on current and future architectures. Many microbenchmarks, such as STREAM and GUPS, allow users to measure the hardware performance of the systems they use to run their workloads, while Spatter provides the ability to analyze the interaction between hardware, software, and data that more closely resembles the conditions the hardware would experience while running a workload. As many hardware platforms continue to become more complex and heterogeneous, it is important for flexible workflows to exist which can allow users to capture access patterns directly from applications and to "replay" them while performing scaling studies for evaluation and benchmarking on a wide cross-section of existing architectures.

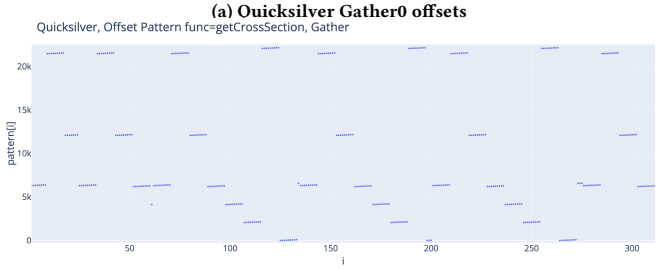
In this section, a number of experiments are performed to demonstrate the ability of the PIN and NVBit based workflows described in Section 2 along with the Spatter improvements outlined in Section 2.2.5 to collect gather/scatter patterns from full applications and to evaluate their performance on a variety of both CPU and GPU node configurations. In Section 3.4.1, we demonstrate the ability to use the MPI improvements to Spatter to conduct weak scaling studies on the CPUs in Table 2 in addition to throughput studies on the GPUs in Table 3 with patterns collected using the workflow described in this paper. Next, we demonstrate the ability to



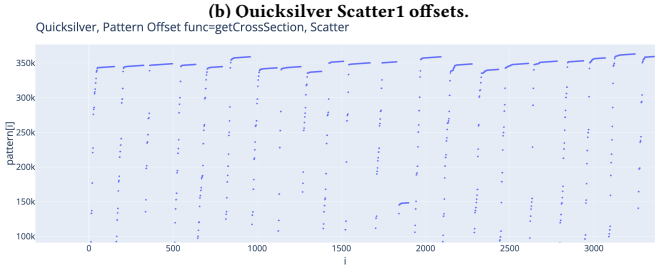
(a) Quicksilver Gather0 offsets



(b) Quicksilver Scatter1 offsets.



(c) Quicksilver Gather0 - zoomed in view



(d) Quicksilver Scatter1 - zoomed in view

```
//! Return multigroup absorption opacity
 GPU_HOST_DEVICE
 inline double get_op_a(uint32_t g) const { return abs_groups[g]; }
 //get_op_a maps to Gather0
```

Listing 3: Branson Gather0 in Cell.h

perform throughput experiments to analyze how improvements across generation of NVidia hardware impact the performance of our collected indirection patterns in Section 3.4.2. Finally, in Section 3.4.3 we combine STREAM and BabelSTREAM benchmark performance across all 10 CPU and GPU platforms with the performance collected from Spatter to evaluate how effectively each platform is utilizing it's available bandwidth on specific patterns.

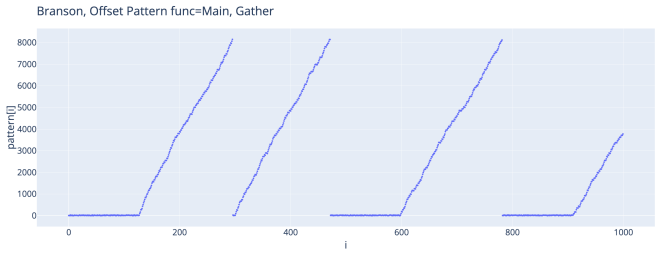
3.4.1 Weak Scaling Experiments. In the first experiments, scaling studies were performed on each of the 99 patterns collected from the 5 applications across the 6 CPU platforms. These patterns were split relatively evenly between gathers and scatters, and the scatters are indicated by the dashed lines in the following figures. Only a subset of these 99 patterns are presented due to space constraints.

The first experiment demonstrates the ability for this workflow to evaluate how the performance of a specific pattern varies across all 6 CPU node configurations. For this, we utilize a gather pattern collected from the XRage application which is performing a simulation of an asteroid collision.

