SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM

Gerasimos Gerogiannis
University of Illinois at Urbana-Champaign, USA
gg24@illinois.edu

Serif Yesil*
University of Illinois at Urbana-Champaign, USA
syesil2@illinois.edu

Damitha Lenadora
University of Illinois at Urbana-Champaign, USA
damitha2@illinois.edu

Dingyuan Cao
University of Illinois at Urbana-Champaign, USA
dc29@illinois.edu

Charith Mendis
University of Illinois at Urbana-Champaign, USA
charithm@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

The widespread use of Sparse Matrix Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix Dense Matrix Multiplication (SDDMM) kernels makes them candidates for hardware acceleration. However, accelerator design for these kernels faces two main challenges: (1) the overhead of moving data between CPU and accelerator (often including an address space conversion from the CPU's virtual addresses) and (2) marginal flexibility to leverage the fact that different sparse input matrices benefit from different variations of the SpMM and SDDMM algorithms.

To address these challenges, this paper proposes *SPADE*, a new SpMM and SDDMM hardware accelerator. SPADE avoids data transfers by tightly-coupling accelerator processing elements (PEs) with the cores of a multicore, as if the accelerator PEs were advanced functional units—allowing the accelerator to reuse the CPU memory system and its virtual addresses. SPADE attains flexibility and programmability by supporting a *tile-based* ISA—high level enough to eliminate the overhead of fetching and decoding fine-grained instructions. To prove the SPADE concept, we have taped-out a simplified SPADE chip. Further, simulations of a SPADE system with 224–1792 PEs show its high performance and scalability. A 224-PE SPADE system is on average 2.3x, 1.3x and 2.5x faster than a 56-core CPU, a server-class GPU, and an SpMM accelerator, respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average 43.4x and 52.4x faster than the GPU and the accelerator, respectively. Further, SPADE has a small area and power footprint.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; **Special purpose systems**.

*Now at NVIDIA. He can be reached at syesil@nvidia.com.

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589054

KEYWORDS

Hardware accelerator, sparse computations, SpMM, SDDMM

ACM Reference Format:
Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. 2023. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589054

1 INTRODUCTION

Two fundamental linear algebra kernels are Sparse Matrix Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix Dense Matrix Multiplication (SDDMM). They are widely used in domains such as machine learning [13, 26, 37, 50], parameter estimation [80], image segmentation [67], atmospheric modeling [9, 49], aerodynamic design [55], matrix factorization [35], and linear algebra solvers [3, 5, 39, 61]. As an example, consider the popular Graph Neural Networks (GNNs) [15, 25, 37, 68]. In GNNs, the message-passing kernels that aggregate information for a vertex from its neighbors can be implemented with SpMMs. Further, the message-passing kernels that aggregate information for an edge from its incident vertices can be implemented with SDDMMs [59, 71].

The SpMM and SDDMM kernels have very similar computational and memory access behavior. They have a unique mix of sparse and dense operands that sets them apart. Indeed, like in other sparse computations, the nonzero structure of the input sparse matrix determines the kernel characteristics. At the same time, the dense matrices in SpMM and SDDMM increase the data reuse and arithmetic intensity in the kernel. However, the data reuse behavior heavily depends on the input sparse matrix structure. As a result, for some matrices, SpMM and SDDMM can exhibit high locality, benefiting from fast local memories (e.g., caches), while for other matrices, they can show highly-irregular accesses and benefit little from fast local memories. In the latter case, main memory bandwidth becomes the bottleneck. In both cases, a non-negligible fraction of the requests is typically served by the lower levels of the memory hierarchy, making latency tolerance a major concern.

In addition, like dense kernels, SpMM and SDDMM can use SIMD execution units effectively but, unlike dense kernels, they are not easily amenable to spatial processing element (PE) arrays due to their hard-to-predict sparse matrix-driven memory access patterns.

Overall, SpMM and SDDMM belong to a special category, which has its unique challenges.

Given the importance of SpMM and SDDMM, there are proposals for hardware accelerators of one or both kernels (e.g., [28, 64, 65]). The Tensaurus accelerator [65], which supports SpMM and other kernels, is a specialized array of PEs with access to HBM memory. It uses a custom compression format to stream sparse data. It provides some hardware configurability that allows it to support different matrix operations. Sextans [64] is an FPGA-based accelerator designed for SpMM. It has an HBM module from which dense and sparse data is streamed to on-chip scratchpads. Extensor [28] proposes a technique to eliminate redundant computation in sparse algebra. While very useful in computations involving more than one sparse matrix, it is less beneficial for SpMM and SDDMM. More details are presented in Section 8.

While these proposals advance the state of the art, the approach they use faces two main challenges. First, moving data from the CPU to the accelerator and back entails substantial overhead. Such overhead results from the spatial separation between CPU and accelerator, and their connection through slow links. Even if the separation could be overcome, non-GPU accelerators still require an address space conversion (e.g., with a DMA engine), as their PEs do not support the CPU's virtual addresses. GPUs support a mechanism for virtual address sharing with CPUs, although it has limited performance [4, 69]. Since future applications are likely to benefit from fine-grain interleaving between CPU and accelerator phases [44, 78, 81], these overheads need to be eliminated.

Second, these designs have only very marginal flexibility and programmability. For SpMM and SDDMM, they do not support different execution strategies based on the non-zero structure of the sparse input matrix. It is well known [29, 77] that such structure determines the flavors of SpMM and SDDMM algorithms that deliver the highest performance. To be future-proof, SpMM and SDDMM accelerators must be programmable.

In this paper, we propose a novel hardware accelerator for SpMM and SDDMM that explicitly focuses on minimizing the overhead of data moves between CPU and accelerator, and on providing programmability without compromising performance. The accelerator is called *SPADE.*[1]

SPADE minimizes data transfer overheads by tightly-coupling accelerator processing elements (PEs) with the cores of a multicore, as if the accelerator PEs were advanced functional units. Each core is associated with one or more SPADE PEs, which share the core's secondary TLB, L2 cache, and last level cache (LLC). While such a design is intrusive, it allows the accelerator to reuse the CPU's memory subsystem and virtual addresses, forgoing any data move or address space conversion. Moreover, the design is scalable: larger multicores with higher memory bandwidth become larger, higher-bandwidth SPADE accelerators.

SPADE attains flexibility and programmability by supporting a special ISA. The ISA does not use fine-grained instructions, which induce instruction fetch and decode overheads; instead, it uses *high-level tile-based* instructions. A core passes these high-level instructions to the PEs, which break them into micro-operations. The resulting programmability allows the accelerator to adapt to the diverse sparsity patterns found in matrices originating from different application domains.

SPADE is designed for high performance. As indicated above, PEs do not have the overhead of fetching and decoding fine-grained instructions. Moreover, the PE pipeline is conceived for latency tolerance. Further, the cache subsystem flexibly allows cache bypassing to reduce cache pollution.

To prove the SPADE concept, we have prototyped and taped-out a simplified four-PE SPADE chip using TSMC 65nm technology. In addition, we have simulated a SPADE system with 224-1792 PEs, and shown that SPADE attains high speedups and is scalable. A 224-PE SPADE system is on average 2.3x, 1.3x, and 2.5x faster than a 56-core Intel Ice Lake CPU, a server-class NVIDIA V100 GPU, and a scaled-up, idealized version of the state-of-the-art Sextans SpMM accelerator [64], respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average 43.4x and 52.4x faster than the GPU and the accelerator, respectively. Furthermore, SPADE has a small area and power footprint.

Overall, this paper's contributions are:

- The SPADE approach to hardware acceleration of SpMM and SDDMM, which attains high performance by eliminating data transfer overheads, flexibility by using a tile-based ISA, and scalability.
- The SPADE architecture, including a pipeline designed for latency tolerance and a flexible memory subsystem.
- Demonstration of SPADE with a simulation-based evaluation of its performance.

2 BACKGROUND

2.1 SpMM and SDDMM Operation

Figure 1 shows the operation of SpMM (top chart) and of SDDMM (bottom chart). SpMM takes an input sparse matrix A and an input dense matrix B, and produces an output dense matrix D. Both dense matrices have K columns. We refer to K as the dense matrix row size. Given a non-zero element (NNZ) in A, (e.g., a), its column index (c_id) is used to index a row of B and its row index (r_id) is used to index a row of D. In SpMM, for every NNZ, the corresponding row of B is multiplied by the NNZ value, and the result is accumulated on the corresponding row of D.

SDDMM takes an input sparse matrix A and two input dense matrices (B and the transposed of C denoted as C^T), and produces an output sparse matrix D that has the same non-zero structure as A. Given an NNZ in A, its r_id is used to index a row of B, and its c_id is used to index a row of C^T. In SDDMM, for every NNZ, an inner product is performed between the corresponding dense rows, and the result is multiplied with the NNZ value, and stored in the corresponding position in D.

In both kernels, one of the dense matrices is accessed using the NNZ r_id and the other dense matrix using the NNZ c_id. We refer to the former (D in SpMM and B in SDDMM) as row matrix or *rMatrix*, and to the latter (B in SpMM and C^T in SDDMM) as column matrix or *cMatrix*. Reuse opportunities for the dense matrices depend on the structure of A: NNZs with the same r_id indicate reuse for the rMatrix, while NNZs with the same c_id indicate reuse for the cMatrix. The NNZ values are used only for a single SIMD operation and are not reused.

[1] SPADE is named after the mix of SPArse and DEnse operands in SpMM and SDDMM.

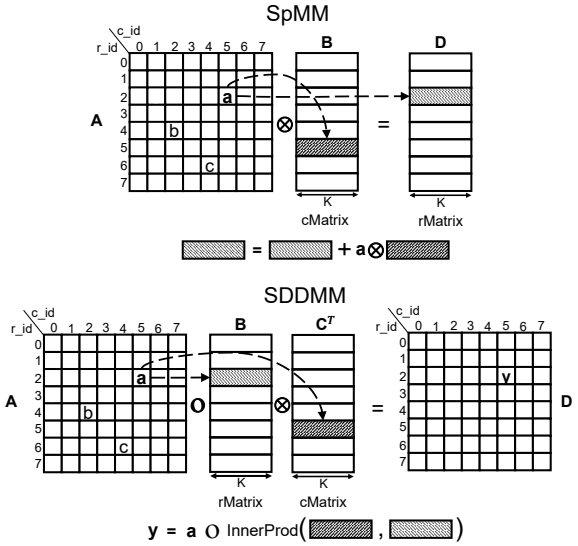


Figure 1: SpMM and SDDMM operation.

2.2 Versatile Architecture for Data Reuse

Matrix tiling is a well-studied method to enhance data reuse in linear algebra kernels. Tiles limit the working set size so that it fits in fast local memory. In dense operations, deriving a good tiling strategy is relatively straightforward. However, in sparse operations, this is not the case, as the effect of the tile size is strongly affected by the sparsity pattern of the matrices [29]. In fact, at high sparsity levels, tiling may be useless due to limited reuse opportunities.

