Evaluating Gather and Scatter Performance on CPUs and GPUs

Patrick Lavin
Georgia Tech
plavin3@gatech.edu

Jeffrey Young
Georgia Tech
jyoung9@gatech.edu

Richard Vuduc
Georgia Tech
richie@cc.gatech.edu

Jason Riedy
Lucata Corporation
jason@acm.org

Aaron Vose
NanoSemi Inc.
Aaron.Vose@nanosemitech.com

Daniel Ernst
Hewlett Packard Enterprise
daniel.ernst@hpe.com

ABSTRACT
This paper describes a new benchmark tool, Spatter, for assessing memory system architectures in the context of a specific category of indexed accesses known as gather and scatter. These types of operations are increasingly used to express sparse and irregular data access patterns, and they have widespread utility in many modern HPC applications including scientific simulations, data mining and analysis computations, and graph processing. However, many traditional benchmarking tools like STREAM, STRIDE, and GPUs focus on characterizing only uniform stride or fully random accesses despite evidence that modern applications use varied sets of more complex access patterns.

Spatter is an open-source benchmark that provides a tunable and configurable framework to benchmark a variety of indexed access patterns, including variations of gather / scatter that are seen in HPC mini-apps evaluated in this work. The design of Spatter includes backends for OpenMP and CUDA, and experiments show how it can be used to evaluate 1) uniform access patterns for CPU and GPU, 2) prefetching regimes for gather / scatter, 3) compiler implementations of vectorization for gather / scatter, and 4) trace-driven "proxy patterns" that reflect the patterns found in multiple applications. The results from Spatter experiments show, for instance, that GPUs typically outperform CPUs for these operations in absolute bandwidth but not fraction of peak bandwidth, and that Spatter can better represent the performance of some cache-dependent mini-apps than traditional STREAM bandwidth measurements.

1 INTRODUCTION
We consider the problem of how to assess the performance of modern memory systems with respect to indexed memory accesses, such as gather and scatter (G/S) operations. Our motivation derives from both applications and hardware. On the application side, there are many instances where memory operations involve loads or stores through a level of indirection (e.g., reg ← base[idx[k1]]). For instance, such indexed memory access is common in scientific data analysis applications that rely on sparse and adaptive data abstractions, including adaptive meshes, sparse matrices and tensors, and graphs, which are our focus. On the hardware side, new CPU architectures have begun to incorporate advanced vector functionality like AVX-512 and the Scalable Vector Extension (SVE) for improving SIMD application performance.

Within this context, our strategy to understanding the interactions between application-relevant G/S operations and modern hardware relies on the development of a microbenchmarking tool. It aims to express critical features of real G/S workloads, derived from applications but abstracted in a way that is easy to adopt by system-oriented stakeholders. These include situations where (1) vendors and hardware architects might wonder how new ISAs (such as AVX-512) and their implementation choices affect memory system performance; (2) application developers may consider how the data structures they choose impact the G/S instructions their code compiles to; and (3) compiler writers might require better data on real-world memory access patterns to decide whether to implement a specific vectorization optimization for sparse accesses. Although these groups could turn to any number of memory-focused microbenchmarks available today [19], we believe a gap still exists in the focused evaluation of system performance for indexed accesses, including G/S workloads.

In light of these needs, we have formulated a new microbenchmarking tool called Spatter. It evaluates indexed access patterns based on G/S operations informed by applications across different language and architecture platforms. More importantly, we believe Spatter can help to answer a variety of system, application, and tool evaluation questions, some of which include: (1) What application G/S patterns exist in the real world, and how do they impact memory system performance? (2) How does prefetching affect the performance of indexed accesses on modern CPU platforms? (3) How does the performance of G/S change when dealing with sparse data on CPUs and GPUs?

The design of the Spatter tool suite aims to address these questions. At a basic level, Spatter provides tunable gather and scatter implementations. These include CUDA and OpenMP backends with knobs for adjusting thread block size and ILP on GPUs.

---

1The source code for Spatter is available at https://github.com/hpcgarage/spatter
and work-per-thread on CPUs. Spatter also includes a scalar, non-ve

torized backend that can serve as a baseline for evaluating the

benefits of vector load instructions over their scalar counterparts.

Lastly, Spatter has built-in support for running parameterized mem-

ory access patterns and custom patterns. We show, for instance,

how one can collect G/S traces from Department of Energy (DOE)

mini-apps to gain insights or make rough predictions about perfor-

mance for hot kernels that depend on indexed accesses (Section 2).

This paper presents the structure of the Spatter benchmark tool,

and then documents experimental results from a number of plat-

forms. Our initial evaluations of Spatter show that newer GPU

architectures perform best in absolute bandwidth for both gather

and scatter operations, in part due to memory coalescing and faster

memories. AMD Naples performs best of all the CPU-based plat-

forms (Broadwell, Skylake, TX2) for strided accesses. A study of

prefetching with Spatter further shows how G/S benefits from

modern prefetching across Broadwell and Skylake CPUs. Spatter’s

scalar backend is also used to demonstrate how compiler vector-

ization can improve G/S with large improvements for both Sky-

lake and Knight’s Landing (Section 5.3). Experiments for three

DOE mini-apps show G/S performance improvements enabled by

caching on CPU systems and by fast HBM memory on GPUs. These

parameterized access pattern studies also suggest that STREAM

bandwidth does not correlate well with specific mini-apps that are

cache-dependent, which further motivates benchmarks like Spatter

that do better.

2 GATHER / SCATTER IN REAL-WORLD

APPLICATIONS

To motivate our interest in G/S performance, we studied several

prominent DOE mini-apps from the CORAL and CORAL-2 procure-

ments [1, 2]. Such software provides a rich source of information

about the computational and memory behavior requirements of
critical scientific workloads in both government as well as academic

environments. Many of these workloads contain important kernels

which stress G/S performance. Indeed, one aim of Spatter is to

leverage such mini-apps as a source of real-world G/S patterns.

Table 5 in the Appendix provides detailed information on the

specific patterns extracted from these applications, so we focus on

the high-level characteristics of each application in this section.

We note that many of these patterns are complex in that the index

defining the G/S does not fit into our categories (broadcast, stride-N,

mostly stride-1) as is discussed in Section 3 and Section 5.

In particular, this work considers mini-apps from CORAL and

CORAL-2, including AMG [28], LULESH [15], and Nekbone [10].

We built these mini-apps targeting ARMv8-A with support for

Arm’s Scalable Vector Extension (SVE) [24] at a vector length of

1024 bits. The resulting executables were run through an instru-

mented version of the QEMU functional simulator [4] to extract

traces of all instructions accessing memory along with their as-

sociated virtual addresses. From this instruction stream, collected

from rank 0, we examine only G/S instructions, and extract the base

address and offset vector for each, along with their frequencies.