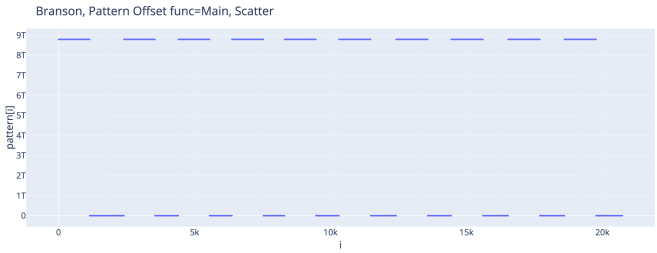
The results of this experiment in Figure 11 illustrate that the 3 systems which use DDR-4 and DDR-5 all perform similarly, while the single socket Grace system outperforms Sapphire Rapids up to 32 cores before plateauing. However, the AMD EPYC 9654P results indicate interesting behavior at low-rank counts, where the node achieves higher performance compared to the Sapphire Rapids DDR node which also uses DDR5. However, the performance of the AMD EPYC 9654P node quickly drops back to levels consistent with the Sapphire Rapids DDR node which also contains DDR5 as its main memory system. Furthermore, the Grace-Grace Superchip achieves the highest bandwidth up to 72 cores before dropping below Sapphire Rapids with HBM when that node is saturated using all 112 cores, despite the Grace-Grace node having a lower STREAM bandwidth due to using LPDDR5X than the Sapphire Rapids HBM node. These results provide important feedback which allows hardware researchers to identify subtle differences in hardware design and their impact on real application access patterns, such as the impact that the number of memory channels, the size and policies of the caches, or the design of the memory controller may impact memory access performance. It also allows facilities to evaluate the same pattern across multiple systems to understand the trade-offs and performance of hardware on indirect access patterns which resemble their applications.

The next weak-scaling experiment provided data for all of the patterns collected from a single application on a single CPU node. For this, we used the 9 patterns collected from the xRAGE application and performed weak-scaling tests for each on the Sapphire Rapids HBM node. The patterns labeled Spatter[1-5] are all gather patterns, while the patterns labeled Spatter[6-9] are scatter patterns.

Using all 112 physical cores across two sockets, these patterns achieve between 341 GiB/s - 470 GiB/s, which is about a quarter of the 1,638.4 GB/s memory bandwidth available on the Sapphire Rapids HBM node. A consistent trend illustrated by Figure 12 is that the scatter patterns from this particular application consistently achieve lower bandwidths than the gather patterns at low core counts. However, many of the gather patterns drop below the scatter patterns as the node becomes saturated. The Spatter 4 pattern exhibits particularly interesting behavior as its performance drops noticeably from 64 ranks to 112 ranks, resulting in it achieving a lower total bandwidth at 112 ranks than it did at 64 ranks. While diagnosing why this may be the case is outside the scope of this paper, the results from this experiment provide another example of how using this workflow to collect real application access patterns to benchmark systems can provide results which indicate where researchers should look more closely at their access patterns and hardware systems.



(a) Branson Gather0 Offsets



(b) Branson Scatter1 offsets.

3.4.2 GPU Throughput Experiments. We performed GPU throughput tests on 4 NVidia GPUs, spanning 3 different generations and 3 different memory systems. The first experiment looked at the same gather pattern from xRAGE analyzed in Section 3.4.1. This pattern was first truncated from its original length of 8,388,968 elements to 524,200 elements to minimize the amount of data transferred. The pattern length was then increased until it reached the full 8,388,968 elements. At this point, the count parameter mentioned in Section 2.4.3 was increased from 1 to 128 by powers of 2 in order to increase the number of indirect accesses accordingly. This resulted in the amount of data transferred ranging from 4MB to 8GB in order to conduct the throughput experiment.

The GH200 achieves the highest bandwidth at 1.78 TB/s as illustrated in Figure 13, less than half of the 4.0 TB/s available bandwidth and about 52% of the measured 3.36 TB/s measured STREAM bandwidth. This is comparable, but a slightly lower ratio of achieved STREAM bandwidth on the H100 (78% or 1.38 TB/s out of 1.76 TB/s), A100 (65% or 914.6 GB/s out of 1.398 TB/s), and V100 (64% or 512 GB/s out of 791 GB/s) GPUs, showing that an improvement in available STREAM bandwidth doesn't necessary translate to a proportional gain in indirect memory access bandwidth. This corresponds to an improvement of 3.5x in indirect access bandwidth from the V100 to the GH200 compared to an increase of 4.44x in the specified achievable bandwidth.

Additionally, each GPU's bandwidth plateaus in the region where 128 MB - 512 MB was transferred during the gather kernel, indicating where each GPU becomes saturated enough to provide enough memory level parallelism to the memory subsystem. Finally, the number of accesses performed for the GPU throughput tests equals the number of accesses performed on 112 cores of the Sapphire Rapids CPU between the 2 rightmost points in the figure, which correspond to expanding the pattern by 64x and 128x, respectively. Given the tools developed to support this workflow, researchers can construct their experiments to provide more direct comparison between CPUs and GPUs by tuning the amount of accesses performed and assessing the size of their data structures and kernels in their applications.