Sparse matrices may display reuse opportunities localized inside tiles (*Local Reuse*) or spread across different tiles (*Distant Reuse*). For an accelerator to be able to support a wide range of matrices, it needs to provide architectural knobs to exploit different reuse behaviors. Ideally, it needs to support: (1) tiles of arbitrary size, (2) techniques to control the concurrent working set of multiple threads in fast shared memories by deliberately ordering tile execution across threads, and (3) the bypass of different levels of the memory hierarchy when there are no reuse opportunities. Note that scratchpad-based solutions enforce strict limitations on tile sizes and are not as versatile as caches.

Accelerators should support these capabilities, and allow programmers to select the tiling parameters, order tile execution across threads to control the concurrent working set, and enforce caching selectively. With this support, accelerators can adapt the computation to the sparsity patterns of the input matrix.

3 MOTIVATION: DATA TRANSFER COST

In most existing accelerator designs, the host memory is separate from the accelerator memory (e.g., [14, 79]), and explicit data transfer to and from the accelerator memory is needed to perform the computation and get the results. In sparse computations, the cost of this data transfer is harder to amortize, since the data reuse in the accelerator is lower than in dense computations. Non-iterative applications and applications whose data does not fit in the accelerator memory [2] are especially sensitive to this overhead.

To quantify the overhead of these data transfers, we measured the execution time of a single SpMM iteration on two machines: (1) a dual-socket Intel Ice Lake CPU server and (2) an NVIDIA V100 GPU connected to its own host via PCIe. The GPU execution time includes the host-to-device and device-to-host data transfer time, and any address mapping or translation overhead. In the GPU experiment, we measure time using CUDA events. Unfortunately, these measurements do not allow us to decouple the address mapping overhead from the data transfer overhead. Hence, we report the value of the combined overhead. We execute 10 different matrices from the popular SparseSuite [17] matrix collection (which we describe in Table 2 of Section 6) and use two different dense matrix row sizes (K equal to 32 and 128).

Figure 2 shows the execution time of the GPU system normalized to the execution time of the CPU system. We show bars for each matrix and K value. The bars are broken down into transfer overhead and kernel execution. We see that, if we just consider the kernel execution, the GPU is always faster than the CPU. However, if we consider both kernel execution and transfer overhead, the GPU is always much slower than the CPU. This is because the transfer overhead accounts, on average, for 97% of the total time.

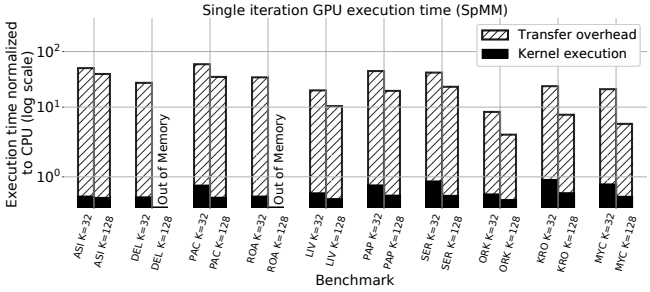


Figure 2: GPU execution times of a single SpMM iteration (including the transfer overhead) normalized to CPU execution times. Times are broken down into transfer overhead and kernel execution. The y scale is logarithmic.

4 OVERVIEW OF THE SPADE SYSTEM

SPADE is built on two principles: attain high performance by eliminating any data move between CPU and accelerator, and attain flexibility without hurting performance by using a high-level tile-based ISA for the accelerator.

As hinted in Section 3, the overhead of moving data from the CPU to the accelerator and back can be substantial. It results from the spatial separation between CPU and accelerator, which often implies transfers through the slow PCIe bus. It is also caused by the fact that non-GPU accelerators do not typically support the CPU's virtual addresses, and an address space conversion is required (e.g., with a DMA engine). GPUs support a mechanism for virtual address sharing with CPUs, although it is inefficient [4, 69]. Since future applications are likely to benefit from fine-grain interleaving between CPU and accelerator phases [44, 78, 81], this overhead needs to be eliminated.

SPADE avoids this overhead by tightly-coupling accelerator processing elements (PEs) with the cores of a multicore. Each core is associated with one or more SPADE PEs, which share the core's secondary TLB (STLB), L2 cache, and lower cache hierarchy. While such a design is intrusive, it allows the accelerator to reuse the CPU's memory subsystem and virtual addresses, and forgo any

data move or address space conversion. One could think of the SPADE PEs as advanced functional units for SpMM and SDDMM.

The second principle driving SPADE's design is programmability without lowering performance. Non-programmable accelerators risk obsolescence as the algorithms they accelerate evolve. For example, graph structures have been evolving, as social networks create power-law graphs. Such graphs, represented as sparse matrices, change the SpMM and SDDMM algorithm flavors that perform best. Hence, for programmability, SPADE has an ISA. However, using fine-grained instructions carries the overheads of instruction fetching and decoding. To avoid these overheads, SPADE uses a *high-level tile-based* ISA. A CPU core passes these instructions to the PEs, which then break the instructions into micro-operations. The result is programmability and high performance.

4.1 Tight Integration with the Host

SPADE is composed of many PEs that execute tiles of SpMM and SDDMM operations, as required by the instructions supplied by a simple general-purpose core called Control Processing Element (CPE). The PEs are integrated in the cores of a multicore to be able to: (1) directly access the cores' memory system and (2) use the cores' virtual addresses and STLBs.

Figure 3 shows the multicore architecture. Each CPU core is associated with a few (1-4) SPADE PEs which, for energy efficiency, are clocked at a fraction of the core's frequency (e.g., 1/4). The PEs share the core's STLB (like the DMA engines in [24]), the core's L2 cache, and the lower cache hierarchy. Each PE has an L1 data cache and, because many of the sparse data structures do not use caches efficiently, a Bypass Buffer (BBF) that optionally allows PE accesses to bypass the cache hierarchy. No instruction cache is needed, as explained later. Overall, this coupling of CPU core and SPADE PE(s) makes the system scalable: larger multicores with higher memory bandwidth will result in larger, higher-bandwidth accelerators.

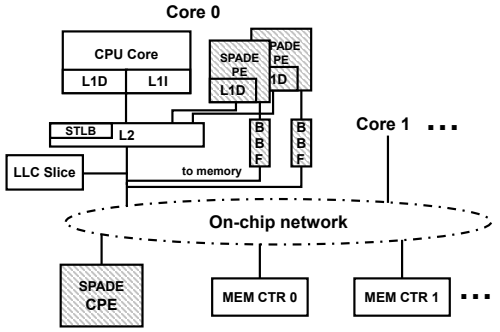


Figure 3: Integration of SPADE processing elements (PEs) and the Control Processing Element (CPE) in a CPU multicore.

Programs interleave CPU-mode execution sections with SPADE-mode execution sections. In the former, only CPU cores are active; in the latter, only the SPADE PEs and the CPE are. Before a SPADE-mode section starts, the pages of the matrix data structures are pinned in physical memory, so that SPADE PEs do not suffer page faults. SPADE PEs can suffer TLB misses.

Because SPADE PEs have their own L1 cache and can access data through the BBF, especial cache operations are needed in the transitions between the two modes. Specifically, before transitioning from SPADE to CPU mode, the SPADE PEs' L1 caches are written back to the L2 caches and invalidated, and the BBFs are written back to main memory and invalidated.

Similarly, before transitioning from CPU to SPADE mode, two actions are done. First, the CPU cores' L1 caches are written back to the L2 caches and invalidated. Second, any data currently in the cache hierarchy that, in the upcoming SPADE-mode section, could be accessed by the SPADE PEs through the BBFs needs to be written back to memory and invalidated from the cache hierarchy. This is to ensure that the accesses through the BBFs will get the correct data versions and that no stale data will be left in the caches after the next SPADE-mode section. If the programmer or compiler cannot identify the aforementioned data, all cached data may need to be written back and invalidated.

The CPE gets the PEs to start, stop, or execute any instruction. To do so, the CPE uses a mechanism similar to the Intel Architecture MWAIT instruction [33], which enables a processor to wait for a store operation to an address. Specifically, the CPE has a few memory-mapped registers for each PE called *Input* registers. When the CPE writes to one of the Input registers of a given PE, that PE is informed in hardware, unless notifications are masked. With this mechanism, the CPE can trigger operations on a PE.

Putting all this together, programs execute as follows. When the CPU cores run, the PEs are paused. When the program wants to switch from CPU mode to SPADE mode, the cores perform the selective cache writebacks and invalidations described above, notify the CPE, and pause. The CPE writes an initialization instruction in the Input registers of all the PEs, and then the tile-based coarse-grained instructions that each PE needs to execute. Every time that the CPE writes to an Input register, the corresponding PE is informed. That PE reads the Input register that contains the instruction and executes the instruction. As soon as a PE has read an Input register, the CPE is informed, and the CPE can then overwrite the Input register with a new instruction. After the SpMM or SDDMM operation completes, the CPE commands the PEs to write back and invalidate the L1s and BBFs, and terminate execution. The program then resumes in CPU mode.

4.2 Tile ISA and Programming

The instructions that the CPE sends to PEs to execute are SpMM or SDDMM operations on a tile. Before these instructions are generated, a compiler or a programmer analyzes the sparse input matrix and decides on a set of good configuration parameters. These parameters include the size of the tiles of the sparse input matrix (Row Panel Size and Column Panel Size, as shown in Figure 4(a)), the order of tile execution, and whether PE accesses to the different data structures should bypass the caches. The row and column panel sizes chosen determine the layout of the matrix in memory, as shown in Appendix A. Both in the appendix example and in our evaluation, we use the COO format.

At run time, a program running on the CPE takes the layout of a tiled matrix and generates instructions for the PEs. Specifically, assume that we want to perform an SpMM or SDDMM operation as shown in Figures 4(a) and 4(b). The figures show all the parameters and, in shaded pattern, the tiles.