The problem sizes are chosen so as to prioritize a realistic working

set with 64 MPI ranks per node with one thread per rank, while

the number of iterations is less emphasized. For these apps, it is

expected that multiple kernel iterations will have many patterns in

common. More information on how we configured these codes is

found in Table 2.

Table 1 shows the G/S characteristics extracted from several

kernels selected from the aforementioned mini-apps, along with the

percentage of data motion performed by G/S operations. The

reported G/S data motion percentages are conservative, as current

data records all scalar loads and stores them as being 64 bits, while

a significant fraction of 32-bit scalar data types is expected.

Examination of the G/S behavior results in the observation of

a small number of common pattern classes: uniform-stride, where
each element of a gather is a fixed distance from the preceding

element; broadcast, where some elements of a gather share the

same index; mostly stride-1, in which some elements of a gather

are a single element away from the preceding element; and more

complex strides, in which elements of a gather have a complicated

pattern containing many different strides.

We can make a few high-level remarks about Table 1. First, gath-
ers are more common than scatters. Secondly, G/S can account for

high fractions of total load / store traffic (last column; up to 67.6%,
or just over two-thirds, in these examples). Thirdly, the appearance

of differing categories of stride types suggests that there are multi-
ple opportunities for runtime (inspector / executor) and hardware

memory systems to optimize for a variety of G/S use-cases, which

Spatter can then help evaluate.

2.1 Open Source Techniques for Pattern

Analysis

The application analysis done in this work depends on a custom,
closed-source QEMU functional simulator that uses an SVE vector

size of 1024 and data from the first rank of an MPI run, which may

slightly differ from other ranks, along with post-processing scripts

to extract the most utilized G/S patterns. We recognize this is a

limitation of the current work in that trace capture and analysis

can be one of the most time-consuming portions of an analysis of

G/S patterns.

A useful open-ended project that would improve pattern inputs

for not just Spatter but many other application analysis frameworks

would be a tool based on either DynamoRIO (which supports AVX-

512 and SVE instructions) or Intel Architecture Code Analyzer

(IACA) that generates this type of data in a less time-consuming

fashion and that performs correlation and clustering across all ranks

of an application for CPU and GPU codes.

3 DESIGN OF THE SPATTER BENCHMARK

We have developed Spatter because existing benchmarks like STREAM [19]

and STRIDE [22] focus on uniform stride accesses and are not config-
urable enough to handle non-uniform, indirect accesses or irregular
patterns. For more information on related benchmarks, see Sec-

tion 6. Figure 1 shows a conceptual view of the Spatter benchmark.
The design of the benchmark is described further below.

3.1 Kernel Algorithm

Spatter represents a memory access pattern as a short index buffer,

and a delta. At each base address address of x + i, a gather or

scatter will be performed with the indices in the index buffer. The

Patrick Lavin, Jeffrey Young, Richard Vuduc, Jason Riedy, Aaron Vose, and Daniel Ernst
### Table 1: High-Level Characterization of Application G/S Patterns.

<table>
<thead>
<tr>
<th>Application</th>
<th>(Extracted Patterns)</th>
<th>Selected Kernels</th>
<th>Gathers</th>
<th>Scatters</th>
<th>G/S MB (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td>mostly stride-1</td>
<td>hypre_CSRMatrixMatvecOutOfPlace</td>
<td>1,696,875</td>
<td>0</td>
<td>217 (17.8)</td>
</tr>
<tr>
<td>LULESH</td>
<td>(uniform-stride)</td>
<td>IntegrateStressForElements</td>
<td>828,168</td>
<td>382,656</td>
<td>155 (22.4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InitStressTermsForElements</td>
<td>1,121,844</td>
<td>1,153,827</td>
<td>291 (67.6)</td>
</tr>
<tr>
<td>Nekbone</td>
<td>(uniform-stride)</td>
<td>ax_e</td>
<td>2,948,940</td>
<td>0</td>
<td>377 (33.3)</td>
</tr>
<tr>
<td>PENNANT</td>
<td>(fixed-stride, broadcast)</td>
<td>Hydro::doCycle</td>
<td>728,814</td>
<td>0</td>
<td>93 (13.9)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mesh::calcSurfVecs</td>
<td>324,064</td>
<td>0</td>
<td>41 (39.5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QCS::setForce</td>
<td>891,066</td>
<td>0</td>
<td>114 (45.5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>QCS::setQCnForce</td>
<td>1,214,318</td>
<td>323,800</td>
<td>197 (64.5)</td>
</tr>
</tbody>
</table>

### Table 2: Details for Selected Applications and Kernels Used for G/S Pattern Extraction.

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>Problem Size / Changes</th>
<th>Kernel Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td></td>
<td>Arguments -problem 1 -n 36 36 36 -P 4 4 4, also mg_maxIter in ang.c set to 5 to limit iterations.</td>
<td>Entirety of each of the functions listed in Table 1.</td>
</tr>
<tr>
<td>LULESH</td>
<td>2.0.3</td>
<td>Arguments -i 2 -s 40, also modifications to vectorize the outer loop of the first loop-nest in IntegrateStressForElements.</td>
<td>The first loop-nest in IntegrateStressForElements. Arrays [xyz]_local[8] as well as B[3][8] give stride-8 and stride-24. Also, the entirety of the InitStressTermsForElements function.</td>
</tr>
<tr>
<td>Nekbone</td>
<td>2.3.5</td>
<td>Set ldim = 3, ifbrick = true, iel0 = 32, ielN = 32, nx0 = 16, nxN = 16, stride = 1, internal np and nelt distribution. Also, niter in driver.f set to 30 to limit CG iterations.</td>
<td>First loop in ax (essentially a wrapped call to ax_e) contains the observed stride-6.</td>
</tr>
<tr>
<td>PENNANT</td>
<td>0.9</td>
<td>Config file sedovflat.pnt with meshpars 2160 1 1 1.25 and cstop 5.</td>
<td>Entirety of each of the functions listed in Table 1.</td>
</tr>
</tbody>
</table>

**Figure 1:** An overview of the Spatter benchmark with inputs and outputs.

**Algorithm 1** Gather Kernel

```plaintext
for i in 1..N do
    src = src + delta * i
for j in 1..vector_length do
    dst[j] = src[idx[j]]
```

The basic gather algorithm. Scatter is performed analogously. False sharing is prevented by giving each thread its own dst buffer for gather, and src buffer for scatter.

