The Sparse Data Reduction Engine

Chopping Sparse Data One Byte at a Time

Jonathan C. Beard

The Sparse Data Reduction Engine

Chopping Sparse Data One Byte at a Time

Jonathan C. Beard
Arm Research
Austin, Texas
jonathan.beard@arm.com

ABSTRACT

Sparse data and irregular data access patterns are hugely important to many applications, such as molecular dynamics and data analytics. Accelerating applications with these characteristics requires maximizing usable bandwidth at all levels of the memory hierarchy, reducing latency, maximizing reuse of moved data, and minimizing the amount the data is moved in the first place. Many specialized data structures have evolved to meet these requisites for specific applications, however, there are no general solutions for improving the performance of sparse applications. The structure of the memory hierarchy itself, conspires against general hardware for accelerating sparse applications, being designed for efficient bulk transport of data versus one byte at a time. This paper presents a general solution for a programmable data rearrangement/reduction engine near-memory to deliver bulk byte-addressable data access. The key technology presented in this paper is the Sparse Data Reduction Engine (SPDRE), which builds previous similar efforts to provide a practical near-memory reorganization engine. In addition to the primary contribution, this paper describes a programmer interface that enables all combinations of rearrangement, analysis of the methodology on a small series of applications, and finally a discussion of future work.

CCS CONCEPTS

• **Hardware → Emerging technologies**; *Memory and dense storage*;

ACM Reference Format:
Jonathan C. Beard. 2017. The Sparse Data Reduction Engine. In *Proceedings of MEMSYS 2017, Alexandria, VA, USA, October 2–5, 2017,* 15 pages.
https://doi.org/10.1145/3132402.3132431

1 INTRODUCTION

Scientific computation promises to better the state of the human race in ways previously not possible [21]. Some of the biggest and most interesting problems contain sparse and irregular data access patterns [28, 29]. Sparse data is found in domains ranging from machine learning [17] and data mining [33] to high performance computing [9]. There has been much work on data structure organization for the most efficient access from a software standpoint [8, 22], however, little has been done at the hardware layer except for a few notable efforts [5, 24]. Past studies have shown that much of the data brought into the last level cache goes unused before eviction [30]. Through these studies it is shown that even applications with relatively regular data access patterns could utilize bandwidth more effectively. With the projected death of Moore's Law almost upon us, bandwidth and low latency cache memory must be used as optimally as possible. Reducing superfluous data movement, so called "Dark Bandwidth" [2], has a direct impact on performance for a wide variety of applications critical to the modern world. Dark bandwidth also has a direct impact on power consumption, each byte moved consumes energy. This work's primary contribution is a scalable method of rearranging data near-memory to bring in only what is needed. The key technology, the Sparse Data Reduction Engine (SPDRE), is to this authors knowledge, one of the first such engines that works within a standard page-based virtual memory system and is capable of operating in a non-uniform memory access environment while not breaking standard coherence models. In addition to the primary contribution, this paper includes a programmer interface that enables all combinations of rearrangement, analysis of the methodology on a small series of applications, and finally a discussion of future work.

The energy of computation is now largely dominated by data movement. The energy required to load, compute, and write, using a 7nm process, has been published in academic literature at around 50pj [4]. The majority of this energy, approximately two thirds of it, is consumed by the random access memory and interconnect. While it is clear that industry research is driving lower energy memory technology, the advancements are at a much slower pace than compute technologies. This observation was made in 1995, and still holds true today [34]. In the absence of significant technological breakthroughs, there are many things that can be done to improve the overall utilization of the bandwidth provided by modern systems. Some have proposed byte addressable memory as a solution [32] to bandwidth and cache utilization. This idea implemented directly as byte addressable is likely impractical given the ratio of commands to data that would result. This paper proposes a hardware methodology that enables bulk byte addressability, which is likely far more scalable. It is an extension of work by Gokhale et al. [23] with additions to work within modern systems (i.e., with paging, virtual memory, non-uniform memory access, heterogeneous memory types, etc.) that have largely not been addressed by previous works.

MEMSYS 2017, October 2–5, 2017, Alexandria, VA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5335-9/17/10…$15.00
https://doi.org/10.1145/3132402.3132431

As compute systems scale out, the problem of providing usable bandwidth at a reasonable power level only multiplies. This problem, first identified as a major obstacle within the high performance computing community, is now recognized as an obstacle to data center density and efficient scalability [18]. Memory systems, including all the technology necessary to feed data to a processor (e.g., memory controller, DRAM bus, memory modules, non-volatile storage) have evolved around the philosophy that bulk transport is the most efficient way to provide data to the processor cores. Software on the other hand, often needs smaller grains of data. As an example, modern random access memory often delivers data to the last level cache (LLC) in 64-byte blocks, however, programmers often write software using, at most, 64-bit data types, leading to a 12.5% utilized cache line as the worst case. Data structures have evolved to attempt to provide better utilization of the bulk transfers that occur in hardware, however, the best efforts often fall short and those that work are not generalizable [31], being tied to either low-level hardware features or to specific properties of the algorithm. Many previous studies demonstrate that cache line utilization, is in general, low [30]. Cache line utilization at the LLC is an excellent proxy for utilization of the bandwidth delivered to the processor core which the technologies described in this paper aim to improve.

Sparse data can come in many forms outside of the canonical matrix filled with mostly zeros. Compressed matrix representations often trade space for indirect memory accesses. Even when all the entries within a given range of memory are filled with nonzero values, the pattern of computation might access every other byte. This type of sparseness falls into another category, known as irregularity [12]. This paper will lump both of these categories into sparse data, making a distinction only when necessary. At the core of the sparse data problem is another widely discussed problem: data movement [7]. This work considers these two problems synonymous. With the combined definitions solidified, a metric can be developed to assess the impacts of sparseness of an application's access patterns on the hardware. Two heavily related metrics that impact sparseness are cache line utilization and reuse distance of data. Cache line utilization is a proxy for the effective bandwidth at each level within the memory hierarchy (e.g., L1-D cache line utilization is a proxy for L1-D to L2 bandwidth utilization). Reuse distance influences cache line utilization. As an example, if a line is brought into the L1-D from the L2, but only half of it is used before evicting it back to the L2 then the bandwidth utilization from the L1-D to L2 is only 50%. If that line is reused at the L2 and brought back into the L1-D on subsequent accesses then that line's impact on L1-D to L2 bandwidth is still the same as before, 50%. Its reuse distance was high enough that the full cache line couldn't be reused at the L1-D. A shorter distance could have increased the effective L1-D bandwidth by enabling that line to be 100% utilized before eviction. In effect, sparseness can be local in real hardware, as the aforementioned example illustrated. This oft-overlooked interplay of reuse and utilization can lead to excessive cache line movement in exclusive caches. The technique described in this paper aims to reduce sparseness by improving cache line utilization and maximizing reuse of data within the memory hierarchy by putting the programmer in charge of what data is made contiguous.

Data rearrangement broadly refers to any technique that takes a non-contiguous section of memory and makes it contiguous by executing a mapping function. Contiguous data can be fetched using efficient bulk data movement techniques (e.g., DRAM burst [16]). In addition to making data movement more efficient, packing data also pays dividends within the cache hierarchy, increasing the effective cache size by compressing the contained data before it ever reaches the cache. With packed data, it also becomes more efficient to use vector operations. Bulk synchronization is a term used in high performance computing to describe a split-apply-combine technique of programming sometimes synonymous with MapReduce. At a high level, the aforementioned techniques: split a larger data set (either virtually or physically), functions act independently and in parallel on the split data, and lastly the results are combined. It would be extremely useful to have hardware acceleration for this splitting which is often done using a memory copy orchestrated by the main CPU core. The SPDRE enables multiple non-aliasable memory windows to be created out-of-core, freeing the core for actual compute versus data movement (overlapping execution and communications). The following sections will elaborate on the background, methodology, and initial evaluation results for the SPDRE.

2 BACKGROUND

Data rearrangement requires some compute to be placed in or near memory in order to calculate offsets, memory addresses, iterate over loops, etc. At the most basic level this could be done by a simple state machine, however, more general rearrangements require slightly more programmable compute. In the broadest sense, data rearrangement looks like many of the processing in-/near-memory (PINM) concepts produced by academic and commercial research entities over the past fifty years. Dozens of incarnations of PINM technologies have been tried, with only burgeoning niche success to date (e.g., Emu Solutions [10]). Notable systems include Computational Ram [11], Terasys [14], and DIVA [15] to name but a few. A summary of more current PINM technologies is given by Zhang et al. [35] and Balasubramonian et al. [1].

These earlier PINM systems largely were non-integrable with modern commodity operating systems (e.g., BSD Unix, Linux, etc.). While more advanced in physical design, the current generation of compute near data systems largely have the same issue. The systems to date that appear more viable rely on an accelerator offload approach where memory space is partitioned between the near-memory processors and the main compute system where data movement is orchestrated via explicit movement commands. Data rearrangement has also been attempted in several past forms. The most notable attempt to date has been the Impulse memory controller [5], which packs cache lines through scratch-pad memory within the memory controller. More recent work, such as DReAM [**?**], offer limited solutions that address within bank page-conflicts. The closest work related to that presented here is the data rearrangement work of Lloyd and Gokhale [23].