The CPE first creates the *Initialization* instruction, shown in Figure 4(c), which is passed to all PEs to initialize them. The instruction

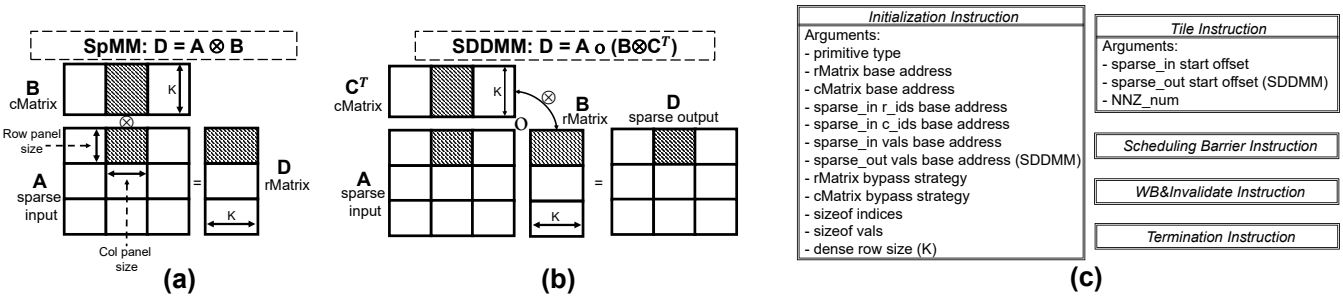


Figure 4: Matrix operations and instructions supported by SPADE.

includes, as arguments: the primitive type (SpMM or SDDMM); base addresses of the rMatrix and cMatrix; base addresses of the r_ids, c_ids, and vals arrays of the tiled input sparse matrix (shown in Appendix A) which, together with the *sparse_in_start_offset* metadata of Appendix A, generate the addresses of the beginning of each tile data; for SDDMM only, the base address of the vals array of the tiled output sparse matrix; the cache hierarchy bypass strategy for the row and column matrices (the two choices for each of them are bypass or no bypass); the size of the indices and vals (which vary depending on the size of the matrix and the type of the data); and the size of the dense matrix row K (shown in Figures 4(a) and 4(b)). On reception of this instruction, the PEs store it in special registers and reconfigure some of their hardware.

After that, the CPE creates *Tile* instructions that it sends to individual PEs. A tile instruction, shown in Figure 4(c), specifies the tile to work on. Its arguments are data from the tiling metadata shown in Appendix A: the offset of the first nonzero of the tile in the r_ids, c_ids, and vals arrays of the input sparse matrix; for SDDMM only, the offset of the first nonzero of the tile in the vals array of the output sparse matrix; and the number of nonzeros in the tile. The PE that receives this instruction processes the tile. There are no upper/lower bound constraints on the tile size.

The figure shows three more instructions, *Scheduling Barrier*, *WB&Invalidate*, and *Termination*. They are discussed next.

4.3 Tile Scheduling

The CPE freely assigns tiles to the PEs. However, in SpMM, the assignment must respect one constraint: all the tiles in a *row panel* should be assigned to the same PE. A row panel is the set of tiles that span the same set of rows of the matrix, as shown shaded in Figure 5(a). The reason is that the SpMM operation on two tiles of the same row panel updates the same rows of the rMatrix. To avoid data races, only one PE processes such tiles. As an example, Figure 5(a) shows the order of assignment of tiles to two PEs. PE 0 (in white circles) gets the tiles in Row Panel 0 and then those of Row Panel 2; PE 1 (in black circles) gets the tiles in Row Panel 1 and 3. This assignment restriction does not apply to SDDMM.

A schedule like Figure 5(a) maximizes inter-tile rMatrix reuse, but hinders inter-tile cMatrix reuse. Specifically, consider the tiles assigned to PE 0 only (i.e., those with white circles), and assume that Tiles labeled 2 and 6 in Figure 5(a) have non-zeros in the same columns. As PE 0 executes Tile 6, it reuses cMatrix data brought into caches during its execution of Tile 2. However, this *Distant Reuse*, as described in Section 2.2, can be missed if the execution of Tiles 3, 4, and 5 evicts the cMatrix data.

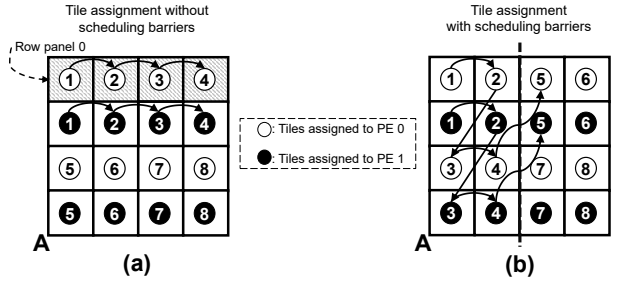


Figure 5: Tile scheduling without (a) and with (b) barriers.

When an analysis of the input matrix determines that this reuse of cMatrix data is important, we want to force a schedule like in Figure 5(b). In the figure, PE 0 and PE 1 first execute tiles in column panels 0 and 1, and once they are both done, they move to the next two column panels. Both PEs help each other reuse cMatrix data. This behavior is attained with the SPADE *Scheduling Barrier* instruction. The CPE passes Tiles 1, 2, 3, and 4 in Figure 5(b) to PE 0 and, after PE 0 has read the instruction for Tile 4, the CPE sends it the barrier instruction. Similarly, after PE 1 has read the instruction for its own Tile 4, the CPE sends it a barrier. The CPE will not send any new tile instruction to any PE until *both PEs* have read the barrier instruction—signifying that both PEs have completed their last assigned tile.

A similar procedure is used to terminate a SPADE-mode section. The CPE sends the *WB&Invalidate* instruction to all PEs. When a PE reads it, the PE writes back and invalidates its L1 cache and BBF, while the CPE sends the *Termination* instruction to the PE. When the PE has completed the writebacks and invalidations, it reads the termination instruction and pauses. When the CPE notices that all PEs have read the termination instruction, it stops SPADE-mode execution and CPU-mode execution can restart.

Finally, to ensure correctness in the presence of BBFs, there are two data layout requirements. First, the dense matrix row size (K) must be a multiple of the cache line size and, therefore, dense rows start at cache line boundaries. Second, in SDDMM, the first nonzero value of each tile in the output sparse matrix must be at the beginning of a cache line. To satisfy these conditions, padding may be inserted in the cMatrix, rMatrix, and the output sparse matrix of SDDMM.

4.4 Processing Element Pipeline

The PE pipeline is designed for latency tolerance. It is composed of three logical stages: sparse front-end, vOp Generator, and dense back-end (Figure 6). We describe each stage next.

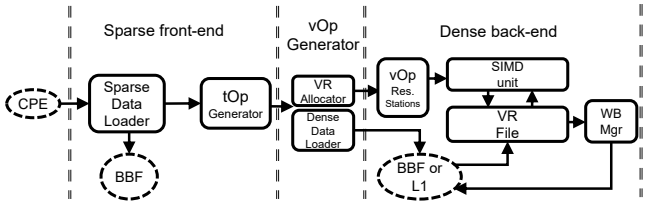


Figure 6: SPADE processing element pipeline overview.

Sparse front-end. This stage receives the tile instruction from the CPE and performs two functions: issue requests for the sparse matrix data and generate the addresses of the dense matrix data. As discussed in Section 4.2, the instruction has all the information needed to load the sparse data. Since the sparse data is not reused, the loads are issued by a Sparse Data Loader to the BBF and bypass all caches. As the data is received, the hardware uses the information in the *r_ids* and *c_ids* arrays to generate the addresses of the dense cMatrix and rMatrix rows needed. Specifically, for each value in the *vals* array, it generates a *Tuple Operation (tOp)* with: {begin address of rMatrix row, begin address of cMatrix row, value from the vals array}. For SDDMM, there is a fourth field: address of the entry in the val array of the output matrix that will receive the result.

vOp Generator. The PE operates as a simple out-of-order vector engine and has a Vector Register File (VRF). Each Vector Register (VR) stores a full cache line. However, a tOp refers to dense matrix rows, which may extend over multiple cache lines. Hence, the vOp Generator stage breaks down each tOp into potentially multiple vector-long operations called *Vector Operations (vOps)*. The two operands of a vOp are cache-line sized and thus fit in VRs.

Next, for each vOp, the vOp Generator allocates two VRs for the two operands, and the Dense Data Loader requests the operands from memory. These dense accesses are directed to either the L1 or the BBF, depending on the cache bypassing strategy chosen. A vOp includes: {VR ID for the first operand, VR ID for the second operand, value from the vals array}. In addition, for SDDMM, the vOp also includes a destination VR ID (for the scalar output data), and the offset of the output data in the destination VR. Eventually, in SDDMM, this destination VR will contain scalar outputs from multiple vector operations that generate the data in the destination memory line. For SpMM, the implicit destination VR is the VR of the rMatrix input. In all cases, the VRs allocated are tagged with the memory line addresses requested. Note that, before allocating a register, the hardware checks the VRF tags to see if the requested data is already in a register from a previous operation. If so, the data is not requested from memory.

Dense back-end. This stage pushes the newly-generated vOps to the vOp Reservation Stations. Once the two operands of a vOp are ready, the operation is dispatched to a pipelined SIMD unit, which executes multiply-accumulate and multiply-reduce operations. The result is stored in the VRF. Since there are no explicit store instructions, a Write-back Manager unit periodically writes back the contents of dirty VRs to the memory subsystem.

This pipeline is designed for latency tolerance, maximizing the overlap of memory accesses with each other and with computation. First, the sparse front-end coalesces and overlaps read requests to sparse data. Second, the pipeline overlaps sparse data read requests issued in the front-end with requests issued in the subsequent two stages (dense data read and write requests and sparse data write requests). Finally, the back-end uses out-of-order execution, overlapping read and write requests with each other and with computation.

5 KEY ARCHITECTURE COMPONENTS

In this section, we provide more architectural details.

5.1 Detailed Pipeline Analysis

To understand the pipeline better, we explain it with an SpMM example. Figure 7 illustrates with numbers in circles the correspondence between the pipeline structures, the pipeline functionality and the memory accesses. It also shows the processing of the first tile of an input matrix in COO format.

Tile instruction ⓪. Figure 7 shows the three arrays (r_ids, c_ids and vals) of the sparse matrix. The tile instruction has the offset of the tile (*sparse_in_offset*) relative to the beginning of the arrays (given in an earlier initialization instruction) and the number of non-zeros (*NNZ_num*) of the tile. In our example, these values are 0 and 7. This information is enough to address all the tile non-zeros.

Loading non-zeros ①. The Sparse Data Loader generates cache-line sized requests for data from the 3 one-dimensional arrays. We assume that a cache line contains 4 elements. The figure shows the response for the first request. For as long as there are free entries in the Sparse Load Queue and the total tile *NNZ_num* has not been reached, a new request is issued every cycle. In our example, the next request would bring the next four elements of the arrays. However, because *NNZ_num* is 7, only three of the elements of the second request would be used. No more requests will be issued.

Determining dense addresses and generating tOps ②-③. Elements from the Sparse Load Queue are popped and tuples of the form (r_id, c_id, val) are isolated. In the figure, we show the first tuple, namely (0, 2, a). The r_id and c_id are used to index an rMatrix row and a cMatrix row, respectively. The base addresses of the dense matrices (which have been given in an earlier initialization instruction) are incremented by the r_id and c_id displacements. In our example, r_id is multiplied with the size of a dense row (K) and added to the rMatrix base to create the address range rMatrix[0,0:K]; the same is done with c_id to create cMatrix[2,0:K]. The tOp Generator module takes these two address ranges plus the non-zero value (a) and creates a Tuple Operation (tOp) (Figure 7).

