Programmable Acceleration for Sparse Matrices in a Data-movement Limited World

Arjun Rawal
Computer Science Department
University of Chicago
arjunrawal4@cs.uchicago.edu

Yuanwei Fang
Computer Science Department
University of Chicago
fywkevin@cs.uchicago.edu

Andrew A. Chien
Computer Science Department
University of Chicago &
Argonne Natl Lab
achien@cs.uchicago.edu

***Abstract*—Data movement cost is a critical performance concern in today's computing systems. We propose a heterogeneous architecture that combines a CPU core with an efficient data recoding accelerator and evaluate it on sparse matrix computation. Such computations underly a wide range of important computations such as partial differential equation solvers, sequence alignment, and machine learning and are often data movement limited. The data recoding accelerator is orders of magnitude more energy efficient than a conventional CPU for recoding, allowing sparse matrix representation to be optimized for data movement.**

We evaluate the heterogeneous system with a recoding accelerator using the TAMU sparse matrix library, studying $>$369 diverse sparse matrix examples finding geometric mean performance benefits of 2.4x. In contrast, CPU's exhibit poor recoding performance (up to 30x worse), making data representation optimization infeasible. Holding SpMV performance constant, adding the recoding optimization and accelerator can produce power reductions of 63% and 51% on DDR and HBM-based memory systems, respectively, when evaluated on a set of 7 representative matrices. These results show the promise of this new heterogeneous architecture approach.

***Index Terms*—heterogeneous architecture, accelerator, memory bandwidth wall, sparse matrix multiplication**

I. Introduction

Sparse matrices and computation are important in a wide range of computations, and represent efficient algorithms. For example, they are widely used in partial-differential equation solvers in scientific computing. Graph computations – of interest for social networks, web structure analysis, and recently machine learning – are also often realized as sparse matrix computations. Deep learning is an area of growing excitement, and both training and inference computations typically involve large sparse matrix computations.

For several decades, the memory system (DRAM access) has been the critical bottleneck for sparse matrix computation performance [26], [28], [45]. This is not because DRAM interfaces could not be scaled up with parallelism, but rather because of the significant cost associated with the increased hardware and power required. Consequently, approaches to sparse matrix computation that decrease the number of DRAM memory accesses required have been widely studied [15], [43]. We focus on a heterogenous architecture that adds a capability for energy-efficient recoding, orders of magnitude cheaper than that with conventional CPU cores [19], [21]. This heterogeneous architecture can reduce memory bandwidth requirements, and thereby increase performance on sparse matrix computation.

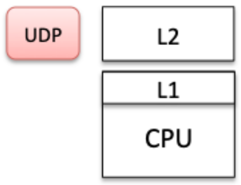


Fig. 1. CPU-UDP Heterogenous Recoding Architecture

In 2019, with transistor features now smaller than 10 nanometers, well past the end of Dennard Scaling and in the midst of the demise of Moore's Law, the cost balance of computing systems has shifted decisively. With 20 billion transistor chips, arithmetic operations and logic are cheap and fast, but data movement is the key cost and the performance limit [36], [47].

In such a world, the choice of how to represent information (encoding, representation) is a critical factor in computation and storage efficiency. The design of CPU's, long optimized for large and growing datatypes (16-bit, 32-bit, 64-bit, ...) as well as regular containers (vectors, arrays, hashmaps, trees, ...) makes them ill-suited for fast recoding (representation transformation). Thus, we consider a heterogeneous approach that combines a traditional CPU for efficient arithmetic computation with a data transformation accelerator (UDP [21]) for efficient recoding, as shown in Figure 1. Together, these are applied to the challenge of sparse matrix computation.

We evaluate this architecture on a wide variety of sparse matrices taken from the TAMU sparse matrix library [5]. This is the greatest variety of realistic sparse matrices available, and is widely viewed as the most representative. The architecture is modeled based on 14nm CMOS technology and varied DDR4 and HBM2 DRAM memory configurations. Application of efficient sparse matrix structures based on a first difference then Snappy [9] compression show a geometric mean of 2.4x better performance for memory-limited SpMV, and up to 6x lower memory power at the same performance. These results suggest that this heterogeneous approach not only provides overall system benefits, but also efficiency.

Specific contributions of the paper include:

- Proposal of a heterogeneous architecture that utilizes a data transformation accelerator together with CPU cores to achieve novel capabilities for recoding
- Evaluation of the heterogeneous architecture, using sparse matrix computation that yields 2.4x performance improvement at fixed memory power
- Evaluation of the heterogeneous architecture, using sparse matrix computation that yields 30-84% power reduction, varying by matrix, while providing the same performance
- Demonstrate that optimization across data encodings is only possible with our heterogeneous architecture. CPU architectures show >30x worse recoding performance, eliminating any benefit from data representations.

The rest of the paper is organized as follows. Section II presents key background on sparse matrix computation, Section III presents the heterogeneous CPU-UDP architecture, and discusses how it is applied to accelerate sparse matrix computation. Section IV describes the methodology for our evaluation based on the TAMU sparse matrix library, and careful system modeling. Section V presents our evaluation of the proposed CPU-UDP architecture on sparse matrix computation. Section VI presents key related work, showing the novelty of our approach, and comparing it to prior results. Finally, Section VII recaps our results and points out some interesting future directions.

II. Background

A. Sparse Matrix Vector Applications

Sparse matrix-vector multiplication (SpMV) has long been a focus of research and efficient implementation because of its central and growing importance for a broad range of scientific computing, numerical methods, graph analysis, and machine learning. Scientific simulation and modeling such as computational fluid dynamics [24], [48], cosmological/physics simulations [12], [42], and dot-matrix sequence alignment [34], [41], typically solve partial differential equations whose solutions compute the simulation evolution such as waves, particle motion, etc. Efficient algorithms focus computational effort on the most significant space and phenomena, producing sparse matrix structure, often with only a few percent nonzeros in matrixes with millions of nonzeor elements.

In graph analysis, most real-world datasets are sparse [2], [3], [6]. For example, the Netflix prize dataset [3] is a matrix with 480K users (rows) and 17K movies (cols) but only 100 million of the total possible 8 billion ratings are available. Similarly, very few of the possible edges are present in web graphs. It is important to store and manipulate such data as sparse matrices and keep only non-zero entries.

