# The Superstrider Architecture: Integrating Logic and Memory towards non-von Neumann Computing

Sriseshan Srikanth and Thomas M. Conte
Georgia Institute of Technology
Atlanta, Georgia
Email: seshan@gatech.edu; conte@gatech.edu

Erik P. DeBenedictis and Jeanine Cook
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM
Email: epdeben@sandia.gov; jeacook@sandia,gov

***Abstract*—We present a new non-von Neumann architecture, termed "Superstrider," predicated on no more than current projected improvements in semiconductor components and 3D manufacturing technologies, which should offer orders of magnitude advances in both energy efficiency and performance for many high-utility problem classes. The architecture is described, which is based on computing on row-wide memory words to accelerate sparse matrix algebraic operations that are normally implemented as scalar operations. A cycle-accurate simulation demonstrates potential performance improvements on existing High Bandwidth Memory (HBM) on the order of 50× that increases to 1000× or more when implemented using a fully integrated 3D technology and compared to a simple baseline. Further refinement may change these numbers, but the magnitude of the opportunity suggests further work.**

***Keywords—Moore's law; superstrider; processor-in-memory; component; von Neumann; sparse matrix; associative array***

## I. Introduction

Realization of Moore's law created a world-wide information revolution and economic expansion due to the exponential growth in the number of components per integrated structure [1] (chip) and the computing applications it enabled. Flatlining of line width on chips is evidence that Moore's law is at least changing. Moore's article included a projection of an exponentially increasing number of devices per 2D chip over time, but there are now numerous examples of (memory/storage) chips that are scaling in the third dimension, making it no longer necessary for line width to scale to maintain the vision originally proposed by Moore.

A traditional von Neumann architecture implements the stored-program concept, with both data and instructions being stored in a common memory and moved to the compute unit over a bus. Although the von Neumann architecture and Moore's law are largely responsible for the performance of today's computers, the von Neumann architecture has negatively impacted speed and energy efficiency, particularly with respect to data movement, to a point that renders it unsustainable. The cost of moving data between caches and main memory accounts for a large portion of total energy in several application domains and is cited as the key challenge in future exascale systems [2].

Although memory and storage are now implemented in 3D structures, they are currently being used only in computers of the typical von Neumann architecture. So even if scaling in the third dimension can extend Moore's law, unless we mitigate the von Neumann bottleneck that is created by the bus between processor and memory, we will not see computational efficiency (performance and energy efficiency) grow at the rates that drove economic growth over the past several decades.

In this work, we make use of additional design flexibility brought out by the shift from 2D to 3D [3] and the associated possibility of collocating logic and memory. Some very effective algorithms, such as merging (a type of sorting), become inefficient at large scale simply due to the large amount of data movement between logic and memory chips. Using a merge network as an example, Fig. 1 shows how 3D integration makes the needed interconnection pathways possible and efficient.

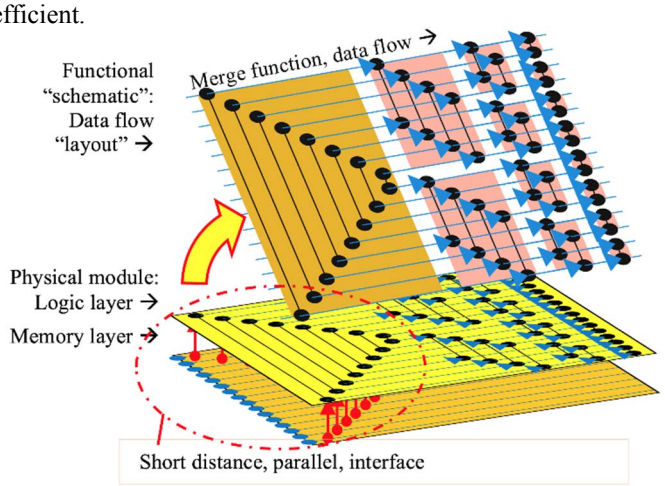


Fig. 1: For highest performance, functions are defined as schematics and laid out to reduce communications latency and energy, such as the bitonic merge network on the top. (lower) A 3D physical module with tight coupling between logic and memory allows short connections instead of conversion of signals to high energy levels and off-chip delays. An external von Neumann processor can be interfaced with one of the layers to establish compatibility with existing software. The red curve shows a representative data movement step in merging, which has no off-chip links.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Superstrider belongs to a special class of accelerators, which, in conjunction with a traditional processor, alleviates the von Neumann bottleneck by collocating some computational functions and their associated memory access, enabling increased performance and energy efficiency over a broad class of applications (e.g., data analytics, graph processing, scientific/linear algebra). Superstrider itself is not a von Neumann architecture, as it combines computation and memory into a single component that is directed by a control unit and can be integrated with a standard processor.

## II. The Superstrider Architecture

The Superstrider architecture performs a limited set of operations using conjoined logic and memory, enabling it to efficiently execute several algorithms related to sparse matrices, which are the primary computational kernels in several scientific applications. Its organization is non-von Neumann in that it computes very close to memory rows with no processor in the traditional sense. This eliminates the need to move data (referred to as *records* or key-value pairs) from the processor to memory over a bus, thereby eliminating the von Neumann bottleneck.