Allocating vector registers and generating vOps ④. The core operation of SPADE is a SIMD operation between a sparse value and dense rMatrix/cMatrix rows. In order for the SIMD operations to take place, the rMatrix and cMatrix operands must be fetched from memory and stored in VRs. The pipeline is equipped with a Vector Register File (VRF) with a Vector Length (VL) equal to the system cache line size. However, a dense row may extend over multiple cache lines. For this reason, the tOp is broken down into finer-grained Vector Operations (vOps) whose operands are cache-line sized. In our example, we assume that the dense row size K extends over 2 cache lines (i.e., K=2*VL). Hence, 2 vOps are generated from each tOp; one for the first half of the dense rows and one for the second half of the dense rows.

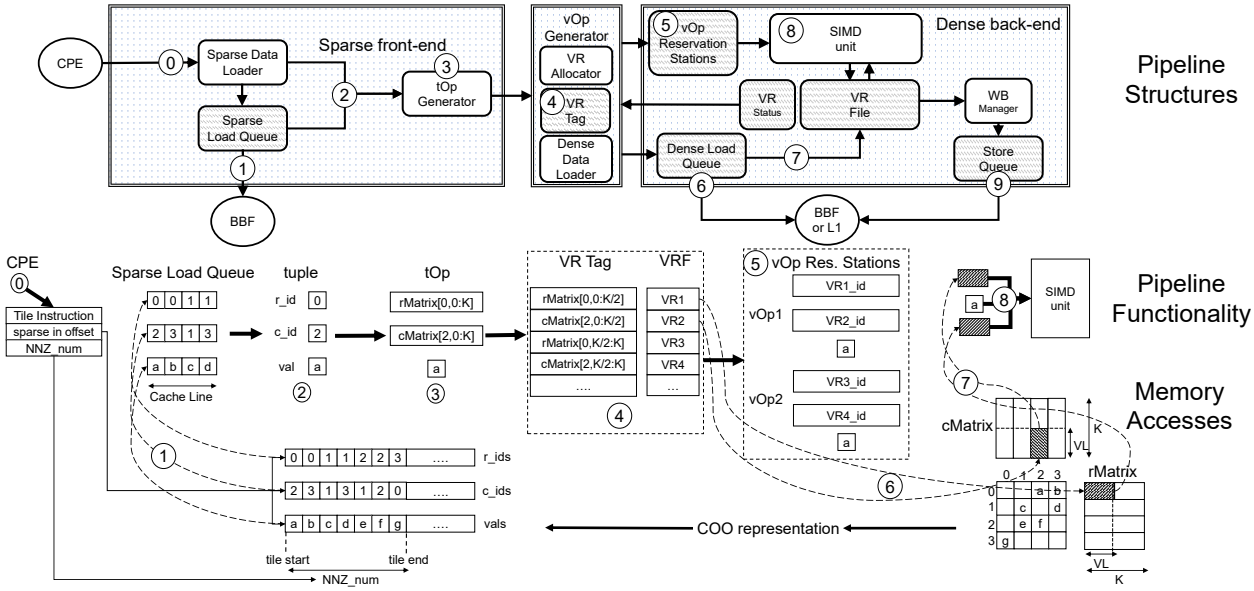


Figure 7: Pipeline structures, pipeline functionality, and memory access behavior.

The vOp Generator module has a VR Tag structure, which records the addresses of the cache-line-sized data stored in each VR. In our example, the vOp Generator module picks four empty registers and tags them with the four addresses of the two vOps. As shown in the figure, rMatrix[0,0:$K/2$], cMatrix[2,0:$K/2$], rMatrix[0,$K/2$:K], and cMatrix[2,$K/2$:K] are assigned to VR1, VR2, VR3, and VR4.

A desired vOp operand may already be in the register file from a previous operation. Consequently, before filling an entry in the VR Tag, the vOp Generator checks whether the address is already in the VR Tag. If so, the vOp Generator picks the corresponding VR for the vOp. The VR Tag structure is organized as a content-addressable memory (CAM).

There is also an auxiliary VR Status RAM that has additional status bits for each VR. They indicate if the entry is dirty or unused. They can be updated on a VR access and allocation/deallocation.

Pushing vOps into the vOp reservation stations ⑤. Once the VRs of vOps have been allocated, the vOps are pushed to slots in the vOp Reservation Stations for out-of-order execution. Figure 7 shows the two slots used by vOp1 and vOp2. A vOp carries additional information beyond the VR ids. It carries the input scalar value (a in our example) and, not shown in the figure, information that records: (i) inter-vOp dependencies and (ii) offsets of sub-cache line operands. These offsets are translated to masks to prevent overwriting unwanted cache-line contents.

Loading dense values ⑥. The Dense Load Queue issues requests for the operands of the vOps. These are cache-line requests for rMatrix and cMatrix data. If the VRs corresponding to these addresses already contain the data from a previous operation, no request is issued to memory.

Dense data arrival and execution ⑦-⑧. The vOps execute out of order. A vOp waits in a vOp reservation station if an operand has not yet arrived from memory or has a data dependence with an operand of an earlier vOp. Once the vOp receives all its operands and its dependencies are resolved, the vOp is dispatched for execution in the pipelined SIMD.

Data dependence tracking between vOps is simple for SpMM and SDDMM. Consider SpMM first. A vOp can only write to an rMatrix cache line and, furthermore, it must read the line first. Moreover, when the vOp Generator module accesses the VR Tag for a vOp, it marks that the selected VR will be both read and then written. As a result, the only data dependence that can exist between two different vOps is RAW—i.e., the second vOp will read the value that the first vOp will write. The dependence is detected and recorded when the vOp Generator accesses the VR Tag for the second vOp.

With this design, the pipeline may process vOps out of program order. This is not a problem because the accumulation operation of SpMM is associative. The same argument is true for SDDMM, except that the output is a single scalar rather than a vector.

Writing data back to memory ⑨. SPADE does not have explicit store instructions. vOps simply update the VRF. Hence, dirty VRs must be written back to memory in the background to free-up VRs to allocate new operands. Two extreme approaches are: (1) to write back a VR as soon as it gets dirty and (2) to write back VRs only when the VRF is all filled with dirty VRs. Unfortunately, the first approach causes an excessive number of stores to the memory subsystem, and the second one places writebacks in the critical path. Therefore, SPADE has a Write-back Manager unit in the pipeline that uses an intermediate approach: it initiates writebacks when the fraction of VRs that are dirty exceeds a certain threshold, and stops when the fraction falls below a second threshold.

5.2 Cache Bypassing

The ability of SPADE to provide a flexible cache hierarchy, with the capability to bypass the caches is important. The reason is that different input matrices can use the cache hierarchy very differently. To understand the choices, consider the different data structures in an SpMM and SDDMM operation.

Table 1: Microarchitecture of SPADE and its host CPU multicore system, modeled after a 2-socket Ice Lake with 56 cores total.

Parameter	Ice Lake Core	SPADE PE	Parameter	Ice Lake Core	SPADE PE	Parameter	Ice Lake Core	Ice Lake Total	SPADE PE	SPADE Total
Frequency	2.6/3.5GHz (base/turbo)	0.8GHz	**Physical vector registers**	224	64 (WB thres: 0.25, 0.15)	**L1D size**	48KB	2.625 MB	32KB	7.2MB
Issue width	5 uOp / cycle	1 vOp / cycle	**Load queue size**	128 entries	32 (dense) + 6 (sparse)	**L1I size**	32KB	1.75 MB	-	-
ROB size	352 entries	N/A	**Store queue size**	72 entries	8 entries	**BBF**	-	-	32 entries 64B/entry	0.45MB
SIMD FP units	3	1 (single precision)	**Phys. register allocation**	Register alias table	VR tag CAM	**Victim cache**	-	-	16KB 2-way 64B/entry	3.6MB
OoO execution support	160 scheduler entries	32 vOp res. station entries	**Frontend complexity**	High: L1I, branch predictor, decode, etc.	Low: Simple address calculation	**L2 size**	1.25 MB	70 MB	1.25 MB per 4 PEs	70 MB
Integer units	yes	no (hardwired address calculation)	**Frontend - Backend interface**	Allocation queue 140 entries	tOp queue 16 entries	**LLC size**	1.5 MB	84 MB	1.5 MB per 4 PEs	84 MB
Speculative execution	yes: shadow RAT etc.	no	**# of cores or PEs in 2 sockets**	56	224	**TOTAL DRAM BW**	max theoretical: 410GB/s		max. theoretical: 410GB/s; max. observed: 304GB/s	

For the input sparse matrix data, since each tuple is read once, used for a single tOp, and never reused again, it is best to bypass all caches and avoid polluting them.

When it comes to the rMatrix data, recall that individual rMatrix rows are reused only by a single PE to avoid data races. Since L2 caches and lower-level caches are shared by multiple PEs, one needs to be careful to cache rMatrix data, lest it pollute the caches for other PEs. Consequently, we identify three cases, depending on the pattern of non-zeros in the sparse input matrix and the size and order of tiles processed by a PE, as determined by barriers.

First, if the VRs have high reuse in the VRF, then caching the rMatrix does not offer significant benefits and, therefore, it is best to bypass the caches. Second, if the size and schedule of tiles is such that the distance between reusing rMatrix lines is very large, then VR reuse in the VRF will be low, and it is best to cache the lines, so they can still be in the caches when needed. The third case is like the second one except that the working set of reused rMatrix lines is small at any time, while the overall footprint of rMatrix lines is large. Caching all the rMatrix lines would pollute the caches for small gains. Hence, we augment the BBF with a small *Victim Cache*. The few rMatrix lines in the working set bypass the caches and use the victim cache in the BBF.

For the cMatrix data, caching is the best choice. The reason is that, inside a tile, data is processed in row order. As a result, the VRs in the VRF rarely have good reuse. Finally, for the output sparse matrix of SDDMM, cache bypass is most beneficial. The reasons are that bypassing eliminates cache pollution and, in addition, the output sparse matrix has high reuse in the VRF, as the vOps that access a given output cache line are bunched up.

6 METHODOLOGY

A. Architectures: Table 1 lists the architecture parameters of the simulated SPADE accelerator and its host CPU multicore. The latter is modeled after a dual-socket Intel Ice Lake server with 28 cores per socket. Its architectural details are taken from the literature [32, 74]. The SPADE PEs cycle at only 0.8GHz and, for hardware simplicity, only issue one tOp and vOp every cycle. A PE has 64 physical vector registers. As soon as 25% of them are dirty, the writeback manager starts writing them back until only 15% are dirty. Four PEs share the L2 of a CPU core. Neither L2 nor the LLC are partitioned. We evaluate SPADE with cycle-level simulations using SST [60]. To simulate DRAM, we integrate DRAMsim3 [41] in SST.