The next experiment performed throughput tests for patterns from each of our 5 applications on the GH200 GPU. Similar to the previous experiment, the patterns were first truncated and then expanded by sweeping the count parameter from 1 to 128 in order to increase the amount of memory operations performed by the GPU on each run. The differences in the original pattern lengths results in each pattern achieving different data transferred values while using an identical count of 128.

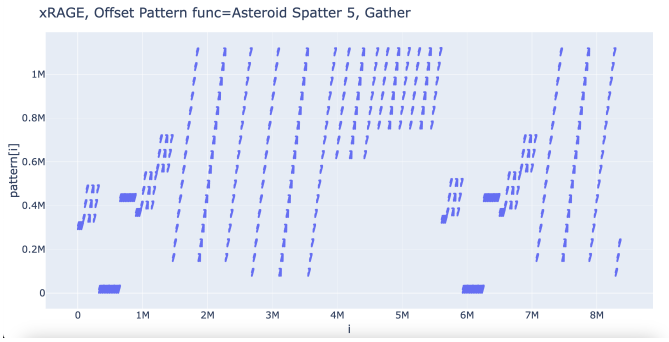
Similar to the CPU results in Figure 12, the data in Figure 14 illustrate that for all 5 applications the scatter patterns perform significantly worse than the gather patterns. This is likely due to scatter patterns not being thread safe and requiring atomic operations to ensure correctness due to aliasing on memory addresses caused by multiple entries in the offset array pointing to the same address in memory. Additionally, all of the patterns indicate a plateauing around 64-128 MB, similar to the results obtained for the single gather pattern in Figure 13. Finally, the variety of access patterns and types of indirect kernel result in a wide range of performance, with a difference in pattern performance of 3x-4x, demonstrating how this workflow can identify indirect access patterns which are better behaved and which ones may be important bottlenecks to focus on.

3.4.3 Performance Relative to STREAM. In order to properly compare changes to memory systems across generations, we want to see not only improvements in the performance reported by Spatter, but also a greater utilization of of the available memory bandwidth. To visualize this, [13] introduced bandwidth-bandwidth plots. These plots, shown in Fig. 15, allow us to see how each pattern performs as a percentage of the STREAM (or BabelSTREAM) Copy bandwidth. These plots use the same data as the weak scaling and GPU throughput tests, but we only plot the highest performance achieved for each pattern on each platform. As before, gather patterns are shown with solid lines and scatter patterns with dashed lines.

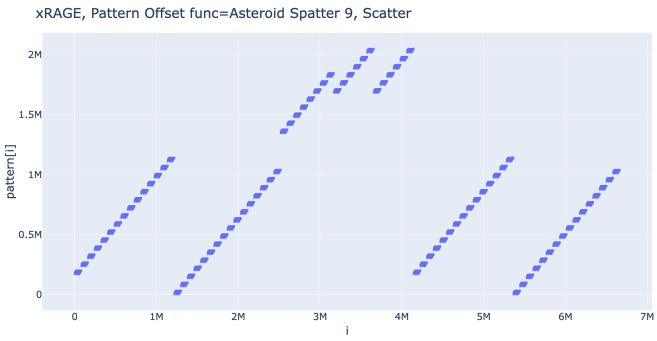
In these log-log plots, *the performance reported by Spatter for a specific pattern is shown as a function of that platform's STREAM bandwidth.* Thus, the following are all true, and are provided to guide understanding of the plots:

(1) The $y{=}x$ line shows the STREAM or BabelSTREAM bandwidth for each platform
(2) All of the performance results for a given platform will form a vertical line
(3) Two points that have the same y-value have the same performance but utilize a different fraction of the available memory bandwidth on those platforms
(4) Two points that are the same distance below the $y{=}x$ line have different performance but utilize the same fraction of available bandwidth on those platforms

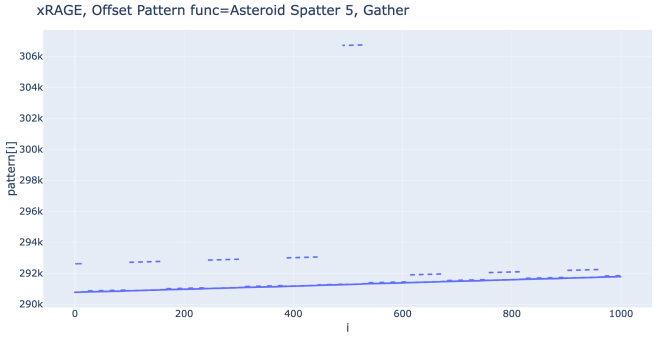
The last two points are particularly important: taken together, they mean that we can use these plots to not only compare how performance is different between platforms, but also how the fraction of available bandwidth differs for different patterns and across platforms.