### A. Overview of Superstrider Operation

The operational primitive of Superstrider is to perform in-situ sorting (Section IIB) and compression (combining values of records with identical keys, discussed in Section IIF) at the granularity of a memory row. To achieve this, we organize memory into a tree where the nodes are entire physical rows of the memory, where each row comprises records in sorted order and a *pivot* key that distinguishes subtrees. By setting the fundamental unit of data to be a memory row, we achieve high bandwidth utilization and eliminate bank conflicts.

Superstrider currently implements a fixed set of fundamental algorithms to support sparse matrix computation. In this paper, we focus specifically on the addition or accumulation phase of sparse matrix multiplication. In dense multiplication of matrices, $\mathbf{C} = \mathbf{AB}$, each record $c_{ij}$ of $\mathbf{C}$ is the vector dot product of a row of $\mathbf{A}$ and a column of $\mathbf{B}$, such that $c_{ij} = \sum_k a_{ik}b_{kj}$. If we define $c_{ij}^{(k)} = a_{ik}b_{kj}$, many $c_{ij}^{(k)}$ do not exist when $\mathbf{A}$ and $\mathbf{B}$ are sparse. When the sparsity pattern is too irregular to exploit, this can be treated as a data processing problem on a series of records of the form $\{ i, j, c_{ij}^{(k)} \}$. Superstrider takes vectors of records in the form $\{ i, j, c_{ij}^{(k)} \}$, *sorts* them into a standard form, then *sums* all the values $c_{ij}^{(k)}$ for a given $(i, j)$. This is precisely the accumulation phase of sparse matrix multiply.

Therefore, in the context of such an accumulation phase, an addition operation is used to compress records with identical keys. However, Superstrider implementations of other data processing problems may employ a different collision function [4] to achieve compression. In this paper, we limit the collision function to addition.

Fig. 2 depicts an overview of Superstrider's architecture and operation. Our Superstrider implementation involves several components:

- A memory array where the rows organized as a tree; where each row stores at most $K$ records.
- An "open row" buffer and an accumulator of size $K$ each, to buffer input and intermediate processed data.
- A merge network to sort records in the open row buffer and the accumulator taken together. When laid out in a different manner and used in reverse, such a network is used to eliminate empty records that manifest as a result of compression.
- A pool of function units consisting of comparators and adders in order to aid with compression.

We dedicate the remainder of this section to explain each of these in further detail.

### B. The Merge Operation and Sorted Invariant

Superstrider's sort/merge capability is key to its efficient operation. Shown at the top of Fig. 1, it is based on a bitonic merge network that was first proposed decades ago for use in specialized chips [5]. This network takes two sequences of 8 numbers, each sequence in sorted order, i. e. a bitonic sequence, as input on the left, which flow through the network and are output as single sequence of 16 numbers in sorted order at the right. The vertical lines are comparators, and when two numbers reach the ends of a vertical line, they are compared and swapped so that the larger number is on top. A four-stage network is shown in Fig. 1, where each stage of the network outputs sorted sequences of decreasing size (e.g., bitonic sequences of 8 are merged in the first stage, the second stage merges sequences of 4, the 3rd stage merges bitonic sequences of 2, etc.).

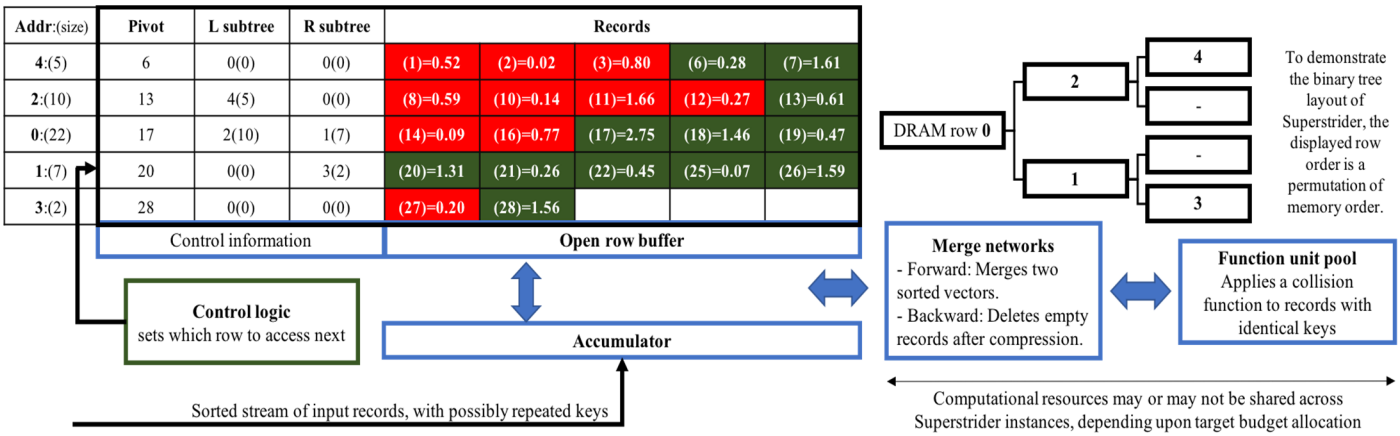


Fig. 2: Superstrider architecture and functionality. In a typical operation, such as Addvec (Section IIE), pre-sorted records in the open row buffer and the accumulator are merged using the merge network (Section IIB) and records with identical keys are summed (Section IIF) using the function unit pool. The result is a vector of sorted records, each of which has a unique key. These are then written into the binary tree layout of memory, based on pivot values.

