Sparse Matrix Multiplication on a Many-Core Platform

By

PEIYAO SHI

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Venkatesh Akella

Member, Dr. Soheil Ghiasi

Committee in charge
2018

Abstract

Sparse matrix-vector multiplication (SpMV) is a critical operation in scientific computing and engineering applications. This thesis explores implementing SpMV kernels on a many-core array. Eight functionally equivalent SpMV implementations are created for a fine-grained many-core platform with independent shared memory modules and floating-point (FP) capabilities. These implementations are considered against one general-purpose processor chip (Intel Core-i7 3720QM) and one graphics processing unit (GPU) chip (NVIDIA Quadro 620). The designs for the many-core array, general-purpose processor, and GPU are evaluated using the metrics of throughput per area and throughput per watt when operating on a set of twenty-seven unstructured sparse matrices of varying dimensions sourced from a wide range of domains including directed graph, circuit simulation problems, computational fluid dynamics problems, structural problems, and theoretical/quantum chemistry problems.

Since different scale methodologies and data types are used, throughput, power and area results are scaled to 32 nm and single-precision FP values for the general-purpose processor, GPU and fine-grained many-core implementations. The improvement in throughput per watt achieved from experiments is 69× on average among all simulated matrices versus the general-purpose processor implementations, and 94× on average versus the GPU implementations. The improvement in throughput per area achieved from experiments is 54× on average versus the general-purpose processor implementations, and 40× on average versus the GPU implementations.

Acknowledgments

I would like to first thank my advisor Professor Bevan Baas for all of his help and guidance throughout my graduate career. Professor Baas is a brilliant researcher who has guided and supported me throughout my graduate career, especially for the completion of my research towards MS degree over the past two years. He has helped me grow into a more productive researcher and taught me the necessity of having high ambitions and goals. I learned the importance of self-discipline and have been able to explore new research points independently. His guidance allowed me to see both the details and the big picture.

I would also like to thank Professor Soheil Ghiasi and Professor Venkatesh Akella for serving on my MS Plan I thesis committee for all of their guidance in the process. Thank you for taking the time to read this thesis and providing me with valuable feedback.

Thank you to Professor Bevan Baas, Professor Hussain Al-Asaad, and Professor Venkatesh Akella, for whom I have had the privilege to serve as a teaching assistant for EEC 180B, EEC 180A, and EEC 170, respectively. These experiences taught me to be an effective leader, enhanced my ability to communicate clearly, and helped me develop a command of the subject matter.

I thank all of the students in the VLSI Computation Laboratory, past and present. Thank you to all of the VCL members I have had the privilege of working alongside, Satyabrata Sarangi, Shifu Wu, Timothy Andreas, Mark Hildebrand, Jin Cui and Renjie Chen, from whom I received useful feedback.

I am especially grateful for Dr. Jon Pimentel, from whom I continued the many-core Sparse matrix-vector multiplication (SpMV) research.

I would also like to thank my family for all of their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Matrix-vector multiplication of the form $y = Ax$ is a basic tool of linear algebra, and as such has numerous applications in many areas of mathematics, as well as in applied mathematics, statistics, physics, economics, and engineering. Matrices that contain mostly zero values are called sparse, distinct from matrices where most of the values are non-zero, called dense [3]. The input matrix A and input vector x can be either dense or sparse. The output vector y is dense. Figure 1.1 shows both the dense and sparse matrix vector multiplication. To multiply a row vector by a column vector, the row vector must have as many columns as the column vector has rows.

As dense matrix-vector multiplication has been widely used in a wide variety of applications, sparse matrix-vector multiplication (SpMV) of the form $y = Ax$ is becoming more common, especially in many scientific and engineering applications, such as linear programming problems, combinatorial problems, graph analytics, and even in the entire subdomain of machine learning, such as natural language processing.

It is computationally expensive to represent and work with huge sparse matrices, which size can reach million by millions, as though they are dense, and much improvement in performance can be achieved by using representations and operations that specifically handle the matrix sparsity. In case of repeated $y = Ax$ operation involving the same input matrix A but possibly changing numerical values of its elements, A can be preprocessed to reduce both the parallel and sequential run time of the SpMV kernel [4].

$$\underset{\text{Dense Matrix A}}{\begin{pmatrix} 1 & 1 & 0 & 2 & 3 & 1 \\ 1 & 3 & 5 & 0 & 4 & 1 \\ 0 & 2 & 6 & 1 & 3 & 1 \\ 0 & 5 & 1 & 3 & 4 & 1 \\ 1 & 1 & 2 & 6 & 4 & 0 \\ 0 & 2 & 2 & 3 & 1 & 7 \end{pmatrix}} \times \underset{\text{Dense Vector X}}{\begin{bmatrix} 1 \\ 2 \\ 5 \\ 3 \\ 6 \\ 5 \end{bmatrix}} = \underset{\text{Dense Vector Y}}{\begin{bmatrix} 32 \\ 61 \\ 60 \\ 53 \\ 55 \\ 64 \end{bmatrix}}$$

$$\underset{\text{Sparse Matrix A}}{\begin{pmatrix} 0 & 0 & 0 & 0 & 2 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}} \times \underset{\text{Dense Vector X}}{\begin{bmatrix} 1 \\ 2 \\ 5 \\ 3 \\ 6 \\ 5 \end{bmatrix}} = \underset{\text{Dense Vector Y}}{\begin{bmatrix} 12 \\ 5 \\ 12 \\ 5 \\ 6 \\ 2 \end{bmatrix}}$$

Figure 1.1: Example matrix-vector multiplications showing dense vectors and (top) dense matrix, and (bottom) sparse matrix containing mostly zero data values.

Using parallel processing for SpMV is challenging and not as efficient as dense matrices because the ratio of computation to memory access is low. Complementary strategies for workload decomposition and matrix storage formatting to achieve uniform processor utilization and efficient use of memory bandwidth regardless of the matrix's non-zero structure are necessary and required for achieving good performance on today's parallel architecture.

Many custom matrix formats and algorithms have been developed to exploit both the structural properties of a given matrix and the organization of the underlying platform architecture to meet these design objectives [5]. More than 60 SpMV algorithms and sparse matrix formats have been proposed on general-purpose processors [6], GPUs [7], FPGAs [8, 9], and many-core platforms [10, 11], which illustrates the current trend of increased parallelism in high performance computer architectures. However, specialized or supplementary formats ultimately burden the application with significant practical cost, like extra preprocessing time or inspection and formatting, which could be tens of thousands of times greater than the SpMV operation itself; and excess storage overhead because the original Compressed Sparse Row (CSR) [12] matrix is likely required by other routines and cannot be discarded. Normally, sparse matrices will not be maintained in custom compressed format, instead preferring general-purpose one such as the CSR format, which is free of architecture-specific blocking and used to improve portability for different applications.

As a result of a number of inherent performance limitations, like the algorithmic nature of the kernel, the architectural nature of the platform and the sparsity patterns of the matrix, only a fraction of the peak performance of current computing platform can be achieved when implementing SpMV on them. With number of cores on one platform or die scaling to hundreds or thousands, current SpMV algorithms designed for traditional architectures cannot be used due to differences within architectural features such as intra-processor communication, shared memories, and off chip I/O [13, 14].

This work presents high performance, area and power efficient, scalable CSR SpMV implementations, which were simulated on a fine-grained many-core array of low-powered, simple Multiple Instruction Multiple Data (MIMD) [15] processors (AsAP3). Each SpMV implementation consists a set of small modular program kernels operating on each core, making them scalable to different array sizes.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 starts with going over SpMV on modern multi-core and many-core processors. Then the target fine-grained many-core architecture, AsAP3 (KiloCore) is explained, which is used throughout the thesis. The extensive simulation methodology, which is used to generate the results from the proposed SpMV algorithms, is also introduced. A brief introduction to sparse matrix collection from University of Florida [5], as well as all simulated matrices chosen for this thesis are given at the end.

Chapter 3 first gives the main kernels that make up the proposed many-core SpMV methods, followed by proposed mappings of these implementations onto a generic 2D mesh based on AsAP3 architecture. Lastly, different phases of SpMV methods are analyzed and results from various implementations are compared.

Chapter 4 presents power and area efficiency comparison for all the implementations on a many-core processor array (AsAP3).

Chapter 5 presents the most power and area efficient implementations on AsAP3, compared to the-state-of-art implementations on Intel general-purpose processor and Nvidia GPU platforms.

Finally, Chapter 6 gives a brief summary of the thesis and proposes future projects.

Chapter 2

Background

2.1 SpMV on Modern Multi- and Many-core Processors

As computers changes to multi- and many-core processors, this scaling progression necessitates a new shift to implement sparse matrix-vector multiplication kernels with large processor arrays [16].

As modern processors continue to exhibit wider parallelism, workload imbalance can quickly become the high-order performance limiter for segmented computations, such as CSR SpMV. Current research adapted parallelizing SpMV algorithms to take advantage of multiple processing cores.

2.1.1 Sparse Matrix Compressed Format

A sparse matrix is a matrix consisting of mostly zero elements. Sparse matrices are usually stored in a compact format, i.e., only non-zero elements are preserved. Bell and Garland [17] proposed some widely-used sparse matrix storage formats, including Compressed Sparse Row (CSR), ELLPACK (ELL), Coordinate (COO), and Hybrid ELL/COO (HYB), each of which has its own storage requirement, computational characteristic, and accessing method for the non-zero elements of the sparse matrix.

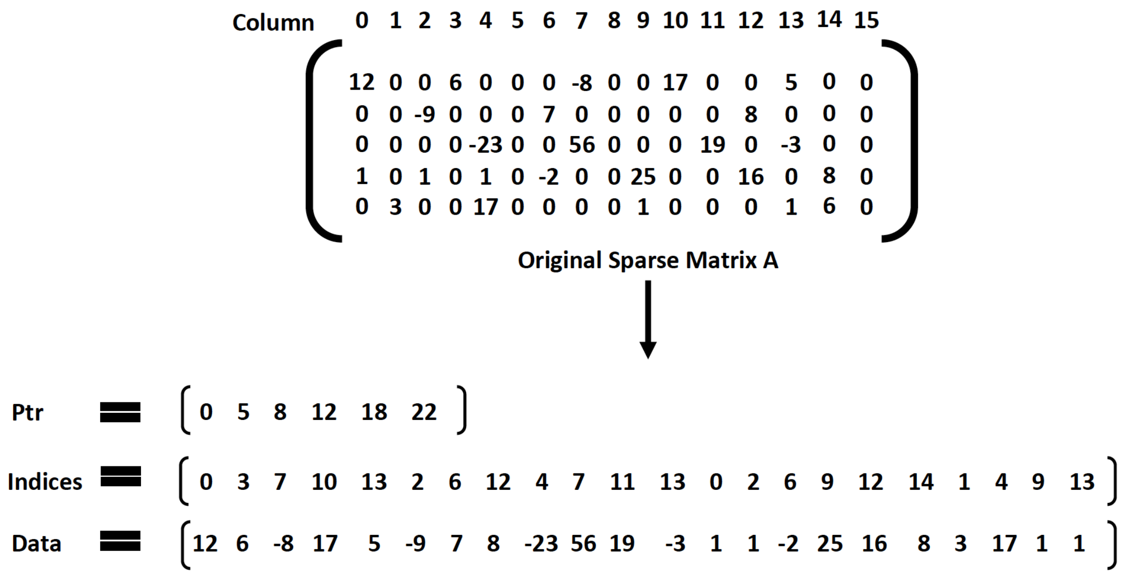
In addition to the matrix storage format, how to process the multiplication with a dense vector x in parallel is another question, which includes two main points: balancing the workload among the distinct cores/threads; and accessing the matrix entries and the vector values efficiently.

The standard approach of parallelizing the CSR, ELL, COO and HYB formats is to distribute the rows among distinct cores/threads [18].

According to the Roofline model [19], one method of visualizing program performance and determining the program boundedness (compute or memory), the operational intensity (in flops/byte or flops/word) was measured by $I = W/Q$, where W is the total number of useful operations that the algorithm performs, and Q is the total number of words it transfers (reads or writes). Memory boundedness refers to a situation where the time to complete a computation is decided primarily by the amount of data transferred from memory (i.e. memory speed). SpMV is characterized by a very low flops/byte ratio because the kernel performs $O(NNZ)$ operations on $O(N{+}NNZ)$ amount of data, where NNZ represents number of sparse multiplications (number of non-zero elements in matrix A) and N refers to number of non-zero elements in vector x. Since SpMV is obviously a memory-bound implementation for current computing platforms and systems, many specialized formats are also designed to reduce memory I/O via index compression. Blocking is a common extension of the storage formats above, where only a single index is used to locate a small, dense block of matrix entries [20]. The yaSpMV BCCOO format is block-compressed COO variation that uses bit flags to store the row indices in line with column indices [21]. Other sophisticated compression methods attempt to optimize the bit-encoding of the matrix, often at the expense of significant formatting overhead [22, 23].

Instead of storing the whole matrix in one specific format, another commonplace scheme called hybrid partitions the matrix into separate regions, each of which may be stored in a different format. The HYB format combines an ELL portion with a COO portion [17]. Also multi-level format is very popular in this research area. The pOSKI autotuning framework explores a wide set of multi-level blocking schemes [24]. The compressed sparse block format is a nested COO-of-COO representation where tuples at both levels are kept in a Morton Z-order [25].

However, increasing the SpMV performance through innovative matrix formatting brings huge real costs. For most applications, general-purpose encoding such as CSR format, is preferable for in-memory representation, compared to other custom codings. CSR coding has no architecture-specific blocking, reordering, comments, etc., which means it has high portability for different platforms and systems. In addition, CSR is the most popular format used in SpMV operations due to its space efficiency and fast access latency. It consists of three arrays: ptr, indices, and data,



$$\begin{array}{c|cccccccccccccccc} \textbf{Column} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ \hline & 12 & 0 & 0 & 6 & 0 & 0 & 0 & -8 & 0 & 0 & 17 & 0 & 0 & 5 & 0 & 0 \\ & 0 & 0 & -9 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ & 0 & 0 & 0 & 0 & -23 & 0 & 0 & 56 & 0 & 0 & 0 & 19 & 0 & -3 & 0 & 0 \\ & 1 & 0 & 1 & 0 & 1 & 0 & -2 & 0 & 0 & 25 & 0 & 0 & 16 & 0 & 8 & 0 \\ & 0 & 3 & 0 & 0 & 17 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 6 & 0 \end{array}$$

$$\downarrow$$

$$\textbf{Ptr} = \begin{bmatrix} 0 & 5 & 8 & 12 & 18 & 22 \end{bmatrix}$$

$$\textbf{Indices} = \begin{bmatrix} 0 & 3 & 7 & 10 & 13 & 2 & 6 & 12 & 4 & 7 & 11 & 13 & 0 & 2 & 6 & 9 & 12 & 14 & 1 & 4 & 9 & 13 \end{bmatrix}$$

$$\textbf{Data} = \begin{bmatrix} 12 & 6 & -8 & 17 & 5 & -9 & 7 & 8 & -23 & 56 & 19 & -3 & 1 & 1 & -2 & 25 & 16 & 8 & 3 & 17 & 1 & 1 \end{bmatrix}$$

Figure 2.1: Example of the CSR storage format. It consists of three arrays: *ptr* stores the cumulative number of non-zero elements up to each row, *indices* stores the column indices of the non-zero elements, and *data* stores the values of non-zero elements.

which store the cumulative number of non-zero elements up to each row, the column indices of the non-zero elements, and the values of non-zero elements, respectively. Given a sparse matrix A, its CSR representation is illustrated in Figure 2.1.

2.1.2 Related Work

SpMV is typically a memory-bound kernel for the majority of sparse matrices on multi- and many-core platforms. In particular, SpMV is characterized by a very low flops/byte ratio, since the kernel performs $O(NNZ)$ operations on $O(N{+}NNZ)$ amount of data, indirect memory references as a result of storing the matrix in a compact format and irregular memory accesses to the right-hand side vector due to sparsity. Its bandwidth utilization is strongly dependent on the sparsity patterns of the matrix and the underlying computing platform.

Consequently, most optimization efforts proposed in the literature over the past years have focused on reducing traffic between caches and main memory, primarily by compressing the memory footprint of the matrix using segmented reduction [17], compressed block [25, 26], ELL variant [27] and data prefectching [28].

With the advent of many-core architectures, including GPGPUs and the Intel Xeon

Phi [29] processors, the performance landscape has become more diverse. For example, the large number of cores available makes SpMV performance more sensitive to workload distribution between cores/threads. As the performance of SpMV becomes more and more dependent on the matrix structure and the underlying computing platform, there is no universal solution to achieve high performance. Blindly applying optimization can actually hinder performance, which further emphasizes the value of picking and optimizing SpMV.

Even though each optimization can achieve significant gain for some matrices, it may cause non-negligible slowdowns in others. Optimizing SpMV typically involves preprocessing steps to analyze the sparse matrix structure, and may include format conversion, parameter adjustment, etc. Although the overhead of this step can be amortized for applications that reuse the same matrix multiple times, it can outperform any performance when the solver converges with fewer iterations, which is often the case in preconditioned solvers [30], or when the structure of the matrix changes frequently, e.g. in graph applications. This is why many recent efforts have focused on designing more lightweight optimizations [31, 32].

As variation increases, the row-based CsrMV implementations within Intel MKL [33] and NVIDIA cuSPARSE [34] are progressively unable to map their workloads equitably across parallel threads. In this direction, many clever partitioning and redistribution methods have been proposed to improve performance stability for all sparse matrices, and reduce overhead to make it suitable for solvers that require small number of iterations to convergence. These methods rely on each processor's ability to effectively access system I/O, shared memory, or processors other than neighbors. This is a reasonable expectation for multi-core arrays, but not always for large many-core arrays.

A high throughput SpMV called the Merge-Based SpMV was created for the GPU [7]. It presents a strictly balanced method that operates directly upon the CSR format without preprocessing for the parallel computation of SpMV. Regardless of the sparse structure patterns, its fair decomposition based on 2D merging strictly limits the amount of work assigned to each processing element while traditional CsrMV methods often subject to orders-of-magnitude performance changes in similarly sized data sets. In contrast, its approach provides predictable performance that is substantially uncorrelated with the non-zero distribution among rows , which is also the main reason that this method is chosen as the main benchmark when comparing to the implementations on many-core platform since it broadly improves upon the performance stability of current CsrMV methods.

For extended reading, Williams et al. explore a range of methods for multi-core general-purpose processors [5, 35], and Filippone et al. provide a comprehensive survey of SpMV techniques for GPUs [36].

2.2 Targeted Many-Core Processor: AsAP3

This thesis proposes SpMV implementations on a large array of processors that only communicate with the nearest neighbor and have limited long-distance communication. Only the processor at the edge of the array can access the chip I/O, and the chip contains 12 explicit global shared memories. Due to local communication and arbitrarily accessible large shared memory, the proposed many-core SpMV designs are limited to streaming data through the array.