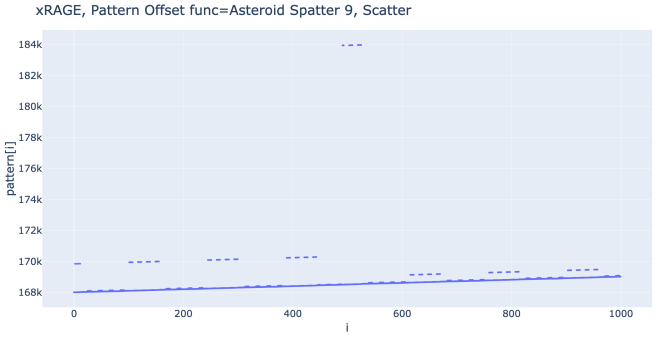
(a) xRAGE Spatter 5 gather visualization.



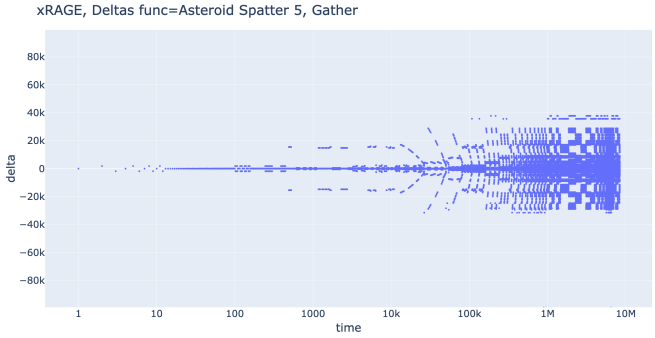
(b) xRAGE Spatter 9 scatter visualization.



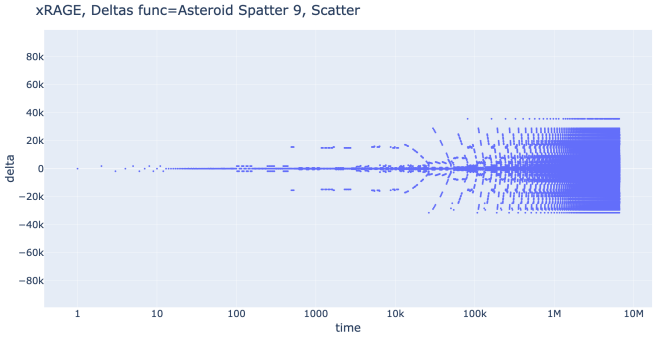
(c) xRAGE Spatter 5 zoomed visualization.



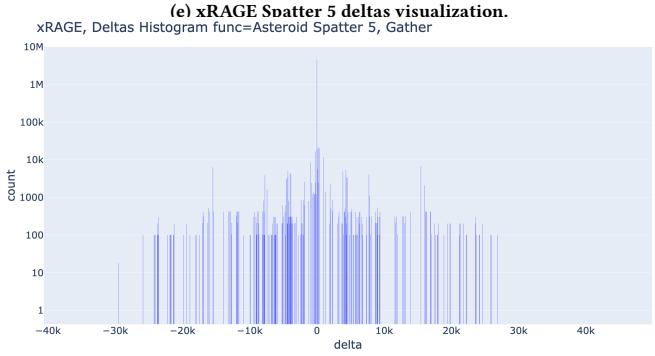
(d) xRAGE Spatter 9 zoomed visualization.



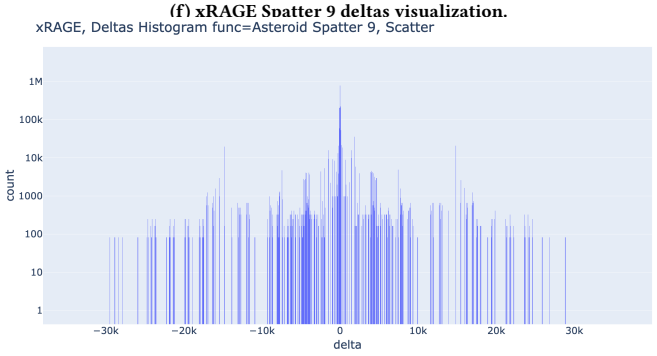
(e) xRAGE Spatter 5 deltas visualization.



(f) xRAGE Spatter 9 deltas visualization.



(g) xRAGE Spatter 5 histogram of deltas visualization.



(h) xRAGE Spatter 9 histogram of deltas visualization.