In machine learning, SpMV is an essential kernel in many popular algorithms such as sparse principal component analysis (PCA), or kernelized SVM classification and regression [38]. Sparse PCA computes a covariance matrix from a sparse dataset. It involves multiplication of one feature sparse vector by all other feature vectors in the matrix dataset. Kernelized SVM classifiers and regression engines compute the squared distance between two sparse feature vectors by calculating the inner-product.

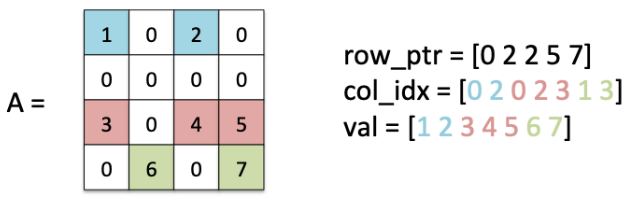
B. The Memory Bandwidth Challenge

The challenge of sparse matrix computation, as embodied in SpMV is that there are few computation operations (FLOPS) per byte of data. For large sparse matrices in particular, this produces a severe and growing mismatch between peak potential computation rate and memory bandwidth. The history of the challenge goes back to early 1990s. Cray Y-MP C90 enjoys sufficient memory bandwidth and architecture support for segmented sum operators for SpMV [13]. Nowadays, many applications require very large sparse matrices, such as the ones in machine learning, are at the scale of 100 billion parameters per matrix [30]. Over the past two decades, the exponential increase in microprocessor computation rate has far outstripped the slower increase in memory bandwdith. For example, McCalpin reported in an SC keynote in 2016 [4], that flops/memory access ratio has increase from from 4 (in 1990) to 100 (in 2020). This ratio continues to double every 5 years with no sign of slowing. As a result, a long history of declined bytes/flops in computer architectures [39] is observed.

Increasing flops is straightforward and easy, which can be done by simply adding more compute resources on the chip. On the other hand, the data bandwidth of off-chip main memory scales poorly, due to pin and energy limitations. The ITRS projects that package pin counts will scale less than 10 percent per year [39]. At the same time, per-pin bandwidth increases come with a difficult trade-off in power. A high-speed interface requires additional circuits (e.g. phase-locked loops, on-die termination) that consume static and dynamic power. These factors often make the main memory bandwidth a key cost, and thereby a system performance bottleneck in many large SpMV computations.

C. TAMU Sparse Matrix Collection

The TAMU Sparse Matrix Suite Collection [5], is the largest, and the most diverse representation suite of sparse matrices available. It is an actively growing set of sparse matrices that arise in real applications. It is widely used by the numerical linear algebra community for the performance evaluation of sparse matrix algorithms. Its matrices cover a wide spectrum of domains, including those arising from problems with underlying 2D or 3D geometry (as structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, and other discretizations) and those that typically do not have such geometry (optimization, circuit simulation, economic and financial modeling, theoretical and quantum chemistry, chemical process simulation, mathematics and statistics, power networks, and other networks and graphs). We use this collection, and selected matrices from the collection, for the evaluation throughout the paper.



$$A = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 4 & 5 \\ 0 & 6 & 0 & 7 \end{bmatrix}$$

```
// basic SpMV implementation
// y = A*x, A is in CSR format
for (int i = 0; i < m; i++) {
    double temp = y[i];
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
        temp = temp + val[j] * x[col_idx[j]];
    y[i] = temp;
}
```

Fig. 2. CSR format, and a basic CSR-based SpMV implementation.

III. A Heterogeneous Computing Architecture to accelerate SpMV

Performance of sparse matrix computations on conventional CPUs is limited by memory bandwidth, address generation and irregular memory references. Typical computing hardware system balance combined with low flops per matrix value (non-zero), and representation size makes memory bandwidth a critical limit. Address generation overheads arise from complex matrix representations, designed to reduce size. Irregular references arise from these complex formats. We address all three of these issues by introducing a complementary computing element that enables a dense representation with linear packing in memory, eliminating the need for complex address generation.

In the rest of the section, we first present an overview of the SpMV basic CPU implementation and its performance. Next, we describe the proposed heterogeneous CPU-UDP architecture and outline the optimized computation flow it enables. We close the section by a brief explanation of the UDP micro-architecture, whose efficient recoding performance is an essential enabler of the heterogeneous architecture.

A. Sparse Matrix-Vector Product (SpMV) Overview

We consider the SpMV operation $y \leftarrow Ax$, where A is a sparse matrix, and x, y are dense vectors. The SpMV kernel is as follows, $(i, j) : \forall a_{i,j} \neq 0 : y_i \leftarrow y_i + a_{i,j} \cdot x_j$, where $a_{i,j}$ denotes an element of A. SpMV has a low computational intensity – for an SpMV, each $a_{i,j}$ is used exactly once, and requires only two floating point operations. There can be reuse of x and y, so with optimized implementations, memory access for A dominates the time to execute SpMV.

B. SpMV on CPU

A common matrix representation for SpMV is Compressed Sparse Row (CSR), shown in Figure 2. The CSR format for sparse matrices consists of three arrays: (1) **row_ptr** array which saves the start and end pointers of the non-zeros of the rows. It has size $m+1$, where m is the number of rows of the matrix, (2) **col_idx** array stores column indices of the non-zeros, and (3) **val** array stores values of the non-zeros. A simple implementation of SpMV using CSR is shown in Figure 2. This implementation enumerates the stored elements of A by streaming both **val** and **col_idx**, and loads and stores each element of y only once.

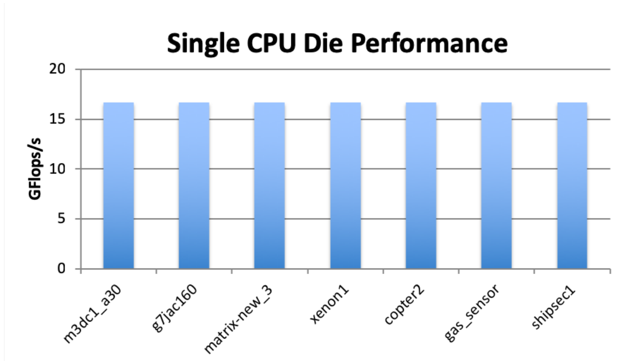


Fig. 3. Single Die CPU SpMV Performance, 100GB/s DDR system (memory-bandwidth limited).