The results from the SpMV were measured from the third generation of the Asynchronous Array of Processors (AsAP3), or KiloCore, developed at the University of California, Davis, VLSI Computation Laboratory by Bohnenstiehl et al. [1, 37–40]. In AsAP3, each core, each packet router inside each core, and each independent memory module contains its own local programmable clock oscillator in an independent fully synchronous clock domain [41, 42]. Communication on chip is accomplished by two complementary means: a very high-throughput and low-latency circuit-switched network [43] and a very-small-area packet router [44, 45]. Each circuit or packet link terminates in a dual-clock FIFO memory [46]. Diverse applications have been implemented in AsAP3, including Advanced Encryption Standard (AES) encryption [47], 4095-bit code length low-density parity-check (LDPC) decoders [48], 100-B database record sorting [49], 32-bit floating point [50]. More basic information of AsAP3 can be found from the original generations, AsAP [41, 51–55] and AsAP2 [56–62]. Energy efficient 32-bit adder design [63] and method for transient frequency distortion compensation [64] can also be good points of entry. A block diagram showing the AsAP3 chip, highlighting the communication, can be found in Figure 2.2.

AsAP3 is an example of a fine-grained many-core system with a fixed-point data path and has 1000 simple independently-clocked homogeneous programmable processors and routers, which is best suited for computationally-intensive applications and kernels. Using the AsAP3 many-core architecture, its simplified instruction set architecture with 72 40-bit instructions, 128×40-bit

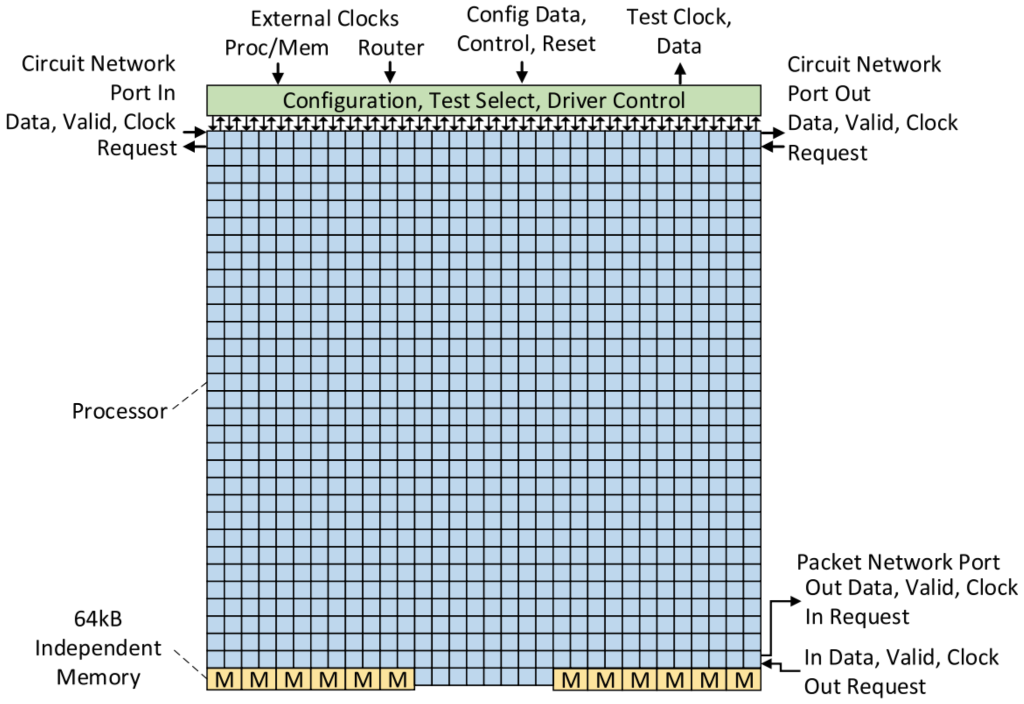


Figure 2.2: Block diagram of the KiloCore chip, containing 1000 independent processors and 12 shared memories. Both of circuit and packet router network chip I/O are highlighted, showing the communication between processors [1].

of instruction memory, 512 bytes of data memory, and two 128-bytes FIFOs for inter-processor communication across clock boundaries per processor are the main considerations for implementing applications on it.

Each processor implements a 16-bit fixed point data path, including a MAC unit having a 40-bit accumulator, and a pair of processors adjacent to the shared memory can transfer data through the memory. Although the natural word width of the data paths and memories is 16-bit, through software other word widths are easily handled, for example, 32-bit floating point [50] and 10-Bytes sorting keys for 100-Bytes data records [49]. Each processor has its own clock that uses a specific frequency depending on the workload, and can turn off its oscillator when it stalled, so that the processor consumes negligible energy. For modeling purposes, the traits from the fabricated 32 nm PD-SOI CMOS chip were physically measured, where each processor takes up 0.055 mm^2 of area, each shared memory occupies 0.164 mm^2 of area, and can operate up to a maximum clock frequency of 1.70–1.87 GHz at 1.10 V [56]. Figure 2.3 is a KiloCore chip die micrograph showing

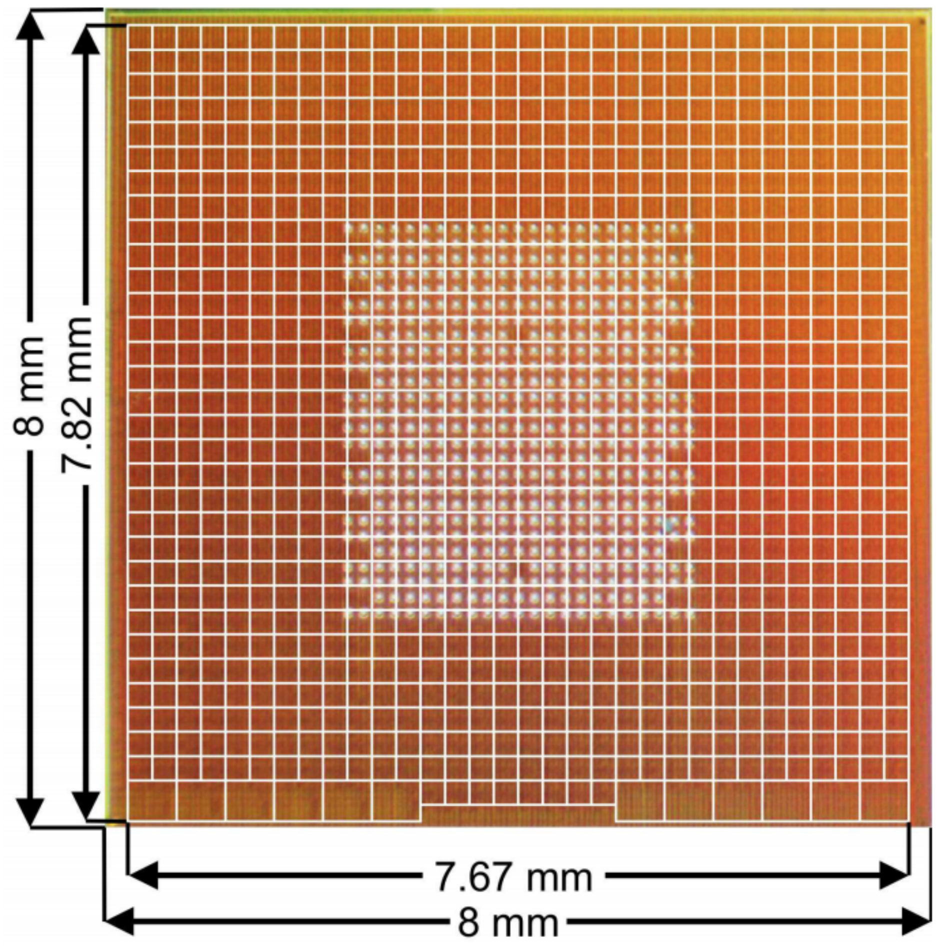


Figure 2.3: KiloCore Chip Micrograph [1].

outlines of the 1000 cores and 12 independent memories, and 564 C4 solder bumps for flip-chip mounting in the center of the array.

2.2.1 Scaling to Many-Core

Challenges of implementing SpMV on large processor arrays include balancing the computational load among the processors and transmitting the data between processors and memories. Many matrices don't scale well with more cores because of these challenges. To get around this issue, many SpMV methods on large scale array processor, such as GPU SpMV algorithms, take advantage of shared caches or shared memories to easily access the dense vector, and have independent threads work on different parts of one row in parallel [7].

For AsAP3, calculating the partial products of rows is computationally simple in a processor array, all the processing cores are either responsible for processing independent parts of one row or mapped into a data path to perform one part of the sorting network to pass the sparse data points to the processing array. However, as array becomes larger, high-bandwidth long-range communications and access to shared memory become more difficult, and in some cases, are removed to achieve more processing areas and processing units, as done in the many-core array by Truong et al. [56].

Another main problem is that many commonly used compressed matrix formats do not scale well to a many-core architecture because they require long extra preprocessing time, or mapping their workloads equitably across parallel threads/cores. These more complex formats do not apply to the many-core array architecture and are not explored in this thesis. Our target architecture is low power, small area, many-core systems, thus eliminating more complex matrix compression methods such as Blocking Coordinate format (BCCOO).

2.2.2 Basic Methodology

Section 3.2 explores implementing SpMV on a many-core platform as the size of the matrix is scaled to include a different number of processors and shared memories. All implementations are simulated using the following methods.

Simulation Methodology

Before starting the simulation, intuitive *matlab* programs were utilized to create CSR format based matrices as the input data for all simulations, and then a simulator was used to model an arbitrarily sized many-core array. The scalable multi-core architecture using two loop-based state simulators was written in C++ by Bohnenstiehl et al. [1]. Each processor core was emulated as a separate task, allowing multi-threaded operations. The core simulation is synchronized based on the timing of data transmission through the modeling circuit network.

The original method previously presented of implementing SpMV on many-core processors was also implemented on the AsAP3 platform [65]. These SpMV kernels were designed to be modular and work on a large array of processors with only data memory within each processor. Expanding on previous work, both of data memory within each processor and the independent shared memory module have been gathered together for dealing with larger matrix and improving throughput per watt versus throughput per area. Two scaling methods were explored: adding more processors to the array and expanding the amount of shared large memory. The latter provides increased I/O bandwidth and storage capacity for loading vector at the expense of requiring more area usage.

Measurement Methodology

Power usage is estimated based on physical chip measurements for each type of operation, memory accesses, network access, oscillator activity, and leakage. These figures are scaled for voltage based on measured scaling characteristics.

Area usage is physically measured from the fabricated 32 nm PD-SOI CMOS chip, where each processor takes up 0.055 mm^2 of area, and each shared memory occupies 0.164 mm^2 of area.

Comparison Methodology

With the uniqueness of the computation platform and compressed matrix format, it is impossible to directly and fairly compare SpMV on the many-core platform with other published SpMV implementations. So that I am looking specifically at CSR format based sparse matrices, chosen from the University of Florida sparse matrix collection [2], a large and actively growing set of sparse matrices that arise in real applications.

The state-of-the-art SpMV benchmark is the merge-based CsrMV, created by Duane Merrill et al. [7], as it measures the total throughput of all the matrices from the University of Florida sparse matrix collection [2]. It conducts the broadest SpMV evaluation to date to demonstrate the practical shortcomings of exiting CsrMV implementations on real-world data.

Therefore, it would be uninformative to try to add a system power to our SpMV results, so that only the processing power was counted for all implementations. With the above constraints in mind, I created SpMV kernels to compare simulation results against common unoptimized CSR format based SpMV kernels from merge-based CsrMV [7], which were implemented on two platforms, a laptop general-purpose processor and a laptop GPU.

For the general-purpose processor implementation, the SpMV program was run on the Intel Core-i7 3720QM 22 nm chip with a clock frequency of 2.6 GHz, die size of 160 mm^2 and a TDP of 45 W.

For the GPU implementation, the SpMV program was run on the Nvidia Quadro K620 28 nm chip with a clock frequency of 1.05 GHz, die size of 148 mm^2 and a TDP of 41 W.

2.3 Matrix Database

The role of sparse matrices from real applications in the development, testing, and performance evaluation of sparse matrix algorithms has long been recognized. Almost all research articles that include the performance analysis section of the sparse matrix algorithm include matrices from actual applications or the simulations of parametric matrices that may actually occur.

Large, easy-to-access and actively growing sparse matrix collections in real-world applications are critical to the development and testing of sparse matrix algorithms. The University of Florida sparse matrix collection [2] meets this need and is the largest and most widely used collection available.

A complete list of these groups is too long to include here, there are a summary of numbers of problems for different kinds of applications in Table 2.1, in which there are total 2272 matrices in the collection coming from 359 different authors and 50 different editors.

Figure 2.4, 2.5 plot the matrix size dimension and NNZ of total 2272 matrices coming from real life versus the year when the matrices were created.

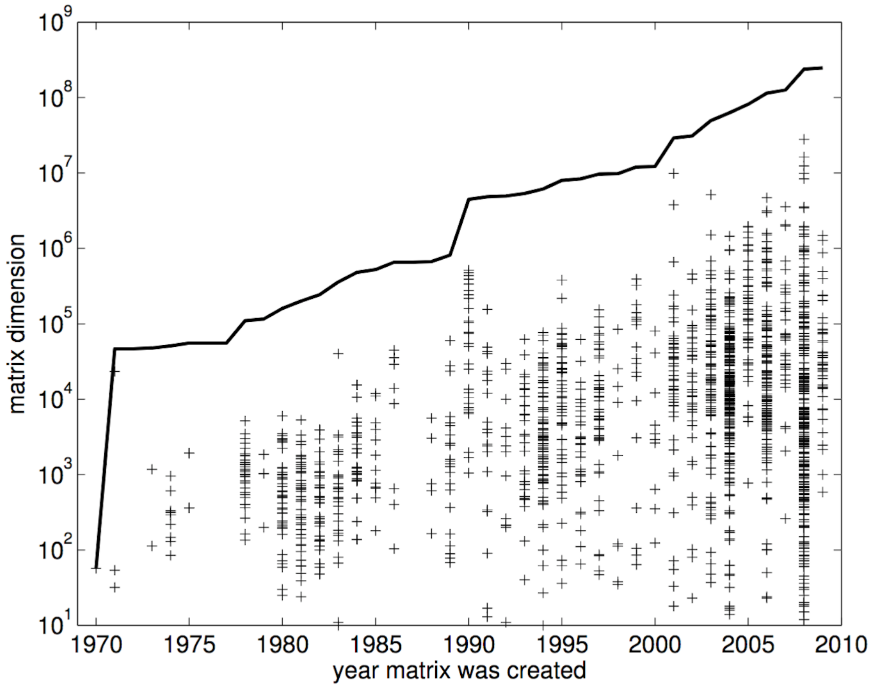


Figure 2.4: Matrix dimension (the largest of row/column dimension if rectangular) versus years created. The solid line is the cumulative sum [2].

Table 2.1: Summary of all matrices from the University of Florida Sparse Matrix Collection.

1516		**problems with no 2D/3D geometry**
	342	linear programming problem
	299	combinatorial problem
	251	circuit simulation problem
	135	optimization problem
	88	directed graph
	70	chemical process simulation problem
	68	economic problem
	68	random problem
	61	theoretical/quantum chemistry problem
	56	power network problem
	23	least squares problem
	23	undirected graph
	11	counter-example problem
	10	statistical/mathematical problem
	8	bipartite graph
	4	frequency-domain circuit simulation problem
756		**problems with 2D/3D geometry**
	288	structural problem
	166	computational fluid dynamics problem
	94	2D/3D problem (other than those listed elsewhere)
	44	electromagnetics problem
	42	model reduction problem
	35	semiconductor device problem
	31	thermal problem
	28	materials problem
	13	acoustics problem
	12	computer graphics/vision problem
	3	robotics problem

Figure 2.6 shows the distribution of matrix dimensions and nonzeros more clearly. Most of the matrix dimensions are between 3^{10} and 5^{10}, and NNZ are normally between 4^{10} and 6^{10}. The largest matrix in the collection has a dimension of 28 million with 760 million nonzeros. All 27 sparse matrices used through this thesis are obtained from the University of Florida sparse matrix collection [2], the largest and most widely used collection of sparse matrices from a wide range of domains, including linear programming, circuit simulation, directed graph, undirected weighted random graph and combinatorial problem [2]. The sparsity patterns of all matrices used to evaluate SpMV performance are shown in Table 2.2, which is sorted in matrix column size. Matrix and

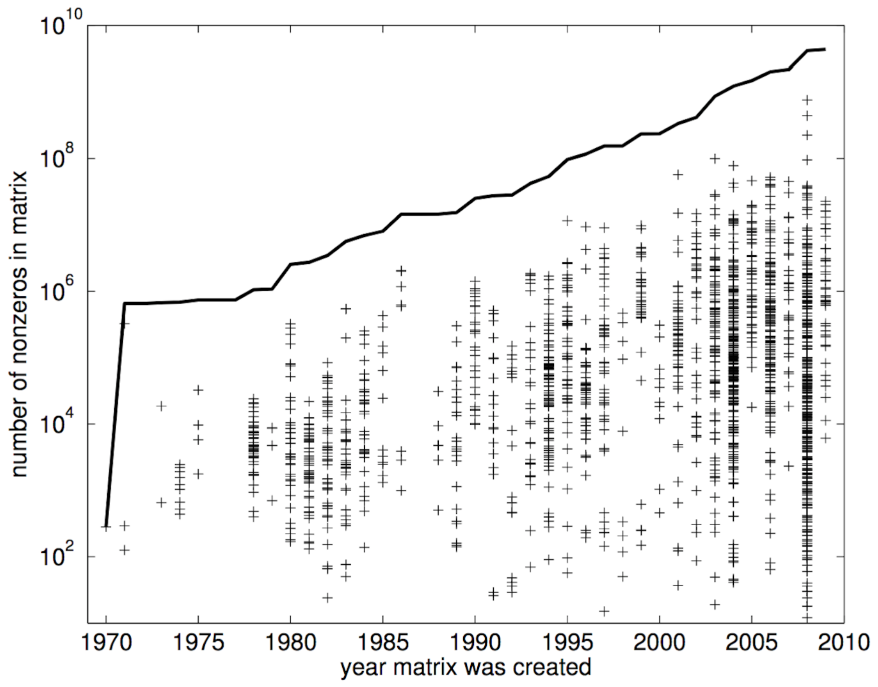


Figure 2.5: Number of nonzeros per matrix versus year created. The solid line is the cumulative sum [2].

vector data of 17 integer matrices is in 16-bit fixed point format and the other 10 real matrices is in single-precision 32-bit IEEE-754 format. The median column size of all simulated matrices is 6184, while the average density is 0.00943.

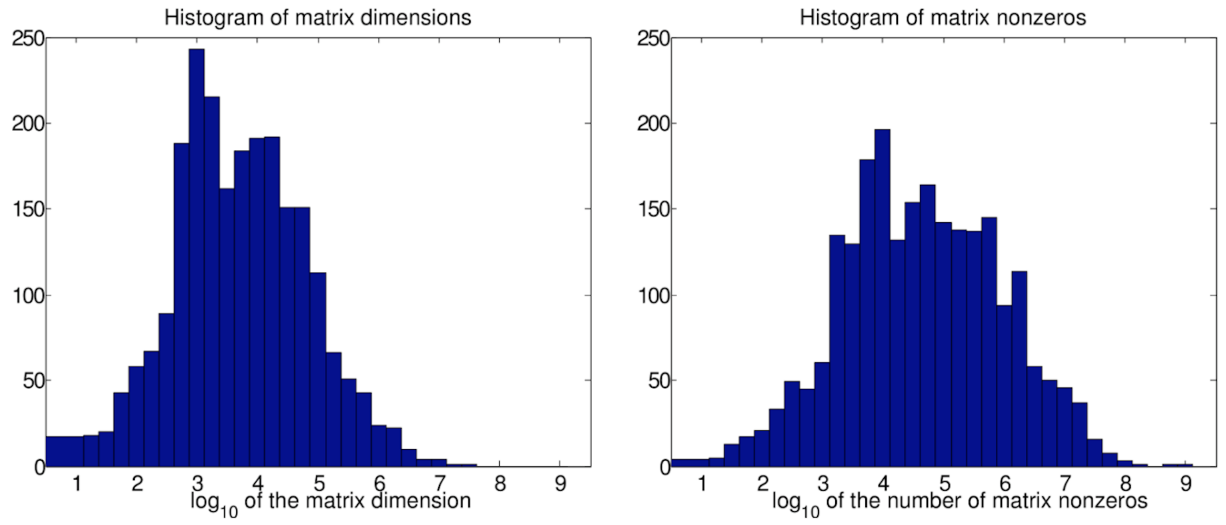


Figure 2.6: Overall histogram of matrix dimensions and nonzeros. Matrix dimension is the largest of row/column dimension if rectangular [2].