Figure 10: Visualization of GS Patterns from xRAGE.

The bandwidth-bandwidth plot for the GPU platforms is shown in Fig. 15a. Looking at the diagonal $y=x$ line, we see how the total available memory bandwidth has increased across the past last 4 generations of NVidia GPUs. However, Spatter shows us that the gains in STREAM bandwidth have not always translated to a proportional increase for all memory access patterns. For instance, a number of patterns perform worse on the A100 than they did on the V100, including *AllAbosorb12* and *Marshak dd20*. Additionally, while most patterns increase by an amount greater than STREAM bandwidth between the A100 and the H100, as shown by the lines being steeper than the STREAM line in that region, we see the

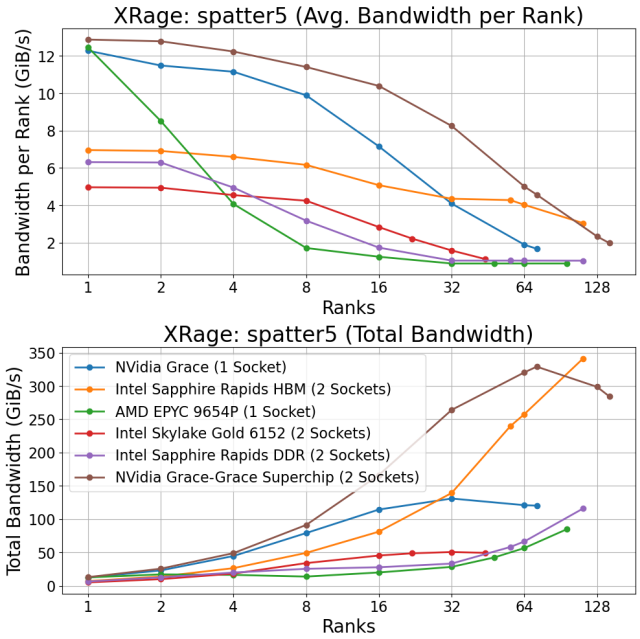


Figure 11: Weak-scaling Spatter results across NVidia, AMD, and Intel Nodes for an xRAGE gather pattern. Top: Average bandwidth measured per rank; Bottom: Total bandwidth measured across the entire node.

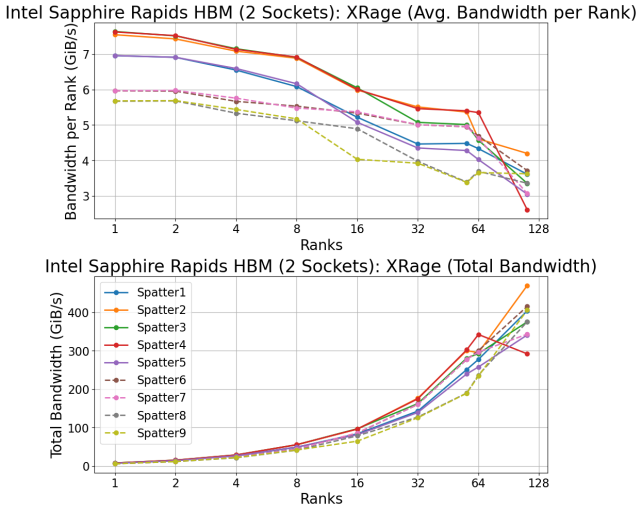


Figure 12: xRAGE patterns Spatter results on Intel SPR HBM Nodes. Top: Average bandwidth measured per rank; Bottom: Total bandwidth measured across the entire node.

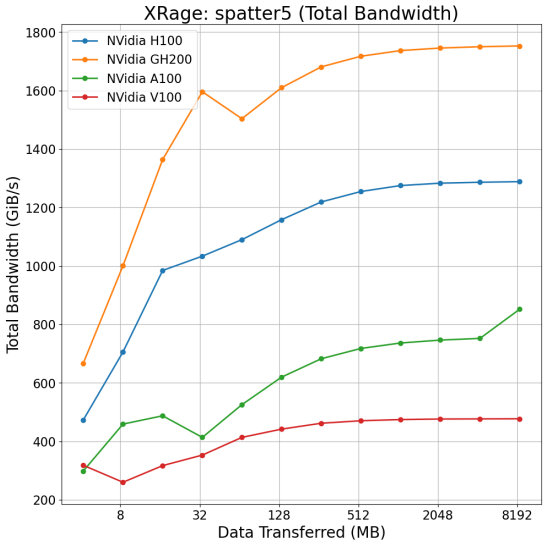


Figure 13: Spatter results across multiple generations of NVidia GPUs for an xRAGE gather pattern.