In Figure 3, we plot the SpMV flops on a CPU-only system, modeled assuming a system with DDR4 and memory bandwidth of 100GB/s.[1] The state-of-art SpMV algorithms [15], [46] and libraries (e.g. BLAS) for a many-core architecture can easily saturate all the DDR4 channels on a single die. Thus, CPU SpMV performance is bounded by maximum memory bandwidth.

C. CPU-UDP Heterogeneous Architecture

Our heterogeneous architecture employs high-performance, energy-efficient recoding to mitigate the memory bandwidth limit in SpMV applications. The Unstructured Data Processor (see Section III-E for description) is such a general-purpose data recoding accelerator with software programmability. It sits on the same die with CPU cores and is integrated into the memory system to reduce the data movement overhead for performance. A single UDP accelerator (64-lanes) is around half the area of an x86 core + L1, and <5% of a core+associated L1/L2/L3 caches. So it consumes 1% the area of a 4-core Xeon chip area [21], and in a modern system perhaps 0.13% of a modern 32-core chip.

Figure 4 illustrates the CPU-UDP heterogeneous architecture. The CPU has normal access to caches, but can also access the UDP's local memory directly. The UDP local memory is mapped onto the CPU's address space as uncacheable (data won't appear in the cache memory hierarchy) as shown in Figure 5. When data is recoded into this UDP memory space (see Figure 7), the library routine initiates lightweight DMA operations (like memcpy) that transfer blocks of data from the DRAM to the UDP memory with high efficiency. The DMA

[1]The 100GB/s estimate is the fastest per-die DDR memory system available today – 2-die AMD Epyc system [1].

engine [44] acts as a traditional L2 agent to communicate with the LLC controller. The green lines in Figure 4 show the idea. A more aggressive approach (not shown) would integrate UDP seamlessly into the CPU memory system at the word level. The DMA engine is responsible for moving data to/from the memory controller from/to UDP local memory. It works with the snooper sitting on the memory bus that intercepts the related requests. This is very different from the memory integration in GPUs and PCIe-attached FPGA accelerators, which maintains separate address space and suffers from expensive off-chip data copy across address space.

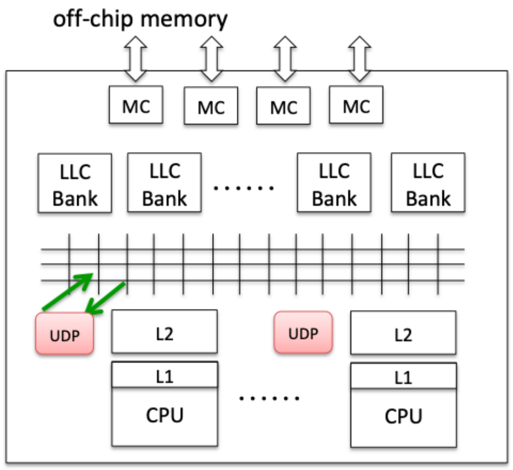


Fig. 4. CPU-UDP Heterogeneous Architecture, and integration of the UDP into the chip NoC fabric

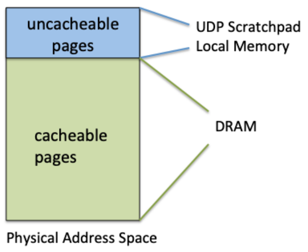


Fig. 5. UDP local memory is exposed as part of the CPU Address Space.

D. SpMV on CPU-UDP Architecture

The use of UDP/recoding enables efficient use of the system's memory bandwidth. First, sparse matrix data is streamed as contiguous address data, allowing the DDR system to be used at maximum efficiency. Second, the sparse matrix data is compressed with a custom, aggressive compression format on top of CSR format that reduces its size significantly. Third, the matrix is presented as CSR, avoiding any costly address generation. Together, these techniques maximize the use of available memory bandwidth to accelerate SpMV. Specifically, we use a combined Delta, Snappy, and Huffman encoding on top of the block CSR format to further compress the matrices and reduce the off-chip memory traffic. The Unstructured Data Processor (UDP) serves as a powerful programmable recoding engine for such need, and allows these recoding transformations to be written in software. This programmability enables a broad variety of recoding approaches. In Figure 6, the Delta-Snappy-Huffman encoded CSR matrix blocks are streamed into the CPU-UDP chip, UDP executes the decompression algorithm to recover the blocks in CSR format. CPU executes matrix-vector multiplication on the native CSR format. The tiled SpMV code only needs to add two function calls for the value block and the index block decompression, as Figure 7 shown. This modular approach eases programming effort as well as avoiding complex addressing computations on the CPU. No other changes to the SpMV implementation are required. But in the future, if better representations are discovered, they can be implemented for the UDP/recode engine to further improve performance without requiring CPU code change.

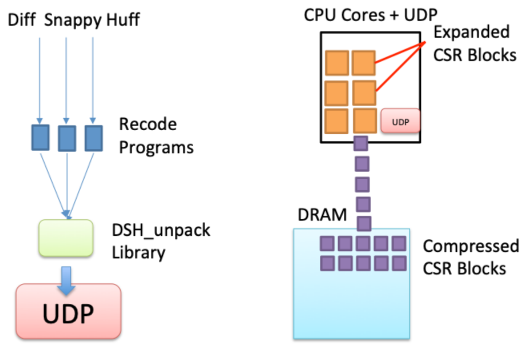


Fig. 6. SpMV running on CPU-UDP Architecture.

```
for(int i=0;i<M/B;i++){
 for(int k=i*B;k<(i+1)*B;k++){
   recode(DSH_unpack,ccol_idx,col_idx,k,B);
   recode(DSH_unpack,cvalues,values,k,B);
   for(int j=row_ptr[i];j<row_ptr[k+1];j++)
        y[i]+=val[j]*x[col_idx[j]];
   }
}
```

Fig. 7. Recoding-enhanced SpMV implementation. The `recode` calls encapsulate use of the UDP/recode engine and can be customized for future compressed formats.