Table 2.2: Data for the 27 2-dimensional sparse matrices used in the benchmarking throughout this thesis, including the name, dimensions, number of non-zero elements, average number of non-zero elements per row, data type, data format and word width.

Matrix Name	Row Size	Column Size	Non-zeros Number	NNZ /Row	Data Type	Data Format	Word Width
GD01-a	311	311	645	2.07	real	floating point	32 bit
GD97-c	452	452	460	1.02	real	floating point	32 bit
GD00-c	638	638	1041	1.63	real	floating point	32 bit
G10	800	800	38352	47.94	integer	fixed point	16 bit
GD01-Acap	953	953	645	0.68	real	floating point	32 bit
GD96_a	1096	1096	1677	1.53	real	floating point	32 bit
Trec11	235	1138	35705	151.94	integer	fixed point	16 bit
Trec12	551	2726	151219	274.44	integer	fixed point	16 bit
complex	1023	1408	46463	45.42	real	floating point	32 bit
Rosen2	1032	3080	47536	46.06	integer	fixed point	16 bit
Franz9	19588	4164	97508	4.98	real	floating point	32 bit
C8-mat11	4562	5761	2462970	539.89	integer	fixed point	16 bit
cis-n4c6-b4	20058	5970	100290	5	integer	fixed point	16 bit
Ip_d6cube	415	6184	37704	90.85	integer	fixed point	16 bit
Trec13	1301	6561	654517	503.09	integer	fixed point	16 bit
IG5-14	6735	7521	173337	25.74	integer	fixed point	16 bit
pcb3000	3960	7732	57479	14.51	integer	fixed point	16 bit
p6000	2095	7967	19826	9.46	integer	fixed point	16 bit
foldoc	13356	13356	120238	9	integer	fixed point	16 bit
EAT_RS	23219	23219	325592	14.02	integer	fixed point	16 bit
IG5-17	30162	27944	1035008	34.31	integer	fixed point	16 bit
as-caida	31379	31379	106762	3.4	real	floating point	32 bit
TF17	38132	48630	586218	15.37	integer	fixed point	16 bit
ch7-7-b5	35280	52920	211680	6	integer	fixed point	16 bit
rail582	582	56097	402290	691.22	real	floating point	32 bit
Andrews	60000	60000	760154	12.67	integer	fixed point	16 bit
rail507	507	63516	409856	808.39	real	floating point	32 bit

Chapter 3

Implementations of Sparse Matrix-Vector Multiplication on a Many-Core Platform

3.1 Sparse Matrix-Vector Multiplication Kernels

The SpMV implementations, described in Section 3.2, utilize basic program kernels in each processor of the array. Each kernel is designed for modularity, so the kernels can be easily used in any processor in any part of the array, making it easy to scale. Distributing, sorting, or processing kernels do not require specific information or knowledge about processor location or run size, so it's easy to program different implementations. Each processor on the target platform has a 128×40-bit instruction memory, limiting each kernel to just 128 assembly instructions. Three main SpMV scenarios including eight different implementations are described in this chapter, as shown in the following list:

1. *BigMemSnake.*
    - *BigMemSnake.*
2. *BigMemPara.*
    - *BigMemParaOne.*

- *BigMemParaTwo.*
- *BigMemParaTwoNnz.*
- *BigMemParaFour.*
- *BigMemParaFourPad.*

3. *BigMemSubPara.*

   - *BigMemSubParaFour.*
   - *BigMemSubParaFourTable.*

BigMemSnake includes only one implementation and it is itself. *BigMemPara* includes five implementations: *BigMemParaOne*, *BigMemParaTwo*, *BigMemParaTwoNnz*, *BigMemParaFour* and *BigMemParaFourPad*. *BigMemSubPara* includes two implementations: *BigMemSubParaFour* and *BigMemSubParaFourTable*. The naming of each different implementation depends on the name of the scenario, the number of rows in the processing array, and different kernels used. For example, *BigMemParaFourPad* represents the implementation using four processing arrays with Padding NNZ Distribution kernel, based on *BigMemPara* scenario.

3.1.1 Matrix Preprocessing

Since the original matrix data has a lot of zero elements without any compression, before starting the simulation, intuitive *matlab* programs were utilized to create CSR format based matrices as the input data to the simulator for all implementations.

Snake Preprocessing

The Snake preprocessing program generates CSR format based matrices as the input data to the implementation *BigMemSnake*. As shown in Algorithm 1, it first generates a dense vector with the same length as the matrix column size. For simplify, the vector items are all set to 1's, loading to output *Matrix*. Then all non-zero column indices and values are loaded to output *Matrix* one by one. For each nonzero, its row index is compared with the previous one. Once the row index of current non-zero is different from the previous one, which means the end of current row has been achieved, then a token is added to the output *Matrix* as a symbol for the end of this row. Multiple tokens are added sequentially if there are continuous rows without any non-zero. For simplify, the value of token is set equal to column size of the matrix since all indices are zero-based in all the implementations.

Algorithm 1 Pseudocode of *Snake* Preprocessing

```
$MatrixCol, MatrixRow, MatrixVal \leftarrow Find(Matrix)$
$ColSize, RowSize \leftarrow Size(Matrix)$
for ($i = 0; i \leq ColSize; i{+}{+}$) do
    $Matrix[i] \leftarrow 1$                                   # Store vector $X$ values
end for
$k \leftarrow 1$
for ($j = 1; j \leq length(MatrixCol); j{+}{+}$) do
    if $MatrixRow[j] == k$ then
        $i \leftarrow i{+}1;$                                  # Store matrix $A$ values of current row
        $Matrix[i] \leftarrow MatrixCol[j] - 1;$
        $i \leftarrow i + 1;$
        $Matrix[i] \leftarrow MatrixVal[j];$
    else
        for ($q = 1; q \leq MatrixRow[j] - k; q{+}{+}$) do
            $i \leftarrow i{+}1;$                              # Store *token* at the end of row
            $Matrix[i] \leftarrow Colsize;$
        end for
        $i \leftarrow i{+}1;$                                  # Store matrix $A$ values of next row
        $Matrix[i] \leftarrow MatrixCol[j] - 1;$
        $i \leftarrow i + 1;$
        $Matrix[i] \leftarrow MatrixVal[j];$
        $k \leftarrow MatrixRow[j];$
    end if
end for
$i \leftarrow i + 1;$
$Matrix[i] \leftarrow Colsize;$                                # Store *token* at the end of row
```

SubParallel Preprocessing

The SubParallel Preprocessing program generates CSR format based matrices as the input data to the implementation *BigMemParaOne*, *BigMemSubParaFour* and *BigMemSubParaFourTable*. As shown in Algorithm 2, it first generates a dense vector which length is the same as matrix column size. For simplify, the vector items are all set to 1's, loading to output *Vector*. Then all non-zero column indexes and values are loaded to output *Matrix* one by one. At last, the NNZ per row is accumulated to output *NNZPerRow* depending on the row index of each non-zero.

Algorithm 2 Pseudocode of *SubParallel* Preprocessing

```
$MatrixCol, MatrixRow, MatrixVal \leftarrow Find(Matrix)$
$ColSize, RowSize \leftarrow Size(Matrix)$
for $(i = 0; i \leq ColSize; i{+}{+})$ do
    $Vector[i] \leftarrow 1$                                  # Store vector $X$ values
end for
$i \leftarrow 1;$
for $(j = 1; j \leq length(MatrixCol); j{+}{+})$ do
    $Matrix[i] \leftarrow MatrixCol[j]-1;$                   # Store matrix $A$ values of current row
    $i \leftarrow i + 1;$
    $Matrix[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
end for
for $(j = 1; j \leq length(MatrixRow); j{+}{+})$ do
    $NNZ[MatrixRow[j]] \leftarrow NNZ[MatrixRow[j]] + 1;$
end for                                                      # Store $NNZ$ per row
```

ParallelTwo Preprocessing

The ParallelTwo Preprocessing program generates CSR format based matrices as the input data to the implementation *BigMemParaTwo* and *BigMemParaTwoNnz*. As shown in Algorithm 3, it first generates a dense vector, which length is the same as matrix column size. For simplify, the vector items are all set to 1's, loading to output *Vector*. Then all non-zero column indexes and values are divided into two groups in a round-robin order. The first and second groups are loaded to output *Matrix1* and *Matrix2* one by one. At Last, the NNZ per row is accumulated to output *NNZPerRow* depending on the row index of each non-zero.

Algorithm 3 Pseudocode of *ParallelTwo* Preprocessing

```
$MatrixCol, MatrixRow, MatrixVal \leftarrow Find(Matrix)$
$ColSize, RowSize \leftarrow Size(Matrix)$
for $(i = 0; i \leq ColSize; i{+}{+})$ do
    $Vector[i] \leftarrow 1$                                  # Store vector $X$ values
end for
$i \leftarrow 1;$
for $(j = 1; j \leq length(MatrixCol); j{+}2)$ do
    $Matrix1[i] \leftarrow MatrixCol[j]-1;$ # Distribute and Store matrix $A$ values for $1^{st}$ processing row
    $i \leftarrow i + 1;$
    $Matrix1[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
end for
$i \leftarrow 1;$
for $(j = 2; j \leq length(MatrixCol); j{+}2)$ do
    $Matrix2[i] \leftarrow MatrixCol[j]-1;$ # Distribute and Store matrix $A$ values for $2^{nd}$ processing row
    $i \leftarrow i + 1;$
    $Matrix2[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
end for
for $(j = 1; j \leq length(MatrixRow); j{+}{+})$ do
    $NNZ[MatrixRow[j]] \leftarrow NNZ[MatrixRow[j]] + 1;$
end for                                                # Store $NNZ$ per row
```

ParaFour Preprocessing

The ParallelFour Preprocessing program generates CSR format based matrices as the input data to the implementation *BigMemParaFour*. As shown in Algorithm 4, it first generates a dense vector, which length is the same as matrix column size. For simplify, the vector items are all set to

1's, loading to output *Vector*. Then all non-zero column indexes and values are divided into four groups in a round-robin order. Four groups are loaded to output *Matrix1*, *Matrix2*, *Matrix3* and *Matrix4* respectively.

Algorithm 4 Pseudocode of *ParallelFour* Preprocessing

```
  $MatrixCol, MatrixRow, MatrixVal \leftarrow Find(Matrix)$
  $ColSize, RowSize \leftarrow Size(Matrix)$
  for ($i = 0; i \leq ColSize; i{+}{+}$) do
    $Vector[i] \leftarrow 1$                                # Store vector $X$ values
  end for
  $i \leftarrow 1;$
  for ($j = 1; j \leq length(MatrixCol); j{+}4$) do
    $Matrix1[i] \leftarrow MatrixCol[j]-1;$ # Distribute and Store matrix $A$ values for 1st processing row
    $i \leftarrow i + 1;$
    $Matrix1[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
  end for
  $i \leftarrow 1;$
  for ($j = 2; j \leq length(MatrixCol); j{+}4$) do
    $Matrix2[i] \leftarrow MatrixCol[j]-1;$ # Distribute and Store matrix $A$ values for 2nd processing row
    $i \leftarrow i + 1;$
    $Matrix2[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
  end for
  $i \leftarrow 1;$
  for ($j = 3, 4; j \leq length(MatrixCol); j{+}4$) do
    $Matrix3[i], Matrix4[i] \leftarrow MatrixCol[j] - 1;$
    $i \leftarrow i{+}1;$                # Distribute and Store matrix $A$ values for 3rd & 4th processing rows
    $Matrix3[i], Matrix4[i] \leftarrow MatrixVal[j];$
    $i \leftarrow i + 1;$
  end for
```

ParaFourPad Preprocessing

The ParallelFourPad Preprocessing program generates CSR format based matrices as the input data to the implementation *BigMemParaFourPad.* As shown in Algorithm 5, it first generates a dense vector, which length is the same as matrix column size. For simplify, all vector items are set to 1's, loading to output *Vector.* Then all non-zero column indexes and values of each row are divided into four groups in a round-robin order.

If the NNZ can be divided by 4, each group is loaded one by one to the corresponding output. Otherwise, the column index token (*ColSize*) and value *zero* are added to the appropriate group in a round-robin order, depending on the NNZ of each row.

Finally, the existing NNZ and *zero* added per row are accumulated to output *NNZPerRow* depending on row index of each non-zero.

3.1.2 Nonzero Distribution

The Nonzero distribution kernel, shown in Algorithm 6, obtains NNZ per row from matrix preprocessing and determines the NNZ per row that each processing array receives. The NNZ of each row is not always divisible by the number of rows processed, and is distributed to the processing row according to:

$$k = (NNZ \ \% \ NumRows) \tag{3.1}$$

$$NNZDis[j] = [\frac{NNZ}{NumRows}] + (j < k); \quad j = i, ..., (i + k) \ \% \ NumRows \tag{3.2}$$

$$i = (i + NNZ \ \% \ NumRows) \ \% \ NumRows \tag{3.3}$$

where *NNZDis*[*j*] is the number of elements distributed to the processing row j. The remainder of elements k after distribution is added to the processing row based on previous distribution results. The starting point (row i) that processes the next row distribution is determined based on the addition result of the current starting point and the remainder of the previous row.

For example, the Algorithm 6 indicates that for two-array processing, the NNZ of each row is divided by two, and the rest is added to one of the two processing rows in a round-robin order, depending on the previous allocation order.

Algorithm 5 Pseudocode of *ParallelFourPad* Preprocessing (continued on the next page)

```
$MatrixCol, MatrixRow, MatrixVal \leftarrow Find(Matrix)$
$ColSize, RowSize \leftarrow Size(Matrix)$
for $(i = 0; i \leq ColSize; i{+}{+})$ do
    $Vector[i] \leftarrow 1$                                   # Store vector $X$ values
end for
for $(j = 1; j \leq length(MatrixRow); j{+}{+})$ do
    $NNZ[MatrixRow[j]] \leftarrow NNZ[MatrixRow[j]] + 1;$
end for                                                  # Store $NNZ$ per row
$a1, a2, a3, a4 \leftarrow 1;$
for $(j = 1; i \leq RowSize; j{+}{+})$ do
    if $NNZ[j] == 0$ then                               # Padding empty row
        $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}], Matrix3[a3{+}{+}], Matrix4[a4{+}{+}] \leftarrow ColSize;$
        $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}], Matrix3[a3{+}{+}], Matrix4[a4{+}{+}] \leftarrow 0;$
    else
        for $k = 1; k \leq floor(NNZ[j]/4); k{+}{+}$ do
            $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}] \leftarrow MatrixCol[i, i + 1] - 1;$
            $Matrix3[a3{+}{+}], Matrix4[a4{+}{+}] \leftarrow MatrixCol[i + 2, i + 3] - 1;$
            $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}] \leftarrow MatrixVal[i, i + 1];$
            $Matrix3[a3{+}{+}], Matrix4[a4{+}{+}] \leftarrow MatrixVal[i + 2, i + 3];$
            $i \leftarrow i{+}4$                # Distribute and Store matrix $A$ values for 4 processing rows
        end for
```

Algorithm 5 Pseudocode of *ParallelFourPad* Preprocessing (continued)

```
if $mod(NNZ[j], 4) == 1$ then
    $Matrix1[a1] \leftarrow MatrixCol[i] - 1;$
    $a1 \leftarrow a1 + 1;$
    $Matrix1[a1] \leftarrow MatrixVal[i];$
    $a1 \leftarrow a1 + 1;$
    $i \leftarrow i{+}1$                # Padding and Store matrix $A$ values when $NNZ$ remainder is 1
    $Matrix2[a2], Matrix3[a3], Matrix[a4] \leftarrow ColSize;$
    $a2, a3, a4 \leftarrow a2 + 1, a3 + 1, a4 + 1;$
    $Matrix2[a2], Matrix3[a3], Matrix[a4] \leftarrow 0;$
    $a2, a3, a4 \leftarrow a2 + 1, a3 + 1, a4 + 1;$
end if
if $mod(NNZ[j], 4) == 2$ then
    $Matrix1[a1], Matrix2[a2] \leftarrow MatrixCol[i, i + 1] - 1;$
    $a1, a2 \leftarrow a1 + 1, a2 + 1;$
    $Matrix1[a1], Matrix2[a2] \leftarrow MatrixVal[i, i + 1];$
    $a1, a2 \leftarrow a1 + 1, a2 + 1;$
    $i \leftarrow i{+}2$                # Padding and Store matrix $A$ values when $NNZ$ remainder is 2
    $Matrix3[a3], Matrix4[a4] \leftarrow ColSize;$
    $a3, a4 \leftarrow a3 + 1, a4 + 1;$
    $Matrix3[a3], Matrix4[a4] \leftarrow 0;$
    $a3, a4 \leftarrow a3 + 1, a4 + 1;$
end if
if $mod(NNZ[j], 4) == 3$ then
    $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}], Matrix3[a3{+}{+}] \leftarrow MatrixCol[i, i + 1, i + 2] - 1;$
    $Matrix1[a1{+}{+}], Matrix2[a2{+}{+}], Matrix3[a3{+}{+}] \leftarrow MatrixVal[i, i + 1, i + 2];$
    $i \leftarrow i{+}3$                # Padding and Store matrix $A$ values when $NNZ$ remainder is 3
    $Matrix4[a4{+}{+}] \leftarrow ColSize;$
    $Matrix4[a4{+}{+}] \leftarrow 0;$
end if
```

Algorithm 6 Pseudocode of *Nonzero Distribution*

```
$SendFirst:$
$go\ to\ CalculateNNZ$
$return$
$FirstOutput \leftarrow NNZdiv2{+}ExtraNNZ1$          # Store $NNZ$ per row to two processing arrays in order
$SecondOutput \leftarrow NNZdiv2 + ExtraNNZ2$
if $NNZMod2 == 0$ then
    $go\,to\,SendFirst$                                # Decide next order
end if
$SendSecond:$
$go\ to\ CalculateNNZ$
$return$
$SecondOutput \leftarrow NNZdiv2{+}ExtraNNZ1$         # Store $NNZ$ per row to two processing arrays in order
$FirstOutput \leftarrow NNZdiv2 + ExtraNNZ2$
if $NNZMod2 == 0$ then
    $go\,to\,SendSecond$                               # Decide next order
else
    $go\ to\ SendFirst$
end if
$CalculateNNZ:$
$NNZ \leftarrow Input$                                 # Distribute $NNZ$ per row to two processing arrays
$NNZdiv2 \leftarrow floor(NNZ/2)$
$NNZmod2 \leftarrow NNZ - NNZdiv2 * 2$
if $NNZMod2 == 0$ then
    $ExtraNNZ1 \leftarrow 0$
    $ExtraNNZ2 \leftarrow 0$
else
    $ExtraNNZ1 \leftarrow 1$
    $ExtraNNZ2 \leftarrow 0$
end if
$return$
```

3.1.3 Core Boundary

The Core Boundary kernel, shown in Algorithm 7, obtains column size of matrix from matrix preprocessing and determines the column boundary and number of vector items that each processing core stores. Vector length (*ColSize*) is not always divisible by the number of processing cores, and is distributed among processing cores according to:

$$k = (\mathit{ColSize}\ \%\ \mathit{NumRows}) \tag{3.4}$$

$$\mathit{RowArray}[j] = [\frac{\mathit{ColSize}}{\mathit{NumRows}}] + (j < k); \quad j = 0, ..., \mathit{NumRows} \tag{3.5}$$

where *ColSize* is the column size of matrix, equals to the number of vector items that the processing network stores and *RowArray*[*j*] is the number of column indexes of matrix A that each processing array stores. The remainder k after distribution is added to the processing rows in order, starting from row 0.