We are able to realize efficient sorting by using only the merge capability of a bitonic network because we enforce the invariant the records in memory rows and the accumulator are always in sorted order. Merging requires just $\log_2(n)$ stages, where $n$ is the number of records being merged. Note that realizing such an efficient sorting paradigm not only aids in insertion/lookup of records into a Superstrider tree, but also aids faster compression.

### C. Control Logic

The control logic in Superstrider was designed to complement its other unique features, but the control logic will be described only briefly due to space limitations. Superstrider is a state machine overall with one state transition per memory access, with the control logic holding the main part of the state. The control logic receives signals from the data path containing information like the number of records whose key is less than the pivot or whether a subtree exists or not. The control logic produces simple commands like whether to swap the accumulator and open row buffer or to leave them as they are, compute the address or the next row to be accessed, and which function to "call" on the next row.

The most important high-level Superstrider functions include *Addvec* (add vector of records to the tree) and *Normalize*. These operate on a memory row and both use the merge network and the pool of function units to accomplish their tasks. Before describing these functions, we first describe the Superstrider data layout and control fields.

### D. Memory/Data Layout

A Superstrider row may span several physical memory banks, but for simplicity, we use a single bank to illustrate the data layout. Fig. 2 shows rows of $K = 5$ records from a Superstrider bank. The fields shown to the left of the red and green records are control fields that are described in Table 1. The contents of each record are in the format "(key) = value", so a term of $c_{ij}^{(k)}$ appears with the notation $(n(i, j)) = c_{ij}^{(k)}$, where $n(i, j)$ is an invertible function mapping two integers to one. While we perform compression (Section IIF) to remove duplicate keys from the same row before adding them to the tree, we explain in Section IIE that it is possible for the same key to appear in different rows. At a later stage, we remove such duplicate keys via normalization (Section IIH).

A bank has an open row, or a row whose contents have been transferred to the open row buffer shown in blue at the bottom of the bank. The control fields to the left of the buffer are likely to be a few dozen bits, as shown in Table 1, with the

**Table 1:** Control fields

| | |
|---|---|
| Addr:(size) | The Superstrider row number (Addr) and computed size of the subtree rooted at this row. |
| Pivot | The pivot value for sorting; is set to $p = K/2$. Elements of the left/right subtree are less than/greater than or equal to this value. |
| L subtree(size) | The Superstrider row number (Addr) of the left subtree and size of the left subtree. |
| R subtree(size) | The Superstrider row number (Addr) of the right subtree and size of the right subtree. |

⓪ Initial memory state: empty

| Addr. | Pivot | L subtree | R subtree | DRAM Rows | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| Accumulator | | | | | | | | |

①Addvec function call
(a) Input to Accumulator
(b) Copy from Accumulator to memory
(c) Pivot = (17)

| Addr. | Pivot | L subtree | R subtree | DRAM Rows | | | | |
|---|---|---|---|---|---|---|---|---|
| 0:(5) | 17 | 0(0) | 0(0) | (2)=0.02 | (14)=0.09 | (17)=0.49 | (20)=0.27 | (22)=0.45 |
| Accumulator | | | | (2)=0.02 | (14)=0.09 | (17)=0.49 | (20)=0.27 | (22)=0.45 |

②Addvec function call
(a) Input to Accumulator
(b) Merge memory row 0 with Accumulator
(2)(13)(13)(14)(17)(18)(20)(20)(22)(27)
(c) Compression function call
(2)(13)(14)(17)(18)(20)(22)(27)
(i) Sends green list to right child (>= root)
(ii) Adjust size of root (to 8)
(d) Pivot (child) = (20)

| Addr. | Pivot | L subtree | R subtree | DRAM Rows | | | | |
|---|---|---|---|---|---|---|---|---|
| 0:(8) | 17 | 0(0) | 1(5) | (2)=0.02 | (13)=0.54 | (14)=0.09 | | |
| 1:(5) | 20 | 0(0) | 0(0) | (17)=0.49 | (18)=0.63 | (20)=0.88 | (22)=0.45 | (27)=0.20 |
| Accumulator | | | | (13)=0.50 | (13)=0.04 | (18)=0.63 | (20)=0.61 | (27)=0.20 |

③Addvec function call
(a) Input to Accumulator
(b) Merge memory row 0 with Accumulator
(2)(7)(13)(13)(14)(18)(26)(28)
(c) Compression function call
(2)(7)(13)(14)(18)(26)(28)
(i) Sends red list to left child (< root)
(ii) Adjust size of root (to 12)
(c) Pivot (child) = (13)

| Addr. | Pivot | L subtree | R subtree | DRAM Rows | | | | |
|---|---|---|---|---|---|---|---|---|
| 2:(4) | 13 | 0(0) | 0(0) | (2)=0.02 | (7)=0.68 | (13)=0.61 | (14)=0.09 | |
| 0:(12) | 17 | 2(4) | 1(5) | (18)=0.82 | (26)=0.25 | (28)=0.99 | | |
| 1:(5) | 20 | 0(0) | 0(0) | (17)=0.49 | (18)=0.63 | (20)=0.88 | (22)=0.45 | (27)=0.20 |
| Accumulator | | | | (7)=0.68 | (13)=0.07 | (18)=0.82 | (26)=0.25 | (28)=0.99 |

NOTE: (18) does not match ground truth because it has not yet been merged, so is colored yellow.