E. UDP Micro-architecture

The UDP is a software programmable data transformation engine, optimized for high performance. Highly effective, dictionary-based decoding or decompression algorithms introduce many branches between small code blocks. This leads to poor performance on CPUs. Essentially, the encoded format contains a sequence of tag-value pairs, with the corresponding operation to decode the value stored in the tag field. However, CPUs suffer from poor branch prediction on the operation dispatch, which can lead to 80% cycle waste due to frequent pipeline flushes [21].

We briefly describe the unstructured data processor (UDP) [18], [21], which excels at branch-intensive tasks, especially data recoding tasks, with high efficiency. The UDP is an MIMD parallel accelerator (Figure 8) with each lane pairing its own scratchpad memory bank(s). Parallel lanes exploit the data parallelism often found in encoding and transformation tasks, and the lane architecture includes support for branch-intensive codes, computation on small and variable-sized application-data encodings, and programmability.

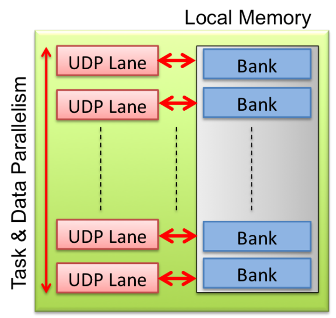


Fig. 8. A 64-lane UDP Accelerator.

The UDP accelerator consists of 64 parallel UDP lanes. Each lane contains three key units: 1) Dispatch, 2) Symbol Prefetch, and 3) Action (see Figure 9). The Dispatch unit handles multi-way dispatch (transitions), computing the target dispatch memory address for varied transition types and sources. The Stream Prefetch unit prefetches stream data, and supports variable-size symbols. The Action unit executes the UDP actions, writing results to the UDP data registers or the local memory.

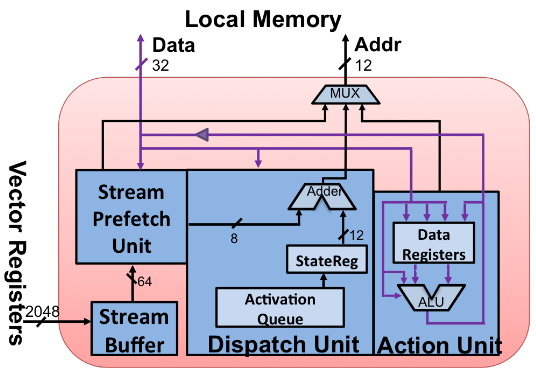


Fig. 9. UDP Lane Micro-architecture.

Multi-way dispatch is the key micro-architecture feature of the UDP lane to gain efficiency from for branch-intensive applications. The UDP selects efficiently from multiple targets by using the symbol (or any value) as a dynamic offset. Compared to branch offset, multi-way dispatch can process several branches in a single dispatch operation, and avoids explicit encoding of many offsets. Compared to branch indirect, multi-way dispatch avoids an explicit table of branch targets, producing placement coupling challenges discussed below. As a result, compared to both, multi-way dispatch shuns prediction, depending on a short pipeline for good performance.

To support multi-way dispatch, the UDP compiler deals with precise relative location constraints directly. It converts UDP assembly to machine code, and creates an optimized memory layout using the Efficient Coupled Linear Packing (EffCLiP) algorithm [20] that resolves the coupled code block placement constraints. Together, EffCLiP and UDP achieve dense memory utilization and a simple, fixed hash function – integer addition. This enables a high clock rate and energy efficient execution. In effect, EffCLiP achieves a "perfect hash" for a given set of code blocks. The UDP assembler back-propagates transition type information along dispatch arcs, and then generates machine binaries using machine-level transitions and actions.

Together, multi-way dispatch for variable symbol size and scratchpad memory provides each UDP lane high efficiency for data recoding tasks. MIMD parallelism across lanes provides UDP sufficient throughput by exploiting the block-oriented pattern in SpMV applications. A more complete description of the UDP is available [**?**], [18], [19], [21].

IV. Methodology

A. System Modeling

To evaluate the effectiveness of compression, we compare traditional CPU software, Google Snappy version 1.1.3. compiled using gcc version 5.5.0. The resulting software was run on a local machine called river-fe running CentOS Linux 7 (Core). This system has two Intel Xeon E5-2670 v3 processors, each of which has 12 cores operating at 2.30 GHz and a 30 MB cache. Each socket can support up to 12 threads. River-fe has 512 GB of RAM (16 x 32 GB TruDDR4 Memory 2133 MHz). The system has 96 TB of total disk storage (8 x 6 TB 7.2K 6 Gbps HDD and 12 x 4 TB 7.2K HDD). However, we study two more aggressive memory systems, because even a few cores is plenty to keep up with a 100GB/s memory system on SpMV.

The UDP-part of the heterogeneous accelerated system is modeled using a 64-lane cycle-accurate simulator written in C++. The UDP is implemented in SystemVerilog RTL. Performance reported in prior publication [21] for 28nm CMOS is extrapolated for a 14nm process and 64-bit extension, which takes the previously reported speed and power from 1GHz and 864mW to 1.6Ghz and 160mW.[2] The Snappy, Huffman, and Delta performance is measured on the UDP simulator using their respective versions for UDP.

We use two examples of high-performance memory systems to estimate memory power savings. For DDR, we use the AMD Epyc system [1], amongst the most aggressive DDR systems that with two compute die and eight DDR4 memory controllers achieves 200GB/s. We assume only one die, so the corresponding peak is 100GB/s. We model energy costs of reading DDR DRAM and shipping to the CPU to be 100pJ/bit. For HBM2, we model a peak bandwidth for 4 stacks to be 1 terabyte/second, and 8pJ/bit [16].

[2]The performance and power for UDP is heavily dominated by SRAM access time and energy, so this scaling is based on CACTI estimates and reflects two full process generation (TSMC 28nm to TSMC 14nm) and the shift to FinFET transistors.