Similarly, the elements distributed to each processing row are further evenly distributed among the cores according to:

$$k = (\mathit{RowArray}[i]\ \%\ \mathit{NumCols}); \quad i = 0, ..., \mathit{NumRows} \tag{3.6}$$

$$\mathit{CoreArray}[i][j] = [\frac{\mathit{RowArray}[i]}{\mathit{NumCols}}] + (j < k); \quad j = 0, ..., \mathit{NumCols} \tag{3.7}$$

Where *NumCols* is the number of processing cores of each processing array in the network, and *CoreArray*[*i*][*j*] is the number of column indexes of the matrix A that each processing core stores. The remaining k after the allocation is added to the processing core in order, starting from core 0.

Each processing core maintains the boundary of the column index, which it needs to handle the multiplication, given by:

$$\mathit{CoreArrayLow}[i][j] = \mathit{CoreNum}; \quad i = 0, ..., \mathit{NumRows}, \quad j = 0, ..., \mathit{NumCols} \tag{3.8}$$

$$\mathit{CoreArrayHigh}[i][j] = \mathit{CoreNum} + \mathit{CoreArray}[i][j] \tag{3.9}$$

$$\mathit{CoreNum} = \mathit{CoreNum} + \mathit{CoreArray}[i][j] \tag{3.10}$$

Where *CoreNum* is the number of column indexes that have been stored in the processing array so far. *CoreArrayLow*[*i*][*j*] and *CoreArrayhigh*[*i*][*j*] are the lowest and highest column indexes of the matrix A that each processing core stores.

To simplify loading vector processing, each processing core also retains records of the lowest column index of the matrix A it processes to make the correct decision to flush other elements to other cores. Since all implementations use data memory and large shared memory, the numbers of vector elements stored in these two kinds of memories are recorded for each processing core, given as follows:

$$NumCoreHold = CoreArrayHigh[i][j] - CoreArrayLow[i][j] \tag{3.11}$$

$$NumPassEast = CoreArrayHigh[i][j] - CoreArrayLow[i][NumCols-1] \tag{3.12}$$

$$DMemHold = 240 \tag{3.13}$$

$$BigMemHold = NumCoreHold - DMemHold \tag{3.14}$$

For example, pseudocode 7 shows that for three processing cores, the number of column indexes per row is divided by three, and the rest is added sequentially from the rightmost core to the leftmost core.

Algorithm 7 Pseudocode of *Core Boundary*

```
$NumPerRow \leftarrow floor(ColSize/NumRow)$
for $(i = 0; i < NumRow; i{+}{+})$ do
  $RowArray[i] \leftarrow NumPerRow$                # Distribute row length to all processing arrays
end for
for $(i = 0; i < ColSize \ \% \ NumRow; i{+}{+})$ do
  $RowArray[i] \leftarrow RowArray[i]{+}1$          # Distribute extra part to the corresponding arrays
end for
for $(i = 0; i < NumRow; i{+}{+})$ do
  for $(j = 0; j < NumCol; j{+}{+})$ do
    $CoreArray[i][j] \leftarrow floor(RowArray[i]/NumCol)$
  end for                              # Distribute each array length to the whole processing array
  for $(k = 0; k < RowArray[i] \ \% \ NumCol; k{+}{+})$ do
    $CoreArray[i][k] \leftarrow CoreArray[i][k] + 1$
  end for                              # Distribute extra part to the corresponding cores
end for
for $(k = 0; k < NumRow; k{+}{+})$ do
  $CoreNum \leftarrow 0$                             # Set boundary to the corresponding cores
  for $(j = NumCol - 1; j \geq 0; j{-}{-})$ do
    $CoreArrayLow[k][j] \leftarrow CoreNum$
    $CoreArrayHigh[k][j] \leftarrow CoreNum + CoreArray[k][j] - 1$
    $CoreNum \leftarrow CoreNum + CoreArray[k][j]$
    $CoreNumLow \leftarrow CoreArrayLow[k][j]$
    $NumCoreHold \leftarrow CoreArrayHigh[k][j] - CoreArrayLow[k][j] + 1$
    if $j \neq NumCol{-}1$ then                     # Set number to send to next core
      $NumPassEast \leftarrow CoreArrayHigh[k][j] - CoreArrayLow[k][NumCol - 1]$
    end if
    $DMemHold \leftarrow 240$          # Set limitation to the corresponding data memory & shared memory
    $BigMemHold \leftarrow NumCoreHold - DMemHold$
  end for
end for
```

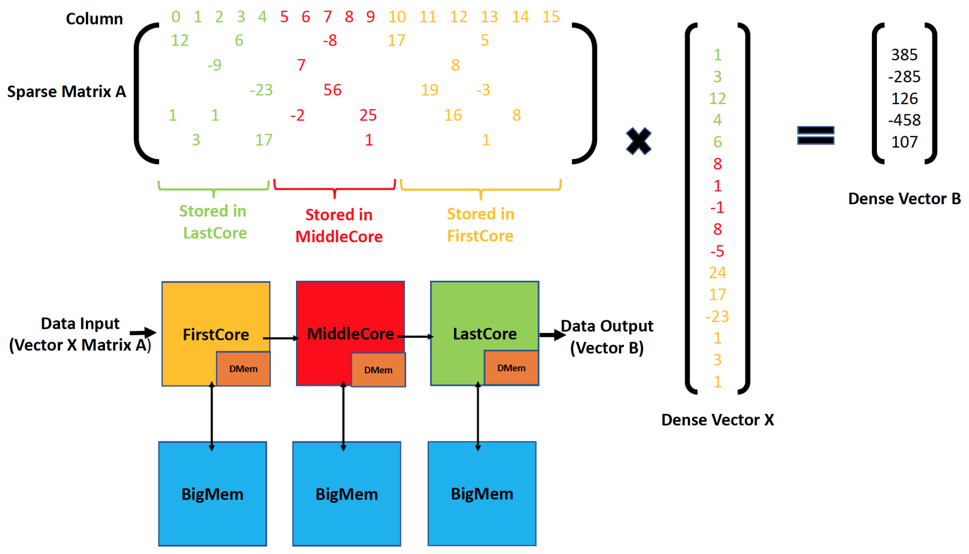


Figure 3.1: An array of 3 processing processors loading dense vector to their data memory and corresponding shared memory. The vector X values are evenly divided among three processing cores. Vector elements correspond to the lowest columns of matrix A for which rows of X are stored in the last core. Similarly, the first core stores vector elements correspond to the highest column index of A. Each core, except for the last, passes X values downstream before storing X values in memory.

3.1.4 Loading Vector into BigMem and DMem

The Loading Vector kernel shown in Algorithm 8 involves obtaining the dense vector data X from matrix preprocessing and streaming vector data through a set of cores that are connected in a chain. First, *vector* is stored in the core array (either during programming or inflow), and then flows into the A matrix data. In the process of loading the *vector* into memory, the data memory on each core is filled first, and then all other items are loaded into the large shared memory connected to the core. The total number of vector items that each core and its connected large memory hold depends on the column boundaries stored in each processing core, as shown in equations 3.11-3.14.

The example allocation of loading vector kernels is performed in 2x3 blocks, with three processing cores in each row, as shown in Figure 3.1. For simplify, the chain length here is chosen to be three because such a chain contains three basically similar processing cores. In addition to the FirstCore to get the input from the matrix preprocessing and the LastCore outputs to the off-chip, the most common MiddleCore gets input from left core and outputs to the right core. The length of loading chain can be increased by adding more MiddleCores to store more X dense vector elements.

Algorithm 8 Pseudocode of *Loading Vector* (continued on the next page)

```
$NumPassEastDiv1024 \leftarrow floor(NumPassEast/1024)$
$NumPassEastMod1024 \leftarrow NumValEast \ \% \ 1024$
if $NumPassEast < 1024$ then
  for $(i = 0; i < NumPassEast; i{+}{+})$ do
    $EastOutput \leftarrow Input1$                    # Loading vector $X$ to next core
  end for
else
  $LoadingDMem:$                                  # Loading vector $X$ to next core repeatedly
  for $(i = 0; i < 1024; i{+}{+})$ do
    $EastOutput \leftarrow Input1$
  end for
  $NumPassEastDiv1024 \leftarrow NumPassEastDiv1024 - 1$
  if $NumPassEastDiv1024 > 0$ then
    go to $LoadingDMem$
  end if
  if $NumPassEastMod1024 \neq 0$ then
    for $(i = 0; i < NumPassEastMod1024; i{+}{+})$ do
      $EastOutput \leftarrow Input1$
    end for
  end if
end if
for $(i = 0; i < DMemHold; i{+}{+})$ do
  $Dmem \leftarrow Input1$                           # Loading vector $X$ to data memory
end for
$NumBigMemDiv1024 \leftarrow floor(BigMemHold/1024)$
$NumBigMemMod1024 \leftarrow BigMemHold \ \% \ 1024$
```

3.1.5 Sorting Network

The sorting kernel shown in the Algorithm 9 involves obtaining sparse matrix data points (non-zero values and corresponding column indexes) from matrix preprocessing and sorting the matrix data through a set of cores connected in 4x2 blocks.

First, the column index, non-zero value and NNZ per row are assigned to the core array,

Algorithm 8 Pseudocode of *Loading Vector* (continued)

```
if $BigMemHold < 1024$ then
    for $(i = 0; i < BigMemHold; i{+}{+})$ do
        $BigMem \leftarrow Input1$            # Loading vector $X$ to the big shared memory
    end for
else
    $LoadingBigMem:$            # Loading vector $X$ to the big shared memory repeatedly
    for $(i = 0; i < 1024; i{+}{+})$ do
        $BigMem \leftarrow Input1$
    end for
    $NumBigMemDiv1024 \leftarrow NumBigMemDiv1024 - 1$
    if $NumBigMemDiv1024 > 0$ then
        go to $LoadingBigMem$
    end if
    if $NumBigMemMod1024 \neq 0$ then
        for $(i = 0; i < NumBigMemMod1024; i{+}{+})$ do
            $BigMem \leftarrow Input1$
        end for
    end if
end if
```

Algorithm 9 Pseudocode of *Sorting Network*

```
GetNNZPerRow :
NNZ ← Input1                                          # Get NNZ left per row
if NNZ == 0 then
  go to EndOfRow
else
  SortingBit1 :
  Bit1 ← Input0_next & 2                              # Get $2^{nd}$ LSB of each column index
  if Bit1 == 0 then
    go to Bit1IsZero
  else                            # Send column index and its corresponding data to right core
    RightOutput ← Input0
    RightOutput ← Input0
    NNZ ← NNZ − 1
    if NNZ == 0 then
      go to EndOfRow                                  # Achieve the end of the row
    else
      go to SortingBit1
    end if
  end if
end if
Bit1IsZero :
EastOutput ← Input0               # Send column index and its corresponding data to east core
EastOutput ← Input0
NNZ ← NNZ−1                                           # Decrement NNZ per row
if NNZ == 0 then
  go to EndOfRow
else
  go to SortingBit1
end if
EndOfRow :                        # Send token to two directions at the end of each row
EastOutput ← −1
RightOutput ← −1
go to GetNNZPerRow
```

and then if the second LSB of the column index of the sparse data point is 0, then the data value from A is forwarded to downstream core, otherwise it is forwarded to upstream core.

Whenever the NNZ of a row counts down to zero, a flag (-1) indicating the end of the row is sent to the downstream and upstream cores. The second LSB of each column index is given by

$$\mathit{Bit1} = \mathit{Column\ index}\ \&\ 2 \tag{3.15}$$

Where *Bit1* represents the second LSB of each column index, and 2 in decimal represents 10 in the binary.

For the second phase of the sorting network, the only difference from the first phase is that the sparse data points are sent to the downstream or upstream cores, depending on the value of the LSB of the column index of the sparse data.

After the sorting network, all sparse data points are flushed to the processing array. Once the last value of the row has been sent, a token is sent to indicate the end of the row.

3.2 Proposed SpMV Architecture Mapping

Three main SpMV scenarios were explored in this section. The following three basic SpMV implementations apply to our target architecture limitations, leveraging simple modular kernels and easily extending the processor array size. Using the different phase kernels in Section 3.3, a total of eight implementations based on three scenarios, were tested in the performance comparison part of Chapter 4 and Chapter 5.

3.2.1 BigMemSnake Accumulation

BigMemSnake is the simplest one of the proposed SpMV implementations and its pseudocode is shown in Algorithm 10. For simplify, as shown in Figure 3.2, it involves streaming data through a set of cores connected in a 1x4 block. This implementation utilizes the snake kernel on each processor in the array, linking them together using a single input and output per processor.

Each processor takes in sparse data points, including column indices, matrix values, and tokens, and then determine whether to multiply the matrix term with the corresponding vector term and refresh the partial sum, or directly flush the token. If the column index of the sparse data point matches the core boundary, then the matrix value A is multiplied by the corresponding vector x value, otherwise the sparse data point is forwarded to the right core. Once the last value of the row has been sent, a token will be sent to indicate the end of the row. When a row end tag is received instead of a valid column, the result is accumulated and output.

To accommodate the 256 bytes processor memory limit on the target architecture, each processor core can hold up to 240 vector elements, and additional vector elements can be stored in big shared memory connected to the processor. Each processor added to the array increases the number of vector elements that can be stored.

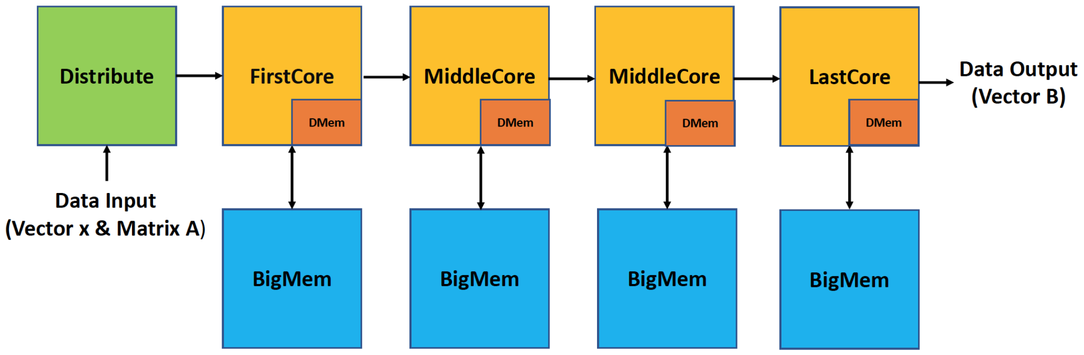


Figure 3.2: BigMemSnake SpMV mapping showing the data path and BigMem connection. The x vector values are evenly divided among the cores. Each core is labeled with its kernel name. Dense column vector x and sparse matrix A data are passed into the array through FirstCore and the resultant dense column vector B is sent out from LastCore. Each core, storing x values in data memory before sending and storing remain x values in the connected shared memory module. After storing x vector values, each core passes x values downstream, except for the LastCore.

Algorithm 10 Pseudocode of *BigMemSnake* (continued on the next page)

```
$LoadingVector$
$CheckColumn:$
$Column \leftarrow Input$                                   # Get column index
if $CoreNumlow \leq Column \leq CoreNumHigh$ then
    $go\ to\ Multiply$            # Go to Multiply since the column index is inside the boundary
else if $Column < CoreNumlow$ then             # Pass column index and its corresponding data
    $EastOutput \leftarrow 0$
    $EastOutput \leftarrow Column$
    $EastOutput \leftarrow Input$
else                                            # Pass token when achieving the end of the row
    $EastOutput \leftarrow 0$
    $EastOutput \leftarrow Column$
end if
$Multiply:$
$MemAddress \leftarrow Column - CoreNumLow$                # Get the corresponding memory address
if $MemAddress \leq DMemHold$ then
    $go\ to\ MultiplyDMem$
else          # Loading the corresponding vector item from big memory and do the multiplication
    $VectorVal \leftarrow BigMem[MemAddress - DMemHold]$
    $AccSum \leftarrow Input * VectorVal + AccSum$
    $go\ to\ CheckColumnContinued$
end if
$MultiplyDMem:$ # Loading the corresponding vector item from data memory and do the multiplication
$VectorVal \leftarrow DMem[MemAddress]$
$AccSum \leftarrow Input * VectorVal + AccSum$
$CheckColumnContinued:$
$Column \leftarrow Input$                              # Get next column index
```

Algorithm 10 Pseudocode of *BigMemSnake* (continued)

```
  if $CoreNumlow \leq Column \leq CoreNumHigh$ then
    go to $MultiplyContinued$                # Column index is inside the boundary
  else
    $EastOutput \leftarrow 1$
    $EastOutput \leftarrow AccSum$           # Pass partial sum when achieving the end of the row
    go to $CheckColumn$
  end if
  $MultiplyContinued$ :
  $MemAddress \leftarrow Column - CoreNumLow$
  if $MemAddress \leq DMemHold$ then
    go to $MultiplyDMemContinued$
  else     # Loading the corresponding vector item from big memory and do the multiplication
    $VectorVal \leftarrow BigMem[MemAddress - DMemHold]$
    $AccSum \leftarrow Input * VectorVal + AccSum$
    go to $CheckColumnContinued$
  end if   # Loading the corresponding vector item from data memory and do the multiplication
  $MultiplyDMemContinued$
  $VectorVal \leftarrow DMem[MemAddress]$
  $AccSum \leftarrow Input * VectorVal + AccSum$
  go to $CheckColumnContinued$
```

3.2.2 BigMemSubPara Accumulation

The *BigMemSubPara* is another more complex SpMV implementation and its pseudocode is shown in Algorithm 11. As shown in Figure 3.3, there are three phases involved:

In the first phase, NNZ per row and matrix data are assigned to the sorting network in a row-and-loop order.

The sorting network then sorts the data based on the LSB and second LSB of the column index of each sparse data point. Each sorting core takes sparse data point, including its column index and value, and then determines where it is sent depends on the LSB and second LSB of its column index, and then flush the sparse data point to the corresponding processing array.

In the final phase, data is transmitted through a set of processing cores connected in 4x4 blocks. The processing phase uses the snake kernel on each processor in the array, linking them together using a single input and output from each processor. Each processing core determines whether to multiply the matrix term with the corresponding vector term and refresh the partial sum, or directly flush the token. If the column index of the sparse data point matches the core boundary, the A matrix value is multiplied by the corresponding x vector value, otherwise the sparse data point is forwarded to the next core. After the last value of the row has been sent, a token will be sent to indicate the end of the row. Once a row end tag is received instead of a valid column, the result is accumulated and output.