Fig. 3: Addition of three records to an initially empty Superstrider

remainder of the row divided into *K* data records. The magnitude of *K* will vary depending on both the row length and size of user-defined records, but a representative range could be from *K*=200-2,000. This layout may look similar to a hashed array tree with *K*-element leaves, however, it is a binary tree with *K*-elements nodes because of its pivot based organization.

As mentioned previously, Superstrider's primary primitive is merge, with data organized optimally in memory via a tree-based sort. Each row is considered a node in the tree, with the root of Superstrider's overall tree at address zero. We use a pivot in each row that is slightly different than that found in the algorithmic literature for sorting. All keys in the left subtree are less than the pivot; all keys in the right subtree are greater than or equal to the pivot. However, each row has records whose keys can be less than, equal to, or greater than the pivot. In Fig. 2, records in the row whose key is less than the pivot are colored red, the others green, and the records are always sorted.

### *E. Addvec*

Addvec adds a vector with up to *K* records to the tree present in memory. All the records are added in parallel.

Fig. 3 shows how record vectors of size $K = 5$ are added to an empty memory using a tree-based algorithm. Records whose key is less than the pivot are colored red; all others are shown in green.

The memory is initially empty, which is step 0. In step 1, Addvec copies the input from the accumulator to memory as the root of the tree (i.e., 0:(5)) and sets the pivot to the key of the middle record, which in the example is 17. Pivots do not change once set.

Step 2 shows a new sorted vector of records (length *K*) in the accumulator being added to the root of the tree in memory. The records in the accumulator and memory are merged, which results in records with identical keys ((13) and (20) both occur twice in the merged record). This merged vector of records (of length 2*K*) is then compressed, which adds the values in records with identical keys (e.g., records (13)=0.50 and (13)=0.04 become a single record (13)=0.54). So after compression, we have three records that are less than the pivot value (shown in red, (2), (13), (14)) and five that are greater than or equal to the pivot value (shown in green, (17), (18), (20), (22), (27)). At this point, the algorithm determines which of these vectors to store in memory and which to keep in the open row buffer, and on which child to recursively call Addvec. To maintain consistency of the tree, the Addvec algorithm recursively operates on vectors that are either all less than or all greater than or equal to the pivot value (i.e., all red or all green). Using the pivot (17), a right subtree child is created and written to memory, leaving the keys that are less than 17 in the root (open row buffer) and those greater than or equal to 17 in the right subtree. The size of the root is changed (from 5 to 8) to reflect the new number of records tracked from the root and the child pivot is set to the key of the middle (third) record, or 20.

Addvec is recursively called on the left or right child in the open row buffer, depending upon the relative number of red and green records. This iteration repeats until the proper subtree does not exist, i. e. a leaf of the tree has been reached and the iteration attempts to move below the leaf. At this point, a new leaf node is created with the accumulator value and Addvec completes.

### *F. Compression*

Compression occurs when the same key appears in several records. The merge at the beginning of each step results in equivalent keys being adjacent in the row of records. The pool of function units identifies these adjacent records and with the help of the backward merge network, replaces them with a single record with the common key and a combined value. This shortens the list by some number of records. For accumulation in our sparse vector multiplication example, values are combined by floating point addition, but other applications may use a different collision function.

The simulator that constructed the diagram in Fig. 3 also computes a ground truth representation of the correct **C** matrix. Cells are colored yellow if their value does not match the ground truth. At some point later in algorithm processing (normalization), these two records pictured in yellow will appear in the same row and will be compressed.

### *G. Multiset, Set, and Standard Forms.*

The sort algorithm implemented in memory allows the tree to have an unusual structure, leading to multiple forms. A row may contain records whose key is less than the pivot, but these records can also be in the left subtree. Likewise, a row may contain records whose key is greater than or equal to the pivot, but such records can also be in the right subtree. For multiset form in general, a key can be found in a leaf node and/or in any ancestor node up to the root, or in any combination of these nodes. This property is essential for computational efficiency.

During normalization, all duplicate keys are removed from the tree, converting it from multiset to set form. Allowing only unique keys in the set (i. e., no duplicates) abides by the strict definition of set and requires that all keys in the left/right subtree are less than/greater than or equal to all keys in the row above (parent node) rather than just the pivot.

We define standard form as the set form with the additional constraint that all rows except the last have exactly *K* records. Standard form is unique.

### *H. Normalize*

Normalization compacts and reorganizes multiset trees with incompletely compressed (yellow) cells and rows of irregular length to the set or standard form. It is recursively called to adjust the division of records between each of its subtrees and the open row, with the division chosen to make the number of records in the left subtree a multiple of *K* and to fill the memory row completely if there are enough records to do so. Normalization also ensures that all the records in the left/right subtree are less than/greater-than-or-equal to any record in the tree node.