B. *Compression: Delta-Snappy-Huffman*

We run compression on matrices sourced from the Texas A&M University sparse matrix collection. We consider 369 matrices drawn from the largest 20% of those in the collection. The matrices range in number of non-zeros from 1.0E+6 to 8.0E+8, with a median of 4.9E+6 (about 5 million). The sparsity of the selected matrices varies widely from 9.4E-7% (one-millionth) to 19%, with a median of 0.019%. The set contains matrices with banded, diagonal, and symmetric structure, as well as unstructured matrices. We use the bytes per non-zero element metric to evaluate compression effectiveness, so the original storage format does not matter. We also consider seven representative matrices, *copter2*, *g7jac160*, *gas_sensor*, *m3dc1_a30*, *matrix-new_3*, *shipsec1*, and *xenon1* to look at memory system power savings.

We generate a Huffman tree for each sparse matrix by sampling a subset of the 8KB blocks. The number of blocks sampled was varied (up to 40% of the total number of blocks) to get good coverage. Snappy is selected because of its popularity. Delta encoding of the matrix indices provides large benefits for matrices that are symmetrical and have diagonal structure, as it turns arithmetic series into easily compressible repeating integers. The delta encoding step on its own provides no benefit, but combined with a compression algorithms helps to reduce the bytes per non-zero value significantly.

V. Evaluation

A. *Sparse Matrix Transformation Performance*

For SpMV, the matrix is kept in compressed form in memory, brought to the processor thereby reducing the number of bits moved from the memory, and then it is decompressed "on the fly", enabling the actual sparse matrix computation to proceed unchanged. Our empirical experiments showed that delta-snappy-huffman produced the best compression results. So, the decompression process contains these three transformations, run in the reverse order – huffman decode, snappy decode, inverse delta – that run as a series of steps in a single lane of the UDP. First, the fully compressed block is run through a Huffman decoding program. Then, the data passes through a snappy decompression stage. Finally, the data is sent through a delta-decoding step, at which point the block is fully decompressed back to 8KB. These steps allow the compressed blocks to be sent from memory to UDP to the CPU. This transformation can be run in parallel on all 64 lanes of the UDP, with enough memory per lane to store the 8KB block and the output of each individual step.

Compressed Size: We compare the effectiveness of Snappy on a CPU versus the Delta-Snappy-Huffman combination on the heterogeneous accelerator, the UDP, using the compressed size (see Figure 10). The baseline CSR representation is 4 bytes (index) and 8 bytes (double float), for a total of 12 bytes per non-zero. The Snappy on CPU (32KB block size) produces a geometric mean of 5.20 bytes per non-zero value. On the UDP, Snappy compression on the delta-encoded blocks, produces a geometric mean of 5.92 bytes per non-zero value. Adding Huffman encoding further reduces to 5.00 bytes per non-zero. The combined UDP Delta-Snappy-Huffman surpasses the compression of the CPU based snappy, despite being limited to a small 8KB block size.

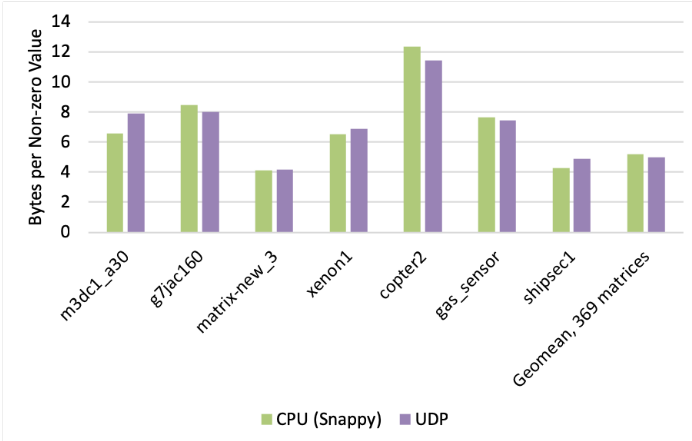


Fig. 10. Comparing Compressed Size with CPU (Snappy) and UDP (Delta-Snappy-Huffman).

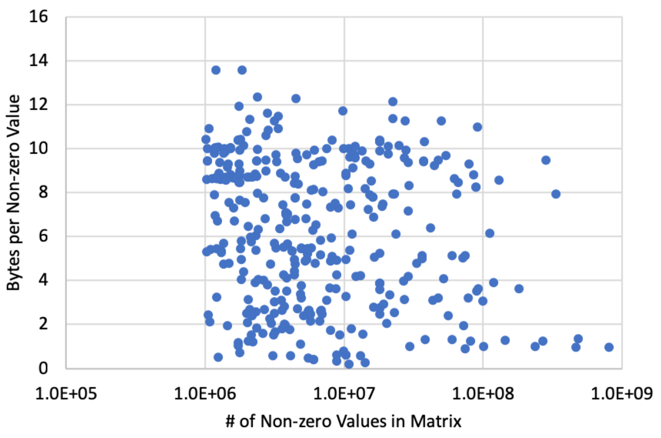


Fig. 11. Bytes per Non-zero Value vs # of Non-zeros.

We show compressed size data in a scatterplot with size for the largest matrices (all those with more than one million non-zeros) in the TAMU Sparse matrix collection in Figures 11. These results show no clear correlation of matrix compression ratio and size, but good compression overall is achieved by the Delta-Snappy-Huffman combination.