To fit within the processor memory limitation of 256 bytes on our target architecture, each processor core can hold up to 240 vector elements, and additional vector elements can be stored in shared large memory connected to the processor. Each processor added to the array increases the number of vector elements that can be stored.

Algorithm 11 Pseudocode of *BigMemSubPara* (continued on page 40)

```
$LoadingVector$
$CheckColumn:$
if $Input\_next \geq CoreNumLow$ then # Check if the incoming column index corresponds to this core
    go to $Multiply$                          # Do $Multiply$ if column index inside the boundary
else
    $EastOutput \leftarrow Input$             # Else pass column index and its corresponding data
    $EastOutput \leftarrow Input$
    go to $CheckColumn$
end if
$Multiply:$
$MemAddress \leftarrow Input - CoreNumLow$ # Get the memory address of the corresponding vector item
$EastOuput \leftarrow -2$
```

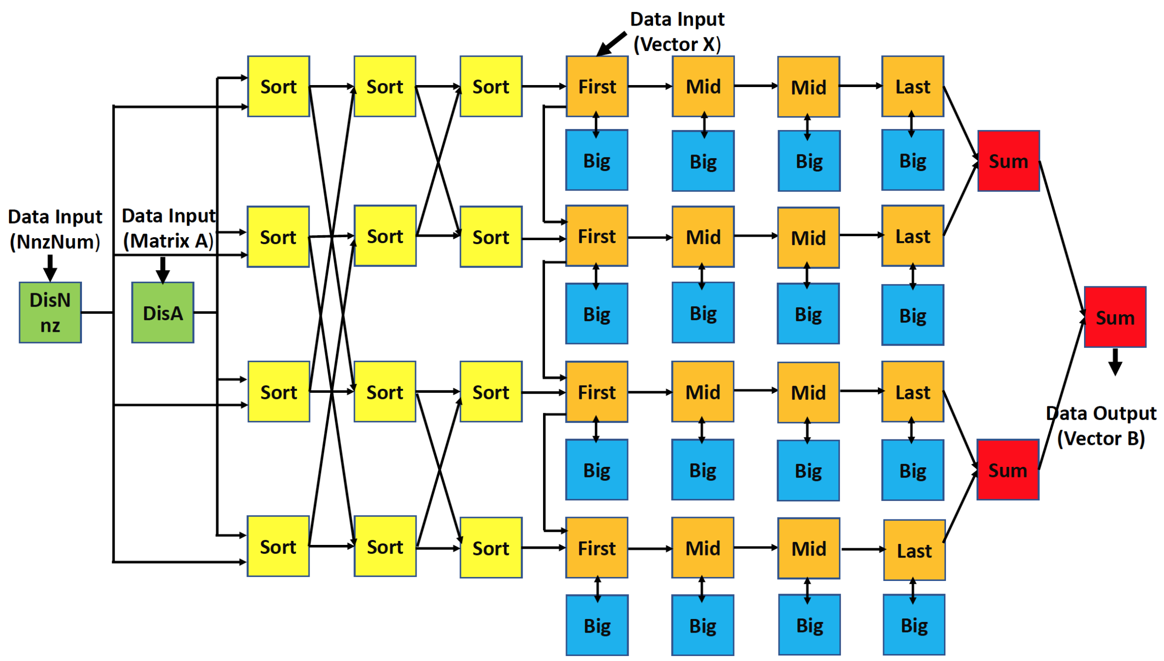


Figure 3.3: BigMemSubPara with LSB based sorting SpMV mapping showing the data path and BigMem connection. The implementation consists of four phases, distribution network, sorting network, processing array, and collection network. NNZ of each row is received by the DisNnz core, which sends a token downstream once transmitting all nonzeros of current row. LSB based sorting network uses the LSBs of the column index to route data to the appropriate processing row. Each processing core, storing x values in memory before transmitting sorted A values. After sorting A matrix values, each core passes A values downstream depends on the column index of each element, except for the LastCore. Each row computes partial products in parallel. Products are accumulated in the collection network before being to off chip.

Algorithm 11 Pseudocode of *BigMemSubPara* (continued)

```
if $MemAddress \leq DMemHold$ then
  go to $MultiplyDMem$
else                    # Loading vector item from big memory and do the multiplication
  $VectorVal \leftarrow BigMem[MemAddress - DMemHold]$
  $EastOutput \leftarrow Input * VectorVal$
  go to $CheckColumn$
end if
$MultiplyDMem$ :        # Loading vector item from data memory and do the multiplication
$VectorVal \leftarrow DMem[MemAddress]$
$EastOutput \leftarrow Input * VectorVal$
go to $CheckColumn$
```

3.2.3 BigMemPara Accumulation

BigMemPara is another proposed implementation of SpMV and its pseudocode is shown in Algorithm 12. As shown in Figure 3.4, it involves three phases:

In the first phase, the matrix distribution kernel assigns matrix data to the beginning of processing network in loop order, and the DisNnz core allocates NNZ per row to the end of processing array.

For the processing array, matrix data A is streamed through a set of processing cores connected in 4x3 blocks. Unlike the snake kernel described in section 3.2.2, the processing kernel used here doesn't need to stream token at the end of each row since NNZ per row has been routed to the end of each processing row to indicate the end of each row. The processing phase uses a simplified kernel on each processor in the array, linking them together using a single input and output per processor. Each processing core determines which core each sparse data point fits in based on the core boundary, then multiplies the matrix term by the corresponding vector term to refresh the partial sum. The corresponding vector x value is fetched from either data memory within each core or independent shared memory module based on the column index of the sparse data point. Each big shared memory module is arbitrarily accessed by two processing cores from adjacent rows.

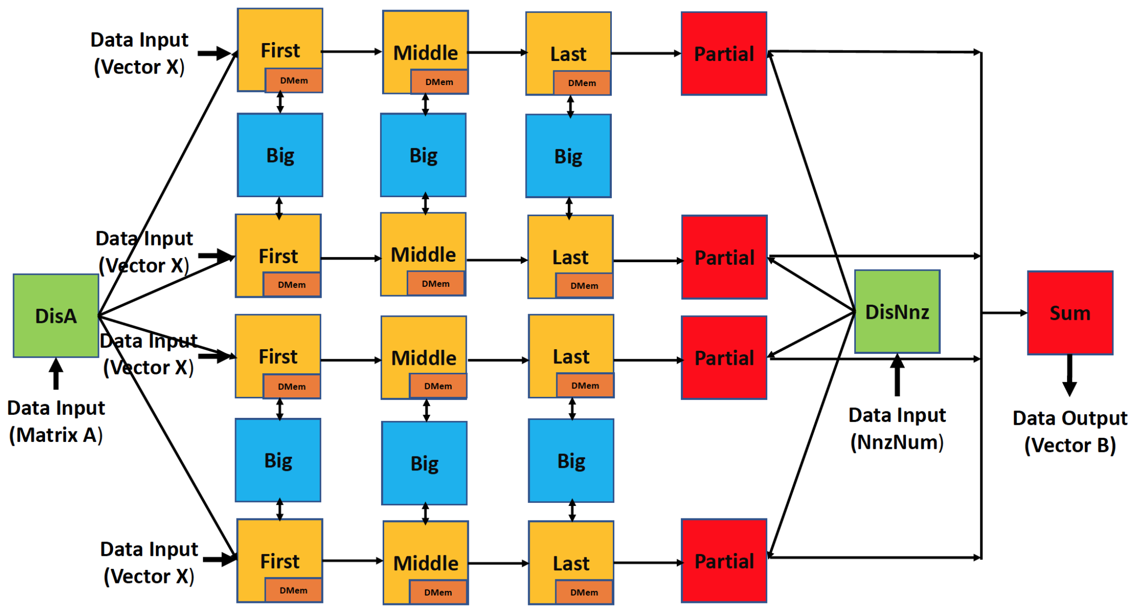


Figure 3.4: BigMemPara SpMV mapping showing the data path and BigMem connection. The implementation consists of three phases, distribution network, processing array, and collection network. DisNnz core sends nonzeros of each row to the appropriate processing row in round-robin order. Each processing core, storing x values in memory before transmitting A matrix values downstream, except for the LastCore. Each big shared memory module is arbitrarily accessed by two processing cores from adjacent rows. Each row computes partial products in parallel. Products are accumulated in the collection network before being to off chip.

If the column index of the sparse data point matches the core boundary, matrix value A is multiplied by the corresponding vector x value, otherwise the sparse data point is forwarded to the right core. Once the NNZ of the row counts down to zero, the result is accumulated and flows into the output.

To fit within the processor memory limitation of 256 bytes on our target architecture, each processor can hold up to 240 vector elements, and extra vector elements can be stored in the shared big memory connected to that processor. Different from the two SpMV implementations described above, one big shared memory will be connected to two processing processors in this implementation because the downstream and upstream cores have the same column boundary, which means they will access the same part of the whole vector. During operation, both processing cores will arbitrarily access the same shared memory to save time and power. Each processor added to the array increases the number of vector elements that can be stored.

Algorithm 12 Pseudocode of *BigMemPara*

```
$LoadingVector$
$CheckColumn:$
if $Input\_next \geq CoreNumLow$ then # Check if the incoming column index corresponds to this core
  go to $Multiply$                     # Do $Multiply$ if column index inside the boundary
else
  $EastOutput \leftarrow Input$        # Else pass column index and its corresponding data
  $EastOutput \leftarrow Input$
  go to $CheckColumn$
end if
$Multiply:$
$MemAddress \leftarrow Input - CoreNumLow$ # Get the memory address of the corresponding vector item
$EastOuput \leftarrow -1$
if $MemAddress \leq DMemHold$ then
  go to $MultiplyDMem$
else              # Loading the corresponding vector item from big memory and do the multiplication
  $VectorVal \leftarrow BigMem[MemAddress - DMemHold]$
  $EastOutput \leftarrow Input * VectorVal$
  go to $CheckColumn$
end if
$MultiplyDMem:$ # Loading the corresponding vector item from data memory and do the multiplication
$VectorVal \leftarrow DMem[MemAddress]$
$EastOutput \leftarrow Input * VectorVal$
go to $CheckColumn$
```

3.3 Different Phase Kernels Analysis

This work explores the Sparse Matrix-Vector Multiplication, one of the most common and useful scientific implementations, and uses the KiloCore (AsAP3) processor to quantify the results. Three main scenarios: *BigMemSnake*, *BigMemSubPara* and *BigMemPara*; and a total of eight SpMV implementations on the many-core platform were simulated. The three main scenarios

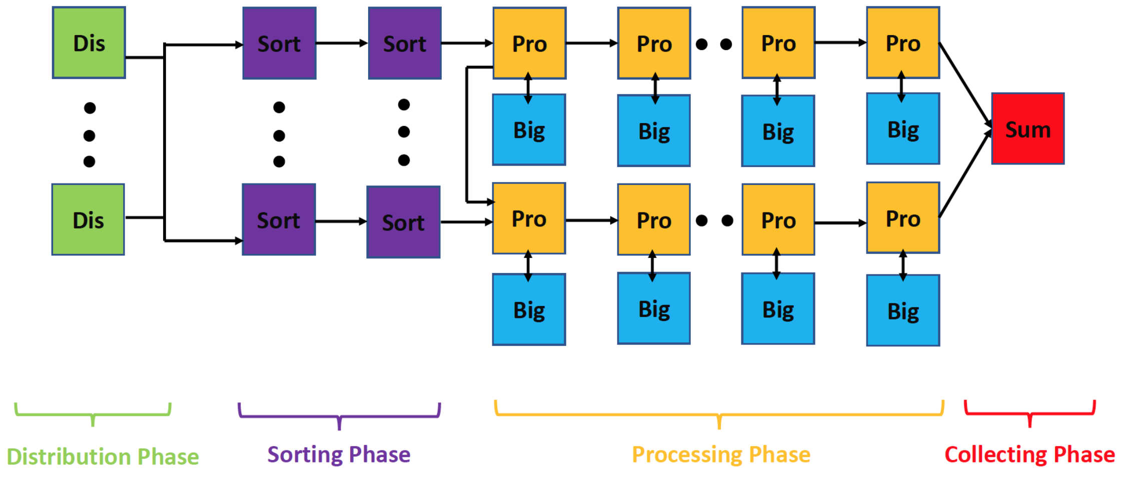


Figure 3.5: A diagram showing the general design including Distribution, Sorting, Processing, Collecting phases.

were general mapped onto the chip in Section 3.2. Each scenario can be divided into up to four different kinds of phases and each phase is built from a similar set of kernels, as shown in Figure 3.5. I will focus on comparing the different kernels of three phases in this section: distribution network, sorting network and processing array. For the three main SpMV scenarios: *BigMemSnake* only includes processing array, *BigMemPara* includes distribution network and processing array, and *BigMemSubPara* covers all three phases: distribution network, sorting network and processing array. Different kernels within three phases that affect the throughput per watt and throughput per area achieved by the SpMV implementations are analyzed as following.

3.3.1 Processing Array Phase

The previous method to implement SpMV on a many-core processor was also implemented on the AsAP3 platform [65]. These processing array kernels were designed to be modular and run on a large number of processors with only data memory within each processor. In order to extend the previous work, big shared memory modules have been collected for large matrices with different sizes. Two scaling methods were explored: adding more independent shared memory module to the processing array and expanding the amount of processing cores. Both methods provide increased I/O bandwidth and storage capacity with loading vectors at the cost of the area consumption. In order to distinguish, the previous method is called *DMem* implementation, and the improved

Table 3.1: Performance Comparison of *Bigmem* and *DMem* Implementations on the P6000 sparse matrix. Throughput per area versus throughput per watt for various SpMV implementations operating on the P6000 sparse matrix are plotted in Figure 3.6. The highest throughput per area versus throughput per watt of *Bigmem* and *DMem* Implementations are in bold.

Implementation Name	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
DMemSnake	**42**	282
DMemParaOne	26	336
DMemParaTwo	35	266
DMemParaTwoNnz	37	271
DMemParaFour	17	231
DMemParaFourPad	31	247
DMemSubParaFour	32	371
DMemSubParaFourTable	36	**387**
BigMemSnake	210	**4182**
BigMemParaOne	215	2950
BigMemParaTwo	**254**	3000
BigMemParaTwoNnz	213	2867
BigMemParaFour	124	1922
BigMemParaFourPad	216	2722
BigMemSubParaFour	35	590
BigMemSubParaFourTable	48	632

method is called *BigMem* implementations for the rest part of thesis.

Since the percentage of time each processor actively processes information varies by processor in the array, the bottleneck of a large number of processing cores in an array is highlighted. When SpMV runs on a large amount of data, the lack of activity on the first processor will be filled up in a continuous run, allowing runs to overlap. Other processors are underutilized because they wait for new inputs to process or output records to stop. The percentage of activity is far below 100% utilization because they are waiting for neighboring processors, which reduces throughput. The longer the array chain, the greater the throughput per watt and throughput per area are hurt. Since connecting big shared memory module to a processing array can greatly reduce the array size and alleviate bottlenecks between adjacent cores, it plays a great role on improving total throughput per watt and throughput per area.

From Figure 3.6, which shows the performance comparison of *BigMem* implementations

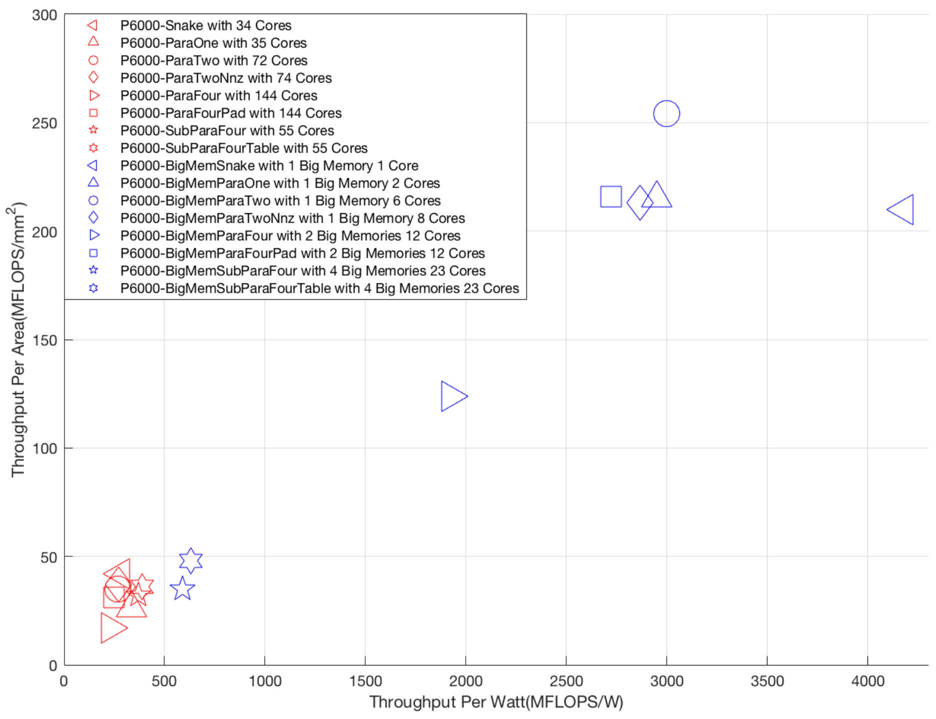


Figure 3.6: Performance Comparison of *Bigmem* and *DMem* Implementations on the P6000 sparse matrix. Throughput per watt and throughput per area of all implementations are shown in Table 3.1. For the most efficient *Bigmem* implementation, the improvement in throughput per watt and throughput per area over the most efficient *DMem* one are 6.05× and 10.81× respectively. The P6000 sparse matrix simulated is a relatively small sparse matrix from the entire database. For larger matrices, the throughput per watt and throughput per area improvements are more significant since the number of cores *DMem* implementations used increase dramatically, compared to *Bigmem* ones.

and *DMem* implementations on matrix P6000, all implementations that use big shared memory module achieve higher throughput per watt versus throughput per area than implementations with only data memory within each core. For all *Bigmem* implementations, as shown in Table 3.1, the average improvement in throughput per watt and throughput per area over *DMem* ones are 5.14× and 7.89× respectively. For the most efficient *Bigmem* implementation, the improvement in throughput per watt and throughput per area over the most efficient *DMem* one are 6.05× and 10.81× respectively.

Since each data entry needs to flow through the entire chain, and the access time to the shared memory is negligible compared to the traffic latency between cores in chain, the length of the processing core chain is the bottleneck affects throughput per watt versus throughput per area. The P6000 sparse matrix simulated is a relatively small sparse matrix from the entire database. For larger matrices, the throughput per watt and throughput per area improvements are more significant since the number of cores *DMem* implementations used increases dramatically, compared to *Bigmem* ones. *DMem* implementations are not further discussed or compared in this thesis since using a combination of shared memory and data memory can significantly reduce the number of processors required, which greatly improves throughput per watt and throughput per area.

3.3.2 NNZ Distribution Phase

Basic non-zero distribution kernel has been described at Section 3.1.2 on page 27. Therefore, part of the difficulty of this kernel is to effectively determine which core in the distribution network last received data.

Basic NNZ Distribution

Figure 3.7 shows the basic NNZ distribution kernel for two processing arrays, the NNZ of each row will be divided by 2, and the rest will be added to one of the two processing rows in a round-robin order, depending on the previous allocation order.