This shifting of records to/from the open row from/to a subtree uses the Normalize function (as well as min and max functions) and the Addvec function, respectively. Addvec shifts records from a row to a subtree. The records are removed from the open row starting at the pivot through the rightmost record and are sent to the subtree, with the pivot being the leftmost record in the new subtree. Normalize moves records from a

subtree to the open memory row, using either the min($n$) or max($n$) functions. These functions are called to find and remove the $n$ smallest or $n$ largest keys in a subtree and return their records. The records are then added to the open row. Obviously, this activity cannot remove more records from a subtree than exist and cannot fill the row beyond its $K$-record capacity.

If the input tree is in set form, the output of Normalize will be in standard form. All left subtrees will have a multiple of $K$ records, and hence will have rows filled to exactly $K$ records. Since every subtree except the rightmost leaf node of the entire tree is a left subtree of some node, only this rightmost node is not guaranteed to have $K$ records. This corresponds to standard form. If the input tree is in multiset form, the output of Normalize will be in set form. The Normalize function makes use of an estimate of the size of subtrees to compute how to divide the records. However, the number of records in a subtree will change during normalization when records merge. Even though merging may leave rows incompletely filled, Normalize still puts the tree in set form with no duplicates.

## III. Experimental Framework

We implemented a memory cycle-accurate simulator for the Superstrider architecture and compared its performance to a von Neumann architecture baseline. We call the simulator memory cycle-accurate because it faithfully preserves cycle timing of memory. However, HBM has a protocol for moving data from the physical memory to the controller. We model this protocol using published timing figures [6], but we extrapolate the protocol to hypothetical HBM successors with wider interfaces where the timing is speculative. At the widest possible interface width, the timing simulated represents a fully integrated logic in memory model.

The simulator accurately counts the number of cycles for the merge network and the function unit pool, however, we do not consider wire delay in these networks. We find that the incremental benefits (even with optimistic performance projections obtained by ignoring wire delays) due to relatively larger networks are rather modest, thereby rendering their complexity and overheads unjustified.

### A. Front End

A front end generates sorted records to feed the Superstrider HBM+logic structure as well as to measure performance of a von Neumann baseline. The baseline implements no reordering optimizations on the sparse input matrices and does not simulate caches because there is negligible spatial locality in sparse matrix inputs. The baseline traverses the HBM banks as rectilinear memory, i.e., without the binary tree format of the Superstrider algorithm. We use University of Florida matrices [7] as well as pseudo-randomly generated sparse matrices as input to the Superstrider simulator. The latter is especially useful in performance analysis because they enable arbitrarily large inputs, and we use these to demonstrate the benefits of the Superstrider paradigm in Section IV, with the random sparse matrix multiplication input generating 27 million non-zero records as the Superstrider input stream, where the key of each record is a pseudo-random number between 0 and 27 million.

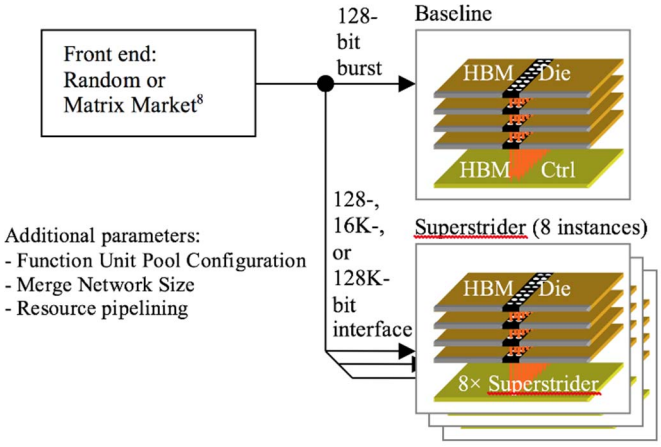


Fig. 4: Experimental framework. Baseline is an HBM stack with controller (8 channels) but no cache anywhere. For compatibility, Superstrider is implemented as 8 instances, one per HBM channel, with a sweep performed over additional parameters.

### B. Memory Model

To make reasonable comparisons with a conventional processor, the simulator illustrated in Fig. 4 models 8 identical instances of Superstrider, each connected to an HBM channel. Each HBM channel, simulated with 1 rank, comprises 8 16,384-bit wide physical DRAM banks.

Each simulated Superstrider record contains an integer key and a single precision floating point value, or 8 bytes per record. To achieve maximum algorithmic efficiency, each Superstrider row is set as wide as possible, i. e., $K$ = 2,048 records or 128K bits wide, which is the total row width offered by combining all 8 banks together. A Superstrider row is strided across its 8 banks to avoid bank conflicts, thereby realizing high channel bandwidth utilization.

The physical configuration of a bank as well as the timing parameters that govern row access time are obtained from the High Bandwidth Memory (HBM) JEDEC standard [6]. However, for simplicity, we ignore the overheads due to DRAM refresh and read-to-write delays. We assume that the HBM is clocked at one-fourth the frequency of that of logic.

### C. Simulated Parameters

In the simulator, we vary several parameters to explore the design space of the Superstrider architecture. We perform experiments to understand the sensitivity to performance of varying the following characteristics: (1) function unit pool configuration, (2) merge network size, (3) "tightness" of the logic/memory integration (interface width), and (4) resource pipelining.