A primary concern with past systems, and something that largely stands between concept and commercial success of true in-memory compute (see Section 2.1 for definition) is the cost of commodity memory chips. Memory modules are relatively cheap, adding additional logic to memory (for in-memory processing) would make the modules cost prohibitive. Until recently, the performance of the legacy memory system that we have today has not been an issue. Modern compute architectures are now orders of magnitude faster than memory. Adding to the gulf between performance of memory and compute is the energy efficiency gap. Memory consumes far more energy than compute. These conditions make it far more likely that a PINM solution, of which data rearrangement is one type, will succeed. Going forward, it is more appropriate to refer to this class of technologies as processing in-/near-memory (PINM) [2] versus simply as processing in-memory (PIM).

Outside of economics and system software other major hurdles exist for data rearrangement hardware: coherence (specifics addressed in Section 2.1), translation for page-based virtual memory, and programmability (addressed in Section 3). With a coherent cache network [27], data can exist within the cache that must be flushed in order to have the most valid copy in memory before rearranging, this must be dealt with carefully to avoid hurting performance. The virtual memory issue, briefly discussed in [2], limits the amount of data that can be rearranged over without consulting an external facing memory management unit. This can be costly, and is largely avoided in this work by assuming contiguous huge pages whose contained physical addresses can be calculated through a simple offset calculation.

The next subsections will describe the types of rearrangement that are possible followed by a description of the API used.

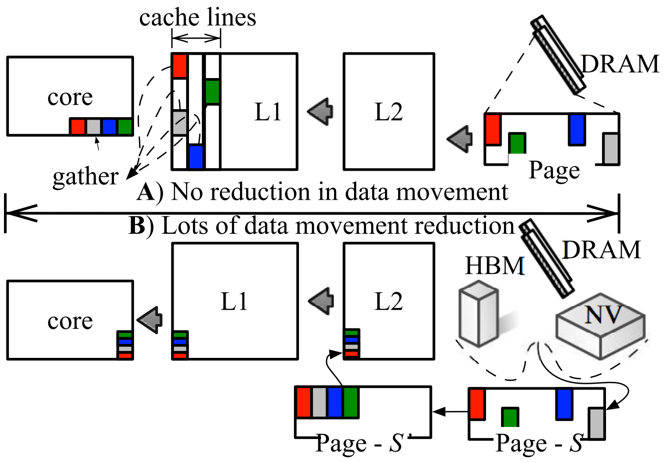
cache lines
core
L1
L2
DRAM
Page
gather
A) No reduction in data movement
B) Lots of data movement reduction
HBM
DRAM
NV
core
L1
L2
Page - S
Page - S

Figure 1: Diagram A shows an in-register rearrangement. When compared to an near-memory rearrangement shown in Diagram B, near-memory rearrangement can often provide: fewer fills to the cache, better cache line utilization, lower latency between gather operations, and reduced energy of data movement.

2.1 Types of Rearrangement

There are many potential locations within a memory hierarchy to perform computation, and subsequently data rearrangement. This complicates the definition when discussing hardware specifics. In general, the various locations of a memory hierarchy can be partitioned into categories based upon their proximity to the host compute element (processor core). These categories are: in-cache, in-register, in-memory controller, and in-memory. All of the aforementioned categories also have a "near" equivalent, where instead of direct integration at the circuit level, compute elements for the rearrangement device are integrated on a common bus integrated with the component. As an example, a vector processor integrated at the sense amplifier level would be an in-memory device, whereas, a compute element located within an cache attached at the level of the read and write ports is considered "near" memory. Each hierarchy location has very specific characteristics which contribute unique flavors to the resulting data reduction engine. In order to provide a standardized label to rearrangement types, this work will use the convention that the destination of the memory from the rearrangement engine will receive the name, as an example, Diagram A from Figure 1 shows an in-register arrangement because the work is typically done within the load queue and the destination is the register file. Data rearrangement is properly considered a subset of broader PINM technology.

In or Near-cache/-Register Most rearrangements available on modern processors fall into the in-/near-cache or in-register category. They are commonly termed gather-scatter operations. These types of instructions are useful for dense workloads that can be blocked for maximal usage of cache lines brought into the compute core between LLC and the load queue. These instructions, however, do nothing to improve bandwidth utilization for sparse data applications with poor cache line utilization. As a comparison between near-memory rearrangement, Figure 1 shows an example of an in-register arrangement.

In-memory-controller In-memory controller (IMC) rearrangement is a technique somewhere in between simple scatter-gather operations and full in-memory storage rearrangement. IMC is defined as a means to make contiguous limited sections of memory via a hardware scratchpad located at the memory controller. These techniques often use the unused section of the virtual address space (most architectures utilize less than the full 64-bits of the possible address range) to map to the scratchpad region. There are advantages and significant drawbacks to this approach which will be discussed in Section 4. This technique reduces overall memory traffic through packing of data into cache lines from the memory controller forward. The length of the rearranged segment of data is limited by the scratchpad size. There are considerable synchronization issues that must be handled when using this approach for multicore systems such as, how to share memory from a local IMC scratchpad to other cores (potentially on other sockets) and how to keep the memory within the scratchpad coherent with the memory from which it was originally pulled from (non-contiguous memory).

In or Near-memory/-storage The rest of this work focuses on rearrangement near a memory device that is not fully connected to the on-chip coherence network. On-die memory devices such as the hybrid memory cube (HMC) and high bandwidth memory (HBM) fall into this category, as do standard DIMMs and non-volatile storage media. In or

Near-memory/-storage data rearrangement has the potential for efficient fully programmable rearrangements with more significant efficiency gains than either of the aforementioned approaches. The logic behind this increase in programmability is simple: more compute logic can be made available to drive the rearrangement versus closer to the core itself. The challenges to be addressed in order to make rearrangement function at a systems level, however, are far greater for this category of device. The most significant of these challenges are: memory coherence, synchronization (explained in detail below), the API through which the programmer directs the rearrangement, and lastly translation itself. The rest of this work will focus on high bandwidth memory, on-socket rearrangement, however, almost all of the techniques are fully applicable to both on or off package memory and in-/near-storage rearrangement.

Before any details are discussed for SPDRE and near-memory rearrangement, a few assumptions must be made clear. When discussing SPDRE, it is necessary to refer to more than a single view of the memory space. The original non-contiguous memory to be rearranged from is referred to as S, the rearranged contiguous space S'. When multiple regions S' exist, a subscript will be used S'_i to denote that fact. Further assumptions are enumerated below:

(1) Any rearrangement from a non-contiguous space S to a contiguous space S' will result in a new physical mapping for S' (and potentially a new virtual memory mapping, which is the approach taken in the simulator discussed later in this section).
(2) The contiguous region S' has the ability to map back (for synchronization) to the original non-contiguous space S.
(3) The non-contiguous space S from which S' is constructed could be located on multiple physical memory pages. In general, these could be non-contiguous physical pages. For the SPDRE, either contiguous pages or one huge page to gather from is assumed (future work looks at more efficient mechanisms to utilize non-contiguous pages for the memory region S). It was found that for most workloads, utilization of the input/output memory management unit (IOMMU) would bottleneck rearrangement [2].
(4) Data in region S is either not interleaved across multiple memory controllers, or is behind a logic controller which gathers from internal interleaved memory channels or elements.

The term sparse data reduction engine (SPDRE) (generically shown in Figure 2) will be understood to consist of the following components:

(1) a central compute element that has the capability of sending a rearrangement command to a device which is not on its cache coherence network
(2) a compute element or state machine capable of recognizing the command at (or near) the memory device and capable of translating addresses based on physical offsets from a base page address (either through offset calculations or other means)
(3) capable of signaling when the rearrangement is done (or the first page of S' is available) for the host processor to load data from (the SPDRE, as well as many PINM technologies are page-atomic)

Coherence issues arise when making S contiguous in S' as in Figure 2 when cache lines are resident in a modified state within the host compute element which contain addresses that map to region S prior to issue of rearrangement command. This could result in a read-after-write with reference to S or a write-after-write if S' is synchronized back to S and the host copy is modified as well. For both of these cases, the cache lines that contain elements of S resident within the coherence domain of the shared memory space must be flushed to memory to ensure that S' contains the most recent values (i.e., sequentially consistent memory view). There are various techniques to reduce the number of flushes necessary. These will not be the subject of this work directly, however, flushing only modified lines from S to S' can be done in an efficient manner. When programming the SPDRE, one requirement that enables more efficient management of coherence is knowing the bound of S prior to issuing a rearrangement command.