As baseline machines to compare to SPADE, we use a CPU, a GPU, and the state-of-the-art Sextans SpMM accelerator [64], which is an FPGA-based accelerator connected to its host through PCIe. The CPU baseline platform is the dual-socket Intel Ice Lake server described above. The GPU baseline platform is a server-class NVIDIA V100 connected to its host through PCIe, with 16GB of global memory and 900GB/s achievable global memory bandwidth. We simulate Sextans using SST and DRAMsim3. For fairness, we scale-up Sextans by using 16 PE groups (PEGs), each with 16 PEs clocked at 0.8 GHz. Each PE is equipped with 16 Processing Units (PUs). We scale-up the capacity of the on-chip scratchpads to 170MB (which is more than the total capacity of the SPADE cache subsystem). We additionally: (1) do not take into account AXI-related FPGA limitations, (2) disregard intra-PEG load imbalance, and (3) idealize the Sextans computation engine by only accounting for the time needed for memory accesses. Finally, we compress the sparse data so that each {row_id, col_id, val} tuple consumes 8B, and implement the address interleaving used by the authors. Although SPADE could also benefit from such optimizations, we do not employ them for SPADE.

B. Benchmarks: We use 10 large graphs from SparseSuite [17] coming from different domains (Table 2). We present our evaluation for two different dense row sizes (K=32 and K=128). We group matrices based on whether they benefit from tiling or scheduling barriers. If they always benefit from them, we say the matrices have a high *Restructuring Utility* (RU); if they sometimes benefit from them (e.g., in only one of SpMM or SDDMM, or only for K=128), we say they have a medium RU; for the remaining matrices, we say that they have a low RU. Table 2 shows the RU of the matrices.

C. Software kernels for the baseline machines: For the Ice Lake server, we use the Inspector-Executor (IE) kernel from Intel's MKL [70] library for SpMM. We believe that MKL IE already employs optimizations such as tiled execution. Since MKL does not include an SDDMM kernel, we use the SDDMM implementation from TACO [38] (similar to prior work [28]). TACO does not include an input-aware kernel. The kernels were compiled using the

Table 2: Benchmark graphs evaluated.

Graph (Short Name)	Domain	Nodes (Mill.)	Edges (Mill.)	Density (Order)	Restructuring Utility
asia_osm (ASI)	Road graph	11.9	25.4	10^{-7}	Low
com-LiveJournal (LIV)	Social network	4.0	69.4	10^{-6}	Medium
com-Orkut (ORK)	Social network	3.1	234.4	10^{-5}	High
coPapersCiteseer (PAP)	Citation graph	0.4	32.1	10^{-4}	Medium
delaunay_n24 (DEL)	Geometry problem	16.8	100.7	10^{-7}	Low
kron_g500-logn20 (KRO)	Synthetic graph	1.0	89.2	10^{-5}	High
mycielskian17 (MYC)	Mathematics (fractals)	0.1	100.2	10^{-2}	High
packing-500x100x100-b050 (PAC)	Numerical simulations	2.1	35.0	10^{-5}	Low
road_usa (ROA)	Highway graph	23.9	57.7	10^{-7}	Low
Serena (SER)	Environmental science	1.4	64.1	10^{-5}	Medium

2022.2.1 version of IntelOne's DPC++/C++ compiler. For the GPU, we use the SpMM kernel from Nvidia's cuSPARSE library [48], and the SDDMM kernel from dgSPARSE [30] using CUDA 11.6. We profiled both the transfer time and the kernel execution time.

In our baselines, we use the CSR format for high performance. In SPADE, we use the COO format. This is conservative, since SPADE could also benefit from CSR's reduced memory footprint.

D. Chip prototype: We prototyped in a chip a simplified SPADE design called *miniSPADE* and taped it out using TSMC 65nm technology. Each miniSPADE die (Figure 8) consists of 4 in-order SPADE PEs. Each PE is equipped with a bypass buffer (BBF) and an L1 cache. The die also includes a shared L2 cache and a memory buffer. At 65nm, the die size is 1.75mm x 1.00 mm, dominated mainly by SRAM for the caches and memory buffer. At 200MHz, the power consumption of the die is 30mW. Due to the simplified PE design and the large feature size, miniSPADE is not a fully representative depiction of the actual envisioned system. However, it serves as a proof-of-concept for various SPADE features (e.g., front-end, tOps, vOps, and cache bypassing).

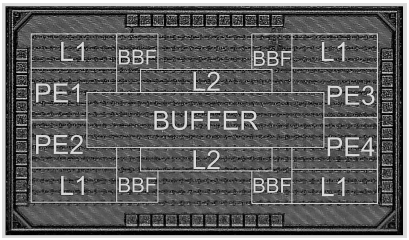


Figure 8: The miniSPADE die.

E. Area and power estimation: To estimate the area and power of SPADE, we model the L1D, BBF, victim cache, and all the pipeline memory structures (CAMs, RAMs, and registers) using CACTI [11] targeting 32nm. For the single-precision FP SIMD unit, we use the numbers from [20]. Our synthesis of miniSPADE suggests that additional logic (e.g., multiplexers and finite state machines) account for less than 5% of the miniSPADE pipeline area. Thus, for SPADE, we conservatively assume that the additional logic accounts for 20% of its pipeline area and power. Since Ice Lake uses 10nm technology, we scale both the power and area of SPADE down to 10nm by using the scaling factors from [66]. Finally, we use CACTI to estimate the L2 and LLC power consumption and DRAMsim3 for the DRAM.

7 EVALUATION

A. Performance comparison to CPU and GPU: We consider three SPADE variants: *SPADE Base*, *SPADE Opt*, and *SPADE2 Base*. The SPADE Base variant does not use any flexibility knobs: sparse matrix tiles always have a row panel size equal to 256 rows and a column panel size equal to the total number of columns in the matrix; the rMatrix and cMatrix data is accessed without bypassing caches; and there are no scheduling barriers.

To configure SPADE Opt, we test: (1) three different tile row panel sizes and three column panel sizes; (2) accesses to rMatrix that bypass or do not bypass the cache hierarchy; and (3) for the medium-sized column panels that we test, not inserting or inserting scheduling barriers to limit the LLC working set. For the small and large column panels, we do not use scheduling barriers. The parameters tested are shown in Table 3. For the MYC sparse matrix, which has a very small number of rows, we also test a row panel size of 16 to mitigate load imbalance. Although SPADE supports a large space of different parameter settings, we limit our analysis to these. Then, we set SPADE Opt to be, for each individual matrix, the version with the best-performing parameter settings that we tried. SPADE Opt is the proposed system variant in this paper.

Table 3: Parameters examined to set SPADE Opt.

Dense Row Sizes	Row Panel (RP) Sizes	Column Panel (CP) Sizes	rMatrix Caching Strategies	Scheduling Barriers
K=32	64, 256, 1024	8192, 524288, all_columns	No bypass, bypass	No, yes for CP=524288
K=128	64, 256, 1024	2048, 131072, all_columns	No bypass, bypass	No, yes for CP=131072

SPADE2 Base is a scaled-up version of SPADE Base with double PE count, DRAM bandwidth, LLC size, and link latencies.

Figure 9 shows the speed-up of the server-class GPU (*assuming zero host-device data transfer overhead*), SPADE Base, SPADE Opt, and SPADE2 Base over the baseline CPU server. For matrices that do not fit in the GPU memory (i.e., DEL and ROA for K=128) we assume a GPU speedup over the CPU of 1.

As shown in Figure 9, the speedups of SPADE Base, and the effectiveness of SPADE's flexibility as demonstrated by SPADE Opt, vary for each matrix category. First, for *low RU* matrices, the SPADE Base speedups are lower. This is because matrices in this category show limited reuse opportunities and, therefore, the SPADE Base speedup mainly stems from better utilization of the available memory bandwidth. In addition, we see that the benefit of SPADE Opt's flexibility is smaller. The reason is that SPADE Base with very large tiles is already a good fit for this category. Furthermore, without accounting for the host-device data transfer, the GPU is faster than SPADE Opt, due to having an achievable memory bandwidth (900GB/s) that is much higher than the maximum observed bandwidth of SPADE Opt (304GB/s). Although SPADE2 Base only has a bandwidth of 600GB/s, it is competitive with the GPU baseline.

For *high* and *medium RU* matrices, both the SPADE Base speedup and the flexibility benefits of SPADE Opt are higher. A sizable portion of the requests are served from caches with low-latency and without introducing DRAM bandwidth pressure. The SPADE PEs can utilize this pattern, by being able to satisfy a larger number of requests from the caches while long-latency main memory ones are pending, hence hiding the latency of the latter. By using the flexibility knobs, SPADE Opt outperforms the GPU for most matrices in these categories. The speedups of SPADE2 Base are higher.

Overall, on average across all the environments shown in Figure 9, SPADE Base, SPADE Opt, and SPADE2 Base are 1.67x, 2.32x,

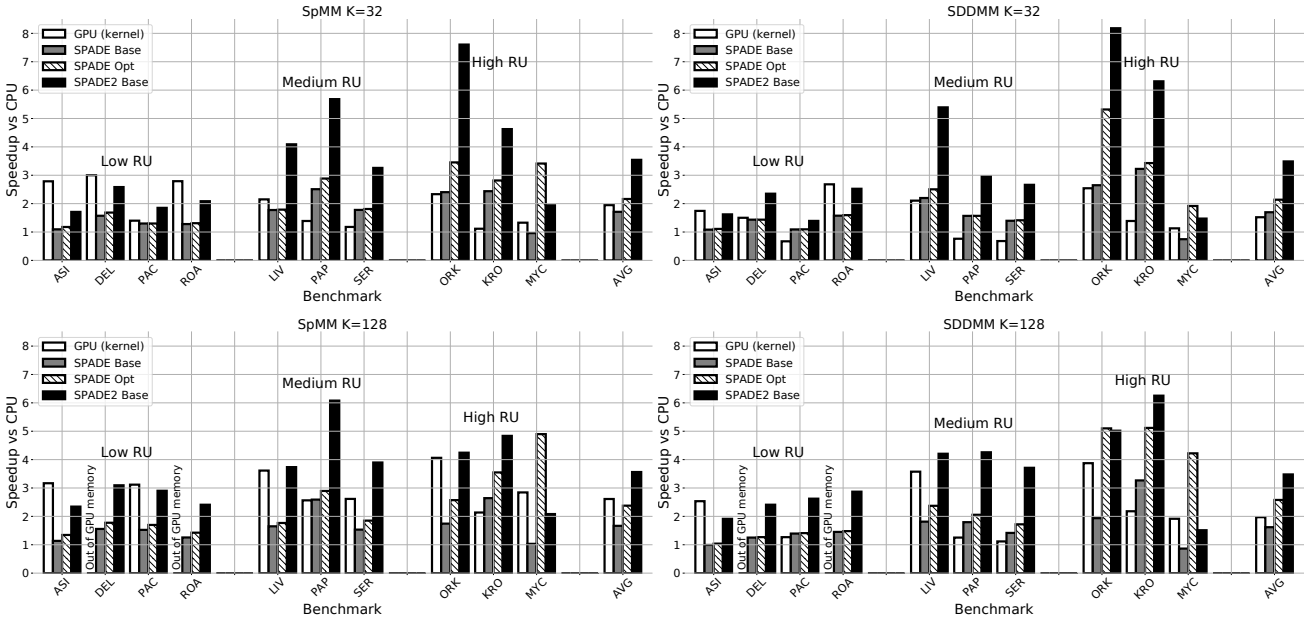


Figure 9: Speedup of the SPADE variants and the GPU (ignoring data transfers) over the CPU. Matrices are grouped into low, medium, and high Restructuring Utility (RU).