#### 1) Function Unit Pool Configuration

A pool of function units are made available to facilitate compression. These can be global or shared across the HBM channels (Superstrider instances), or can be distributed/partitioned or private per HBM channel depending upon the target design budget allocated.

We evaluate three scheduling schemes:

**Partitioned/N**. The pool of $N$ function units is statically partitioned and distributed across all channels equally.

**FCFS Greedy/N**. The pool of $N$ function units is globally shared across channels and allocation/scheduling is done on a first-come first-serve basis. It is greedy in that the scheduler allocates any available units to an incoming compress request, even if a sufficient number of them is not available to perform the addition in a single time step.

**Infinite#.** We also evaluate an upper limit policy where there are an infinite number of function units available, guaranteeing constant (single cycle) access time.

*2) Merge Network Size*

A merge network is responsible for merging the open row buffer and the accumulator and for deleting empty records after compression. We assume that each channel has a merge network associated with it close to the open row buffer and accumulator to minimize wire length. As explained in Section II, this network is a $\log_2 n$-level bitonic structure, with each level taking multiple pipelined cycles depending upon the number of comparators available. For simplicity, we assume that there are enough ports to the network to feed all its comparators simultaneously.

Recall that a Superstrider row spans 8 banks, each of which is 16,384 bits wide. This means that the open row buffer as well as the accumulator can house 2,048 records each. We simulate three merge network sizes by varying the number of single-cycle two-record comparators available: 4, 256, 2,048. However, because we observe low marginal utility from increasing the size of the merge network all the way to support 2,048 comparisons per cycle, we conclude such complexity as unwarranted, and omit presentation of their results.

*3) Interface width: Near-Memory vs. In-Memory Logic:*

We simulate a near-memory compute paradigm by modeling Superstrider as an HBM controller chip. Although the row buffer is 16,384 bits wide in an example HBM configuration, data is provided to the base layer I/O in bursts that are only 128 bits wide. This means that such a near-memory logic configuration is limited by this narrow burst width, although a memory access reads an entire row into the memory row buffer.

By simulating higher interface widths we increase the "nearness" of near-memory compute, thereby increasing the "tightness" of the coupling between logic and memory, making it in-memory compute at the limit. While a production HBM has a 128 bit interface (burst) width, we simulate interface widths of 16,384 bits and 128K bits using HBM timing. However, in the absence of off-the-shelf implementations, we hypothesize the physical realization and timing of the larger interface widths.

*4) Resource Pipelining*

As the Superstrider instances are mutually independent, we allow for interleaving between components across these instances, and, thereby benefit from channel-level parallelism.

In addition, we optionally allow for pipelined execution within each Superstrider instance:

**Non-pipelined**. There is no pipelining between the operation of the components (open row buffer, accumulator, merge network, function unit pool), as they process any given row.

**Pipelined**. This builds upon the simplistic approach above by allowing adjacent components to overlap execution. For example, as a row is being read out from memory in bursts, it can proceed to the first stage of the merge network in same-sized bursts without having to wait for the entire row to be first read. Similarly, the last stage of the merge network can be overlapped with the first stage of the function unit execution, the last stage of which can be overlapped with the first stage of the deleting network. This fine-grained pipelining can be implemented using FIFOs.

**Pipelined with Write Buffer**. The pipelining described above is limited to a single row because we need to have finished processing a row in its entirety before we know the address of the next row. As such, there is a window of time where the memory channel is inactive while it waits for the row processing to finish. Subsequently, there is another window of time where the processing logic is inactive when the just-processed row is being written back to memory. To increase the overlap between logic and memory components, we employ a write buffer to store processed rows and flush them out while a subsequent row is being processed.

## IV. RESULTS

### A. Data transfers

The principal advantage of Superstrider is that it mitigates the von Neumann bottleneck by reducing the number of bandwidth-limiting and energy-consuming transfers between the processor and memory. In conventional processors, cache-line utilization (including hardware prefetching) for sparse matrices is extremely low. In contrast, Superstrider makes effective use of an entire row and there is no extraneous traffic. In fact, to benefit from bank level parallelism and extract maximal algorithmic efficiency, recall that we stride a Superstrider row across its 8 banks, thereby making effective use of 8×2048 byte wide rows at a time.

For an estimation of energy saved due to reduced memory traffic, we count the number of times logic accesses memory at a DRAM row granularity. We find that Superstrider accesses over 121× fewer physical rows from memory than the von Neumann baseline.

We will see in the next section that the amount of computational resources required for orders of magnitude speedup is relatively low. A detailed power model is beyond the scope of this paper, but it is well known [2] that data transfers are the primary contributors to energy consumption. Clearly, the significant reduction in memory traffic described above renders a proportional reduction in system energy.

### B. Performance

The Superstrider algorithm reduces the number of transfers between logic and memory thereby saving energy and reducing wasted bandwidth, or in other words, the Superstrider architecture circumvents the von Neumann bottleneck. As we shall describe below, *even the most resource-constrained configuration results in close to 50× performance improvement over von Neumann baseline.* Upon subsequently removing various resource bottlenecks from our Superstrider implementation, simulation shows an additional speedup of close to 80×.

For the memory+logic configurations of Section III, we now present sensitivity of performance of each configuration to the simulation parameters as outlined in Section III C.