### 3.2 Backend Implementations

Spatter contains Gather and Scatter kernels for three backends: Scalar, OpenMP, and CUDA. A high-level view of the gather kernel is in Figure 2, but the different programming models require that the implementation details differ significantly between backends. Spatter provides performance tuning knobs for both the OpenMP and CUDA backbones, such as index buffer length and block size.

**OpenMP:** The OpenMP backend is designed to make it easy for compilers to generate G/S instructions. Each thread will perform some portion of the iterations shown in Figure 2. To ensure high performance when gathering, each thread will gather into a local destination buffer (vice-versa for scattering). This avoids the effects of false sharing.

**CUDA:** Whereas in the OpenMP backend, each thread will be assigned its own set of iterations to perform, in the CUDA programming model, an entire thread block must work together to perform an iteration of the G/S operation (shown in Figure 2) to ensure high performance. These backends are similar, in that each thread block gathers into thread local memory to allow for high performance.
The major difference is that each thread block must read the index buffer into shared memory to achieve high performance on Spatter’s indexed accesses.

**Scalar:** The Scalar backend is based on the OpenMP backend, and is intended to be used as a baseline to study the benefits of using CPU vector instructions as opposed to scalar loads and stores. The major difference between this and the OpenMP backends is that the Scalar backend includes a compiler pragma to prevent vectorization, namely `#pragma nowvec`.

### 3.3 Benchmark Input

A Spatter user can evaluate a variety of memory patterns. Spatter accepts either a single index buffer and run configuration as input, or a JSON file containing many such patterns and configurations.

**Pattern Specification:** Spatter currently provides robust mechanisms for representing spatial locality with both standard patterns and more complex, custom patterns for representing indirect accesses. In Spatter, a memory access pattern is described by specifying (1) either gather or scatter (2), a short index buffer, (3) a delta, and (4) the number of gathers or scatters to perform. Spatter will determine the amount of memory required from these inputs. Spatter includes three built-in, parameterized patterns, which are Uniform Stride, Mostly Stride-1, and Laplacian. These all describe small index sets, which should be thought of as the offsets for a single G/S. When combined with a delta, these will describe a memory access pattern. They are described in further detail below.

#### 3.3.1 Uniform Stride

The Uniform Stride index buffer is specified to Spatter with `UNIFORM:N:STRIDE`. It generates an index buffer of size N with stride STRIDE. For example, the index buffer generated by `UNIFORM:8:4` is \([0, 4, 8, 12, 16, 20, 24, 28]\).

#### 3.3.2 Mostly Stride-1

The Mostly Stride-1 index is the result of accessing a few elements sequentially, and then making some jump and accessing a few more elements sequentially. In code, this could be the result of accessing the same few members of an array, or from accessing a sub-block of a matrix. In Spatter, you can specify an MS1 pattern with `MS1:N:BREAKS:GAPS`. The pattern will be length N, with gaps at BREAKS, with gaps of size GAPS. For example, the index buffer generated by `MS1:8:4:20` is \([0, 1, 2, 3, 23, 24, 25, 26]\).

#### 3.3.3 Laplacian

The Laplacian index is based on Laplacian stencils from PDE solvers. Spatter can generate 1-D, 2-D, or 3-D stencils with the pattern `LAPLACIAN:D:L:SIZE`. This creates a D-dimensional stencil, with each “branch” of length L, for a problem size of SIZE. Even though a 2- or 3-D problem can be specified, Spatter still only allocates a 1-D data array. Thus the problem size must be specified in the stencil so that Spatter can calculate the distances of the neighbors in the stencil. For example, the input `LAPLACIAN:2:2:100` generates the classic 5-point stencil \([0, 100, 198, 199, 200, 201, 202, 300, 400]\), which may be more familiar to users in the non-zero-based form, \([-200, -100, -2, -2, 0, 1, 2, 100, 200]\).

#### 3.3.4 Custom Patterns (Complex Accesses)

Finally, if users want to represent a more complex pattern not specified above, they can specify a pattern index buffer as `./spatter -p idx0, idx1, ..., idxN`. This allows users to develop more complex and irregular kernels that often show up in HPC applications. The use of custom patterns is the basis of ongoing research described in Section 7.

**Delta Specification:** To form a full memory access pattern, Spatter needs an index buffer, such as the ones described above, and a delta. The index buffer will be used as the offsets for a gather or scatter with base addresses 0, delta, 2*delta etc.

**JSON Specification:** When running tests, it is common to run many different patterns. To support this, Spatter accepts a JSON file as input that can contain as many configurations as the user wishes. Spatter will parse this file and allocate memory once for all tests, greatly speeding up test sets with many different patterns, and easing data management.

### 3.4 Example

A user specifies a Spatter run with an index buffer, a delta, and the number of gathers or scatters. The simplest example would be to emulate a STREAM-like pattern, which would look like a uniform stride-1 pattern with delta equal to the index length, so that there is no data reuse between gathers. In Spatter, this is expressed as:

```
./spatter -k Gather -p UNIFORM:8:1 -d 8 -l $(((2^x+24)))
```

to run \(2^{24} (-1)\) gathers (\(-k\)), each one 8 doubles beyond the last (\(-d\)), and each using an index buffer of length 8 and uniform stride-1 (\(-p\)). This will produce a STREAM Copy-like number, but it will only be read bandwidth, as a gather reads data from memory to a register. Spatter includes further options for choosing backends and devices and performance tuning that are described in its README.

### 3.5 Benchmark Output

For each pattern specified, Spatter will report the minimum time taken over 10 runs to perform the given number of gathers or scatters. It will also count the bandwidth used by the index buffer, as it is assumed to be small and resident in cache. This measure may not be a true bandwidth in the traditional sense of the word, as many patterns specified to Spatter may allow for cache reuse. Thus, one should think about the bandwidths reported as the rate at which the processor is able to consume data for each pattern.

Optionally, PAPI [27] can be used to measure performance counters. However, we do not demonstrate that feature in this paper.
Evaluating Gather and Scatter Performance on CPUs and GPUs

For JSON inputs, Spatter will also report stats about all of the runs, such as the maximum and minimum bandwidths observed across configurations, as well as the harmonic mean of the bandwidths.

4 EXPERIMENTAL SETUP

Table 3 describes the different configurations and backends tested for our initial evaluation using the Spatter benchmark suite. We pick a diverse set of systems based on what is currently available in our lab and collaborator’s research labs, including a Knight’s Landing system, and a prototype system with ARMv8 ThunderX2 chips designed by Marvell (formerly Cavium). We also include a server-grade and desktop-grade Intel CPU system and several generations of NVIDIA GPUs. Recent AMD GPUs, CPUs, or APUs were not available to us for testing at the time of this writing and are instead left for future work experiments.