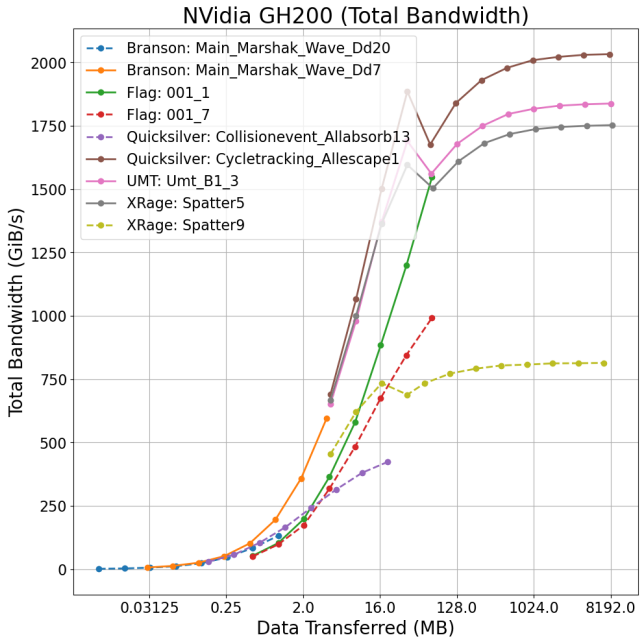


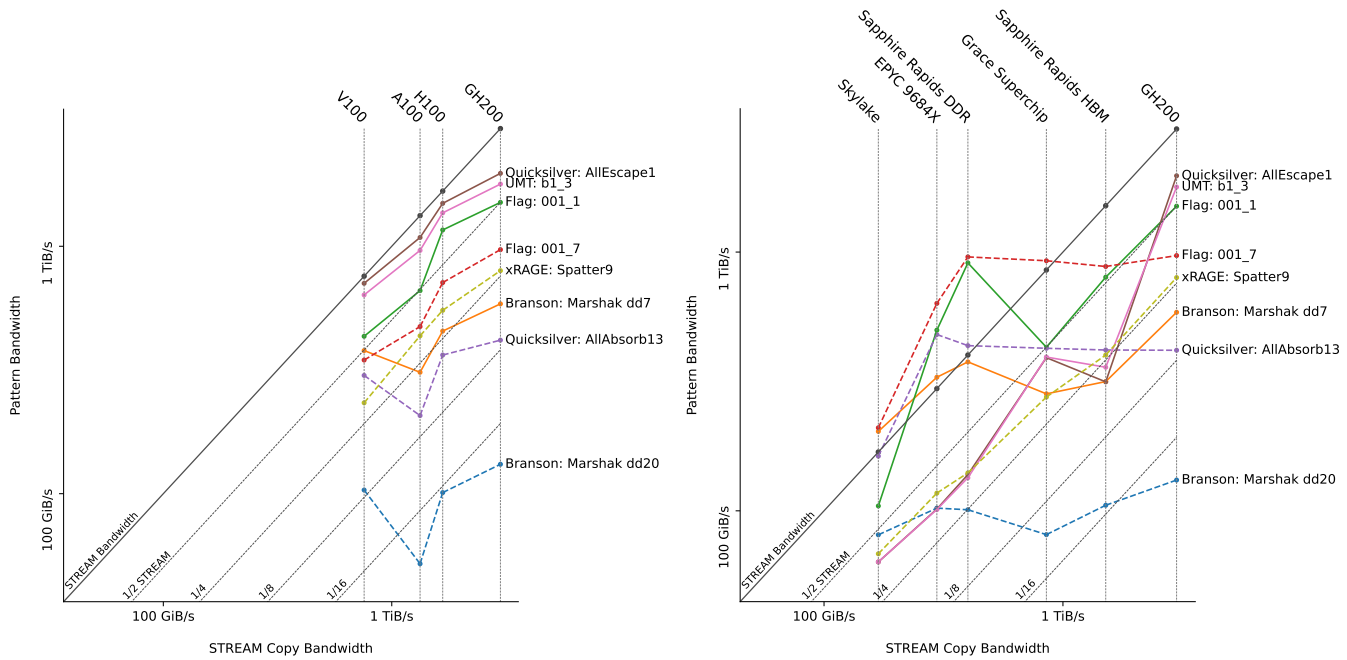
Figure 14: Spatter results of all 5 applications on GH200 GPU.

opposite trend when moving to the GH200. While the GPU trends are easy to analyze, we see a different story when looking at CPUs.