**Function unit pool configuration**. Partitioned/8 yields a speedup of 49-96× for an HBM-based near-memory logic configuration with 4 comparators per channel, depending upon the Superstrider pipelining configuration employed. In other words, allocating just a single function unit and 4 comparators per Superstrider channel yields significant benefits, as shown in Fig. 5.

Partitioned/64 and FCFS Greedy/8 yield identical benefits. In other words, the designer can make a tradeoff between dedicating 8 function units per Superstrider channel for shorter wires/low scheduling overhead, and, sharing 8 units across all channels for improved resource utilization.

In general, we find that FCFS Greedy/64 approaches the performance of Infinite# when 4 comparators are used, meaning that 64 function units are more than sufficient as the bottlenecks are in the sorting network and memory access width.

**Merge network size**. For an HBM-based near-memory logic configuration, increasing the number of comparators per merge network from 4 to 256 yields an additional improvement of 1.8-2.6×, depending upon the resource pipelining scheme employed. Further increasing the merge network size is not useful as the system is bottlenecked by memory access width.

**Interface width**. Increasing the interface width from 128 bits to 16,384/128K bits (or by 128×/1024× respectively) renders an additional improvement of slightly less than 2× (for all resource pipelining and function unit pool configurations) when the system is bottlenecked by a mere 4 comparators.

However, upon *also* increasing the number of comparators per merge network to 256, significant additional improvement is seen when the interface width and the capability of the function unit pool are increased, as shown in Fig. 6. The larger merge network is now able to better keep up with data being delivered due to the increased interface width, rendering improved marginal utility of more powerful function unit pool configurations as well.

Further increasing the size of the merge network to support 2,048 comparisons per cycle realizes very modest improvements when increasing interface width to 16,384 bits and larger. Similarly, the fabrication/implementation cost of achieving logic-memory integration all the way to 128K bits is not sufficiently justified by the relative improvement in performance.

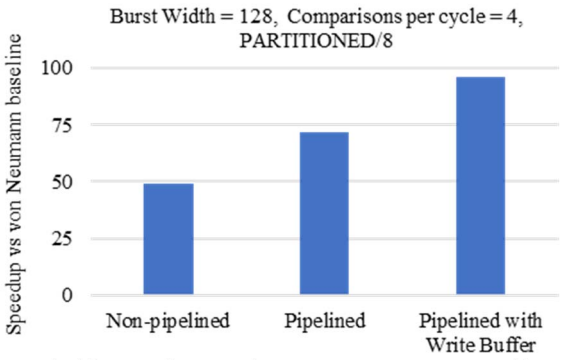


Fig. 5. Significant performance improvements are seen even with simplistic, resource-constrained implementations of Superstrider, owing to alleviation of the von Neumann bottleneck.

**Resource pipelining**. Improving the degree of overlap between logic and memory access components yields additional benefits (about 2×) in a manner similar to that of improving the function unit pool or merge network's capability, as the system becomes bottlenecked by interface width. This is demonstrated in Fig. 5, but applies to other configurations as well.

In the general scenario, the relative order of efficiency is Non-pipelined < Pipelined < Pipelined with Write Buffer. However, in the scenario where there is little overhead in computation (such as with over 2,048 comparators and 64 function units), using write-buffer based pipelining can be detrimental to performance. This is because without a write buffer, the write-back occurs to the same row, resulting in a single precharge latency incurred upon closing that row, post its write. With a write buffer, however, adjacent reads and writes are to different rows, meaning that there is an additional overhead of row activation and precharge. When there is little overhead in computation, this additional row opening and closing DRAM command latencies are no longer hidden. A row remap memory may be designed to remove this limitation.

## V. Comparison with Related Work

Superstrider is a proposed hardware solution to address computational efficiency issues for many algorithms that experience performance degradation due to the von Neumann bottleneck. Sparse matrix multiplication suffers performance loss on current computational platforms due to this bottleneck.

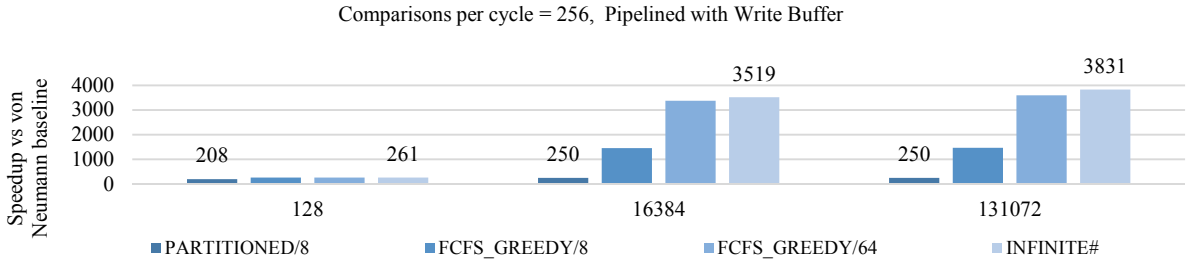


Fig. 6: The utility of increasing the compute resources available to Superstrider is realized only when the memory access bottleneck is loosened. By simulating higher-than-128 interface widths, progressively tighter integration of logic and memory is realized. For space constraints, only the best resource pipelining scheme is shown, although these trends apply to other schemes as well.