and 3.52x faster, respectively, than the CPU. The corresponding speedups over the GPU are 1.03x, 1.34x, and 2.00x. The performance gap between SPADE Opt and SPADE Base reveals that flexibility is a key feature for an SpMM/SDDMM accelerator. Further, the performance of SPADE2 Base suggests that our system is scalable. Our evaluation reveals that SPADE is a superior architecture in comparison to GPUs *even if host-to-device data transfers and address remapping overheads are ignored*. If these overheads are taken into account (Figure 2), SPADE Opt is 43.4x faster than the GPU.

B. Pipeline analysis: To provide further insight into the effect of some of the system optimizations, we perform an optimization analysis for SpMM with K=32. We evaluate 5 different SPADE system configurations by progressively adding system features. Table 4 presents the configurations.

Table 4: Evaluated SPADE configurations.

Config.	SPADE features included
CFG0	Tile instructions, 3-entry sparse load queue, overlap of sparse and dense requests, 16 vOps RS entries, 56 SPADE PEs at 3.2GHz
CFG1	CFG0 with 32 vOps RS entries
CFG2	CFG1 with 224 SPADE PEs at 0.8GHz
CFG3	CFG2 with 6-entry sparse load queue
CFG4	CFG3 + sparse data bypasses the cache hierarchy (≡ SPADE Base)
CFG5	CFG4 + flexible execution (≡ SPADE Opt)

Since SPADE is built for high latency tolerance, we evaluate each of the SPADE configurations for three different link latencies. Specifically, we consider the average round-trip latency of an access from the PE to a memory controller, without including the latencies to access the memory or any of the caches on the way. We call this time the link latency (LL). We set this LL to 60ns (default for our SPADE architecture), 480ns, and 960ns. For CFG5, we only evaluate LL=60ns to reduce the number of experiments required. For each configuration and LL value, Figure 10 shows the number of DRAM accesses, the number of LLC accesses, the average number of requests that the pipelines collectively issue per cycle, and the execution time. The figure shows the geometric mean of the metrics across all the matrices, normalized to CFG0 for 60ns LL.

Compare the different configurations for a given LL. A high number of requests per cycle indicates active pipelines. It can mean either enhanced latency tolerance or decreased latency due to better utilization of the caches. If an increase in requests per cycle is not accompanied by a decrease in DRAM and/or LLC accesses, it is an indicator of increased latency tolerance.

From the figure, we see that the optimizations progressively applied in CFG1, CFG2, and CFG3 result in an increased number of requests per cycle being generated *without the LLC or DRAM accesses decreasing*. Hence, they increase latency tolerance. CFG4 and CFG5 increase the number of requests per cycle while decreasing the LLC and DRAM accesses, and thus decrease the average request latency. We also see that the reduction in execution time induced by the progressive optimizations increases as LL increases.

C. Evaluation of flexibility knobs: We now show how the different flexibility knobs of SPADE benefit different sparse matrices. Recall that the knobs are: different tile sizes, cache bypassing or not, and scheduling barriers or not. First, we perform a sensitivity analysis of how the tile row panel (RP) and column panel (CP) sizes affect the execution time. We disable bypassing and insert no barrier. Figure 11 shows the results for three matrices. It shows the execution times normalized to the execution time of the worst-performing setting. Thus, lower values and darker colors are better. We observe that KRO, which belongs in the High RU category, benefits from tiles with a small CP and a large RP since, in this way, cMatrix reuse opportunities can be maximized. On the other hand, DEL, which has low RU, benefits from tiles with a CP that extends over all of the columns. Finally, for MYC, which has a small number of rows, the selection of small RPs mitigates load imbalance.

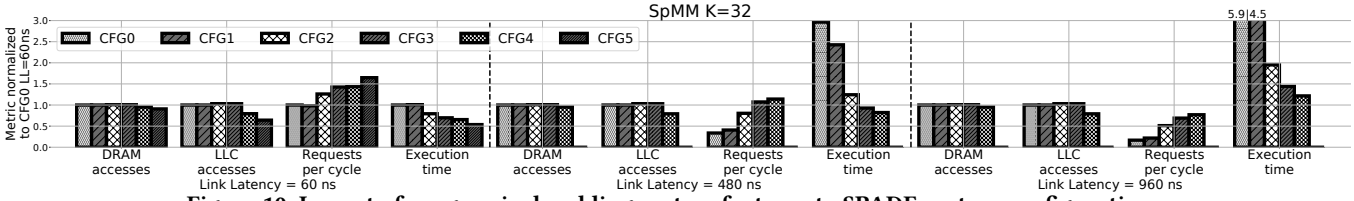
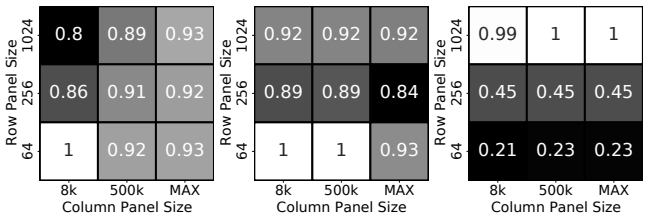


Figure 10: Impact of progressively adding system features to SPADE system configurations.



(a) KRO **(b) DEL** **(c) MYC**

Figure 11: Execution time of SpMM with K=32 for different tile column/row panel sizes, normalized to the worst parameter setting. Lower numbers and darker colors are better.

Table 5 examines the impact of applying scheduling barriers. It shows the percentage change in execution time of the setting with medium RP and CP sizes and no cache bypassing when we apply barriers. Positive numbers are slowdowns. We observe that the effect of barriers is matrix-dependent. It can either increase the execution time (by up to 80.5% in ASI SpMM with K=128) or reduce it (by up to 57.1% in ORK SpMM with K=128).

Table 5: Percentage change in execution time by applying scheduling barriers. Positive numbers are slowdowns.

Algorithm & K value	Low RU				Medium RU			High RU		
	ASI	DEL	PAC	ROA	LIV	PAP	SER	ORK	KRO	MYC
SpMM32	22.8	41.1	4.3	31.7	24.2	0	1	-29.6	-12.6	0
SDDMM32	9	6.5	0.6	5	-15	0	0.6	-48.3	-1.2	0
SpMM128	**80.5**	76.6	12.8	56.2	34	47.7	9.4	**-57.1**	-37	0
SDDMM128	19.6	37.8	8.2	22.9	-16.9	13.2	-3.8	-53.7	-36.9	0

Table 6 examines the impact of applying cache bypassing for the rMatrix. For each benchmark, we pick the setting of tile RP and CP sizes and scheduling barriers that delivered the best performance. On top of this setting, we apply cache bypassing. The table shows the percentage change in execution time when we apply cache bypassing. Positive numbers are slowdowns. We observe that, for most of the benchmarks, rMatrix cache bypassing is beneficial. It can reduce the execution time by up to 32.9% in ORK SpMM with K=128. However, it can also slow down the execution. For example, in KRO SpMM with K=32, cache bypassing leads to a 169.2% increase in execution time. Recall from Figure 11 that KRO benefits from a large row panel. Unfortunately, this row panel size overflows the victim cache in the BBF, leading to many main memory spills.

Table 6: Percentage change in execution time by bypassing the caches for the rMatrix. Positive numbers are slowdowns.

Algorithm & K value	Low RU				Medium RU			High RU		
	ASI	DEL	PAC	ROA	LIV	PAP	SER	ORK	KRO	MYC
SpMM32	-7	-6.7	4.2	-2.3	-0.8	13.9	-1.8	-17.2	**169.2**	0.7
SDDMM32	-3.2	0.1	-0.3	-1.2	-6.8	0.2	-0.1	-7	0.2	0.1
SpMM128	-9.9	-10.4	-9	-12.2	-0.7	-7.1	-14.4	**-32.9**	-6.3	-1.9
SDDMM128	-4.5	-0.3	-1.1	-1.9	-13.2	-6.3	-12.8	-18.7	-8.4	-2.7

Overall, the input-dependent impact of tiling, scheduling barriers, and cache bypassing justifies our emphasis on flexibility.

D. Overhead of mode transitions: When execution repeatedly interleaves CPU and SPADE mode sections, it will repeatedly suffer mode transition overheads. We have measured such overheads for our benchmarks. The transition from SPADE to CPU mode involves writing back the PEs' L1 caches to the L2s, writing back the BBFs and victim caches to memory, and invalidating L1s, BBFs, and victim caches. For our benchmarks, this overhead is on average 0.2% of the SPADE mode duration.

The overhead of the transition from CPU to SPADE mode is generally application dependent. We assume an execution that repeatedly interleaves CPU sections and SPADE PEs executing SpMM, or CPU sections and SPADE PEs executing SDDMM. We additionally assume that the CPU updates a dense matrix that will be an input for SPADE (i.e., cMatrix for SpMM and rMatrix or cMatrix for SDDMM); this is a reasonable assumption for social network GNN applications. For SpMM, all that is needed is to write back and invalidate the CPU's L1 caches. There is no action to take for the two structures that SPADE PEs will access through the BBFs (rMatrix and input sparse matrix): the rMatrix is not accessed by the CPU and the input sparse matrix is only read. For SDDMM, the transition needs to write back and invalidate the CPU's L1 caches and write back and invalidate the rMatrix from the caches. This is because, of the three structures that SPADE PEs access through the BBFs (the rMatrix and the input and output sparse matrices), only the rMatrix is a potential concern: the sparse output matrix is not accessed by the CPU and the input sparse matrix is only read. For our benchmarks, these transition overheads are negligible for SpMM and on average 3.4% of the SPADE mode duration for SDDMM. Overall, transition overheads are small.