OpenMP: To control for NUMA effects, CPU systems are tested using all the cores on one socket or one NUMA region if the system has more than one CPU socket. Some systems like the KNL on Cori have an unusual configuration where the entire chip is listed as 1 NUMA region with 272 threads. For all the OpenMP tests, Spatter is bound to one socket and run using one thread per core on that socket. The following settings are used for OpenMP tests:

1. OMP_NUM_THREADS = <num_threads_single_socket>
2. OMP_PROC_BIND = master
3. OMP_PLACES = sockets
4. KMP_AFFINITY = compact (only for KNL)

An important performance tuning factor is the index buffer length. On CPUs, we find that it is best to use an index buffer that closely matches the hardware vector length, or a small multiple. On the CPUs we have tested, we achieve good performance by using an index buffer length of 16, which is 2-4x the length of the vector registers on our systems.

CUDA: When testing on GPUs, the block size for Spatter is set at 1024 and an index buffer of length 256 is used. These settings allow Spatter to reach bandwidths within 20% of the vendor reported theoretical peak for both gather and scatter kernels. These bandwidths are slightly different than what is typically reported, as gather is designed to only perform reads, and scatter should only perform writes.

Experimental Configurations: Runs of Spatter use the maximum bandwidth out of 10 runs for the platform comparison uniform stride and application pattern tests. STREAM results used for comparisons with Spatter are generated using 256 elements with either STREAM for CPU or BabelStream for GPU, while all Spatter uniform stride tests read or write at least 8GB of data on the GPU and 16GB on the CPU. The difference between CPU and GPU data sizes results from most GPUs having less than 16 GB of on-board memory. The application-specific pattern tests read or write at least 2GB.

5 CASE STUDIES

Spatter is designed to be a flexible tool that can allow the user to run many different memory access patterns and expose many knobs used for tuning. In this section, we use Spatter to investigate several questions regarding CPU and GPU memory architecture including: A) uniform stride access on CPUs, B) uniform stride access on GPUs, C) the effectiveness of G/S over scalar load/store, and D) the performance of trace-derived G/S patterns on CPU.

5.1 CPU Uniform Stride

We increase the stride of memory access and show how performance drops as the stride increase from 1 to 128 doubles on Skylake, Broadwell, Naples, and Thunder X2 systems. Cascade Lake is omitted as it overlaps closely with Skylake. A log-scale is used for the y-axis to make differences between the platforms apparent. Takeaway: Uniform stride patterns show us that peak bandwidth is not necessarily an indication of which architecture will perform best at even moderate strides.

We begin with a basic test: running Spatter with the uniform stride pattern, and increasing the stride by 2x until performance flattens. A stride of 1 is analogous to the STREAM benchmark, except that Spatter will only generate read instructions (gathers) for the gather kernel and write instructions for the scatter kernel, meaning the bandwidths should be slightly different. Figure 3 shows...
Table 3: Experimental Parameters and Systems (OMP Denotes OpenMP, and OCL Denotes OpenCL).

<table>
<thead>
<tr>
<th>System description</th>
<th>Abbreviation</th>
<th>System Type</th>
<th>STREAM (MB/s)</th>
<th>Threads, Backends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knight’s Landing (cache mode)</td>
<td>KNL</td>
<td>Intel Xeon Phi</td>
<td>249,313</td>
<td>272 threads, OMP</td>
</tr>
<tr>
<td>Broadwell</td>
<td>BDW</td>
<td>32-core Intel CPU (E5-2695 v4)</td>
<td>43,885</td>
<td>16 threads, OMP</td>
</tr>
<tr>
<td>Skylake</td>
<td>SKX</td>
<td>32-core Intel CPU (Platinum 8160)</td>
<td>97,163</td>
<td>16 threads, OMP</td>
</tr>
<tr>
<td>Cascade Lake</td>
<td>CLX</td>
<td>24-core Intel CPU (Platinum 8260L)</td>
<td>66,661</td>
<td>12 threads, OMP</td>
</tr>
<tr>
<td>ThunderX2</td>
<td>TX2</td>
<td>28-core ARM CPU</td>
<td>120,000</td>
<td>112 threads, OMP</td>
</tr>
<tr>
<td>Kepler K40c</td>
<td>K40c</td>
<td>NVIDIA GPU</td>
<td>193,855</td>
<td>CUDA</td>
</tr>
<tr>
<td>Titan XP</td>
<td>Titan XP</td>
<td>NVIDIA GPU</td>
<td>443,533</td>
<td>CUDA</td>
</tr>
<tr>
<td>Pascal P100</td>
<td>P100</td>
<td>NVIDIA GPU</td>
<td>541,835</td>
<td>CUDA</td>
</tr>
<tr>
<td>Volta V100</td>
<td>V100</td>
<td>NVIDIA GPU</td>
<td>868,000</td>
<td>CUDA</td>
</tr>
</tbody>
</table>

the results of our uniform stride tests on CPUs. We would expect that as stride increases by a factor of 2, bandwidth should drop by half; the entire cache line is read in but only every other element is accessed. This should continue until about stride 8, as we are then using one double from every cache line. This is what we see on Naples, but performance continues to drop on TX2, Skylake, and Broadwell. Interestingly, Broadwell performance increases at stride-64, even out-performing Skylake. We can further use Spatter to investigate these two points: 1) why does Broadwell outperform Skylake at high strides, and 2) why does TX2 performance drop so dramatically past 1/16?

5.1.1 Disabling Prefetching. To get an idea of what is causing Broadwell to outperform Skylake, we turn prefetching off with Model Specific Registers (MSRs) and re-run the same uniform stride patterns. Figure 4a and b shows the results from this test. For Broadwell, performance does not show the same increase for stride-64 with prefetching off and it instead bottoms out after stride-8. We conclude that one of Broadwell’s prefetchers pulls in two cache lines at a time for small strides but switches to fetching only a single cache line at stride-64 (512 bytes). We can understand the performance discrepancy between Broadwell and Skylake by looking at Figure 4b. Performance drops to 1/16th of peak, as Skylake always brings in two cache lines, no matter the stride. We did not get the opportunity to run on the Thunder X2 without prefetching since it does not have a similar MSR equivalent, but we suspect similar effects are at play: one of the prefetchers likely always brings in the next line, although that only helps to explain performance dropping through stride-16, not through stride-64.

Lesson: By running uniform stride tests on CPUs we are able to (1) identify a number of performance crossover points for intermediate strides and (2) see some interesting differences between the Broadwell and Skylake prefetching behavior.