Aside from synchronization within the coherence network, there are also issues with synchronization within the SPDRE device itself between memory segments. To avoid confusion, the term *synchronization* is used from this point on within the text to refer to the synchronization issues within the SPDRE unless explicitly stated. Synchronization issues arise within SPDRE-like devices near memory when the values within S' are modified, the most obvious being that each modified cache line from S' must be written back to memory (analogous to the problem described for rearranging from S to S' and handled in a similar fashion). A mechanism must exist in order to map the values within S' to those in S. Without this, the near-memory rearrangement device has no mechanism to scatter (reverse the gather operation). An alternative to providing a solution in hardware, would be to have the programmer manually manage the write back from S' to S using the compute core, however, this would largely defeat the purpose of the SPDRE which is to reduce data movement (especially data movement into the main processor core). Any SPDRE built near memory that is intended to provide both read and write capabilities to S', must be able scatter S' to S in order to be efficient. The operation to synchronize from S to S' after the initial rearrangement is also possible, however, in the authors opinion it is not practical (e.g., an instance were multiple windows of a single data segment S' changing the segment S could potentially require updating many windows S' which could be in use or active in multiple thread contexts in multiple compute elements). In the general case, ordering could also not be guaranteed which would make conflict resolution of multiple S'_i windows near memory physically impossible in some cases (i.e., limitations of clock synchronization and stable global clock reference would make temporal determination impossible). To avoid solving intractable problems, the SPDRE considered within this text defines a synchronization function which has the capability to make the changes from S' synchronize (write-back) to the non-contiguous segment S with hardware acceleration near memory, but not in the opposite direction ($S \rightarrow S'$). The synchronization is programmer driven via a `sync` function call and is page-granularity atomic with respect to near-memory write-back ordering. Any higher-level ordering

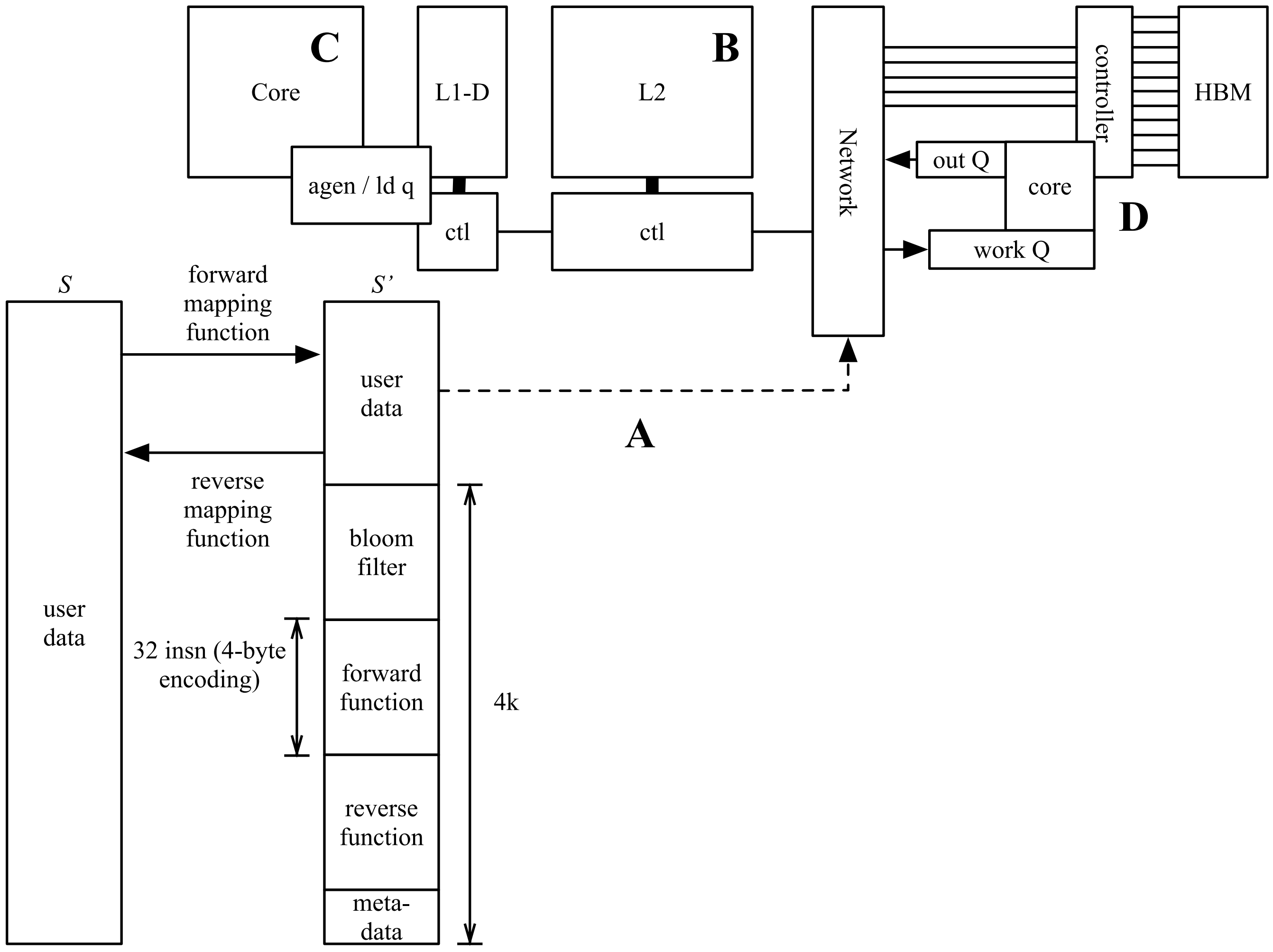

Figure 2: Abstract depiction of Sparse Data Reduction Engine (SPDRE) hardware layout (as simulated). The programmer facing API for the SPDRE enables the user to allocate memory for S' which initializes the SPDRE accessible 4K region laid out in memory immediately ahead (lower numerical address) of the returned user pointer to S'. To initiate a rearrangement from S to a contiguous range S', the programmer provided function is copied to the space labeled "forward function", the user API signals a rearrange command (label "A") via memory mapped IO, the modified lines from S are written back to memory (label "B"), the pages of S are marked as read only by the API (label "C"), then the rearrange command is released to the work queue. At the head of the work queue (label "D") the SPDRE program counter is loaded with the start address of the forward function and the translation registers are set with the base address of S and S' to translate the contiguous physical pages. On completion, or as pages of S' are ready, loads from the core are allowed to proceed. When the programmer wants to synchronize (write-back) modifications of S' to S, the modified cache lines are written back to memory, then the synchronization operation proceeds at the SPDRE vs. in-core.

is expected to be provided via more drastic mechanisms such as a programmer placed `mutex`.

3 SPECIFYING REARRANGEMENT

Specification of patterns via bit-mask, which are practical for simple shuffle operations, no longer scale to larger memory regions (the bit mask would quickly grow linearly to the size data to be rearranged). A near-memory rearrangement API should focus on operators that can scale to larger memory regions. Another often promised feature,

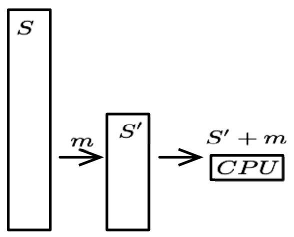

Figure 3: Transparent mapping of S to S' is impossible as the knowledge of m must be maintained and conveyed to both the rearrangement engine and the processor core.

but falsely so, is that of fully transparent rearrangement in memory. Figure 3 shows a segment S being gathered to S' via mapping function m. In order to do useful computation, the CPU core must have knowledge of that mapping function. The transfer from the rearrangement device must either be in a format recognized by the hardware and decoded by it, or included by the programmer. Without that, the information in S' is essentially encrypted via the key m. It should be clear that all but simple offsets must be done with some intervention to transmit the state of m along with S'. Prior works like IMPULSE [5] attempt to interject this mapping function m into the physical address space using a pseudo-physical address space, however, this creates multiple additional dependencies such as synchronizing the remapping table that keeps m and re-running the m for each access to memory in the remapped space. One form of intervention that could result in more transparent rearrangement, that is left to future work, is through application profiling and insertion of code which would actuate a rearrangement and compute, effectively handling m for the programmer. The work in this paper, in contrast, is based on a library approach.

This section describes a series patterns that support all types of rearrangement operations. This section describes an API-based mechanism similar to that proposed by (Lloyd and Gokhale [23]) to rearrange data in-/near-memory. There are three rearrange function modalities:

(1) Fixed offset
(2) Bijective function where given function pointer maps S to S' and S' maps to S.
(3) Dual function where one arrangement S to S' is given by one function and S' to S is given by another function

within our simulated SPDRE which enables a full spectrum of rearrangement possibilities (note: the destination contiguous memory segment must be allocated with a SPDRE specific allocate which is defined later in this paper). The fact that these are exposed as a programmer API should not lead the reader to believe that this is the only way to implement it. Future work will include looking at profile and compiler guided rearrangement.