In all of our experiments, when computing the execution time of an application on SPADE, we have included two overheads: the start-up overhead due to caches starting empty rather than warmed-up and the termination overhead, which is the above overhead of transitioning from SPADE to CPU mode. For our benchmarks, the start-up overhead is on average 0.9% of the SPADE mode duration.

E. Scalability: We perform a strong scaling analysis of SPADE. Figure 12 shows the speedup of different SPADE system sizes over the baseline 224-PE SPADE (as described in Table 1), which has the default link latency of 60ns. SPADE2 Base, SPADE4 Base, and SPADE8 Base are scaled-up versions with 2x, 4x, and 8x the PE count, DRAM bandwidth, LLC size, and link latency. For reference, the figure also shows bars with linear scaling.

SPADE scales well in most benchmarks. The exceptions are MYC and KRO, which have few sparse matrix rows; load imbalance hinders their strong scaling. Superlinear speed-up is observed in some cases due to the increase in LLC size. We repeated this experiment for K=128 and for SDDMM, and observed similar results.

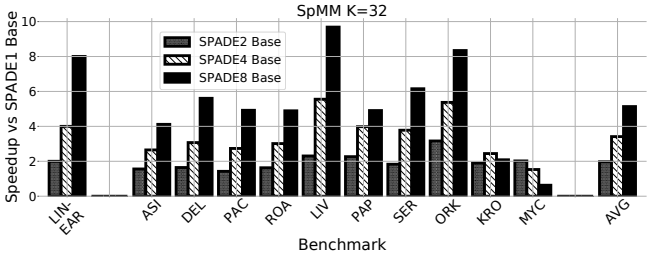


Figure 12: Scaling analysis of SPADE.

F. Performance comparison to an ideal Sextans accelerator: Sextans [64] is a high-performance FPGA-based accelerator for SpMM. Like other accelerators, it does not address the cost of host–device data transfers. It adopts a one-size-fits-all execution model based on data streaming and sequentially-batched phases, which has some similarities with applying scheduling barriers in SPADE. Its execution model suffers from multiple reads to sparse data when the dense row size K increases. It can also lead to multiple accesses to the dense input for large matrices (e.g., ROA, DEL and ASI) when the dense output does not fit on the Sextans scratchpad.

We simulate Sextans and compare it to SPADE. As discussed in Section 6.A, we idealize many parts of the Sextans architecture in our simulation and disregard the host data transfer cost. For these reasons we call it *ideal* Sextans. Note that by removing the FPGA-related limitations, our Sextans version achieves a maximum memory bandwidth utilization of 50%, which is significantly higher than the 15% reported in [64].

Figure 13 shows the bandwidth utilization, DRAM accesses, and speedup of SPADE Opt, all normalized to Sextans (for SpMM K=32). SPADE Opt issues many concurrent requests for both sparse and dense data, across different matrix regions. This results in a 40% higher average bandwidth utilization. In addition, by supporting different parameter settings on a per-matrix basis, SPADE eliminates redundant memory accesses, issuing 32% fewer memory accesses (up to 73% for ROA). Moreover, thanks to SPADE Opt's decoupled and flexible design, SPADE Opt is much faster than ideal Sextans. Its average speedup is a significant 2.4x (and a maximum of 5.1x). Ideal Sextans performs marginally better only for ORK and LIV (10% and 3%). Recall from Table 5 that these matrices benefit from barriers, which resemble the Sextans execution strategy.

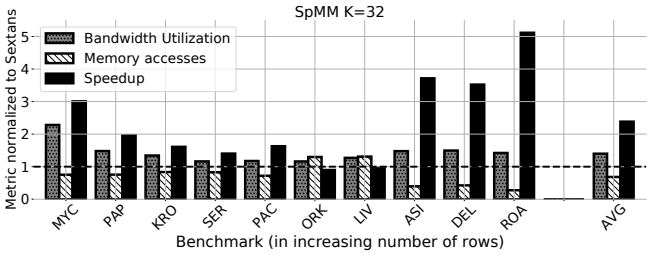


Figure 13: Bandwidth utilization, DRAM accesses, and speedup of SPADE Opt, all normalized to ideal Sextans.

For K=128, it can be shown that SPADE Opt's average speedup is 2.6x. Sextans does not support SDDMM. Finally, including the PCIe data transfer overhead for Sextans, the average SPADE Opt speedup is 52.4x (for a single SpMM iteration).

G. Area and power evaluation: Augmenting a dual-socket Ice Lake [32] with the SPADE PEs, L1 caches, BBFs, and victim caches adds a very small area and power overhead. This is because, compared to the 5-issue speculative Xeon core, a SPADE PE has a simple, single-issue pipeline (Table 1). Further, the PE L1 and victim cache have a single port and low associativity. Using the tools of Section 6.E, we estimate that the 224 SPADE PEs with their L1s, BBFs, and victim caches consume 20.3W and take $24.64mm^2$ at 10nm. Compared to a dual-socket Ice Lake server that has a TDP of 470W and a combined die size of 1000 mm^2 [62], the SPADE power is 4.3% of the host TDP, and its area is 2.5% of the host area.

In SPADE-mode execution, the server disables the Xeon cores and L1 caches, and the SPADE PEs use the server's memory subsystem. In this environment, Figure 14 breaks down the power consumed into the fraction spent by the 224 SPADE PEs, L1s, BBFs, and victim caches; by the L2 caches; by the LLC; and by the DRAM. We conservatively assume that the PE pipelines operate with maximum dynamic power. We see that the SPADE PEs, L1s, BBFs, and victim caches consume on average only 14% of the total power. The power dissipated in the lower levels of the cache hierarchy is also low because the sparse matrix and (sometimes) the rMatrix bypass the caches. Finally, DRAM power accounts for more than 50% of the total power.

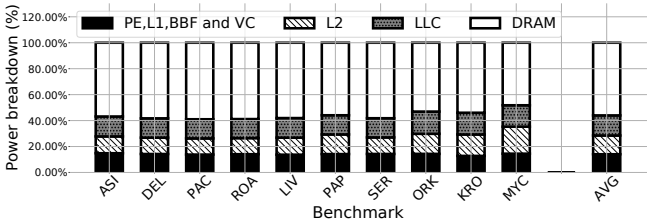


Figure 14: Breakdown of the power consumed in SPADE-mode execution for SpMM and K=32.

8 RELATED WORK

A. SpMM/SDDMM accelerators: Several accelerators have been proposed for either both or one of SpMM and SDDMM [28, 64, 65]. Extensor [28] proposes a technique to eliminate redundant computation in sparse algebra. This technique only yields significant benefits in kernels where more than one matrix is sparse (e.g., Sparse Matrix Sparse Matrix multiplication). For SpMM and SDDMM, Extensor's speedups are lower and they mainly stem from placing specialized units closer to memory. Tensaurus [65] relies on a custom compression format to ensure streaming accesses of sparse data, a specialized PE array, and HBM. Sextans [64] was described in Section 7.F. Recently, the scalability of SpMM was investigated in PIUMA, which is an architecture designed by Intel for graph analytics at scale [1, 2]. Thanks to the huge capacity of the PIUMA memory pool, the number of data moves from host CPU to PIUMA is minimized.

These prior works do not provide a solution for fine-grained interleaving between host and accelerator phases. For example, Tensaurus and Sextans assume that the matrices already reside in the accelerator's HBM and do not account for the low bandwidth of host-device data transfers. In addition, none of these works offers knobs for flexible execution catered to the unique characteristics of SpMM and SDDMM .

B. GNN and sparse Deep Neural Network (DNN) accelerators: There are several accelerator and software optimization proposals

for accelerating GNNs [8, 21, 22, 24, 31, 36, 40, 43, 45, 59, 71–73, 75]. They often focus on overlapping the memory intensive SpMM with compute-intensive dense matrix multiplications. However, these approaches are not applicable to accelerate SpMM and SDDMM as stand-alone primitives. Accelerators [27, 54, 58] and CPU extensions such as SAVE [23] and VEGETA [34] have been proposed to increase the utilization of the compute units in sparse DNN applications. Although these approaches are effective for the low sparsity levels of DNNs (10%-90%), for the high sparsity levels of real world graphs (Table 2), the memory subsystem becomes the bottleneck. In order to accommodate such sparsity levels, SPADE interacts with the memory subsystem in an efficient and flexible manner.

C. Eliminating host-accelerator transfer overhead: In some architectures such as the Qualcom Snapdragon [12], the GPU and CPU are integrated in the same SoC and share the datapath and memory. Although this approach may be viable for mobile devices or desktops, integrating a server-class CPU (e.g., a $500mm^2$ Ice Lake) with a server-class GPU (e.g., a $815mm^2$ V100 [51]) in the same die is currently impractical. Instead, combining the small area footprint of SPADE with the reuse of the area-hungry lower levels of the CPU caches is a more attractive alternative. Recently, NVIDIA introduced Grace/Hopper [52], where GPUs are able to access the host CPU main memory through the high-bandwidth NVLink interconnect. Although such high-bandwidth interconnects can alleviate much of the CPU-GPU data transfer penalty [18], it is still unknown whether this new approach will completely eliminate the inefficiencies of prior CPU-GPU virtual address space sharing methods [4, 69]. With SPADE, we show a way to eliminate both overheads that does not require precious high-performance network resources. Finally, our evaluation reveals that even in the ideal case when both data transfer and address remapping overheads are eliminated in GPUs and accelerators, SPADE still outperforms both GPUs and accelerators (Figures 9 and 13). Recent trends towards built-in HBM in high-end CPU servers [63] are only going to increase the performance benefits of SPADE.

D. Flexible accelerators: Motivated by the importance of flexibility in graph and sparse linear algebra algorithms, researchers have developed accelerators that can be reconfigured to support different algorithm variants or dataflows [16, 19, 42, 47], compression formats [57], data-dependent execution patterns [7], or that use heterogeneity [56] as a flexibility enabler. Transmuter [53] is a CGRA-based architecture that reconfigures its memory type and dataflow to accommodate kernels of varying arithmetic intensity. SPADE supports flexibility through a tile-based ISA, providing flexibility knobs tailored to the unique characteristics of SpMM/SDDMM.

E. Flexible execution strategies in CPU/GPU systems: Input-aware techniques such as enhancing locality through reordering [6, 10, 29, 46], improving load-balancing [30] through row partitioning, or selecting the appropriate execution strategy based on machine learning [76, 77] improve SpMM and SDDMM performance in CPUs and GPUs. These techniques are orthogonal to SPADE.