Transformation Throughput: The UDP takes an geometric mean of 21.7 microseconds across all 369 matrices to decompress a single 8KB block. The CPU-UDP heterogeneous architecture can effectively decompress these matrices – Figure 12 shows the decompression rates achieved by a 64 lanes of the UDP for these particular matrices, comparing them to that achievable on a 32 thread CPU. As Figure 12 demonstrates, a full 64 lane UDP increases the decompression throughput dramatically over Snappy run on even a 32-thread

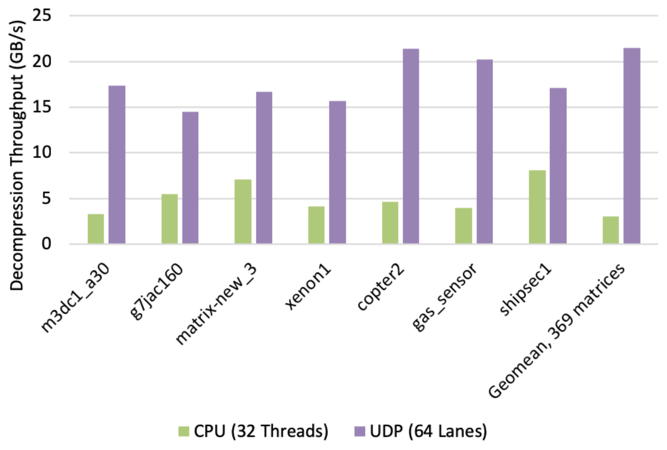


Fig. 12. 32-thread CPU vs 64-lane UDP Decompression Throughput.

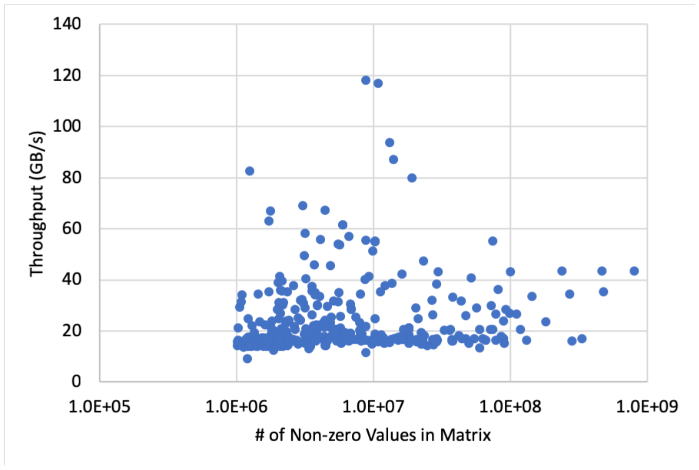


Fig. 13. Decompression Throughput of a 64-lane UDP vs # of Non-zeros.

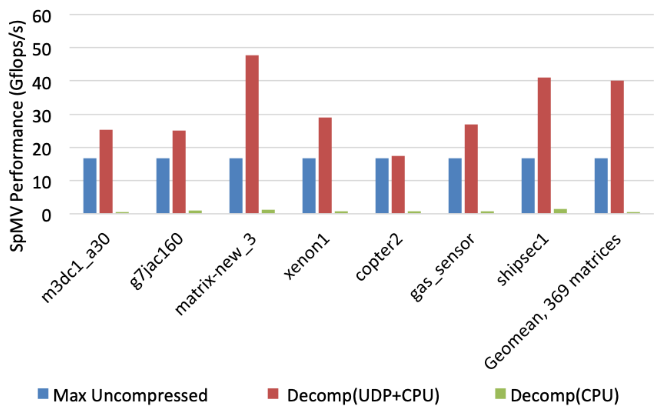


Fig. 14. CPU vs. CPU-UDP SpMV Performance on DDR4 (100GB/s).

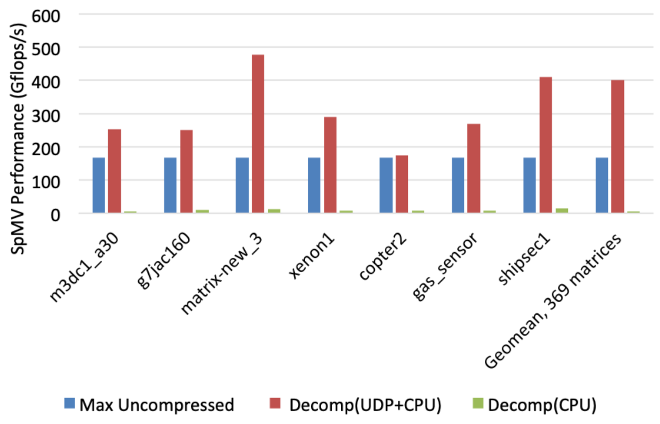


Fig. 15. CPU vs. CPU-UDP SpMV Performance on HBM2 (1TB/s).

CPU. Across the 369 matrix sample set, the UDP increases the geometric mean decompression throughput 7-fold compared to the CPU. On the 7 specific matrices, we observe speedups between 2x and 5x to over 20GB/s. Figure 13 further shows the scatterplot of the UDP performance detail on 369 matrices. UDP's optimized architecture not only delivers much higher decompression performance, it does so as a much lower power (0.16W vs. perhaps 100W), and with a tiny silicon area.

B. SpMV on Heterogeneous Architecture Performance

SpMV Performance: We consider the performance of a SpMV workload on the CPU and CPU+UDP workload, using DDR4 or HBM2, and using a memory bandwidth bound computation with 2 flops per non-zero value. As Figures 14 and 15 display, the heterogenous architecture, geomean Decomp(UDP+CPU), more than doubles the uncompressed SpMV performance on CPU, geomean Max Uncompressed. The primary reason for this speedup is due to the smaller size of the encoded matrix that enables faster transfer to the computation. Using similar encoded approach a CPU alone is impractical, as Decomp(CPU) falls far below the both Decomp(UDP+CPU) and Max Uncompressed. By offloading decompression to the UDP, which is able to handle the memory bandwidth of DDR4 or HBM2, in effect the cost of the decompression is eliminated (area, power). Hence, by reducing the average storage usage per non-zero value from 12 to 5, we achieve a 2.4x increase in achieved gigaflops over CPU only architecture on memory bound SpMV computation.

CPUs for Transformation? It is instructive to consider the case using CPUs for matrix block decompression rather than the UDP before SpMV computations (see the light green bars in Figures 14 and15. The CPU cores provide poor transformation throughput (Decomp(CPU) + SpMV) limits the overall performance (>30x slower), and are far from saturating the memory bandwidth. As a result, CPU-only architectures cannot exploit flexible data transformation on SpMV. The CPU-UDP heterogeneous architecture brings the new recoding capability, opening a new optimization space.