Multiple accessory functions are necessary for the SPDRE to allocate, free, and synchronize memory. A release functionality is provided, however, the functionality could be implemented via reference counting in hardware. The SPDRE specific allocate initializes all the needed data segments for persistent information tracking (see data fields in S' from Figure 2).

```
template < class DST, class SRC >
   static std::size_t rearrange(
        DST * const dst,
        SRC * const src,
        const std::size_t nitems,
        const std::size_t offset );
```

Figure 4: Definition for simple offset rearrange the non-contiguous data from the `src` pointer to a contiguous `dst` pointer given the specified `offset` and `nitems` with respect to the length of source.

```
template < class DST, class SRC >
   static std::size_t rearrange(
        DST * const dst,
        SRC * const src,
        const std::size_t nitems,
        rearrange_func_t< DST, SRC > src_dst );
```

Figure 5: Definition of function to rearrange the non-contiguous data in `src` pointer to the `dst` pointer given the output address from the `src_dst` function and `nitems` with respect to the length of `src`. The `src_dst` function implements a bijection so that the SPDRE device can map $S \leftrightarrow S'$ with a single function.

```
template < class DST, class SRC >
   static std::size_t rearrange(
        DST * const dst,
        SRC * const src,
        const std::size_t nitems,
        rearrange_func_adv_t< DST, SRC > forw,
        sync_func_adv_t< DST, SRC>       back );
```

Figure 6: Definition of rearrange function to gather non-contiguous data from the `src` pointer to the `dst` pointer given the address from the `forw` function $S \to S'$ and the `back` function for $S' \to S$. This splits the bijectivity of Function 5 into two separate functions.

```
template < typename TYPE >
   static void alloc( TYPE **output,
                      const std::size_t nitems );
```

Figure 7: Special SPDRE library allocate virtual memory address for S' memory to pointer given by `*output`

```
template < typename TYPE >
   static void free( TYPE *ptr );
```

Figure 8: Definition of `free` function for SPDRE allocated memory region. It is expected that the programmer has called sync before this op is called otherwise data within S' pointed to by `ptr` will be lost.

```
template < class T >
    static void release( T * const src,
                         const std::size_t nitems );
```

Figure 9: Definition of function to clear memory protections of non-contiguous data segment S. More of a convenience function, given this could be implemented as reference counting, however it is implemented for the tested simulation so it is included here for completeness.

```
template < class S, class S_Prime >
   static void sync( S       * const s,
                     S_Prime * const s_p,
                     const std::size_t nitems );
```

Figure 10: Definition of sync function which is called by the programmer to write data from S' (s) to S (s_p).

The overall usage of the emulated SPDRE from the programmer (API-level) is described in flowchart form within Figure 11. Starting with the allocate call from the user, a memory region is allocated that is compatible with the SPDRE. It is assumed the data segment S resides on a set of contiguous pages (for simulation purposes only, not a limitation of the described method). A rearrange call (one of the methods listed above) initiates rearrangement into segment S'. At the same time as the rearrange call, segment S is write protected (the exact method is an architectural and OS detail), so that the only updates will be to S'. To write data back to S once S' has been written to, the programmer must explicitly call the sync function. This is also true if writes to S' are to be reflected in S before calling free. Upon release of all rearranged segments S' the write protection is removed from segment S.

4 WHERE TO COPY

In Section 2 the Impulse memory controller was described as the closest related work. This paper describes a method that gets around some of the limitations of previous memory controller techniques, while still being memory controller based (in some implementations).

Many memory controller techniques use a scratchpad within the memory controller in order to "construct" a full line from a rearrangement function (described as a mapping function m previously, see Figure 3). On each read of a value x, on LLC miss, a request is issued to the memory controller which then issue requests to memory based on the mapping function to construct a line. The requested virtual address is a line from S', whereas the data requested from memory to construct the line is from S. Each read request of x results in N requests to memory to construct the line (assuming the line isn't still buffered in the scratchpad). This adds significant latency, and duplication of execution of the function m (in the case when x is an LLC miss multiple times), and additional copies if the same address is requested. The same process must be repeated in reverse on a write to S', with the values immediately being exposed to S once the rearrangement function m is run. This presents consistency issues if multiple rearrangement windows S'_i are desired. With programmer guided approaches, it is assumed

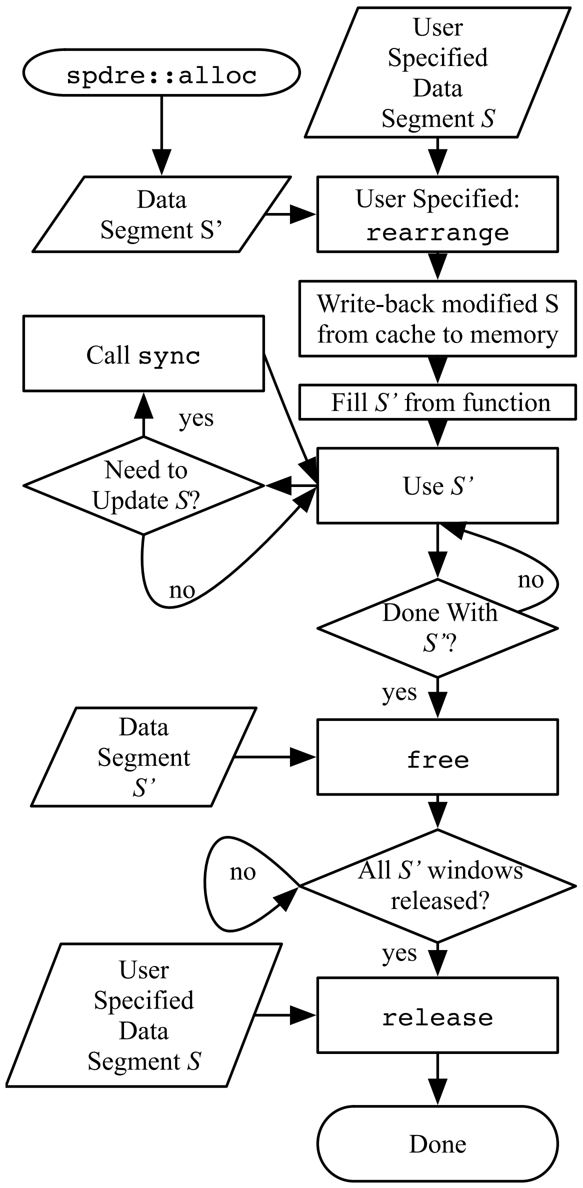

Figure 11: Flow of SPDRE from a programmer (API-level) perspective starting with allocation of segment S'. Note: the free function is not intended to synchronize the data segment S' back to S, some applications need the ability to create a window that can be thrown away despite containing modified data (with respect to S), therefore S' needs to be synchronized to S then the programmer must explicitly insert the call.

that the programmer isn't going to rearrange from memory that she isn't planning on using. With that in mind, this approach is sub-optimal for energy, latency, and bandwidth (although more space efficient). SPDRE takes another approach, improving upon

Impulse and previous reorganization works. These improvements and differences include:

(1) Eagerly reorganizing the data from S to S'_i with the thought of optimizing the transfer and doing as few additional copies as possible. In effect, main memory (fast memory, e.g., HBM) becomes the scratchpad for the SPDRE
(2) Placing all state for the mapping function m and data about modified values within S', S' itself can be moved to any non-uniform memory access (NUMA) node within the system.

5 REDUCTION METHOD

In order to be used within systems that are shared, memory constrained, and with page-based virtual memory systems as they exist today, PINM technologies, such as the SPDRE, must co-habitat constructively with existing technologies. One burden for any out of coherence network processor (including PINM) is that meta-information available to the main processor for managing dirty cache lines and pages is not easily available to the PINM device. This presents an issue for PINM that enable near-memory rearrangements of data. Presented within this section is a method that enables multiple windows of a larger memory segment to be created (exact methodology described below) and then synchronized back to the main memory at a specified interval. The net effect is decreased data movement and increased vectorization potential for many workloads.

5.1 Allocation

The SPDRE device allocates memory through standard mechanisms (e.g., `libc`). The data format of the memory returned, from user space appears exactly as any other array. Laid out in memory ahead of the user accessible memory region returned from the `spdre::alloc` memory region is an SPDRE accessible segment of memory that is marked non-accessible by the host core (a design decision to add a level of safety, see layout in Figure 2). The function m is a bijective function (or two functions), which map the data $S \leftrightarrow S'$ so that a SPDRE can stably translate from an index relative to one set to an index in another, e.g., $S'_i \leftrightarrow S_x$. This bijective function is stored within the head of the set S' which is within the SPDRE accessible segment upon the first rearrange call. At allocation several other data structures are set up, such as a hash filter (described later). Other than the special format of the SPDRE-accessible segment of memory, there is no difference between this memory and any other memory in the system.

5.2 Rearrangement

Rearrangement itself is relatively straightforward. The first step is flushing any dirty lines within the bounds of the source data S that could be in the cache hierarchy. For the described SPDRE implementation, once the modified lines are written back to memory, the pages of S are marked as read only to ensure they are not modified during the gather operation.