5.2 GPU Uniform Stride

As the memory architecture of CPUs and GPUs is quite different, it is worthwhile to see how GPUs handle these uniform stride patterns. Figure 5a shows how a K40c, a Titan Xp, and a P100 perform on the same tests.3 As with the CPUs, we see bandwidth drop by half for stride-2 and by another half for stride-4. However, for the P100 and

3To get high performance on GPUs, the threads within a block all work together to read a pattern buffer into shared memory. This buffer must be much longer than the CPU index buffer (256 indices vs 8) so that each thread has enough work to do.
Evaluating Gather and Scatter Performance on CPUs and GPUs

Figure 5: GPU Gather (a) and Scatter (b) Uniform Stride Bandwidth comparison. 
Takeaway: We are able to use uniform stride patterns to show improvements to the memory architecture of GPUs over time, beyond simply improved bandwidth. We see here that in newer generations, not only do GPUs have more bandwidth, they are also able to utilize a higher percentage of that bandwidth throughout intermediate strides.

Figure 6: Improvement of SIMD Gather Kernel (a) and Scatter Kernel (b) Compared to Serial Scalar Backend. 
Takeaway: By examining the performance of uniform stride patterns with and without vectorization enabled in the compiler, we show that achieving maximum bandwidth on processors such as Knights Landing and Skylake requires vectorization. On the other hand, these instructions can be detrimental to performance on Broadwell.

5.3 SIMD vs. Scalar Backend Characterization
Spatter can also be used to test the effectiveness of different hardware implementations of single instruction, multiple data (SIMD) instruction set architectures (ISAs). In a real-world sense, this capability can be used by compiler writers to answer questions such as whether it would be beneficial to load some addresses with vector instructions and others with scalar instructions to maximize utilized memory bandwidth. Vector versions of indexed load and store instructions help compilers to vectorize loops and can also help avoid unnecessary data motion between scalar and vector registers that might otherwise be required. We can use Spatter to investigate whether these vector instructions are indeed superior to scalar load instructions and whether compiler writers should prioritize vectorized G/S optimizations.
To demonstrate the effectiveness of SIMD load / store instructions, we run Spatter using the gather kernel on multiple platforms with the scalar backend as a baseline. This scalar baseline is then compared to the OpenMP backend as vectorized by the Cray compiler (CCE 9.0) with the resulting percent improvement from vectorization reported in Figure 6 for strides 1-128, as before. The Broadwell CPU performs the worst of all the tested CPUs, showing worse performance with vectorized code in many cases for both gather and scatter. Thus, for a memory heavy kernel, it would likely be better to use scalar instructions than G/S instructions for this architecture. At the same time, this difference may be mitigated somewhat as G/S instructions remove the need to move data between regular and vector registers.

On the other hand, Skylake, Knights Landing, and Naples have better gather performance in the vectorized case. The use of gather instructions on these platforms is clearly justified. Of these three, however, Naples is the only one to not improve in the scatter case as well. This is due to the lack of scatter instructions on Naples. TX2 has no G/S support at all, so it stays close to 0% difference (save for a single outlier in the gather chart). Interestingly, for our three processors with useful G/S instructions, they all gather best in different regions, with Knights Landing best at small strides, Naples for medium strides, and Skylake best at large strides. While we are not able to explain the reason for this performance artifact, we have demonstrated the benefit of G/S instructions over their scalar counterparts. At least for Knights Landing, anecdotal evidence has suggested that using vectorized instructions at lower strides reduces overall unique instruction count and overall request pressure on the memory system.

Lesson: Spatter shows that the G/S instructions themselves lead to higher performance, especially on Knights Landing. G/S instructions have a further benefit over scalar in that the data loaded is already in a vector register, whereas after performing scalar loads, further rearrangement would be needed to move the data into vector registers.

5.4 Application-derived G/S Patterns

While the three previous sections have focused on uniform stride patterns, Spatter is also able to run more complex patterns. To demonstrate Spatter’s ability to emulate patterns found in real applications, we take the top patterns from several DOE mini-apps (as described in Section 2) and run them in Spatter. The patterns that come out of Section 2 are described by a buffer of offsets and a delta. These offsets and deltas can be found in Table 5 in Section A.

In Section 5.4.1, we first look at how these patterns perform in aggregate, and see if they correlate with STREAM bandwidth. In Section 5.4.2, we look at each pattern individually, and look for trends among the applications. Finally, in Section 5.4.3, we show a method for plotting results that allows us to examine absolute and relative performance of patterns at the same time.

5.4.1 Application Pattern / STREAM Correlation. Another question is to what extent application-specific patterns are more informative than STREAM, considering CPUs and GPUs separately. Table 4 shows the harmonic mean of the performance of the patterns. To see if the performance correlates with STREAM, we calculate Pearson’s correlation coefficient, R, as follows:

\[ R = \text{cov}(X, \text{STREAM}) / (\text{std}(X) \times \text{std}(\text{STREAM})) \]  

According to Table 4, in aggregate, LULESH shows poor performance on most CPU platforms. The next section shows that this result is due to the LULESH-S3 pattern, which is a scatter with delta 0. We believe this configuration triggers cache invalidations for multicore backdrops.

We also see that AMG and Nekbone show higher performance than STREAM in general. Inspecting their patterns, the deltas tend to be small, which implies that gathered addresses overlap. Thus, caching effects may explain this observation.

More interestingly, we see that the CPU runs of the Nekbone and PENNANT patterns show poor correlation (close to 0) with STREAM. In the case of AMG, the patterns perform much better than STREAM, whereas in PENNANT, the patterns perform much worse. This difference suggests that Spatter indeed captures distinct behaviors from STREAM, and that the patterns Spatter generates are not well approximated by STREAM on CPUs. For GPU systems, however, the R coefficient shows that STREAM is much better correlated (close to 1) with the Spatter results. This observation may reflect the smaller and simpler memory hierarchy of GPUs compared to CPUs.

5.4.2 Comprehensive Evaluation Across Platforms and Applications. The design of Spatter makes it easy to collect lots of data, over many platforms and patterns, and these results can reveal more than single-number benchmarks like STREAM as discussed in Section 5.4.1. A natural question is whether that data facilitates any qualitative comparisons about broad classes of platforms or applications. For example, what can we say generally about CPU-oriented memory system design today versus GPU-oriented design? Are applications uniform in their patterns, or are they more varied?