Saving Memory Power: Another way to exploit the new capabilities of the heterogeneous architecture is to maintain performance, but reduce the memory system power. UDP power is modeled as 0.16W per 64-lane UDP with sufficient

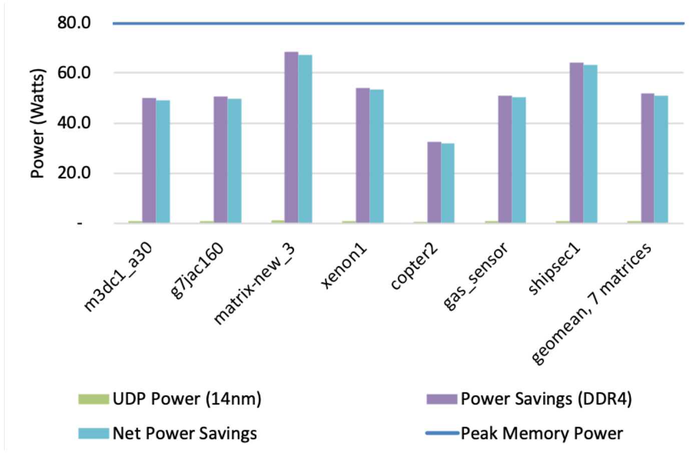


Fig. 16. Raw and Net Memory Power Savings for a 100GB/s DDR4 System.

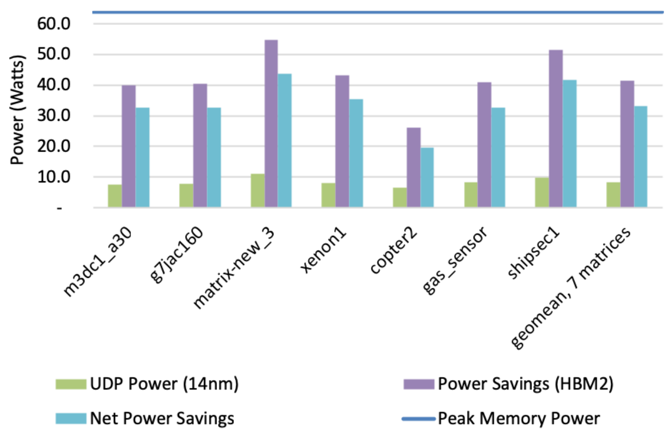


Fig. 17. Raw and Net Memory Power Savings at 1TB/s HBM2 System.

number of UDP's to meet the desired memory rate to feed the computation at same performance (100GB/s or 1TB/s out from UDP's) The plots in Figures 16 and 17 show memory power savings, the added UDP power, and the net power benefit. We compute the maximum memory power by multiplying the maximum data rate by the energy per bit. For the DDR system, this is 100GB/s $\times$ 100pJ/bit $\times$ 8 bits/byte $=$ 80W. For the HBM2 system, 1000 GB/s (four HBM2 stacks) $\times$ 8pJ/bits $\times$ 8 bits/byte $=$ 64W. When evaluating on the 7 representative matrices the UDP saves an average 51W (out of 80W) when using DDR4, and 33W (out of 64W) when using HBM2.

VI. Related Work

A. SpMV Software Optimization

Optimizing SpMV on multi-core platforms and GPUs has received great research interest [10], [11], [15], [17], [33], [46]. These efforts propose clever algorithms and code structure optimizations to better utilize the memory bandwidth and the computation power of the processor in a robust fashion. Merge-based SpMV and runtime decomposition [33] is used to improve the correlation between runtime and the problem size. As a result, the performance scaling is more stable and robust performance gain can be achieved with increased memory bandwidth and flops. Our heteregeneous architecture that integrates a UDP can apply to the throughput-oriented processors and maximize the available memory bandwidth for better performance. All the software optimization techniques can be applied to the UDP-integrated system as well.

Another direction in software optimization is auto-tuning. The SpMV kernels and storage formats are varied and the optimizer picks the best one for execution [17], [29], [40]. These techniques free the programmers from low-level manual tuning, but require many different implementations to be built. Another interesting thread is the SpMV-based operator integration into traditional data analytical systems [14]. It applies a classic query-like global optimization across scheduling, formats, and execution for better performance. Compared to all of these, our use of a general programmable compression approach produces simpler code and good performance.

B. SpMV Format Optimization

The naive sequential implementation (Figure 2) is simple and can be easily parallelized for performance. Many block-oriented, customized data storage formats and algorithms [10], [15], [27], [31], [32], [46] have been proposed to further compress and improve the SpMV performance. For example, one novel data structure as bitmasked register block [15] is proposed for multi-threaded SpMV computation. It saves upto 3.5x memory bandwidth on a range of SpMV kernels. The best suitable encoding format of the matrix is chosen based on the matrix properties and machine characteristics [29], [40]. In contrast, our approach requires no specialized coding and format design for the CPU, rather using a single algorithm written in software for the UDP that reduces data movement work.

C. SpMV Hardware Acceleration

In the hardware community, efficient SpMV execution is also an important research question. Multiple designs have been proposed to build a dedicated accelerator for deep learning workloads with sparse neural net layers [25], [37], [49]. In these designs, circuits are hardwired to direct support multiplication on the sparse NN layers in CSR format. Others have built the embedded processor and FPGA-based acceleration for SpMV [23], [35]. The hardware support for SpMV in these approaches mostly focus on reducing the hardware resources to saturate the memory bandwidth. Similarly, GPUs provide massive parallel streaming processors to accelerate SpMV computations but the performance is still limited by the DRAM bandwidth even with the advanced GDDR packaging technology [15], [27], [46]. The scarce memory bandwidth is the number-one limiting factor for performance in these hardware systems. In an accelerator-rich heterogeneous architecture, our UDP accelerator can co-exist with these accelerators. It further reduces the memory bandwidth need, feeding more data to the throughput-oriented accelerators.

D. Compression Hardware Accelerators

Both academia and industry designed and developed hardware accelerators for compression [7], [8], [22]. PCIe attached compression accelerators such as Microsoft Xpress FPGA acceleration [22] and Intel Quick Assist compression chipset [8], provides a 2-5GB/s compression throughput per device. IBM builds an SoC-style network processor (PowerEN) with on-chip compression accelerator [7], achieving 1.5 GB/s throughput. Our heterogeneous UDP accelerator has several advantages. First, it is programmable, allowing the compression algorithm to be adapted to the data. This gives a significant compression ratio benefit. Second, it is high performance, achieving > 10 GB/s decompression throughput. And, third, it integrates directly into the memory system, avoiding movement to a PCI bus. Our CPU-UDP heterogeneous architecture benefits from the tight in-memory hierarchy integration, achieving flexible and cheap data sharing across CPUs and accelerators.