Initiation of rearrangement without modifying any of the standard memory control channels can be accomplished through memory mapped IO, which forms a control channel with each SPDRE. If a hardware designer is willing to add additional pathways between the memory device and the main processor cores then more efficient

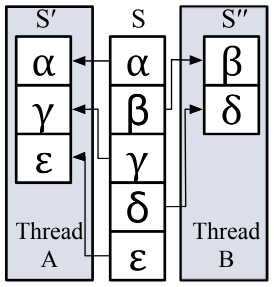

Figure 12: One key feature of the SPDRE is to accelerate creating windows of memory S' from a source segment S without occupying the main core, that are unaliasable to the original segment.

methods such as ARM's AXI bus or other interconnect technology, then the communications pathway could be much more efficient. The memory mapped IO forms a FIFO channel, two such channels make a bidirectional pathway between the SPDRE and the main processor. The minutia of signaling are not directly germane to this method, so the high level description and assumption of command channel existence will suffice.

There are a few types of rearrangements that when combined can make up any needed pattern. The first is the simple offset or filter, where the rearrangement is a fixed distance (shown in Figure 12 on the left, solid outline). The second is a programmable rearrangement where programmable logic controls the offsets (shown in Figure 12 on the right). The programmable logic (either a single or set of functions) form a bijection between S' and S. On execution of a `rearrange` call, the associated mapping function m is copied to the SPDRE-accessible data segment. Transmitting the information for rearrangement was described earlier, however what hasn't been described is how to translate virtual addresses to physical ones within the compute near data device. The exact method is beyond the scope of this text, however it should be obvious that limiting the scope of S to contiguous physical pages is one solution.

Upon completion of a rearrangement in memory, a signal is returned to the host processor to indicate that the data is available. This is the most simple mechanism to describe, however, an implementation could also transmit data as soon as it is available from S' for the host processor to consume. Having the special SPDRE segment contain the functional state m for S' enables this segment (perhaps multiple contiguous pages) to be swapped to disk or moved if the need arises. They can be synchronized back without risk of loosing state assuming the `sync` function is called using the base address for S' (that which was originally passed to the program).

Once the rearranged window S' has been created, it can be used as any other data segment, now with only the data that the processor expects. Given that the memory of S' is a new physical address and a new virtual address, it is completely unaliasable from its parent segment S. Aside from requiring more physical space than some previous techniques, it also introduces some additional design decisions for synchronization. The easiest one to visualize is when multiple threads contain unique subsets of S, e.g., $S'_0, S'_{\ldots}, S'_N$. The data contained within each S'_x must at some point be written back to the main data set S (e.g., for synchronization events, for updates to subsequent operations, etc.). Within the SPDRE, this synchronization is page atomic and user driven via API to the hardware accelerator. As previously mentioned, synchronization is only allowed in one direction $S' \rightarrow S$. This prevents the need for an all to all mapping table as would be required by some prior works.

One innovation over earlier data rearrangement methods is the ability to maintain the state of modified ranges (relative to the main window of memory) independent of the behavior of the main processor. As discussed previously, the SPDRE device sits outside the coherence network and is intended to be movable cross-socket, used by multiple threads and windows (see Figure 12), as well as being paged-in/-out at will by the operating system. This creates an issue when the goal is to minimize data movement, ideally clean values from S' will not be written back to S superfluously. The SPDRE needed a mechanism to determine which segments within S' are modified and which are not. To solve this in a space efficient manner, the SPDRE turns to probabilistic filters such as the Bloom Filter [3].

The Bloom Filter [3] is a probabilistic data structure which enables querying with the guarantee of zero false negatives. When storing information about which segments are modified within a specific region, this behavior ensures that all modified information is in fact synchronized to main memory when needed, with the expense that some clean information might be written as well. In order to index the hash filter, offsets from the base address of S' are used (i.e., $i \rightarrow N$). This scheme transferable (stateful) when paging S' to other SPDRE devices within the system so the operating system requires no further modification to make the SPDRE function in a NUMA system. Granularity of the hash filter itself can also be variable. The current approach enables the size of allocation to determine at what granularity the dirty region information should be (i.e., should a region be considered a single cache line, a whole page, or something else). The granularity itself is stored as a simple offset, again contiguous with the rearranged window. The offset itself is fixed for a given window allocation, but it can vary with each allocation. This solution is preferable over others such as bit-sets given that it doesn't grow linearly with the size of the rearranged segment S'.

This paper is about the overall concept not direct implementation, however, a generic implementation would place the SPDRE device at an HBM memory controller (as in Figure 2) so that the SPDRE, or multiple SPDRE accelerators, can interleave across channels to take advantage of maximum memory parallelism. As will be described in Section 6, the simulated SPDRE will only use a single HBM channel emulated using DDR4. Within the current implementation (simulated setup), the SPDRE-accessible data segment has space for 128-bytes of instructions for two mapping functions m which leaves room for 32 instructions assuming a 4-byte encoding if using both a forward and reverse mapping function and 64 instructions if using a bijective forward/reverse function.

6 SIMULATION ENVIRONMENT

The simulation environment used for SPDRE benchmarking is based on the flow of the API in Figure 11. The simulator (setup shown in Figure 13) consists of a separate thread pinned to an isolated core, which receives rearrange function calls from the application threads, just as a physical SPDRE would.

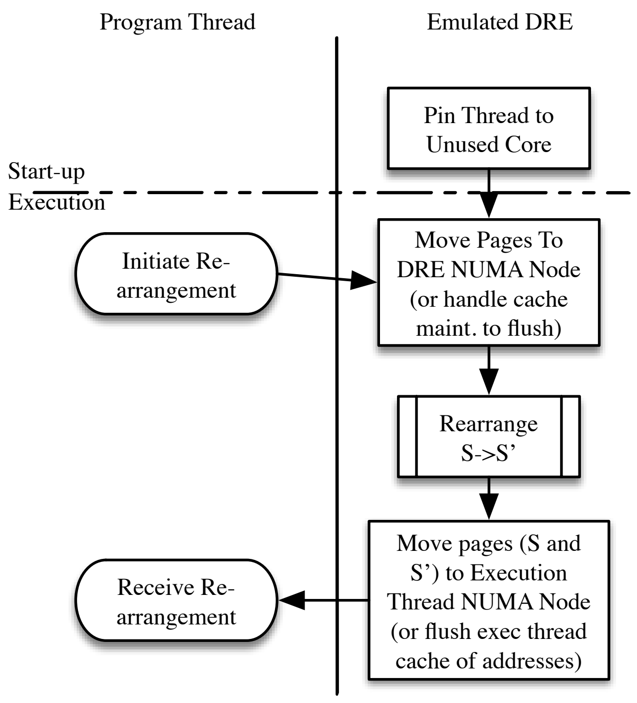
Program Thread
Emulated DRE
Pin Thread to Unused Core
Start-up
Execution
Initiate Re-arrangement
Move Pages To DRE NUMA Node (or handle cache maint. to flush)
Rearrange S->S'
Receive Re-arrangement
Move pages (S and S') to Execution Thread NUMA Node (or flush exec thread cache of addresses)

Figure 13: Depiction of the actions for threaded SPDRE emulator upon `rearrange` call from application thread. First the application thread calls `rearrange`, the addresses needed are computed, a memory fence is used to ensure all writes are present in the original segment S, then actions are taken to ensure that the core used for the SPDRE emulator does not prefetch for the application core. Once the rearrangement is complete, the pages of rearranged memory are either returned to the application core's NUMA node or they are flushed from the cache (to ensure the SPDRE emulator does not act as a prefetch thread).

The motivation of emulating the SPDRE using a pinned thread on an isolated core is to explore the design space far faster than would be possible with a cycle accurate (and full system) simulation, while allowing exploration of system issues within a simpler environment. A secondary motivation, is that on real hardware, the caching behavior can be observed exactly as it would be on a real system with a real SPDRE provided the behavior is managed correctly. The emulator enables capturing:

(1) performance counter statistics for hardware platforms (this allows comparison of on-chip gather instructions compared to near-memory data reorganization)
(2) overall execution time for a set of applications to explore a range of latencies, queueing delay for access to the SPDRE, and latency of IO to send commands to the SPDRE itself

(3) cache behavior statistics for the application cores to model the behavior of rearranged memory on the target architecture's cache hierarchy compared to standard (unmodified) benchmark applications

There are several implementation differences between the simulator code compiled for the target architecture. In order to remove the potential for cache behavior that might not reflect behavior of a non-coherent rearrangement engine. These center around the methodology to prevent accidentally warming the cache of the non-SPDRE compute elements, as well as the overhead of ensuring all writes are present before rearrangement. For all platforms a store barrier must be issued to ensure that all stores are present in-memory before issuing the rearrangement instruction. For caching stats, this issue is ignored as it would not not influence the miss counts or rates. For timing data collection, a fixed latency can be added. The rearrangement itself is platform specific. Intel architectures with NUMA, use a separate NUMA node for the memory rearrangement prevents the application threads from aliasing physical addresses in the cache from the compute element on which the SPDRE thread is pinned to. A flush is not required on the SPDRE simulation core either, as a NUMA page move is performed (changing the physical address), which makes the rearranged data loaded in the SPDRE simulator core impossible to alias. For non-NUMA (current ARM) architectures an explicit cache flush is necessary (and approximately accounted for in execution time via profiling) to prevent the SPDRE core from acting as a pre-push core to the non-SPDRE cores. A cache flush is performed on both the host core and the SPDRE simulator core after rearrangement.