In Fig. 15b, wee have a bandwidth-bandwidth plot for all of our CPU platforms, as well at the GH200 to help us compare to the GPUs. This chart makes even more clear the need for memory system evaluations to include application patterns as close to reality as possible. Consider *Flag: 001_1* for example. This pattern performs close to STREAM on the Intel Skylake and NVidia Grace-Grace platforms. However, when comparing with Intel Sapphire Rapids, it can perform either well above or well below STREAM, depending on whether you have a DDR or an HBM model. The other Flag pattern that is included, *Flag: 001_7*, is similarly odd, performing well above STREAM on some platforms and well below on others.

If we take Fig. 15 as a whole, we see that some platforms are somewhat pessimistic for both CPUs and GPUs, in particular *Marshak dd20*. Increases in STREAM bandwidth have little impact on this scatter pattern.

These bandwidth-bandwidth plots allow us to take a holistic look at how memory systems are evolving over time in relation to the memory access patterns we care about. While GPUs have shown steady improvement in performance for the majority of our patterns, the CPUs in our dataset tell a more complicated story, which must be considered when evaluating system performance.



(a) GPU platform bandwith-bandwidth plot. **(b) CPU platform bandwidth-bandwidth plot.**

Figure 15: Bandwidth-Bandwidth plots. The performance reported by Spatter for each pattern is reported as a function of that platform's STREAM bandwidth.

Those deciding between modern CPUs such as Intel Sapphire Rapids and NVidia Grace must carefully consider the characteristics of their workloads.

4 DISCUSSION

The GS Patterns workflow provides significant enhancements to previously published tools for the analysis and synthesis of sparse memory access benchmarks. While previous tools like Hopscotch [1] and Spatter [13] both provide for the evaluation of sparse patterns with CPUs and GPUs, the missing link has been a path to capture application-specific patterns and use them to synthesize new input microbenchmarks.

However, there are a few caveats with this new approach that may require further research. While new patterns can be quickly pulled from application codes with GS Patterns, comparing patterns from different inputs with the same application still requires a deep knowledge of the application under test as well as inspection of the code that is used to generate sparse microbenchmarks. Visualization approaches like those presented in Section 3.3 provide some initial insight into the differences between regions of interest within an application, but more rigorous statistical analysis is likely required to standardize comparisons across sparse microbenchmarks.

Furthermore, while the process is scripted and uses standard toolsets, the full workflow requires some manual analysis and annotation to optimize the process. Specifically, creating the initial regions of interest or hotspots to limit the search space for GS Patterns requires the user to run a separate tool (vTune, nSight, or possibly perf) and then to annotate their code to limit the runtime of Pin-like tracing tools. We note that the workflow could be run in a more automated fashion without this step, but the runtime of GS Patterns would likely be much longer.

Despite these areas for future enhancement, we feel that the presented approach provides an impressive step forward in the synthesis of new memory microbenchmarks that ties into the co-design of applications and simulation of new and improved memory systems. For example, the generated patterns from GS Patterns could likely be used with little modification as inputs to memory traffic generators for architectural simulation frameworks like gem5 or SST to evaluate future near memory or processing-in-memory designs.

5 CONCLUSION

In this paper we have presented a novel workflow and set of algorithms implemented within GS Patterns which enable the analysis and synthesis of memory access patterns in codes of arbitrary complexity. GS Patterns captures a variety of access patterns during application execution including sparse gather and scatter patterns without relying upon compiler technology recognizing and emitting gather/scatter instructions. To enable rapid performance analysis of these access patterns, we have modified the Spatter microbenchmark to enable replay of GS Patterns access patterns with support for a wide variety of both CPU and GPU hardware architectures.

Performance analysis using Spatter and GS Patterns is presented across a number of latest generation CPU and GPU hardware technologies. Results of our analysis indicate that many applications that exhibit sparse memory access routinely achieve between 25% and 50% of STREAM bandwidth while some patterns achieve less than 6%. In a number of cases, the high STREAM bandwidth of GPU systems delivers high access pattern bandwidth from hardware generation to generation, with some exceptions. As an example, for most access patterns the % of STREAM bandwidth is significantly higher on H100 compared to GH200. In most cases, the tested GPUs

provide significantly higher bandwidth for generated patterns than CPU platforms but not always. In particular the performance of Flag: 001_7 is nearly the same as that achieved on multiple CPU platforms and the performance of Quicksilver: Allabsorb13 is bit lower on GPU relative to CPUs.

This type of detailed analysis, enabled by GS Patterns, illustrates the importance of evaluating a diverse set of memory access patterns with respect to delivered bandwidth. The ability to easily synthesize these access patterns in an automated fashion now enables rapid assessment across a diverse set of applications and hardware technologies.

6 ACKNOWLEDGMENTS

Kevin Sheridan, Galen Shipman, and Jered Dominguez-Trujillo acknowledge support by the National Nuclear Security Administration. Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness. LA-UR-24-24856.