To get a handle on such questions, we take the per-platform and per-pattern data, and render them using small-multiple radar plots as shown in Figure 7 and Figure 8. A single radar in this plot shows the performance of a pattern relative to its stride-1 performance across all CPUs (blue) and GPUs (green). The inner circle represents 100% of stride-1 bandwidth, meaning that any value larger than this must be utilizing caching. This detailed look at the performance gives us a number of insights:

(1) Consider LULESH-S3 in Figure 8. It indeed has very low performance, except on the TX2, which appears to handle the scenario of writing to the same location over and over very well. This behavior could be due to an architectural optimization that recognizes data is being overwritten before it is ever used.

(2) Overall, we see that the GPUs are largely unable to outperform their stride-1 bandwidth. However, this behavior may be changing in newer generations. The V100 values peak above the 100% circle for many of the patterns.

(3) The Naples system (bottom-right of the CPU radars) largely under-performs, save for one set of patterns, Nekbone. Curiously, there is not much that differentiates the Nekbone patterns from LULESH, as both have uniform stride patterns with small deltas. Thus, these patterns may require more thorough profiling and investigation.

(4) Restricting ourselves to the Intel processors, we see in the Gather patterns that improvements to the caching architecture have been made between Broadwell and the new
Table 4: Spatter Results for Mini-apps

<table>
<thead>
<tr>
<th>Platform</th>
<th>AMG (n=36) GB/s (H-Mean)</th>
<th>Nekbone (n=6) GB/s (H-Mean)</th>
<th>Lulesh PENNANT STREAM GB/s (H-Mean)</th>
<th>R-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDW</td>
<td>123</td>
<td>121</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>SKX</td>
<td>328</td>
<td>309</td>
<td>12</td>
<td>35</td>
</tr>
<tr>
<td>CLX</td>
<td>315</td>
<td>287</td>
<td>14</td>
<td>41</td>
</tr>
<tr>
<td>Naples</td>
<td>140</td>
<td>323</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>TX2</td>
<td>270</td>
<td>247</td>
<td>232</td>
<td>28</td>
</tr>
<tr>
<td>KNL</td>
<td>201</td>
<td>190</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>R-value</td>
<td>0.15</td>
<td>-0.04</td>
<td>0.50</td>
<td>-.1</td>
</tr>
<tr>
<td>K40c</td>
<td>108</td>
<td>99</td>
<td>88</td>
<td>14</td>
</tr>
<tr>
<td>TitanXP</td>
<td>496</td>
<td>320</td>
<td>175</td>
<td>21</td>
</tr>
<tr>
<td>P100</td>
<td>703</td>
<td>673</td>
<td>165</td>
<td>19</td>
</tr>
<tr>
<td>R-value</td>
<td>0.66</td>
<td>0.62</td>
<td>0.62</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Skylake and Cascade Lake architectures. We see a further improvement in Cascade Lake when looking at the LULESH scatter patterns, as it outperform Skylake as well. Thus, even within the same architecture family, tweaks to caching and prefetching models can improve performance for hard-to-optimize scatter operations.

(5) To first order, performance appears most sensitive to each pattern’s delta (distance between G/S operations). Looking at the PENNANT patterns, we see a large difference in performance starting at PENNANT-G5. If we look at Table 5, we see that all the patterns before this have deltas less than or equal to 4, and the patterns including PENNANT-G5 and after have deltas larger than 400. Section 5.4.3 further expands on these patterns with a more detailed look at absolute and relative performance for these later PENNANT patterns.

Lesson: Spatter can be used to differentiate performance across architectures and can show how improved caching and prefetching hardware support in CPUs and limited caching in GPUs affects patterns with reuse. Additionally, these results can be correlated with the patterns themselves to show that delta is a primary indicator of performance for G/S operations.

5.4.3 Comprehensive Evaluation of Relative and Absolute Application Pattern Performance. In addition to the high-level takeaways, we can also use Spatter measurements to plot both the absolute and relative performance of patterns and how this relative performance varies between platforms. In Figure 9, we have a few selected gather patterns from PENNANT in (a), and a few scatter patterns from LULESH in (b).

These plots display application pattern performance as a function of stride-1 bandwidth. What this means is that stride-1 bandwidth will appear on the diagonal, and other bandwidths will appear in a vertical line through that point. It also means that all lines with unit slope are lines of constant fractional bandwidth. We have marked some of these lines in the plot for your reference. For instance, you can see that the PENNANT-G12 pattern runs at about 1/16th of the peak bandwidth on Broadwell. This complicated plot structure allows us to see how well a pattern performs on platform X vs platform Y (by comparing the y values of the two points) and also how well a platform utilizes the bandwidth available to it on a given platform (by measuring a point’s vertical distance from the diagonal).

In Figure 9(a), we have 4 different PENNANT patterns plotted, along with Stride-1 and Stride-16 results for reference. At a high level, there is clear left-to-right separation between CPUs and GPUs, due to the former having much less bandwidth available. There are a number of interesting points to discuss:

(1) If we take a look at just the Broadwell and Cascade Lake numbers, we see a slope that is greater than 1. What this means is that Cascade Lake is not only better in absolute terms, but in relative terms as well, utilizing more of its available bandwidth than the Broadwell processor.

(2) A disappointing outlier is Naples, which performs much worse than its stride-1 bandwidth would suggest. This suggests a cache architecture much less capable than the other CPUs. We hope to compare this result with AMD EPYC processors in a future evaluation.

(3) If we shift our attention to just the GPUs, we see that the large strides present in the higher-numbered PENNANT patterns have a large impact on the performance. If we reference Table 5, we see that the delta increases as the pattern number increases. This shows us that while CPUs are able to handle these large deltas relatively well, GPUs have much worse relative performance as the delta increases.

(4) Finally, if we look at both CPUs and GPUs, we see the power of this type of plot: we can see that the CPUs, due to the fact that the patterns contain some data reuse, are able to outperform GPUs on the selected patterns in terms of relative bandwidth.

Figure 9(b) shows two LULESH scatter patterns.

(1) The only platform that does well on LULESH-S3, which has a Scatter with delta 0, is the TX2, which we described in the previous section.

(2) LULESH-S1 appears to distinguish CPUs and GPUs. This pattern has a uniform stride-24 pattern with delta 8. Thus, there is quite a bit of reuse between scatter operations, which is likely
Gather Patterns on CPUs and GPUs

Figure 7: App-derived Gather Patterns. Each circle represents a single pattern. A spoke represents the performance of that pattern on a specific architecture as a percentage of the architecture’s stride-1 bandwidth. The pattern descriptions are in Table 5. Takeaway: As the chart reports relative and not absolute performance, we must emphasize that this does not show CPUs outperforming GPUs, but rather the ability of CPUs to utilize their caches on the chosen patterns.
Evaluating Gather and Scatter Performance on CPUs and GPUs

Figure 8: App-derived Scatter Patterns. Each circle represents a single pattern. A spoke represents the performance of that pattern on a specific architecture as a percentage of the architecture’s stride-1 bandwidth.