Padding NNZ Distribution

Similar to previous work [65], a variant of the basic NNZ distribution kernel was implemented. In order to extend the basic version, this kernel uses explicit zeros to fill each sparse matrix row so that NNZ of each row is an integer multiple of the number of processing rows, and the distribution core doesn't need to track the last received data to fill all processing rows in a loop sequence, which requires the prior allocation order and additional control overhead.

As shown in Figure 3.8, with four processing rows, each row is filled with an appropriate number of zeros, depending on the existing NNZ of this row. The corresponding vector is also padded with zeros to ensure that the result is not affected. After padding zero entries to matrix sparse data points A and dense vector x, the *DistributeNnz* core is simplified a lot because it just

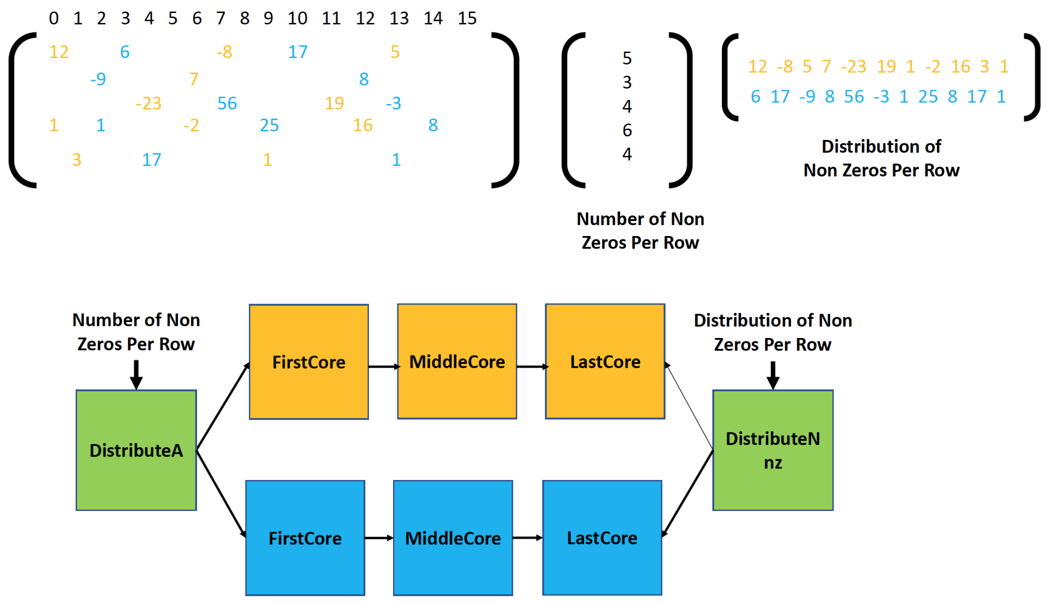


Figure 3.7: Basic NNZ Distribution Kernel on AsAP3. Using two processing rows, the distribution core routes nonzeros per row to each processing array in round-robin order.

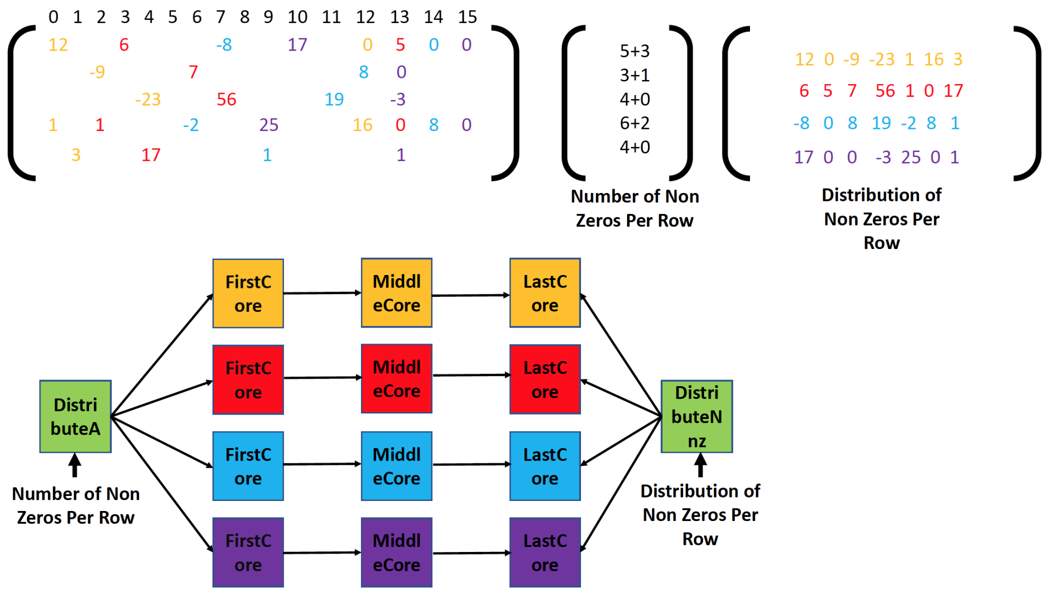
sends count equals to the NNZ per row divided by the number of processing rows to all processing arrays.

Therefore, there is a clear trade-off between the basic kernel and its variant, as additional useless zero entries were added to simplify the NNZ counter allocation instruction overhead. Although the control overhead is reduced by zero padding formatting, once the number of padding zero entries can't be ignored compared to the total NNZ of matrix A, the throughput of the entire kernel are not greatly increased or even decreased. In general, the efficiency of padding kernel increases with the density of matrix A and decreases as the number of processing rows increases.

Table NNZ Distribution

From the two NNZ distribution kernels described above, it is clear that the most important part of improving the efficiency of the kernel is to track the last received data on the distribution network. Therefore, another method that can reduce the control overhead of the NNZ distribution kernel without filling zero entries would be helpful. Since each core has an independent data memory, the lookup table seems to be a good attempt to achieve both goals to improve throughput.

Since NNZ per row is not always divisible by the number of arrays processed, the extra value of the uneven partition can be used to divide the entire table into different parts. For each



Number of Non Zeros Per Row
Distribution of Non Zeros Per Row
DistributeA
DistributeNnz
FirstCore
MiddleCore
LastCore

Figure 3.8: Padding NNZ Distribution Kernel on AsAP3. Using four processing rows, each row is filled with the appropriate number of zeros, depending on the existing NNZ of this row. The corresponding vector is padded with zeros to ensure that the results are not affected. After filling zero entries into A matrix values and x vector values, the distribution network is much simplified because it only needs to route count, which equals to the quotient of the NNZ per row and the number of processing arrays, to each processing array.

section, the cumulative non-zero value sent so far needs to be used as a pointer to the direction of rotation.

For lookup table method, all possible combinations of sending additional non-zero values for each direction are stored in memory along with all possible rotations, where the number of values to store depends on the number of processing arrays of the kernel. The NNZ distribution core receives NNZ of each row, tracks the table rotation, accumulates the NNZ per row and alternately sends a non-zero number to each processing array.

3.3.3 Column Index Based Sorting Phase

The sorting network is another important phase of the *BigMemSubPara* scenario. In the previous work [65], a similar sorting network was used, but the implementations without big shared memory module were inefficient because a large number of dense vector elements take up too many cores. Two different column index based sorting networks with or without shared memory module connection were compared as following.

BigMemSubPara with LSB based sorting

The basic Column Index Based Sorting was described in Section 3.2.2 on page 40, involving three phases, as shown in Figure 3.3.

In the first phase, the NNZ per row and matrix value A are assigned to the sorting network in a row-and-loop order.

The sorting network then sorts the data based on the LSB and second LSB of the column index for each sparse data point. Each sorting processor takes sparse data point, including column index and matrix value, determining where it is sent depends on the LSB and second LSB of its column index, and then flushes the sparse data points to the corresponding processing array.

In the final phase, data is transmitted through a set of processing cores connected in 4x4 blocks. The processing array uses a snake kernel on each processor in the array, linking them together using a single input and output from each processor. Each processing core determines whether to multiply the matrix term with the corresponding vector term and refresh the partial sum, or directly flush the token. If the column index of the sparse data point matches the core boundary, the matrix value A is multiplied by the corresponding vector value x, otherwise the sparse data point is forwarded to the next core. After the last value of the row has been sent, a token is sent to indicate the end of the row. Once a row end tag is received instead of a valid column, the result is accumulated and output.

To accommodate the 256 byte processor memory limit on the target architecture, each processor core can hold up to 240 vector items, and additional vector items are stored in big shared memory connected to the processor. Each processor added to the array increases the number of vector items that can be stored.

BigMemSubPara with LSB based sorting and shared memory

Another *BigMemSubPara* SpMV variant, as shown in Figure 3.9, except for the big shared memory connection, all three phases are identical to Figure 3.3 on page 40.

Unlike the basic kernel above, each big shared memory module are arbitrarily accessed by two adjacent processing cores from same row because two consecutive cores on the same array

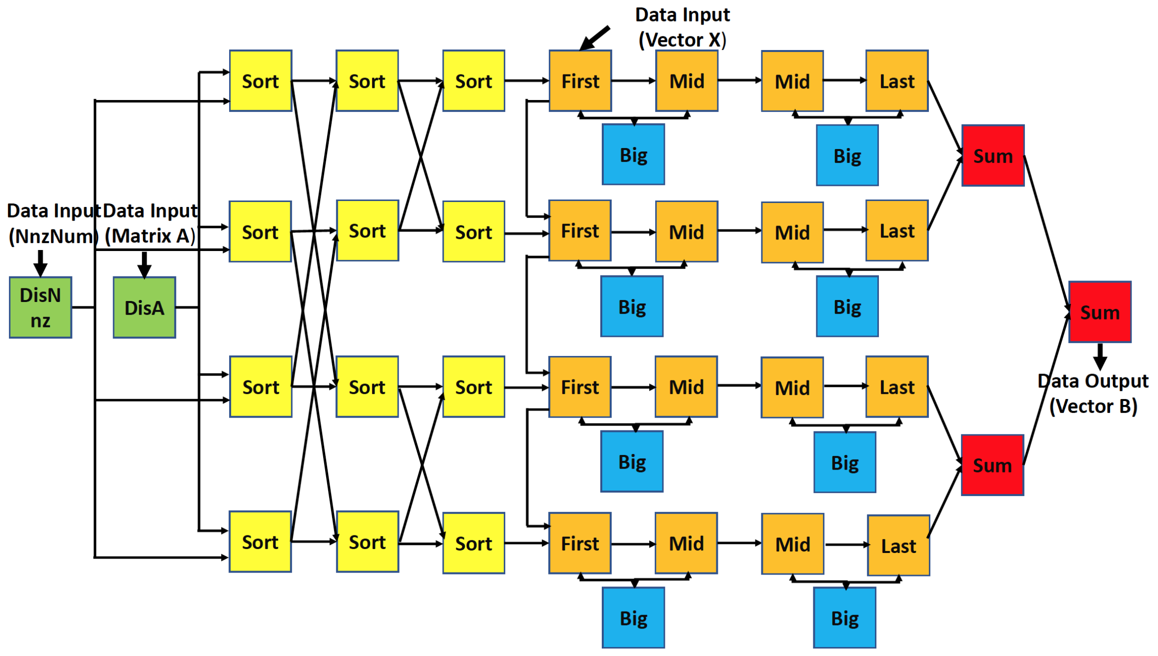


Figure 3.9: BigMemSubPara with LSB based sorting and shared memory SpMV mapping showing the data path and BigMem connection. The implementation consists of four phases, distribution network, sorting network, processing array, and collection network. NNZ of each row is received by the *DisNnz* core, which sends a token downstream once transmitting all nonzeros of current row. LSB based sorting network uses the LSBs of the column index to route data to the appropriate processing row. Each processing core, storing x values in memory before transmitting sorted A values. Each big shared memory module are arbitrarily accessed by two adjacent processing cores from same row. After sorting A matrix values, each core passes A values downstream depending on the column index of each element, except for the LastCore. Each row computes partial products in parallel. Products are accumulated in the collection network before being to off chip.

have consecutive column boundaries, which means they access successive parts of the entire vector. During operation, two adjacent processing cores arbitrarily access the same shared memory to save time and power.

BigMemSubPara with boundary based sorting

Another *BigMemSubPara* SpMV variant, as shown in Figure 3.10, except for the sorting network, both the processing array and distribution network are identical to Figure 3.3 on page 40.

Unlike the basic kernel described above, the sorting network sorts the data according to the column boundaries that are appropriate for the column index of each sparse data point. Each sorting core takes sparse data point, including its column index and matrix value, determining where it is sent to depends on the comparison result of its column index and the corresponding column boundary, and then flushes the sparse data point to each processing row.

In detail, for the first stage of the sorting network, if the column index of the sparse data point is greater than the upper column boundary of the sorting core, the sparse data points and its corresponding column index are sent to the upstream core for another comparison. Otherwise, the non-zero element and its column index will be flushed to the next core for the second stage of sorting. If the column index of the sparse data point is less than the lower bound of this sorting core, the sparse data points and its corresponding column index will be sent to the downstream core for another comparison.

BigMemSubPara with boundary based sorting and shared memory

Similar to Figure 3.9 on page 40, another *BigMemSubPara* SpMV variant is shown in Figure 3.11, each big shared memory module is arbitrarily accessed by two adjacent processing cores from same row since two consecutive cores on the same array have continuous column boundaries, which means they will access successive parts of the entire vector. During processing, both processing cores will arbitrarily access the same shared memory to save time and power.

Compare the four variants of Column Index Based Sorting described above, two basic rules were found as following:

For the sorting network, LSB-based sorting is always more efficient than boundary-based sorting because all sparse data points only need to traverse three sorting cores to reach the processing array. In contrast, the length of the sorting path of the boundary-based sorting network is unstable and variable, which results in more traffic and control overhead, thereby compromising throughput. Although LSB-based sorting requires more processing effort due to the discontinuity of loading vector, the processing workload is negligible compared to the overall core throughput.

For shared memory modules connected to nearby processing cores, the benefits of accessing shared memory arbitrarily are overshadowed by the downside of more processing cores used since the key point affecting the kernel performance is the amount of big shared memory used.

Due to the two rules mentioned above, only LSB-based sorting without shared memory

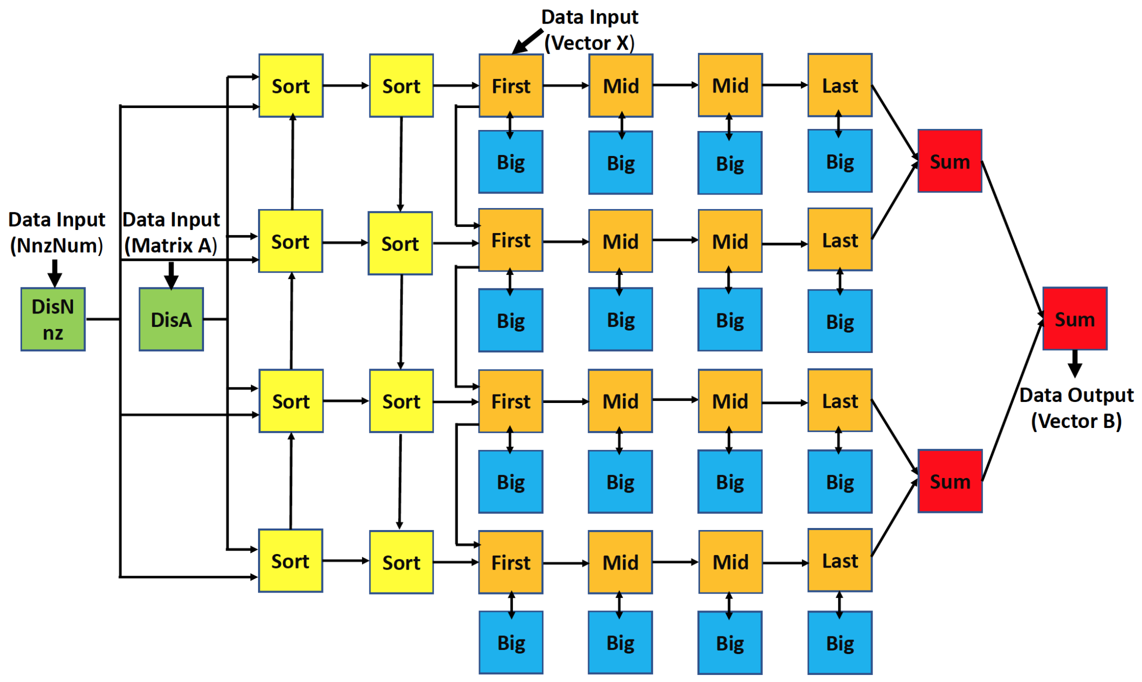


Figure 3.10: BigMemSubPara with boundary based sorting SpMV mapping showing the data path and BigMem connection. The implementation consists of four phases, distribution network, sorting network, processing array, and collection network. NNZ of each row is received by the *DisNnz* core, which sends a token downstream once transmitting all nonzeros of current row. Boundary based sorting network routes data to the appropriate processing row depending on the comparison result between the column index of A matrix values and column boundary of sorting core. Each processing core, storing x values in memory before transmitting sorted A values. After sorting A matrix values, each core passes A values downstream, except for the LastCore. Each row computes partial products in parallel. Products are accumulated in the collection network before being to off chip.

connection was tested for performance comparison of different implementations in Chapter 4 and Chapter 5.

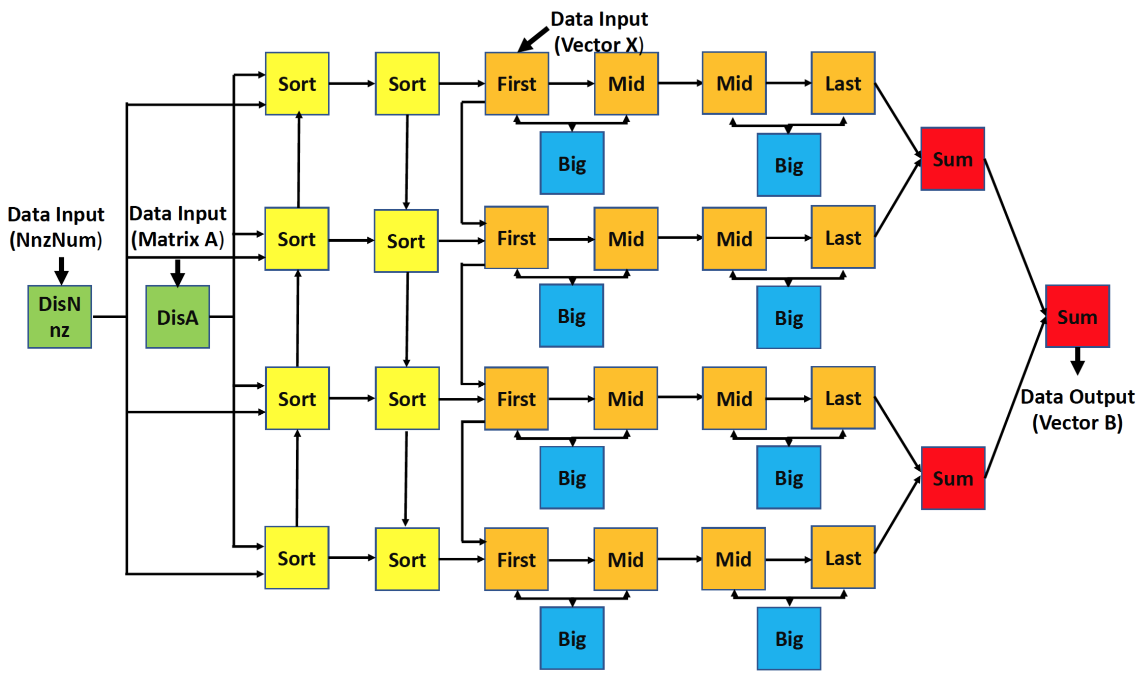


Figure 3.11: BigMemSubPara with boundary based sorting and shared memory SpMV mapping showing the data path and BigMem connection. The implementation consists of four phases, distribution network, sorting network, processing array, and collection network. NNZ of each row is received by the *DisNnz* core, which sends a token downstream once transmitting all nonzeros of current row. Boundary based sorting network routes data to the appropriate processing row depending on the comparison result between the column index of A matrix values and column boundary of sorting core. Each processing core, storing x values in memory before transmitting sorted A values. Each big shared memory module is arbitrarily accessed by two adjacent processing cores from same row. After sorting A matrix values, each core passes A values downstream, except for the LastCore. Each row computes partial products in parallel. Products are accumulated in the collection network before being to off chip.