One drawback of using this type of software emulation is the overhead of the function calls themselves. The function calls within Section 3 have a non-zero effect on the caches. In total, the simulation adds hundreds of instructions and has a non-zero dynamic impact on the cache traffic. The instruction counts will be higher for the application under test (even though rearrangements are performed in a separate thread). There is also the potential for an increase in the number conflict misses and increased cache pollution through use of the SPDRE. This should be kept in mind that the simulated SPDRE results are likely worse than they would be within an architectural simulator. The main advantage of this simulator is that it enables exploration of a larger area of the design space while giving a rough estimate of performance gains from using an near-memory SPDRE-like device.

Caching statistics will be given versus overall speed-up given the lack of description of the clocking mechanisms and trade-offs between an actual hardware implementation and the emulator on real hardware. Caching statistics are, however, indicative of potential speed-up. The logic being, fewer L2 look-ups from L1-D misses and increased vectorization potential from now contiguous data. This speed-up also assumes the rearrangement engine can keep up with the required bandwidth of the core (simulator provisioned as such) and code that utilizes the SPDRE API early enough and large enough to amortize the overhead of offload. All cache stats were captured using the PAPI [25] instrumentation toolkit. All benchmarks were run using Linux version 3.13. Where platforms had frequency scaling capability, it was disabled or fixed to maximum frequency. The compute core used is an Intel based with two E5-2690v3 CPUs and 64GB of DDR4 memory. With two DDR4 channels at $\sim 25\ GB/s$, they emulate a single HBM2 channel roughly in bandwidth characteristics, but not in latency. Latency in the simulator is handled via a global timing mechanism, however, the statistics given for this paper are primarily targeted towards demonstrating the effectiveness at increasing cache bandwidth so this mechanism will be described more thoroughly in future work. The SPDRE emulator itself is linked as a static library to the appropriately-ported benchmarks. The C++ library is compiled with the following compilation flags:

```
-std=c++14 -O2 -mtune=native
```

Several libraries are needed in addition to those already mentioned for various functions including: `libnuma`, `librt`, and the `pthreads` library. These are linked at benchmark application compile time.

```
for(auto index( 0 );
    index < source_length; index += offset )
{
   source[ index ] = workload( source[ index] );
}
```

Figure 14: Most basic fixed offset gather with a simple workload which is shown in Figure 16.

7 APPLICATIONS AND RESULTS

This work looks specifically at how to rearrange code that was difficult to vectorize using in-register data rearrangement techniques. OpenMP [6] versions of both applications are used exclusively. These are early results which look at the impact on cache behavior which is a critical measure of how much data movement is reduced in the system and indirectly indicative of how well an application can vectorize.

7.1 Fixed Gather

The most basic gather task is to gather data at a fixed offset is shown in code form as Figure 14, where the data needed for the workload is in the source array and the workload `workload` is called on it. With the SPDRE, this code can be rewritten as shown in Figure 15.

The SPDRE version is more verbose due to the API, however, the cache performance more than makes up for the added complexity and is far simpler than inserting complicated gather code manually. The PAPI framework is used to capture caching statistics before and after the given code from examples in Figures 14 and 15. This means that the overhead of the SPDRE function calls is included within the SPDRE numbers in Figure 17. Looking at the non-SPDRE data set, the behavior observed for the given strides is exactly as expected. Looking at a stride of four elements (size of 8-bytes, `std::int64_t`), the expected result should have at least 268M cache accesses to the L1-D in order to stride across a 1GB data set. Assuming a 64-byte cache line, the expected result should demonstrate $(\frac{64}{4})$ hits for every line on average, which means there should be a maximum of 67M cache misses. What empirical evaluation demonstrates, however, is that there are quite a few more misses for the lower strides. Further experimentation on the test platform confirms that

```
spdre::alloc( dst, source_length );
const auto num_rearranged(
    spdre::rearrange(   *dst, src,
                         source_length, offset )
);
for( auto dst_index( 0 ); dst_index < ret_val;
        dst_index++ )
{
   (*dst)[ dst_index ] =
        workload( (*dst)[dst_index] );
}
spdre::sync( *dst, src, N_SRC );
spdre::free( *dst );
spdre::release( src, N_SRC );
```

Figure 15: Simple offset in-memory rearrangement using the SPDRE to gather in memory

```
#define workload( x )\
   std::exp( static_cast< double>( x )  / 2 )\
        * 7.0f / 3;
```

Figure 16: Simple workload macro provided for random and strided microbenchmarks.

many of the resultant misses are likely the result of prefetch. The L2 and L3 data are roughly close to what is expected given a next line prefetch of 4 lines (please note that details of prefetch algorithm are assumed from the mathematical ratios, the actual degree of prefetch implemented on the target platform is unknown and likely proprietary to implementation).

The data shown in Figure17 demonstrate a considerable advantage for near-memory rearrangement. The non-SPDRE version of the loop has far more cache misses for the exact same workload, subsequent analysis reveals that most are caused by an overly aggressive prefetch ($\sim$ 95% of misses). The prefetcher on the other hand benefits the SPDRE (next line prefetch is optimal given a contiguous data space). The SPDRE version has roughly the exact number of cache misses that are expected given contiguous access pattern. The SPDRE version also includes misses created by the API itself (allocate/rearrange /synchronize/free) which handicaps it compared to the native code. The only technology that could improve this scheme further would be through directed memory placement/cache stashing which is beyond the scope of this investigation. The next microbenchmark examines random gather versus fixed stride. A Gaussian distribution is used to determine which indices are gathered into the contiguous data segment S'. The authors have designed the SPDRE to utilize small simple off the shelf cores, the inclusion of a random number generator is quite reasonable. The exact same selection function is used for the SPDRE and non-SPDRE versions.

7.2 Random Gather

As an application to gauge feasibility, random gather naively mimics sparse data access patterns. This microbenchmark implementation simply gathers values and performs some mathematical operations

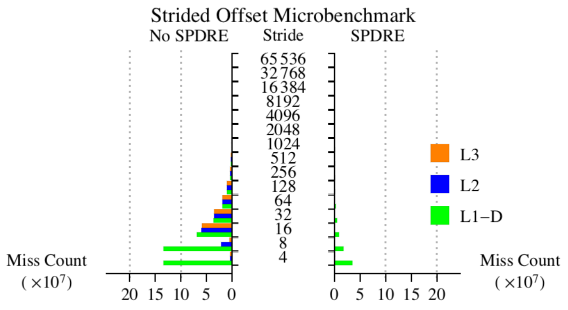

Figure 17: Average cache misses for fixed stride microbenchmark across a 1GB data set.

```
template < class DST, class SRC >
static
DST*
gather_function( SRC * const ptr,
                 const std::size_t index )
{
   static std::random_device rd;
   static std::default_random_engine eng( rd() );
   static std::normal_distribution< double > dist;
   static auto gen( std::bind( dist, eng ) );
   if( gen() >= 0 )
   {
      return( &ptr[ index ] );
   }
   else
   {
      return( nullptr );
   }
}
```

Figure 18: Gather function for random gather. It is assumed that the processor serving as the computation core can access a uniform random source that would enable it to efficiently use Gaussian transforms such as the Box-Muller [26] method. The selected pointer from S is returned as a pointer so that it may be copied to S'.

on them (see code from Figure 16). This is useful not for the complexity of the benchmark, but for an exploration of cache behavior throughout a wide range of data set sizes and randomly chosen gathers. As with the strided gather, the random gather uses a 1GB data set. The data given are averaged across 100 independent executions. The gather function and run code for the SPDRE enabled code is given in Figures 18 and 19 respectively.

PAPI instrumentation was used before and after both run functions (SPDRE and non-SPDRE) to gather caching statistics. The results (shown in Figure 21) are quite similar to the fixed stride, with the SPDRE coming out quite a bit ahead despite the overhead of the emulation environment (allocate, free, synchronize calls). Variation between runs was approximately 1% (single threaded workload).

```
static std::size_t
run( DST **dst,
     SRC * const src,
     const std::size_t N_SRC,
     rearrange_func_t< DST, SRC > function )
{
   /** allocate DRE memory **/
   spdre::alloc( dst, N_SRC );
   const auto ret_val(  spdre::rearrange( *dst,
                                          src,
                                          N_SRC,
                                          function ) );
   /** do something **/
   for( auto dst_index( 0 ); dst_index < ret_val;
            dst_index++ )
   {
      (*dst)[ dst_index ] =
            workload( (*dst)[dst_index] );
   }
   /** synchronize  **/
   spdre::sync( *dst, src, N_SRC );
   spdre::free( *dst );
   spdre::release( src, N_SRC );
   return( ret_val );
}
```

Figure 19: This code example uses the rearrange function from Figure 18 to gather data into the array `*dst` so that the workload loop can stride contiguously across it. Upon execution, the sync function returns the data to the original indices in `src` (S).