Lesson: By examining a number of application-derived G/S patterns, we show that (1) Spatter is able to reproduce unique behavior on CPUs that is not easily modeled by STREAM, (2) Spatter can also be used to discern improvements between architecture generations that go beyond simple bandwidth improvements, and (3) the Spatter benchmark suite can be used to quantitatively rank pattern performance between CPUs and GPUs and identify regimes where the CPUs are the clear winner in terms of relative performance.

6 RELATED WORK

Our primary aim for Spatter is to measure at a low level the effects of sparsity and indirect accesses on effective bandwidth for a particular application or algorithm. While a number of bandwidth-related benchmarks exist, there are no current suites that explicitly support granular examinations of sparse memory accesses. The closest analogue to our work is APEX-Map [26], which allows for varying sparsity to control the amount of spatial locality in the tested data set. However, APEX-Map has not been updated for heterogeneous devices and does not allow for custom G/S patterns.

Similar to Spatter and APEX-Map, the HopScotch microbenchmark [3] suite provides a tunable mechanism for representing mixes of read-only, write-only, and mixed access kernels in a similar fashion as Spatter. Currently, HopScotch includes a large suite of kernels intended to produce many different types of memory access patterns. While their suite does include G/S, we believe our work is complementary, allowing users a large degree of flexibility in the types of access patterns available. In addition, Spatter supports GPUs. One technical difference is Spatter’s interface to the kernel: instead of specifying the entire access pattern up front to the kernel,
we specify an index pattern and a delta. Therefore, Spatter can more effectively mirror apps that generate indices dynamically, and it does not incur the overhead of moving a large index buffer through the memory hierarchy.

In terms of peak effective, or real-world achievable bandwidth, STREAM [20] provides the most widely used measurement of sustained local memory bandwidth using a regular, linear data access pattern. Similarly, BabelStream [8], provides a STREAM interface for heterogeneous devices using backends like OpenMP, CUDA, OpenCL, SyCL, Kokkos, and Raja. Intel’s Parallel Research Kernels [12] also supports an nstream benchmark that is used for some platforms here. The CORAL 2 benchmarks also include a STREAM variant called STRIDE [22], that includes eight different memory-intensive linear algebra kernels written in C and Fortran. STRIDE includes dot product and triad variations but still utilizes uniform stride inputs and outputs. None of these suites support any access pattern aside from uniform stride, which underlines the need for a benchmark like Spatter which includes configurable and indirect access patterns.

Whereas STREAM focuses on a single access pattern, pointer-chasing benchmarks [13] and RandomAccess [18] use randomness in their patterns. Pointer-chasing benchmarks measure the effects of memory latency but are limited in scope to measuring memory latency, and RandomAccess is only able to produce random streams. Spatter cannot model dependencies like pointer chasing, but it contains kernels for modeling random access and can be used for a GUPS-like analysis.

6.1 Heterogeneous Architectural Benchmarking

Memory access patterns have been studied extensively on heterogeneous and distributed memory machines, where data movement has been a concern for a long time. Benchmarks such as SHOC, Parboil, and Rodinia provide varying levels of memory access patterns that are critical to HPC application [5, 7, 25]. For example, SHOC contains “Level 0” DeviceMemory and BusSpeedDownload benchmarks that can be used to characterize GPUs and some CPU-based devices. Likewise, other recent work has investigated vectorization support with hardware and compiler suites for next-generation applications for the SIERRA supercomputers [21]. Spatter intends to be a more focused microbenchmark that supplements these existing benchmark suites and studies. It also aims to provide a simpler mechanism for comparing scatter and gather operations across programming models and architectures.

Other work focuses on optimizing memory access patterns for tough-to-program heterogeneous devices like GPUs. Recent work by Lai, et al. evaluates the effects of TLB caching on GPUs, develops an analytical model to predict the caching characteristics of G/S and then develops a multi-pass technique to improve the performance of G/S on modern GPU devices [17]. Dymaxion takes an API approach to transforming data layouts and data structures and looks at scatter and gather as part of a sparse matrix-vector multiplication kernel experiment [6]. Jang et al. characterize loop body random and complex memory access patterns and attempt to resolve them into simpler and regular patterns that can be easily vectorized with GPU programming languages [14]. Finally, CuMAPz provides a tool to evaluate different optimization techniques for CUDA programs with a specific focus on access patterns for shared and global memory [16].

6.2 Extensions to Other Architectures

One additional motivation for this work is to better implement sparse access patterns on nontraditional accelerators like FPGAs and the Emu Chip. For FPGAs, the Spector FPGA Suite provides several features that have influenced the design of our benchmark suite including user-defined parameters for block size, work item size, and delta settings [11].

Spector uses OpenCL-based High-Level Synthesis and compiles a number of different FPGA kernels with various parameters and then attempts to pick the best configuration to execute on a specific FPGA device. While this process can be time-consuming for FPGAs due to routing heuristics, it could also be incorporated into a benchmark like Spatter to pick and plot the best result for a given configuration (i.e., work item size, block size, and vector length). This is supported but not automated in the current version of Spatter.

Finally, there is also work in computer architecture that explores the area of adding more capabilities to vector units. SuperStrider and Arm’s Scalable Vector Extension both aim to implement G/S operations in hardware [23, 24]. Similarly, the Emu system focuses on improving random memory accesses by moving lightweight threads to the data in remote DRAM [9]. Spatter complements these hardware designs and associated benchmarking by allowing users to test how their code can benefit from dedicated data rearrangement units or data migration strategies.

7 CONCLUSIONS AND FUTURE WORK

This work is motivated by the growing importance of indexed accesses in modern HPC applications and specifically looks at the use of gather and scatter operations in modern applications like the DOE mini-apps investigated in Section 2. Spatter serves as a configurable benchmark suite that can be used to better evaluate these types of indirect memory accesses by using pattern-based inputs to generate a wide class of indexed memory access patterns. The presented experiments suggest how this tool could be used by architects to evaluate new prefetching hardware or instructions for gather and scatter, how compiler writers can inspect the performance implications of their generated code, and potentially how application developers could profile representative portions of their application that rely on these operations.

We envision that the Spatter benchmark will be a tool that can be used to examine any memory performance artifact that exists in sparse codes. The current model that Spatter implements, which is a single index buffer and delta for each pattern, is descriptive of a wide range of patterns that we have seen in DOE mini-apps as well as related benchmarks like STREAM and STRIDE. However, certain aspects of the memory hierarchy cannot be properly examined by the current version of Spatter, especially those relating to temporal locality.