Chapter 4

Comparison of Sparse Matrix-Vector Multiplication Methods for AsAP3

As mentioned in Chapter 2, the throughput per watt and throughput per area of sparse matrix vector multiplication on multi- and many-core platforms has been a keen interest in research. This interest, coupled with the ever-increasing number of processor cores per general processing chip, has led me to focus on using many-core processor arrays for SpMV.

For comparing the throughput, power, and area consumption within all eight SpMV implementations on many-core platform (AsAP3), totally 17 sparse integer matrices used to evaluate SpMV performance are shown in Table 2.2 on page 17. All integer matrix and vector data are in 16-bit fixed-point format. The median column size value for all the integer matrices in the entire database is 7521, while the average density is 0.00945. This work considers matrices with a column size value less than and greater than the median column size value of the University of Florida sparse matrix collection [2].

Throughput data for the implementations on the many-core platform (AsAP3) are obtained with a cycle-accurate C++ simulator. Power measurements from the 32 nm PD-SOI CMOS fabricated chip are input to the simulator to obtain power data. Area usage is physically measured from the fabricated 32 nm PD-SOI CMOS chip, where each processor takes up 0.055 mm^2 of area, and each shared memory occupies 0.164 mm^2 of area.

Two important features will be tested and calculated for all SpMV implementations: throughput per watt and throughput per area;

The throughput per watt is useful for selecting the most power efficient SpMV option, which is the bottleneck for implementing scientific applications on many-core or multi-core platforms.

The throughput per area is useful for selecting the most area efficient SpMV design, which is also very important for decreasing then fabrication cost and increasing area reduction per chip.

The throughput is calculated using the common metric [66, 67], as shown in Equation 4.1:

$$Throughput = \frac{2 \times NNZ}{Execution\ Time} \tag{4.1}$$

For each implementation, the throughput is calculated by dividing the total number of operations by the execution time.

4.1 Power Efficiency Comparison between different implementations

The metrics of throughput per watt is used for power efficiency comparison between different implementations.

Three main scenarios: *BigMemSnake*, *BigMemSubPara* and *BigMemPara*; and a total of eight SpMV implementations using different number of processing rows and kernels on the many-core platform were simulated. The eight implementations, as well as general mappings onto the chip were given in Chapter 3. As shown in Figure 3.5 on page 44, all SpMV implementations contain one or more phases and each phase is build from a set of similar kernels. I will focus on comparing the power efficiency of all implementations by analyzing different kernels of each phase in this section. For the three main SpMV scenarios: *BigMemSnake* only includes processing array, *BigMemPara* includes distribution network and processing array, and *BigMemSubPara* covers all three phases: distribution network, sorting network and processing array.

4.1.1 Processing array phase

For processing array, after simulating 17 sparse integer matrices chosen from Table 2.2 on page 17 on the number of cores ranging from 1 to 23. Adding more cores to the array only increases the size of the run without significantly increasing the throughput per watt.

The main reason is that the logical parts of the processing array of all eight implementations are very similar, passing matrix data to the corresponding core, fetching vector items from the data memory or big shared memory, and flushing the partial sum to the next core.

Theoretically, doubling the number of processing arrays doubles the efficiency throughput, but the ideals can not be achieved due to other factors, including the flow and control overhead of the distribution and sorting network.

Another reason is that due to the limitation of the data path width (16 bits), the maximum size of all simulated matrices is less than 65536 (unsigned 16 bits), and each big shared memory can hold up to 32768 vector items so that length of each processing array doesn't exceed two, which means that the blocking problem caused by too long processing chain when only the data memory within each core using for storing the vector items will be negligible.

Due to the two main reasons mentioned above, it comes to the conclusion that *BigMemSnake* is the most area efficient one among all eight implementations.

Another point to mention is that even the processing logic of *BigMemSnake* is more complicated than *BigMemPara* and *BigMemSubPara* because of the lack of NNZ distribution kernels, by using extra tokens to represent the end of each row, the throughput is not much compromised compared to the power consumption since the processing array is relatively short.

BigMemParaOne recognizes a relatively small increase in throughput while power consumption increases significantly, which occurs as the NNZ distribution kernel is added to simplify the processing logic.

The biggest difference between *BigMemSnake*, *BigMemPara* and *BigMemSubPara* is that big shared memory module can not be shared by neighbour cores for latter scenario because the two neighboring kernels doesn't access the same part of the vector, making loading and fetching process much more complicated and inefficient. That's why the *BigMemSubPara* is always the least efficient one compared to the other two.

4.1.2 Distribution network phase

Since only *BigMemSubPara* and *BigMemPara* contain distribution network, all power efficiency comparison will focus on these two scenarios.

As mentioned in Section 3.3 on page 43, assigning NNZ to all processing rows has a significant impact on the throughput per watt.

Three non-zero distribution kernels were described in Chapter 3 on page 48. Therefore, part of the difficulty of this kernel is to effectively determine which core in the distribution network last received data.

Basic NNZ distribution kernel is a baseline method for non-zero count distribution, and the Padding NNZ distribution kernel and the table NNZ distribution kernel are considered to be potential alternatives to improve throughput per watt and throughput per area.

For *BigMemSubPara* implementations with four computed rows, the table NNZ distribution kernel improves the throughput per watt up to 1.44×, compared to basic one. The main reason is that the table NNZ Distribution kernel uses data memory to store all distribution direction information, so the control overhead of the kernel will be greatly reduced compared to the other two kernels.

Padding NNZ to an integer multiple of the number of processing arrays improves throughput per watt in most cases. Although processing added zero values reduces throughput per watt, padding NNZ simplifies the non-zero count distribution. For *BigMemPara* implementations with four computed rows, the padding NNZ distribution kernel improves the throughput per watt up to 2.87×, compared to basic one.

4.1.3 Sorting network phase

Boundary-based sorting kernel is the baseline method for the sorting network, and LSB-based sorting kernel is an alternative to improve throughput per watt and throughput per area.

As described in section 3.3 on page 49, for sorting networks, LSB-based sorting kernel is always more efficient than boundary-based sorting kernel due to the fixed length of the sorting path. Also as with the two rules found on page 54, only LSB-based sorting kernel without shared memory connection was tested for different implementation performance comparisons in Table 4.1.

For *BigMemSubPara* implementations with four computational rows using boundary-based sorting kernel, switching to LSB-based sorting kernel can improve throughput per watt by up to 1.35× for all simulated matrices.

As N increases and is greater than 4, *BigMemSubPara* implementations with N processing rows become progressively less efficient than the *BigMemSnake* method due to the size limitations of the simulated matrices.

4.1.4 Summary

BigMemSnake is the basic implementation first considered for SpMV.

BigMemPara is a similar scenario to *BigMemSnake*, which adds an extra kernel to calculate the NNZ for each row at the end of each processing array, which simplifies control overhead but also increases power consumption.

BigMemSubPara is the most complex implementation which adds the sorting network to distribute nonzeros of each row to all processing arrays, decreasing the number of processing cores needed but while increasing the total power consumption because of the extra sorting stage.

Additional implementations of *BigMemPara* and *BigMemSubPara* are developed to increase throughput by increasing the number of rows computed in parallel.

Since all simulated matrices are not very large in size compared to the capacity of large shared memory, simpler implementations such as *BigMemSnake* and *BigMemParaOne* achieve the maximum throughput per watt. Complex implementations including extra sorting networks, such as *BigMemParaFour* and *BigMemParaFourPad*, achieve the minimum throughput per watt.

The throughput per watt comparison results of all implementations on all simulated integer matrices from Table 2.2 on page 17 can be found in Table 4.1, and the larger the relative value of particular implementation, the higher its throughput per watt. For most simulated matrices, *BigMemSnake* is the most power efficient implementation with maximum relative value, and *BigMemSubParaFour* is the least efficient one with minimum value, metric (Throughput per Watt) is normalized against the least efficiency one for all other implementations.

Table 4.1: The relative throughput per watt of all 8 implementations on 17 simulated integer matrices (16 bit fixed point) from Table 2.2, metric (Throughput per Watt) is normalized against the least efficiency implementation.

Matrix Name	BigMem Snake	BigMem ParaOne	BigMem ParaTwo	BigMem ParaTwoNNZ	BigMem ParaFour	BigMem ParaFourPad	BigMem SubParaFour	BigMem SubParaFourTable
Andrews	**3.55**	2.74	2.75	2.58	2.96	3.2	1	1.02
pcb3000	**5.95**	4.23	4.62	4.55	3.62	4.28	1	1.01
p6000	**7.09**	5	5.08	4.86	3.26	4.61	1	1.07
TF17	**3.83**	2.80	2.85	2.75	3.53	3.66	1.01	1
foldoc	**6.25**	4.53	4.65	4.36	3.45	4.11	1	1
EAT_RS	**5.72**	4.06	4.51	4.43	4.01	4.15	1	1
ch7-7-b5	**4.25**	3.21	2.98	2.72	2.35	2.96	1	1.02
cis-n4c6-b4	5.27	**6.02**	5.24	4.56	2.73	3.7	1	1.2
IG5-17	**4.26**	3.39	3.85	3.88	3.79	3.84	1	1.01
IG5-14	**4.71**	3.28	3.88	3.88	3.78	3.71	1	1.02
Trec13	**4.28**	3.12	3.77	4.02	3.93	3.93	1	1
C8-mat11	**4.31**	3.14	3.82	4.04	3.98	3.97	1	1.01
Trec11	**4.35**	3.12	3.7	3.94	3.9	3.86	1	1.01
Trec12	**4.3**	3.1	3.61	3.88	3.82	3.87	1	1.01
G10	**4.5**	3.23	3.75	3.85	3.79	3.82	1	1
Rosen2	**4.48**	3.28	3.76	3.86	3.74	3.79	1	1.01
Ip_d6cube	**4.43**	3.19	3.71	3.92	3.7	3.69	1	1.01

4.2 Area Efficiency Comparison between different implementations

The metric of throughput per area is used for area efficiency comparison between different implementations.

Three main scenarios: *BigMemSnake*, *BigMemSubPara* and *BigMemPara*; and a total of eight SpMV implementations using different number of processing arrays and kernels on the many-core platform were simulated. The eight implementations, as well as general mappings onto the chip were given in Chapter 3. As shown in Figure 3.5 on page 44, all SpMV implementations contain one or more phases and each phase is build from a set of similar kernels. I will focus on comparing the area efficiency of all implementations by analyzing different kernels of each phase in this section. For the three main SpMV scenarios: *BigMemSnake* only includes processing array, *BigMemPara* includes distribution network and processing array, and *BigMemSubPara* covers all three phases: distribution network, sorting network and processing array.

4.2.1 Processing array phase

For comparison of throughput per area, the common and majority part of area size used for all the SpMV implementations is processing array, which includes both the processing cores and big shared memory modules.

Since the programmable processors occupy 0.055 mm^2 each and shared memory modules occupy 0.164 mm^2 each so that reducing the control and flow overhead of processing network and making use of the combination of data memory within each core and independent big memory module are the key points for achieving more throughput per area.

After simulating 17 sparse integer matrices chosen from Table 2.2 on page 17 on the number of cores ranging from 1 to 23, *BigMemPara* is the most area efficient scenario for two reasons:

1. Compared to *BigMemSnake*, as described in Section 4.1.4, *BigMemPara* adds an extra kernel to count the NNZ per row at the end of each processing array, simplifying control overhead. Since the area occupied by the extra processing core is negligible compared to the entire implementation, *BigMemPara* is always more area efficient than *BigMemSnake*, especially when increasing throughput by increasing the number of parallel computed arrays in *BimMemPara*.

2. Compared to the *BigMemSubPara*, the extra area occupied by the sorting network is

not negligible compared to the entire implementation. Although the *BigMemSubPara* processing network requires fewer cores than *BigMemPara*, the single-port connection between core and shared memory and size limits for all simulated matrices limit the throughput per area of *BigMemSubPara*.

For *BigMemPara*, the scenario with big memory shared by two cores from two adjacent processing arrays has been analyzed and proved to be the one with highest throughput per area in Section 3.3.3 on page 49. Since the throughput can also be increased by increasing the number of parallel arrays, it is easy to find out that *BigMemParaTwo* is usually more area efficient than *BigMemParaOne* and *BigMemParaFour* because it almost doubles the throughput while just adding one row to processing array, which takes advantage of the big shared memory module shared by two processing arrays.

4.2.2 Distribution network phase

Since only *BigMemSubPara* and *BigMemPara* kernels contain distribution network, comparison of area efficiency will focus on these two scenarios.

For *BigMemSubPara* and *BigMemPara* with four computational arrays, all other NNZ distribution kernels improve throughput per area versus basic NNZ Distribution kernel, with up to 4.5× improvement with Padding NNZ Distribution kernel, and 2× with Table NNZ Distribution kernel.

Versus the *BigMemSnake* implementation, there is a positive correlation between matrix size and throughput per area improvement. As the size of the sparse matrix increases, implementations with more processing arrays (e.g., *BigMemSubPara* implementations with eight or more computed arrays) become more area efficient. However, this trend is not shown in this thesis due to the size limitation of the simulated matrices.

4.2.3 Sorting network phase

Boundary-based sorting is the baseline kernel for sorting networks, and LSB-based sorting kernel is an alternative to improve throughput per watt and throughput per area.

As with the two rules found in Chapter 3 on page 54, only LSB-based sorting kernel without shared memory was tested in Table 4.2 for different implementations performance comparison.

For *BigMemSubPara* implementations with four computed arrays using boundary-based sorting kernel, switching to LSB-based sorting kernel increases the throughput per area by up to 1.38× for all simulated matrices .

For a majority of simulated matrices, switching from boundary-based sorting kernel to LSB-based sorting kernel with table NNZ distribution kernel is efficient for improving throughput per area, by as much as 1.48×.

4.2.4 Summary

BigMemSnake is the basic implementation of SpMV first considered.

BigMemPara is a similar scenario to *BigMemSnake*, which adds an extra kernel to count NNZ of each row at the end of each processing array, which simplifies control overhead but also takes up more area.

BigMemSubPara is the most complex scenario, which adds a sorting network, assigning non-zero values per row to all processing arrays, reducing the number of processing cores required, but increasing the total area occupied due to the extra sorting phase.

Other implementations of *BigMemPara* and *BigMemSubPara* are added to increase throughput by increasing the number of arrays computed in parallel.

Since the *BigMemPara* has been proved to be the most area efficiency scenario among three main ones in section 4.2.1, and the throughput can also be increased by increasing the number of parallel computed arrays so that *BigMemParaTwo* implementation, which takes advantage of the big shared memory shared by two processing arrays, achieving the maximum throughput per area for the vast majority of all the simulated matrices.

The area efficiency comparison results of all implementations on all simulated integer matrices from Table 2.2 on page 17 can be found in Table 4.2, and the larger the relative value of a particular implementation, the higher its throughput per area. For most simulated matrices, *BigMemParaTwo*, *BigMemParaFour* and *BigMemParaFourPad* are the most area efficient implementations with maximum relative value, and *BigMemSubParaFour* is the least efficient one with minimum relative value, metric (Throughput per Area) is normalized against the least efficiency one for all other implementations.

Table 4.2: The relative throughput per area of all 8 implementations on 17 simulated integer matrices (16 bit fixed point) from Table 2.2, metric (Throughput per Area) is normalized against the least efficiency implementation.

Matrix Name	BigMem Snake	BigMem ParaOne	BigMem ParaTwo	BigMem ParaTwoNNZ	BigMem ParaFour	BigMem ParaFourPad	BigMem SubParaFour	BigMem SubParaFourTable
Andrews	1.64	1.75	2.21	1.95	2.21	**2.46**	1	1
pcb3000	4.29	4.43	**5.49**	4.53	3.67	4.88	1	1.02
p6000	6	6.14	**7.26**	6.09	3.54	6.17	1	1.37
TF17	1.61	1.7	2.16	1.90	2.75	**2.85**	1	1
foldoc	4.53	4.76	**5.94**	4.86	4	4.99	1	1
EAT_RS	4.09	4.13	**5.31**	4.37	4.84	4.71	1	1
ch7-7-b5	1.62	1.88	2.28	2	1.39	**2.44**	1	1
cis-n4c6-b4	1.98	3.02	**5.94**	4.89	1.94	3.7	1	1.19
IG5-17	1.27	1.6	**3.22**	2.65	3.21	3.08	1	1
IG5-14	2.57	3.18	**3.32**	2.72	3.28	3.08	1	1
Trec13	2.33	2.87	**2.97**	2.45	**2.97**	**2.97**	1	1
C8-mat11	2.3	2.84	**2.97**	2.42	**2.97**	2.96	1	1
Trec11	2.38	2.35	**3.01**	2.48	**3.01**	2.97	1	1
Trec12	2.37	2.35	**3.03**	2.47	3.01	3.01	1	1
G10	2.45	2.43	**3.13**	2.56	3.12	3.02	1	1
Rosen2	2.57	2.49	**3.17**	2.59	3.11	3.02	1	1
Ip_d6cube	2.56	2.44	2.51	2.5	**2.93**	2.89	1	1

Chapter 5

Comparison of Sparse Matrix-Vector Multiplication Methods on AsAP3 with general-purpose processor and GPU

5.1 Sparse Matrix Data Sets

For performance comparison with all eight implementations on many-core platform and others (general-purpose processor and GPU), totally 10 sparse real matrices used to evaluate SpMV performance are shown in Table 2.2 on page 17. All real matrix and vector data are in single-precision 32-bit IEEE-754 format. The median column size value for all the real matrices in the entire database is 16001, while the average density is 0.00941. This work considers matrices with a column size value less than and greater than the median column size value of the University of Florida sparse matrix collection [2].

5.2 Matrix Library

Since the row-based CsrMV implementations within Intel MKL [33] and NVIDIA cuSPARSE [34] are progressively unable to map their workloads fairly across parallel threads. To ensure a fair comparison of performance evaluation results, the parallel OpenMP C++ merge-based CsrMV implementation on general-purpose processor and CUDA C++ two-level merge-path CsrMV implementation on GPU [7] will be used as benchmark in this thesis for area and power performance comparison. The evaluation precision of all benchmarks is single-precision floating point (32 bit).

I primarily compare against the CsrMV implementations provided by Intel MKL library [33] and NVIDIA's cuSPARSE library v7.5 [34]. To highlight the competitiveness of the merge-based CsrMV method in the absolute, non CSR-based implementation like Cusparse HybMv will also be compared against.