7.3 LULESH

LULESH [19] is an MPI+OpenMP code which uses unstructured access patterns on structured data to represent data motion needed for staggered grid hydrodynamics applications. This indirect access pattern is representative of production codes. This is exactly the type of access pattern that the near-memory SPDRE is designed to tackle. Our modifications keep the unstructured access patterns intact, but perform gathers near-memory vs. in-core. The hybrid version of the code spawns OpenMP threads to work within individual MPI [13] processes on the same loops as in the pure OpenMP version of the code. OpenMP is mainly used around the loops over elements and nodes, but is also used for reductions. This application contains loops that are ideal for overlapping gather near-memory with high throughput vector instructions. Typically this would have to be hand coded for each architecture, however, those equipped with an SPDRE device could gather near memory then use compiler issued vector instructions given the easily recognizable contiguous nature of the data. The code examples use only the OpenMP version, modified to use only a single thread.

The porting of LULESH was done in two ways. The first port is to illustrate a point that the rearrangement distance must be of sufficient granularity and far enough ahead of the computation that needs the data to take advantage of it. This is not a difficult task to accomplish by any means, however, care must be taken (as with

```
static std::size_t
run( DST ** dst,
     SRC * const src,
     const std::size_t N_SRC,
     rearrange_func_t< DST, SRC > function )
{
   auto count( 0 );
   for( auto index( 0 ); index < N_SRC; index += 1 )
   {
      /** use same function to give indices **/
      auto *ptr = function( src, index );
      if( ptr != nullptr )
      {
         /** access normally within core **/
         *ptr = workload( *ptr );
         count++;
      }
   }
   return( count );
}
```

Figure 20: Non-SPDRE loop of code to perform the same operation using the workload macro from Figure 16 as the SPDRE version in Figure 19. It uses the exact same gather function as well to ensure the same number of gather instructions are executed for both the SPDRE and the control

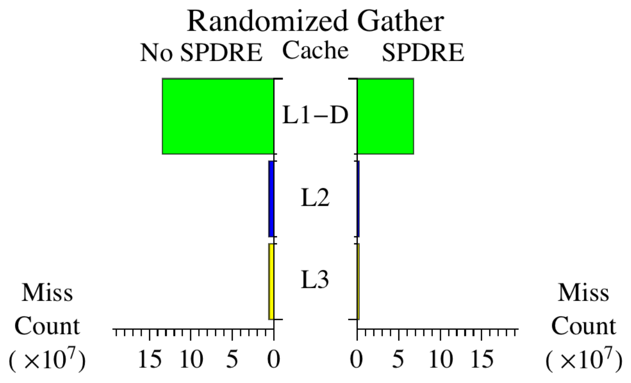

Figure 21: Randomized gather using No SPDRE and SPDRE to compare caching behavior with a control (No SPDRE). The SPDRE version incurs an order of magnitude fewer L1-D cache misses compared to the control (10^7 for SPDRE versus 10^8 for the control).

any advanced feature) not to reduce performance with these code modifications. The below code example is designed to show what not to do with a SPDRE, and the performance degradation that could result if rearrangement points are chosen poorly. Figure 22 shows such an example for the `CalcFBHourglassForceForElems` function from the benchmark file `lulesh.cc`.

PAPI was used to gather statistics before and after this loop, which is gathering from the `Domain` class the correct elements in a contiguous window. The issue with this loop is that the addition of

```
Real_t *xd1, *yd1, *zd1;
spdre::alloc( &xd1, 8 );
spdre::alloc( &yd1, 8 );
spdre::alloc( &zd1, 8 );

rearrange_func_t< Real_t, Domain > yd(
    []( Domain * const ptr,
                 const std::size_t index ) -> Real_t*
    {
       return( &(*ptr).yd( index ) );
    }
);

/** equivalent rearrange functions for xd, yd, xd **/
spdre::rearrange( xd1, &domain, 8, xd );
spdre::rearrange( yd1, &domain, 8, yd );
spdre::rearrange( zd1, &domain, 8, zd );
coefficient =
    (- hourg) * Real_t(0.01) * ss1 * mass1 / volume13;
CalcElemFBHourglassForce(xd1,yd1,zd1,
                hourgam,
                coefficient, hgfx, hgfy, hgfz);
spdre::free( xd1 );
spdre::free( yd1 );
spdre::free( zd1 );
```

Figure 22: Example of a poor port of LULESH to using the SPDRE. The gather is too small to amortize reorganization cost.

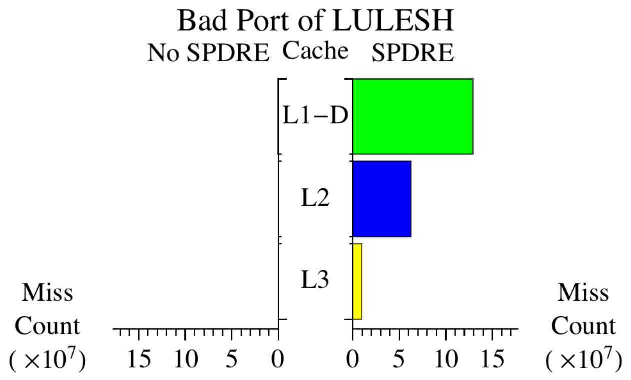

Figure 23: Cache statistics gathered using PAPI for the LULESH application given the poor gather choices made from the code example in Figure 22. The results of PAPI instrumentation of miss counts at the L1-D, L2, and L3 are quite clear (note: LULESH is executed with `-s 20 -i 20` and in single threaded mode), the overhead of calling `allocate` and `free` multiple times is quite high (likely worse due to the simulation method, but the trend would likely be exactly the same for a hardware implementation). This is an example of how the overhead of sparse data reduction near memory can outweigh the benefits when using a software driven programmable approach (ISA mechanisms may not be so limited). A better port, showing sizable improvement (amortizing the cost) is shown next.

more instructions with the `allocate`, `rearrange`,and `free` function calls outweigh the benefit of more contiguous data (empirical observations in Figure 23). It should be noted that a true hardware implementation of the SPDRE would have a much lower overhead compared to the software emulator and subsequently this loop would not be quite as bad, however, is still a good example of what not to do with an near-memory SPDRE.

A second example, which shows the positive benefits for LULESH, places the near-memory gather at a point where it can gather as much data as possible as early as possible. Given the data access patterns from the LULESH `Domain` data structure, a more extensive porting of LULESH to utilize near-memory gather would result in even greater performance. Such extensive modifications are outside the scope of this investigation. A data rearrangement for the `CalcFBHourglassForceForElems` function over a wider range is shown in the code example in Figure 24. The data (from the code example in Figure 24) are gathered into the array pointed to by `d1` right after the assignment of gamma. The elements themselves are used by assigning within the loop over each element as in the code example from Figure 25.

The pointers `xd1`, `yd1`, and `zd1` now point to contiguous segments of memory that were gathered via the SPDRE before iterating over each element. Two versions were tried, one where prefetch is inserted for each of these pointers within the loop and another without the prefetch. The prefetch is quite effective for arranged data as it is cache line contiguous, and can be inserted by the SPDRE library aware compiler far enough ahead to actually be of use. For testing these versions, the iteration counts for LULESH were increased to `i=300` and the problem size was increased to `s=100`.

Figure 26 shows the first version (no prefetch) which is a drastic improvement over the version whose cache miss statistics are shown in Figure 23. Figure 27 shows that adding a prefetch instruction, as each element is iterated over (right after the previous code example from Figure 25 showing assignment of `xd1`, `yd1`, and `zd1`), to fetch the contiguously arranged data makes the SPDRE version more performant than the non-SPDRE version. The prefetch equivalently placed for the non-SPDRE version actually reduced performance.