Many software solutions have been proposed to address this [8-11]. However, little work has been done to address this issue in the hardware space.

Song et. al. introduce a graph processor architecture that represents graph processing as a sparse matrix algebra problem [12]. They propose a novel node architecture that comprises several modules including memory (cacheless), ALU, systolic merge sorter, matrix reader and writer, control, and interprocessor communication. The systolic merger sorter is used for sorting matrix record indices during matrix operations and is the key to graph processing. The ALU module operates on a stream of sparse matrix elements, making it more efficient than operation of data in a register file as in traditional processor architectures. These graph processing nodes are interconnected in a 3D toroidal configuration to form a 3D parallel processor. Through bit-level simulation models using various graph processing kernels, they show orders of magnitude speedup over commercial systems.

A 3D-stacked logic-in-memory (LiM) system architecture for accelerating graph processing proposed [13] has logic layers stacked between DRAM dies that communicate vertically using through silicon vias. Their customized logic for processing sparse matrix data is integrated with a CAM memory customized to specifically support matrix assembly in the SPGEMM benchmark. Results show over two orders of magnitude of performance and energy efficiency improvement over traditional multithreaded implementations. However, they operate only on compressed data, which means that the input matrices have to be static. Not only do we provide comparable benefits (if not better), our architecture is also capable of supporting inputs that require dynamic insertion. Furthermore, our abstraction provides the potential to implement other data irregular applications by modifying the collision function to something other than addition.

Although we simulated only the accumulation phase of sparse matrix multiply (index sort and merge), this accounts for more than 95% of computational throughput for sparse matrix multiply [12]. Neither of the architectures described above implement as tight an integration of logic and memory, Superstrider implements a unique merge capability, is coupled with HBM rather than a more traditional memory device, and it strides/operates on a "super"-sized, very wide memory word. All of these features together realize very large improvements in computational efficiency.

## VI. Conclusion

In this work, we present Superstrider, a 3D architecture that integrates logic in memory to alleviate the von Neumann bottleneck and increase computational efficiency of key scientific algorithms, particularly the sparse matrix accumulation phase in sparse matrix multiplication that suffers from poor cache utilization. We show that Superstrider can potentially provide orders of magnitude speedup for accumulation compared to conventional von Neumann architectures and processing.

This is as a result of improved bandwidth utilization and we attribute this to its unique operational primitives (merge and compress) that operate at the granularity of a memory row, and its novel tree-based representation of sparse matrices. Even the most resource-constrained HBM configuration simulated results in a 50× performance improvement. Furthermore, reasonably increasing the tightness of logic-memory integration and the amount of computational logic resources available renders a *further*, potential improvement of 80×.

## VII. Future Work

The authors have already performed additional theoretical work on Superstrider's generality, reported in Ref. 3. The memory "tightness" can be generalized into an incremental development strategy, like Moore's law. Also, the floating point add and multiply operations are a mathematical semi-ring. If the semi-ring is replaced by, for example, addition and minimum, Superstrider can perform graph operations useful in, for example, big data computations. The generalization of Superstrider is an associative array processor.

It should be possible to broaden Superstrider's function beyond "accumulation." For example, store matrices **A** and **B** in Superstrider and create an empty tree for **C**. Then run a function that computes **C** = **AB** with no processor intervention.

Hardware demonstrations should be possible, even without building any hardware. There are companies selling HBM controller IP that offer samples of their product as an FPGA connected to an HBM stack. If these companies would allow augmentation of the controller IP with Superstrider function, perhaps these product samples could become the first production Superstrider hardware.

## References


[1] G. E. Moore, "Crammng more components onto integrated ciruits," *Electronics Magazine*, vol. 38, no. 8, 1965.

[2] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," presented at the *Proceedings of IEEE International Symposium on Workload Characterization*, 2013.

[3] Zhao, J., Zou, Q., & Xie, Y. (2017). Overview of 3-D Architecture Design Opportunities and Techniques. IEEE Design & Test, 34(4), 60-68.

[4] J. Kepner, "Spreadsheets, Big Tables, and the Algebra of Associatve Arrays," MAA & AMS Joint Mathematics Meeting, Jan 4-7, 2012

[5] Batcher, K. E. (1968). "Sorting networks and their applications". Proc. AFIPS Spring Joint Computer Conference. pp. 307–31

[6] High bandwidth memory (hbm) dram. [Online]. Available: https://www.jedec.org/standards-documents/results/HBM

[7] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[8] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - a high level linear algebra library for gpus and multi-core cpus," in *International Workshop on GPUs and Scientific Applications*, 2010.

[9] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, 2015.

[10] Deveci, M., Trott, C., & Rajamanickam, S. (2017, May). Performance-Portable Sparse Matrix-Matrix Multiplication for Many-Core Architectures. In Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International (pp. 693-702). IEEE.

[11] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix-matrix multiplication for the gpu," *ACM Transacations on Mathematical Software (TOMS)*, vol. 41, no. 4, 2015.

[12] W. S. Song, J. Kepner, V. Gleyzer, H. T. Nguyen, and J. I. Kramer, "Novel graph processor architecture," *Lincoln Laboratory Journal*, vol. 20, no. 1, 2013.

[13] Zhu, Qiuling, et al. "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware." *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013