VII. Summary and Future Work

We study the performance and power benefit of applying a flexible, software programmable data recoding engine (UDP) to form a heterogeneous architecture for the SpMV computation. We systematically evaluate the system using the TAMU sparse matrix library and the result shows a geometric mean performance benefit of 2.4x, power reduction of 63% and 51% for DDR and HMB respectively. Together, these suggest the promise of this new heterogeneous architecture approach. The CPU-UDP heterogeneous architecture provides cheap and flexible data recoding, which opens up a few new research opportunities. Interesting future work includes: novel and customized encodings on top of CSR for matrices with particular structures, performance benefit of other sparse matrix computation using flexible data recoding, and more critical computation and applications that can benefit from data recoding.

VIII. Acknowledgment

This research was supported in part by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration. Support from the CERES Center for Unstoppable Computing is also gratefully acknowledged.

References

[1] Amd optimizes epyc memory with numa. https://www.amd.com/system/files/2018-03/AMD-Optimizes-EPYC-Memory-With-NUMA.pdf.

[2] Livejournal social network. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[3] Netflix prize dataset. https://www.kaggle.com/netflix-inc/netflix-prize-data.

[4] Sc16 invited talk: Memory bandwidth and system balance in hpc systems. http://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/.

[5] Suitesparse matrix collection. https://sparse.tamu.edu/about.

[6] Twitter follower network. https://snap.stanford.edu/data/twitter-2010.html.

[7] The ibm power edge of network processor. http://www.cercs.gatech.edu/iucrc10/material/franke.pdf, 2010.

[8] Intel communications chipset 8955. http://ark.intel.com/products/80372/Intel-DH8955-PCH, 2013.

[9] Google snappy compression library. https://github.com/google/snappy, 2016.

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 781–792. IEEE Press, 2014.

[11] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 18. ACM, 2009.

[12] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation*. CRC press, 2004.

[13] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[14] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[15] A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Parallel & distributed processing symposium (IPDPS), 2011 IEEE international*, pages 721–733. IEEE, 2011.

[16] N. Chatterjee, M. OConnor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. Architecting an energy-efficient dram system for gpus. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 73–84, Feb 2017.

[17] A. Elafrou, G. Goumas, and N. Koziris. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 292–301. IEEE, 2017.

[18] Y. Fang and A. A. Chien. Udp system interface and lane isa definition. Technical report, University of Chicago, 2017.

[19] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, Dec. 2015.

[20] Y. Fang, A. Lehane, and A. A. Chien. Effclip: Efficient coupled-linear packing for finite automata. *University of Chicago Technical Report, TR-2015-05*, May 2015.

[21] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. Udp: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–68. ACM, 2017.

[22] J. Fowers et al. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proc. of FCCM '15*, May 2015.

[23] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 36–43. IEEE, 2014.

[24] M. Germano, U. Piomelli, P. Moin, and W. H. Cabot. A dynamic subgrid-scale eddy viscosity model. *Physics of Fluids A: Fluid Dynamics*, 3(7):1760–1765, 1991.

[25] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[26] M. A. Heroux. Hpccg solver package, version 00. https://www.osti.gov//servlets/purl/1230960, 3 2007.

[27] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.

[28] O. S. Lawlor. In-memory data compression for sparse matrices. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, page 6. ACM, 2013.

[29] J. Li, G. Tan, M. Chen, and N. Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[30] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[31] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[32] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 273–282. ACM, 2013.

[33] D. Merrill and M. Garland. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. In *ACM SIGPLAN Notices*, volume 51, page 43. ACM, 2016.

[34] D. W. Mount. Using the basic local alignment search tool (blast). *Cold Spring Harbor Protocols*, 2007(7):pdb–top17, 2007.

[35] E. Nurvitadhi, A. Mishra, and D. Marr. A sparse matrix vector multiply accelerator for support vector machine. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 109–116. IEEE Press, 2015.

[36] NVIDIA. Titan v has 21 billion transistors. ”https://www.nvidia.com/en-us/titan/titan-v/”, December 2017.

[37] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 27–40. ACM, 2017.

[38] C. Robert. Machine learning, a probabilistic perspective, 2014.

[39] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 371–382. ACM, 2009.

[40] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 99–108. ACM, 2015.

[41] E. L. Sonnhammer and R. Durbin. A dot-matrix program with dynamic threshold control suited for genomic dna and protein sequence analysis. *Gene*, 167(1):GC1–GC10, 1995.

[42] V. Springel. The cosmological simulation code gadget-2. *Monthly notices of the royal astronomical society*, 364(4):1105–1134, 2005.

[43] J. P. Stevenson, A. Firoozshahian, A. Solomatnikov, M. Horowitz, and D. Cheriton. Sparse matrix-vector multiply on the hicamp architecture. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS ’12, pages 195–204, New York, NY, USA, 2012. ACM.

[44] T. Thanh-Hoang et al. A data layout transformation (dlt) accelerator: Architectural support for data movement optimization in accelerated-centric heterogeneous systems. In *Proc. of DATE’16*. IEEE, Mar. 2016.

[45] S. Williams, N. Bell, J. W. Choi, M. Garland, L. Oliker, and R. Vuduc. Sparse matrix-vector multiplication on multicore and accelerators. *Scientific Computing with Multicore and Accelerators*, pages 83–109, 2010.

[46] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing, 2007. SC’07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[47] Xilinx. Ultrascale plus has 21.2 teramacs of dsp performance. ”https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html”, January 2016.

[48] B. Xu and A. Yu. Numerical simulation of the gas-solid flow in a fluidized bed by combining discrete particle method with computational fluid dynamics. *Chemical Engineering Science*, 52(16):2785–2809, 1997.

[49] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 20. IEEE Press, 2016.