The results for LULESH are quite promising for the execution of applications like LULESH on vector architectures. Increasing the L1-D utilization is critical to maintaining high vector throughput due to bandwidth limitations [20]. Technologies like the SPDRE could enable many applications to align data into units that are immediately loadable to vector lanes while at the same time compacting data within the cache hierarchy so more data can fit. A benefit of providing a user-space interface for compacting data near-memory is enabling the compiler to do more vectorization for the programmer versus by hand. In the LULESH example above, providing a restricted pointer for the contiguous segments would likely enable the compiler to emit more optimal streams vector

```
Real_t *d1( nullptr );
/** allocate memory for DRE **/
spdre::alloc( &d1, 24 * numElem );

rearrange_func_adv_t< Real_t, Domain >
gather( [](  Real_t **dst,
             Domain **src,
             const std::size_t index,
             void     *data ) -> bool
{
   const auto * const numElements(
        reinterpret_cast< Index_t* >( data )
   );
   for( Index_t i2( 0 ); i2 < *numElements; i2++ )
   {
      int x( 0 + ( i2 * 24 ) ),
                y( 8 + ( i2 * 24 ) ),
                z( 16 + ( i2 * 24 ) );
      const auto *elemToNode(
        (*src)->nodelist( i2 )
      );

      for( auto i( 0 ); i < 8; i++ )
      {
         (*dst)[ x++ ] =
            (*src)->xd( elemToNode[ i ] );
         (*dst)[ y++ ] =
            (*src)->yd( elemToNode[ i ] );
         (*dst)[ z++ ] =
            (*src)->zd( elemToNode[ i ] );
      }
   }
   return( false );
} );

using sync_t = sync_func_adv_t< Real_t, Domain >;

spdre::rearrange( d1,
                  &domain,
                  24 * numElem,
                  gather,
                  (sync_t)nullptr,
                  (void*)&numElem );
```

Figure 24: Better port of the LULESH application compared to the smaller near-memory gather from Figure 22. This version gathers all 24 elements for all *numElem* at once then operates on them

```
Real_t * const xd1( &d1[ 0  + (i2 * 24) ] ),
       * const yd1( &d1[ 8  + (i2 * 24) ] ),
       * const zd1( &d1[ 16 + (i2 * 24) ] );
```

Figure 25: Assign variables contiguous portions of the gathered array from Figure 24 so that they may be passed to the rest of the code unmodified.

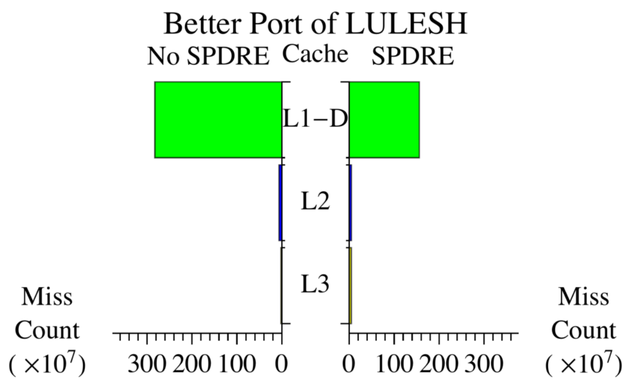

Figure 26: An improved port of LULESH using the code from Figure 24 for LULESH (no prefetch), but increasing the number of elements processed to `-s=`100 and the number of iterations `-i=`300, the SPDRE version pulls ahead of the non-SPDRE equipped example, showing half the number of L1-D misses.

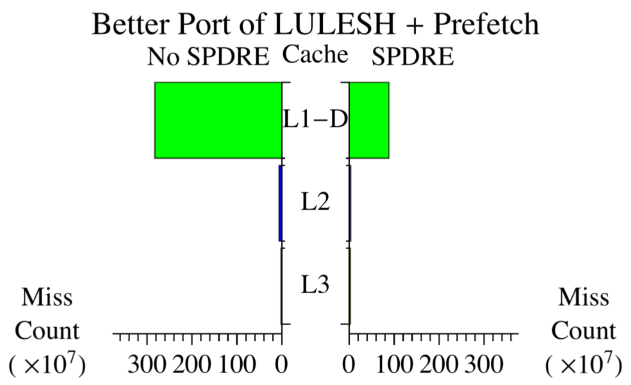

Figure 27: An improved port of LULESH using the code from Figure 24 for LULESH (with prefetch), with the number of elements processed to `-s=`100 and the number of iterations `-i=`300, the SPDRE version with prefetch results in an order of magnitude fewer L1-D misses compared to the non-SPDRE equipped LULESH.

instructions. Future work will include the coordination with vectorization explicitly as well as more machine instruction driven approaches to reduce software overhead.

8 CONCLUSIONS

With the ubiquity of sparse applications and their importance to many very important problems, it is clear that architectures must become more efficient at running them. To make data movement more efficient, however, computer architectures have moved to bulk transfer wherever possible. The unfortunate consequence is that more superfluous data is moved for sparse and irregular workloads. Bulk transfer is needed as an engineering solution to make movement more effective, so the obvious solution is to take a gather-scatter approach as close to memory as possible so that when bulk transfer is used, it is transferring only the needed data.

This work presented a hardware accelerated gather-scatter near-memory targeted at reducing overall data movement within the system. It differs from prior works by giving a solution that is implementable in operating systems with paging, with multicore chips, across sockets, and even with multiple NUMA nodes. The key technology, the Sparse Data Reduction Engine (SPDRE), is to this authors knowledge one of the first such engines to accomplish all of these points. In addition to the primary contribution, this paper includes a programmer interface that enables all combinations of rearrangement, analysis of the methodology on a small series of applications, and finally a discussion of future work.

Early results were shown that demonstrate where methods like SPDRE could be effective and in the case of LULESH where the SP-DRE could fail (when used incorrectly). The results clearly indicate that compacting data near-memory is beneficial for cache usage and could pay dividends for increased vectorization.

9 ACKNOWLEDGMENTS

Partially supported by U.S. DoE FastForward-2 Contract - Subcontract No. B609229

REFERENCES

[1] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro, IEEE* 34, 4 (2014), 36–42.

[2] Jonathan C Beard and Joshua Randall. 2017. Eliminating Dark Bandwidth: a data-centric view of scalable, efficient performance, post-Moore. In *Proc. High Performance Computing Post-Moore (HCPM'17) (Lecture Notes in Computer Science)*.

[3] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[4] Shekhar Borkar. 2013. Role of interconnects in the future of computing. *Journal of Lightwave Technology* 31, 24 (2013), 3927–3933.

[5] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, et al. 1999. Impulse: Building a smarter memory controller. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On.* IEEE, 70–79.

[6] Rohit Chandra. 2001. *Parallel Programming in OpenMP.* Morgan Kaufmann.

[7] Data Movement Dominates [n. d.]. Data Movement Dominates. https://goo.gl/rro35D. ([n. d.]). Accessed March 2017.

[8] Inderjit S Dhillon and Dharmendra S Modha. 2001. Concept decompositions for large sparse text data using clustering. *Machine learning* 42, 1 (2001), 143–175.

[9] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2015. Hpcg benchmark: A new metric for ranking high performance computing systems. *Knoxville, Tennessee* (2015).

[10] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms (IA^3 '16).* IEEE Press, Piscataway, NJ, USA, 2–9. https://doi.org/10.1109/IA3.2016.7

[11] Duncan G Elliott, W Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Custom Integrated Circuits Conference*, Vol. 30. Citeseer, 1–30.

[12] John Feo, Oreste Villa, Antonino Tumeo, and Simone Secchi. 2011. Irregular Applications: Architectures &#38; Algorithms. In *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms (IA3 '11).* ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2089142.2089144

[13] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting.* Springer, 97–104.

[14] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in memory: The Terasys massively parallel PIM array. *Computer* 28, 4 (1995), 23–31.

[15] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, et al. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing.* ACM, 57.

[16] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory systems: cache, DRAM, disk.* Morgan Kaufmann.

[17] Liping Jing, Michael K Ng, and Joshua Zhexue Huang. 2007. An entropy weighting k-means algorithm for subspace clustering of high-dimensional sparse data. *IEEE Transactions on knowledge and data engineering* 19, 8 (2007).

[18] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. 2014. Trends in big data analytics. *J. Parallel and Distrib. Comput.* 74, 7 (2014), 2561–2573.

[19] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. Lulesh 2.0 updates and changes. *Tech. Rep. LLNL-TR-641973* (2013).

[20] Peter M Kogge. 1981. *The architecture of pipelined computers.* CRC Press.

[21] Douglas B Kothe. 2007. Science prospects and benefits with exascale computing. *Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2007/232* (2007).

[22] Ping Li, Trevor J Hastie, and Kenneth W Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 287–296.

[23] Scott Lloyd and Maya Gokhale. 2015. In-Memory Data Rearrangement for Irregular, Data-Intensive Computing. *Computer* 8 (2015), 18–25.

[24] Honghui Lu, Alan L Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. 1997. Compiler and software distributed shared memory support for irregular applications. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 48–56.

[25] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference.* 7–10.

[26] Mervin E Muller. 1959. A note on a method for generating points uniformly on n-dimensional spheres. *Commun. ACM* 2, 4 (1959), 19–20.

[27] Mark S Papamarcos and Janak H Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News* 12, 3 (1984), 348–354.

[28] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.

[29] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures.* ACM, 68–70.

[30] James R Srinivasan. 2011. *Improving cache utilisation.* Technical Report. University of Cambridge, Computer Laboratory.

[31] Vinod Valsalam and Anthony Skjellum. 2002. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience* 14, 10 (2002), 805–839.

[32] R Stanley Williams. 2017. What's Next? *Computing in Science & Engineering* 19, 2 (2017), 7–13.

[33] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. 2014. Data mining with big data. *ieee transactions on knowledge and data engineering* 26, 1 (2014), 97–107.

[34] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[35] Dong Ping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph Greathouse, Mitesh Meswani, Mark Nutter, and Mike Ignatowski. 2013. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness.* ACM, 7.