To increase Spatter’s ability to model memory access patterns, we plan to expand the benchmark suite with the following features: (1) model temporal locality for accesses using time delta
patterns to better represent cacheable access patterns, (2) investigate mathematical and AI techniques for modeling more complex access patterns than can be represented with combinations of stride and delta parameters, and (3) develop new open-source techniques for extracting sparse memory access patterns from applications in a timely fashion. For this last goal, we are currently working on modeling 2D and 3D stencil operations from a proprietary full waveform inversion code used for ocean surveying. Other features that we are investigating for inclusion into Spatter are kernels written with intrinsics as well as new backends for Kokkos, SyCL, and novel architectures like FPGAs or the Emu Chick.

Our goal is also to make Spatter as easy to use as possible, and useful for a wide audience. To aid in this effort, we plan to make the following upgrades to the codebase: (1) support for OpenMP 4.5 and SyCL backends, (2) automation of parameter selection, (3) optimized CPU backends that make use of prefetching and streaming accesses, and (4) make as much of our tracing and trace analysis infrastructure available along with our codebase, which is open-source and available on Github.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1710371 (SuperSTARLU).

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-000R22725.

This research was performed in part during an internship at Cray, Inc.

REFERENCES

## A APPLICATION GATHER / SCATTER PATTERNS

Table 5 lists all the patterns used in evaluation of the Spatter suite.

<table>
<thead>
<tr>
<th>Gather Pattern</th>
<th>Index</th>
<th>Delta</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PENNANT-G0</td>
<td>[2,484,482,0,4,486,484,2,6,488,486,4,8,490,488,6,8,490,488,6]</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G1</td>
<td>[0,2,484,482,4,486,484,6,488,486,6,8,490,488,8,490,488]</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G2</td>
<td>[0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60]</td>
<td>2</td>
<td>Stride-4</td>
</tr>
<tr>
<td>PENNANT-G3</td>
<td>[4,8,12,20,24,28,36,40,44,32,52,56,60,48]</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G4</td>
<td>[0,0,0,1,1,1,1,2,2,2,3,3,3,3]</td>
<td>4</td>
<td>Broadcast</td>
</tr>
<tr>
<td>PENNANT-G5</td>
<td>[4,8,12,20,24,28,36,40,44,32,52,56,60,48]</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G6</td>
<td>[482,0,2,484,482,4,486,484,4,6,488,486,4,8,490,488,6,8,490,488,6]</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G7</td>
<td>[482,0,2,484,482,4,486,484,4,6,488,486,4,8,490,488,6,8,490,488,6]</td>
<td>482</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G8</td>
<td>[2,0,0,0,2,0,0,0,2,0,0,0,2,0,0,0,2,0,0,0]</td>
<td>129608</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G9</td>
<td>[0,0,0,1,1,1,1,2,2,2,3,3,3,3]</td>
<td>388852</td>
<td>Broadcast</td>
</tr>
<tr>
<td>PENNANT-G10</td>
<td>[0,0,0,1,1,1,1,2,2,2,3,3,3,3]</td>
<td>388848</td>
<td>Broadcast</td>
</tr>
<tr>
<td>PENNANT-G11</td>
<td>[0,0,0,1,1,1,1,2,2,2,3,3,3,3]</td>
<td>388848</td>
<td>Broadcast</td>
</tr>
<tr>
<td>PENNANT-G12</td>
<td>[6,0,2,14,8,10,12,22,16,18,20,30,24,26,28]</td>
<td>518408</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G13</td>
<td>[6,0,2,14,8,10,12,22,16,18,20,30,24,26,28]</td>
<td>518408</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G14</td>
<td>[6,0,2,14,8,10,12,22,16,18,20,30,24,26,28]</td>
<td>1036816</td>
<td></td>
</tr>
<tr>
<td>PENNANT-G15</td>
<td>[0,0,0,1,1,1,1,2,2,2,3,3,3,3]</td>
<td>1882384</td>
<td>Broadcast</td>
</tr>
<tr>
<td>LULESH-G0</td>
<td>[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]</td>
<td>1</td>
<td>Stride-1</td>
</tr>
<tr>
<td>LULESH-G1</td>
<td>[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]</td>
<td>8</td>
<td>Stride-1</td>
</tr>
<tr>
<td>LULESH-G2</td>
<td>[0,8,16,24,32,40,48,56,64,72,80,88,96,104,112,120]</td>
<td>1</td>
<td>Stride-8</td>
</tr>
<tr>
<td>LULESH-G3</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>8</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-G4</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>4</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-G5</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>1</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-G6</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>8</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-G7</td>
<td>[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]</td>
<td>41</td>
<td>Stride-1</td>
</tr>
<tr>
<td>NEKBONE-G0</td>
<td>[0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,90]</td>
<td>3</td>
<td>Stride-6</td>
</tr>
<tr>
<td>NEKBONE-G1</td>
<td>[0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,90]</td>
<td>8</td>
<td>Stride-6</td>
</tr>
<tr>
<td>NEKBONE-G2</td>
<td>[0,6,12,18,24,30,36,42,48,54,60,66,72,78,84,90]</td>
<td>8</td>
<td>Stride-6</td>
</tr>
<tr>
<td>AMG-G0</td>
<td>[1333,0,1,36,37,72,73,1296,1297,1332,1368,1369,2592,2593,2628,2629]</td>
<td>1</td>
<td>Mostly Stride-1</td>
</tr>
<tr>
<td>AMG-G1</td>
<td>[1333,0,1,36,37,72,73,1296,1297,1332,1368,1369,2592,2593,2628,2629]</td>
<td>1</td>
<td>Mostly Stride-1</td>
</tr>
<tr>
<td>Scatter Pattern</td>
<td>Index</td>
<td>Delta</td>
<td>Type</td>
</tr>
<tr>
<td>PENNANT-S0</td>
<td>[0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60]</td>
<td>1</td>
<td>Stride-4</td>
</tr>
<tr>
<td>LULESH-S0</td>
<td>[0,8,16,24,32,40,48,56,64,72,80,88,96,104,112,120]</td>
<td>1</td>
<td>Stride-8</td>
</tr>
<tr>
<td>LULESH-S1</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>8</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-S2</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>1</td>
<td>Stride-24</td>
</tr>
<tr>
<td>LULESH-S3</td>
<td>[0,24,48,72,96,120,144,168,192,216,240,264,288,312,336,360]</td>
<td>0</td>
<td>Stride-24</td>
</tr>
</tbody>
</table>