9 CONCLUSION AND FUTURE WORK

This paper proposed *SPADE*, a hardware accelerator for SpMM and SDDMM. SPADE avoids data transfer overheads by tightly-coupling accelerator PEs with the cores of a multicore, as if they were advanced functional units—allowing the accelerator PEs to reuse the cores' memory system and virtual addresses. SPADE attains flexibility and programmability by supporting a *tile-based* ISA. Simulations of SPADE with 224-1792 PEs showed its high performance and scalability. A 224-PE SPADE system is on average 2.3x, 1.3x, and 2.5x faster than a 56-core CPU, a server-class GPU, and a state-of-the-art SpMM accelerator, respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average 43.4x and 52.4x faster than the GPU and the accelerator, respectively. Further, SPADE has a small area and power footprint

Our future work involves distributing SPADE acceleration across the nodes of a large distributed cluster, and augmenting SPADE to support more operations. SPADE can already support Sparse Matrix Vector Multiplication (SpMV) and Sampled Dense Vector-Dense Vector Multiplication (SDDVV). We believe that, with lightweight modifications, it can support a richer set of primitives (e.g., TTV, TTM, and MTTKRP [38]).

APPENDIX A: A TILED SPARSE MATRIX

This section shows the tiled sparse matrix representation that we use. Figure 15(a) shows a 4x4 sparse matrix, represented with the three conventional arrays in COO format. Figure 15(b) shows the representation of the matrix when tiled with a row panel size and a column panel size equal to two. The entries of the r_ids, c_ids, and vals arrays have been reordered, consolidating the per-tile entries together. In addition, we have the tiling metadata: offsets of the beginning of each tile (*sparse_in_start_offset*); number of non-zeros in each tile (tile_NNZ_num); for SDDMM only, the offsets of the beginning of each tile in the output sparse matrix (*sparse_out_start_offset*, necessary because output tiles have to be aligned to cache lines); and which row panels each tile belongs to (*tile_row_panel_id*), needed so that all the tiles in a panel row are executed by the same PE to avoid data races in SpMM.

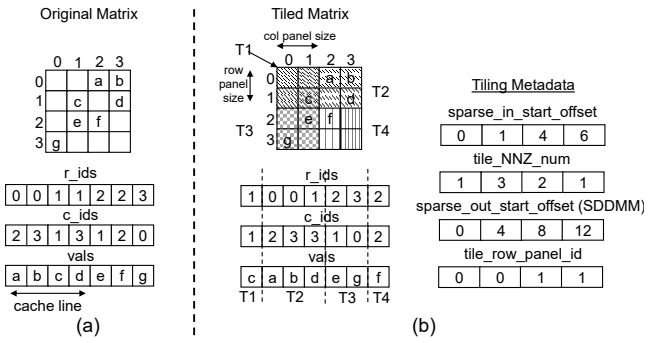


Figure 15: Representation of a tiled sparse matrix.

ACKNOWLEDGMENTS

This work was supported by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by the IBM-Illinois Discovery Accelerator Institute; and by NSF under CNS 1956007 and CCF 2107470. We acknowledge Apple for supporting the miniSPADE chip tapeout. We thank the anonymous ISCA reviewers, whose valuable feedback significantly improved the manuscript.

REFERENCES

[1] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, MidhunChandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. 2020. PIUMA: Programmable Integrated Unified Memory Architecture. arXiv:2010.06277 [cs.AR]

[2] Matthew Adiletta, Jesmin Jahan Tithi, Emmanouil-Ioannis Farsarakis, Gerasimos Gerogiannis, Robert Adolf, Robert Benke, Sidharth Kashyap, Samuel Hsia, Kartik Lakhotia, Fabrizio Petrini, Gu-Yeon Wei, and David Brooks. 2023. Characterizing the Scalability of Graph Convolutional Networks on Intel® PIUMA. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Raleigh, North Carolina.

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1213–1222.

[4] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[5] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In *SpringSim (HPS)*. 75–82.

[6] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[7] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. ALRESCHA: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 249–260.

[8] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[9] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. 2000. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM.

[10] Vignesh Balaji and Brandon Lucia. 2018. When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 203–214.

[11] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[12] Anthony Cabrera, Seth Hitefield, Jungwon Kim, Seyong Lee, Narasinga Rao Miniskar, and Jeffrey S Vetter. 2021. Toward performance portable programming for heterogeneous systems on a chip: A case study with Qualcomm Snapdragon SoC. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[13] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 2015. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 233–242.

[14] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[15] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. 2020. Adaptive universal generalized pagerank graph neural network. *arXiv preprint arXiv:2006.07988* (2020).

[16] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 595–608.

[17] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[18] Anne C Elster and Tor A Haugdahl. 2022. Nvidia Hopper GPU and Grace CPU highlights. *Computing in Science & Engineering* 24, 2 (2022), 95–100.

[19] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. 2021. CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 949–954. https://doi.org/10.1109/DAC18074.2021.9586114

[20] Sameh Galal and Mark Horowitz. 2010. Energy-efficient floating-point unit design. *IEEE Transactions on computers* 60, 7 (2010), 913–922.

[21] Raveesh Garg, Eric Qin, Francisco Muñoz-Martínez, Robert Guirado, Akshay Jain, Sergi Abadal, José L. Abellán, Manuel E. Acacio, Eduard Alarcón, Sivasankaran Rajamanickam, and Tushar Krishna. 2022. Understanding the Design-Space of Sparse/Dense Multiphase GNN Dataflows on Spatial Accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 571–582. https://doi.org/10.1109/IPDPS53621.2022.00062

[22] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. 2020. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 922–936. https://doi.org/10.1109/MICRO50266.2020.00079

[23] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 796–810.

[24] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: Optimizing Graph Neural Networks on CPUs through Cooperative Software-Hardware Techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 916–931. https://doi.org/10.1145/3470496.3527403

[25] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[26] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[27] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM international conference on supercomputing*. 1–12.

[28] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[29] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[30] Guyue Huang, Guohao Dai, Yu Wang, Yufei Ding, and Yuan Xie. 2021. Efficient Sparse Matrix Kernels Based on Adaptive Workload-Balancing and Parallel-Reduction. arXiv:2106.16064 [cs.DC]

[31] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/3437801.3441585

[32] Intel. 2021. Intel Xeon Gold 6348 Processor 42M Cache 2.60 GHz Product Specifications. https://www.intel.com/content/www/us/en/products/sku/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz/specifications.html

[33] Intel. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[34] Geonhwa Jeong, Sana Damani, Abhimanyu Rajeshkumar Bambhaniya, Eric Qin, Christopher J Hughes, Sreenivas Subramoney, Hyesoon Kim, and Tushar Krishna. 2023. VEGETA: Vertically-Integrated Extensions for Sparse/Dense GEMM Tile Acceleration on CPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 259–272.

[35] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. A high-performance parallel algorithm for nonnegative matrix factorization. *ACM SIGPLAN Notices* 51, 8 (2016), 1–11.

[36] Kevin Kiningham, Philip Levis, and Christopher Ré. 2022. GRIP: A graph neural network accelerator architecture. *IEEE Trans. Comput.* (2022).

[37] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[38] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[39] Andrew V Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing* 23, 2 (2001), 517–541.

[40] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. 2021. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 775–788.

[41] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[42] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[43] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. 2020. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Comput.* (2020).

[44] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. 2022. HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 123–133.

[45] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.

[46] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J Sorin, Benjamin C Lee, and Mike O'Connor. 2021. Learning sparse matrix row permutations for efficient SpMM on GPU architectures. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 48–58.

[47] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[48] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. cuSPARSE library. In *GPU Technology Conference*.

[49] Michael K Ng and Zhaochen Zhu. 2019. Sparse matrix computation for air quality forecast data assimilation. *Numerical Algorithms* 80, 3 (2019), 687–707.

[50] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P Sadayappan. 2018. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 32–41.

[51] NVIDIA. 2017. NVIDIA Tesla V100 whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[52] NVIDIA. 2022. NVIDIA Grace Hopper superchip architecture whitepaper. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper

[53] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael O'Boyle, Murray Cole, Chaitali Chakrabarti, David Blaauw, Hun-Seok Kim, Trevor Mudge, and Ronald Dreslinski. 2020. Transmuter: Bridging the Efficiency Gap Using Memory and Dataflow Reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 175–190. https://doi.org/10.1145/3410463.3414627

[54] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news* 45, 2 (2017).

[55] X. Pinel and M. Montagnac. 2013. Block Krylov methods to solve adjoint problems in aerodynamic design optimization. *AIAA journal* 51, 9 (2013), 2183–2191.

[56] Eric Qin, Raveesh Garg, Abhimanyu Bambhaniya, Michael Pellauer, Angshuman Parashar, Sivasankaran Rajamanickam, Cong Hao, and Tushar Krishna. 2022. Enabling Flexibility for Sparse Tensor Acceleration via Heterogeneity. *arXiv preprint arXiv:2201.08916* (2022).

[57] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E Moon, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Extending sparse tensor accelerators to support multiple compression formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1014–1024.

[58] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[59] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[60] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (mar 2011), 37–42. https://doi.org/10.1145/1964218.1964225

[61] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2015. Regularizing graph centrality computations. *J. Parallel and Distrib. Comput.* 76 (2015), 106–119.

[62] David Schor. 2021. Intel launches 3rd Gen Ice Lake Xeon Scalable. https://fuse.wikichip.org/news/4734/intel-launches-3rd-gen-ice-lake-xeon-scalable/

[63] Ryan Smith. 2022. Intel showcases Sapphire Rapids Plus HBM Xeon Performance at ISC 2022. https://www.anandtech.com/show/17422/intel-showcases-sapphire-rapids-plus-hbm-xeon-performance-isc-2022

[64] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[65] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[66] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[67] Narayanan Sundaram and Kurt Keutzer. 2011. Long term video segmentation through pixel level spectral clustering on GPUs. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. IEEE, 475–482.

[68] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[69] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 161–171.

[70] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[71] Minjie Yu Wang. 2019. Deep Graph Library: towards efficient and scalable deep learning on graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://par.nsf.gov/biblio/10311680

[72] Yuke Wang, Boyuan Feng, and Yufei Ding. 2021. TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs. *arXiv preprint arXiv:2112.02052* (2021).

[73] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[74] WikiChip. 2023. Sunny Cove - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[75] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–29.

[76] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 329–341.

[77] Serif Yesil, José E. Moreira, and Josep Torrellas. 2022. Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/3524059.3532369

[78] Bingyi Zhang, Sanmukh R Kuppannagari, Rajgopal Kannan, and Viktor Prasanna. 2021. Efficient neighbor-sampling-based GNN training on CPU-FPGA heterogeneous platform. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[79] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[80] Huasha Zhao, Biye Jiang, John F Canny, and Bobby Jaros. 2015. SAME but Different: Fast and High Quality Gibbs Parameter Estimation. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1495–1502.

[81] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid CPU and GPU training for Graph Neural Networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4582–4591.