This research used resources provided by the Darwin testbed at Los Alamos National Laboratory (LANL) which is funded by the Computational Systems and Software Environments subprogram of LANL's Advanced Simulation and Computing program (NNSA/DOE).

GT CRNCH Rogues Gallery

This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan.

REFERENCES

[1] Alif Ahmed and Kevin Skadron. 2019. Hopscotch: A micro-benchmark suite for memory performance evaluation. In *Proceedings of the International Symposium on Memory Systems*. 167–172.

[2] Daniel Macgee Alex Long, Kelly Thompson. [n. d.]. Github Site - Branson. https://github.com/lanl/branson. Accessed: 2024-06-24.

[3] Patrick S Brantley, Ryan C Bleile, Shawn A Dawson, NA Gentile, M Scott McKinley, Matthew J O'Brien, Michael M Pozulp, David F Richards, David E Stevens, Jonathan A Walsh, et al. 2017. *LLNL Monte Carlo transport research efforts for advanced computing architectures.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[4] Wendy K. Caldwell, Abigail Hunter, Catherine S. Plesko, and Stephen Wirkus. 2019. Verification and Validation of the FLAG Hydrocode for Impact Cratering Simulations. *Journal of Verification, Validation and Uncertainty Quantification* 3, 3 (2 2019). https://doi.org/10.1115/1.4042516

[5] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.

[6] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 353–364.

[7] Michael Gittings, Robert Weaver, Michael Clover, Thomas Betlach, Nelson Byrne, Robert Coker, Edward Dendy, Robert Hueckstaedt, Kim New, W Rob Oakes, et al. 2008. The RAGE radiation-hydrodynamic code. *Computational Science & Discovery* 1, 1 (2008), 015005.

[8] Louis Howell. 2014. *Characterization of UMT2013 performance on advanced architectures.* Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).

[9] Intel. 2020. GTPin - A Dynamic Binary Instrumentation Framework. https://software.intel.com/content/www/us/en/develop/articles/gtpin.html.

[10] Galen Shipman Jered Dominguez-Trujillo. [n. d.]. Github Site - LANL Spatter Site. https://github.com/lanl/spatter. Accessed: 2023-05-28.

[11] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 3–12. https://doi.org/10.1109/IISWC.2009.5306801

[12] Agustin Vaca Valverde Kevin Sheridan, Christopher Scott. [n. d.]. Github Site - GS patterns. https://github.com/lanl/gs_patterns. Accessed: 2023-05-28.

[13] Patrick Lavin, Jeffrey Young, Richard Vuduc, Jason Riedy, Aaron Vose, and Daniel Ernst. 2020. Evaluating Gather and Scatter Performance on CPUs and GPUs. In *The International Symposium on Memory Systems*. 209–222.

[14] John D McCalpin. 1995. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what* 22, 7 (1995).

[15] Zachary James Medin. 2022. xRAGE: A Brief Overview. (November 2022). https://doi.org/10.2172/1900431

[16] P Nowak. 2013. *Unstructured-Mesh Deterministic Radiation Transport. Single Physics Package Code.* Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).

[17] Douglas Michael Pase and Anthony Michael Agelastos. 2019. *Performance of Gather/Scatter Operations.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[18] Jeffrey Young Patrick Lavin, Agustin Vaca Valverde. [n. d.]. Github Site - Spatter Patterns. https://github.com/hpcgarage/spatter-patterns. Accessed: 2024-06-21.

[19] G. M. Shipman, J. Dominguez-Trujillo, K. Sheridan, and S. Swaminarayan. 2022. Assessing the Memory Wall in Complex Codes. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 30–35. https://doi.org/10.1109/MCHPC56545.2022.00009

[20] Galen M. Shipman, Sriram Swaminarayan, Gary Alan Grider, James Westley Lujan, and Robert Joseph Zerr. 2022. Early Performance Results on 4th Gen Intel(R) Xeon (R) Scalable Processors with DDR and Intel(R) Xeon(R) processors, codenamed Sapphire Rapids with HBM. (11 2022). https://doi.org/10.2172/1898330

[21] Alex Skaletsky, Konstantin Levit-Gurevich, Michael Berezalsky, Yulia Kuznetcova, and Hila Yakov. 2022. Flexible Binary Instrumentation Framework to Profile Code Running on Intel GPUs. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 109–120. https://doi.org/10.1109/ISPASS55109.2022.00011

[22] Various. [n. d.]. DyninstAPI: Tools for binary instrumentation, analysis, and modification. https://github.com/dyninst/dyninst/. Accessed: 2024-08-25.

[23] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307