5.3 Details of general-purpose processor and GPU

The general-purpose processor for SpMV is the Intel Core i7-3720QM, and its specifications are shown in Table 5.1. After establishing the merge list and the merge path length, each thread identifies its start and end diagonals and then searches for the corresponding 2D start and end coordinates within the merged grid to effectively execute the sequential CsrMV algorithm. When calculating throughput, the execution time of the multiplication step is divided by the number of iterations and the number of threads used.

The GPU for SpMV is the Nvidia Quadro 620, and its specifications are shown in Table 5.1. Since the entire implementation is divided into two levels, at the coarsest level, the merge path is fairly distributed among the thread blocks to fully occupy the GPU's multiprocessor. Each thread block then proceeds to consume a share of its merge path with a fixed size path chunk. The path chunk length is determined by the amount of local storage resources available to each thread block. After copying the sublist to local shared memory, threads independently perform sequential CsrMV, each consuming exactly items-per-thread to make maximal utilization of the GPU's fixed-size shared memory resources. The time of executing SpMV is measured and divided by the number of instances and iterations to obtain average execution time.

Table 5.1: Details of general-purpose processor and GPU Utilized for SpMV Comparison.

Chip	Technology (nm)	TDP (W)	Area (mm^2)
Intel Core-i7 3720QM	22	45	160
NVIDIA Quadro 620	28	41	148

5.4 Measurement and Simulation Methodology

Two important features will be tested and calculated for all SpMV implementations on various platforms: throughput per watt and throughput per area; The throughput per watt (MFLOPS/W) is useful for selecting the most power efficient SpMV option, which is the bottleneck for implementing scientific applications on many-core or multi-core platforms. The throughput per area (MFLOPS/mm^2) is useful for selecting the most area efficient SpMV design, which is also very important for decreasing then fabrication cost and increasing area reduction per chip. The throughput is calculated using the common metric [66, 67] shown in Equation 4.1 on page 56. The throughput per watt and per area results were plotted to analyze the power and area efficiency of each design.

Due to the different scale methodology, throughput, power and area results are scaled to 32 nm values for the general-purpose processor and GPU implementations. Area is scaled using $1/S^2$ scaling, and throughput and power are scaled using the trend of gate delay and switching energy for 65–14 nm technology nodes [68].

Throughput data for the implementations on the many-core platform are obtained from a cycle-accurate C++ simulator, customized for chip KiloCore (AsAP3) [1]. Power measurements from chips fabricated by 32 nm CMOS PD-SOI are input to the simulator to obtain power data. The chip power measurement and core area are scaled by the relative increase determined by the synthesis in 32 nm CMOS PD-SOI at 0.9 V and 1.8 GHz when adding a single-precision 32-bit FP adder and multiplier with denormal support and switching to a 32-bit data path, the scaling factors for power measurement and core area are obtained from previous work done by Jon J.Pimentel [65].

The general-purpose processor throughput data is gathered from a parallel OpenMP C++ merge-based CsrMV implementation [7], and no architecture-specific optimizations are enabled. Software pipelining, branch elimination, register-blocking, cache-blocking, TLB-blocking, matrix-blocking, or index compression are not explicitly incorporated, and power consumption is estimated using half of the thermal design power (TDP/2).

The GPU throughput data is collected from the CUDA C++ two-level merge path CsrMV implementation [7]. All compiler optimizations are turned on, and power consumption is estimated using half of the thermal design power (TDP/2).

For each implementation, the execution time is only the time of the sparse matrix multiplication operation. For general-purpose processor-based implementations, this excludes the time to read the sparse matrix A and the dense vector x from the file, load them into memory, and save the results to a file. For GPU-based implementations, this also eliminates the time to load data from host memory (i.e., general-purpose processor) to device memory (i.e., GPU), and vice versa. For many-core implementations, this only excludes the time to load the vector x into memory, but includes the time to load the sparse matrix A.

The most efficient designs provide the largest throughput per watt and throughput per area, therefore the most efficient designs are located near the upper-right corner and the least efficient designs are nearest the lower-left corner of each plot.

5.5 Power Efficiency Comparison with other Work

The eight proposed many-core SpMV implementations were simulated on 10 sparse real matrices chosen from Table 2.2 on page 17 using between 1 and 23 processors on the many-core platform, and are effective at improving throughput per watt versus using the methods on the GPUs or general-purpose processors. The implementations on the many-core platform increase throughput per watt by up to 190× versus the general-purpose processor implementations, and by up to 269.5× versus the GPU implementations.

BigMemSnake is typically the most power efficient implementation, but as matrix size (NNZ) increases, other implementations provide a higher throughput per watt.

5.5.1 Small and Sparse Matrix

For a relatively small and sparse matrix, Figure 5.1 shows the per-watt throughput and per-area throughput for all implementations on different platforms when performing SpMV using the sparse matrix GD96_a. The minimum number of cores for storing vector x is first used for the *BigMemSnake* implementation and then added to create other implementations.

Increasing the number of cores for the *BigMemSnake* beyond what is required to store the vector x decreases the throughput per watt. The optimal *BigMemSnake* implementation uses the least amount of cores to store the vector x.

Modify the *BigMemSnake* implementation to use *BigMemParaOne* with extra NNZ counter core to simplify the logic control overhead, thereby increasing throughput per area while decreasing

Table 5.2: Throughput per area versus throughput per watt for various SpMV implementations operating on the GD96_a sparse matrix. All metrics data is plotted in Figure 5.1 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	2.2	16
Quadro K620 Cusparse CsrMv	4	28.8
Quadro K620 Cusparse HybMv	1.3	8.8
i7-3720QM Merge-based	2.4	33.6
BigMemSnake	164	**3600**
BigMemParaOne	**212**	2762
BigMemParaTwo	101	1282
BigMemParaTwoNnz	96	1208
BigMemParaFour	30	545
BigMemParaFourPad	81.5	871
BigMemSubParaFour	11	205
BigMemSubParaFourTable	17.7	272

throughput per watt.

The relatively small size and NNZ of GD96_a make it less meaningful to find alternative implementations for SpMV. The results on Figure 5.1 show that the *BigMemPara* and *BigMemSubPara* implementations attempt to achieve this goal by increasing the amount of parallel processing with multiple processing rows come to failure. As the implementation becomes more complex and uses more cores, throughput per watt and throughput per watt will decrease, as shown in Figure 5.1. The most efficient implementations tend to be smaller and simpler, namely *BigMemSnake*. The least efficient many-core implementations tend to use more cores for a sorting network, including the *BigMemSubPara* implementations.

The least efficient implementation is the Cusparse HybMv algorithm in the NVIDIA Quadro K620. The GPU cannot perform SpMV operations efficiently because the unbalanced workload of each thread due to small column size of the matrix. Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per watt and throughput per area as the least efficient one.

All implementations on many-core platform for GD96_a provide greater throughput per watt than the general-purpose processor and GPU based implementations. As shown in Table 5.2,

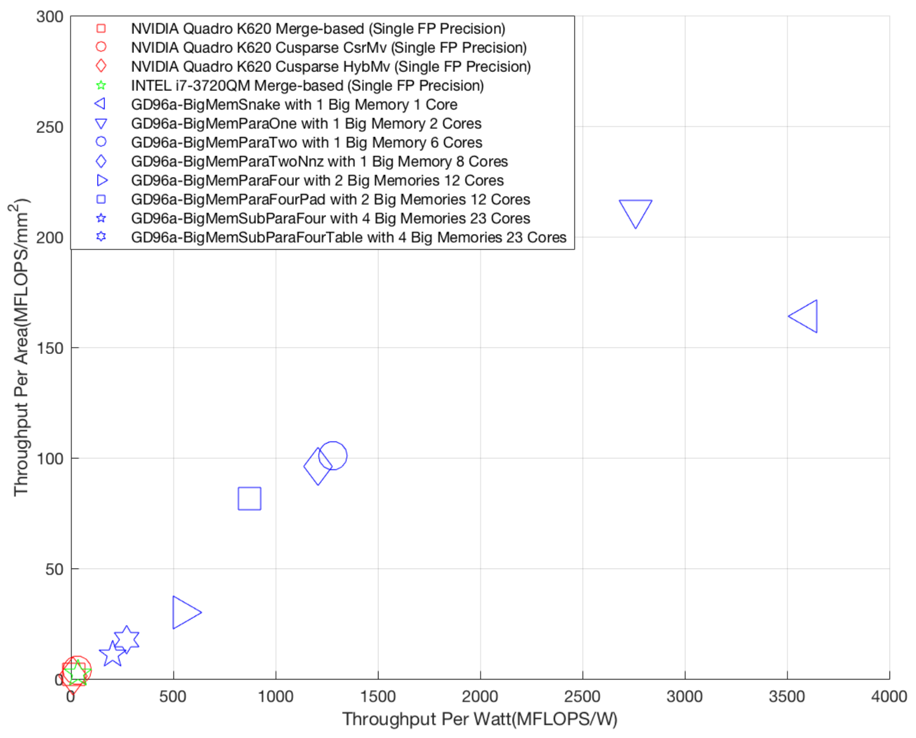


Figure 5.1: Throughput per area versus throughput per watt for various SpMV implementations operating on the GD96_a sparse matrix. All eight implementations on many-core platform for GD96_a are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.2. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 6.1-107.14× and 7.12-125× higher throughput per watt than the general-purpose processor and GPU designs.

implementations on the many-core platform provide 6.1-107.14× and 7.12-125× higher throughput per watt than the general-purpose processor and GPU designs.

5.5.2 Medium Size Matrix

As the size and density of the sparse matrix increases, implementations with multiple processing rows typically become more power efficient.

For relatively large and dense matrices, Figure 5.2 shows throughput per watt versus

Table 5.3: Throughput per area versus throughput per watt for various SpMV implementations operating on the Franz9 sparse matrix. All metrics data is plotted in Figure 5.2 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	30.2	218.2
Quadro K620 Cusparse CsrMv	30.4	219.8
Quadro K620 Cusparse HybMv	47.2	416.7
i7-3720QM Merge-based	41.6	338
BigMemSnake	68.5	1875
BigMemParaOne	146.1	**3200**
BigMemParaTwo	**289.5**	2804
BigMemParaTwoNnz	236.8	2423.7
BigMemParaFour	94.1	1453.1
BigMemParaFourPad	181.2	1967
BigMemSubParaFour	48.4	537.6
BigMemSubParaFourTable	59.3	644.1

throughput per area for all implementations on different platforms when performing SpMV using the sparse matrix Franz9. The minimum number of cores for the storage of vector x is first used for the *BigMemSnake* implementation, and then added to create additional implementations.

The relatively large size and NNZ of Franz9 motivate finding alternative implementations for SpMV. Modify the *BigMemSnake* implementation to use *BigMemParaOne* with an extra NNZ counter core to simplify logic control overhead, which increases throughput per area and throughput per watt because of the medium density of Franz9. The results on Figure 5.2 show that the *BigMemPara* and *BigMemSubPara* implementations attempt to achieve higher throughput per watt by increasing the number of arrays processed in parallel come to failure. Since with more cores, the throughput per watt is reduced because processing cores with relatively small chip sizes typically consume more power than shared memory.

The least efficient implementation is the merge-based algorithm in the NVIDIA Quadro K620. Although the column size of Franz9 is almost the median of all matrices of the sparse matrix database, even if a merge-based algorithm is used to balance the workload between all threads, the GPU cannot perform SpMV operations efficiently. Cusparse HybMv algorithm on Franz9 achieves 1.89× and 1.23× throughput per watt than other general-purpose processor and GPU

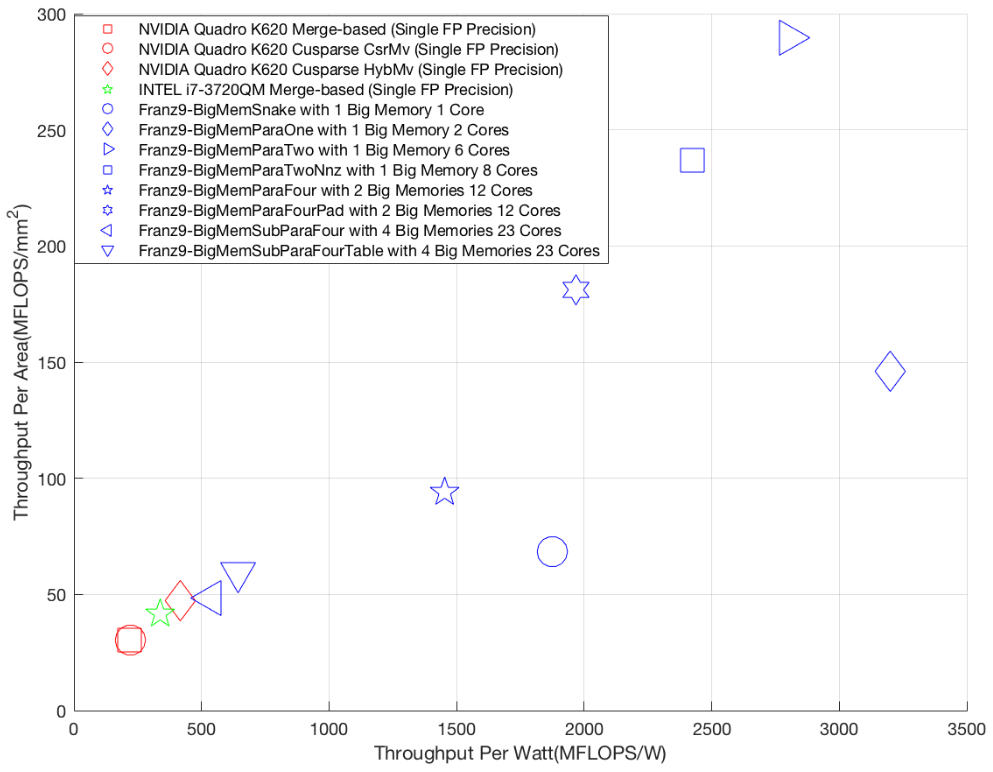


Figure 5.2: Throughput per area versus throughput per watt for various SpMV implementations operating on the Franz9 sparse matrix. All eight implementations on many-core platform for Franz9 are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.3. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 1.59-9.47× and 1.29-7.68× higher throughput per watt than the general-purpose processor and GPU designs.

implementations since the non-zero elements of this matrix are evenly distributed, enabling ELL format to improve throughput per watt versus throughput per area. Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per watt and throughput per area as the least efficient one.

The most efficient many-core implementation tends to achieve a balance between simplifying control logic of processing cores and doubling processing rows, namely the *BigMemParaOne* design.

The least efficient many-core implementation tends to use more cores for the sorting network, including the *BigMemSubPara* implementations.

All the implementations on many-core platform for Franz9 provide greater throughput per watt than the general-purpose processor-based and GPU-based implementations. Compared with the general-purpose processor and GPU designs, the throughput per watt on the many-core platform is increased by 1.59-9.47× and 1.29-7.68× respectively, as shown in Table 5.3.

5.5.3 Large and Dense Matrix

For larger and denser matrices, Figure 5.3 shows throughput per watt versus throughput per area for all implementations on different platforms when performing SpMV using the rail507 sparse matrix. The minimum number of cores to store the vector x is first used for the *BigMemSnake* implementation, and then increased to create additional implementations.

The large column size and NNZ of rail507 prompted the search of alternative implementations for SpMV. Since column size for rail507 is 63516, which means at least two big shared memory modules are needed for storing the vector x. While increasing the number of processing rows for the *BigMemParaOne* to handle multiplication in parallel, it provides nearly 2× and 4× throughput for *BigMemParaTwo* and *BigMemParaFour* implementations and is relatively small in run size because each shared memory module can be accessed by processing cores from nearby rows.

The results on Figure 5.3 show that the *BigMemParaFourPad* achieves the highest throughput per watt because the average NNZ per row of this matrix is 808, which is the densest one among all the simulated matrices and could be divided by 4. The second optimal *BigMemParaFour* implementation uses the same schema mapping as *BigMemParaFourPad*, which uses padding NNZ distribution kernel to provide higher throughput per watt while still delivering the same throughput per area.

The least efficient implementation is the Cusparse HybMv algorithm in the NVIDIA Quadro K620. Since the column size and density of the matrix are the highest for all simulated matrices, the merge based algorithm can effectively perform SpMV operations over the other two algorithms. Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per watt and throughput per area compared to the least efficient one.

The most efficient many-core implementation tends to increase the number of processing

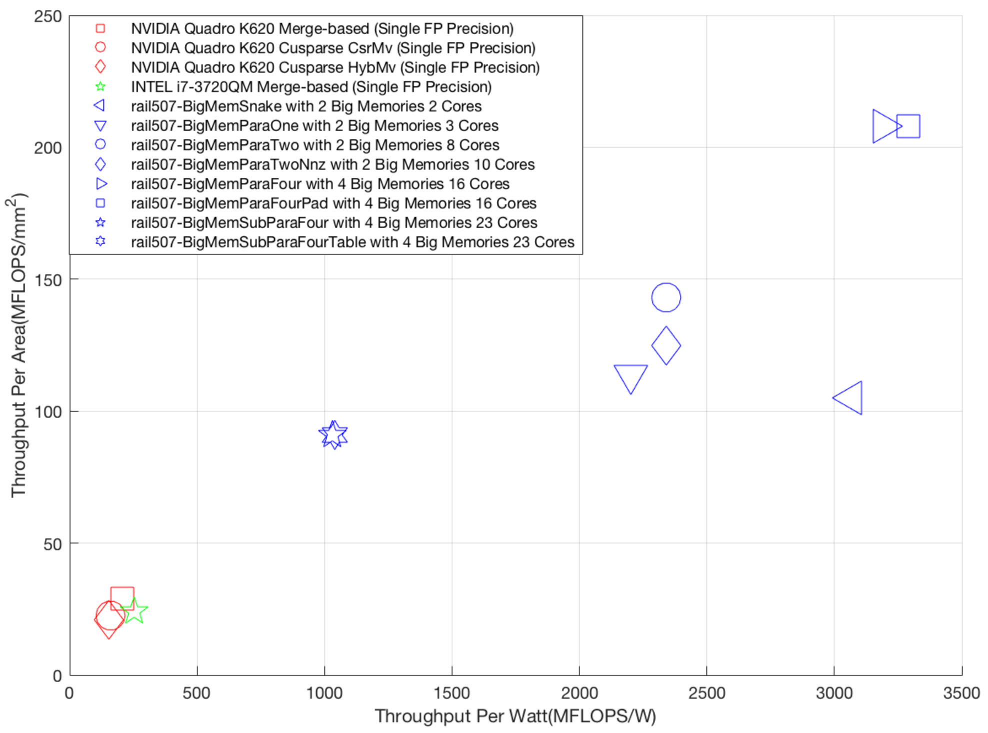


Figure 5.3: Throughput per area versus throughput per watt for various SpMV implementations operating on the rail507 sparse matrix. All eight implementations on many-core platform for rail507 are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.4. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 4.07-13.01× and 4.95-15.81× higher throughput per watt than the general-purpose processor and GPU designs.

rows while reducing control and traffic overhead, namely the *BigMemParaFourPad* design. The least efficient many-core implementation tends to use more cores for the sorting network, including the *BigMemSubPara* implementations.

All the implementations on many-core platform for rail507 offer greater throughput per watt than the general-purpose processor-based and GPU-based implementations. Compared with the general-purpose processor and GPU designs, the throughput per watt on the many-core platform is increased by 4.07-13.01× and 4.95-15.81× respectively, as shown in Table 5.4.

Table 5.4: Throughput per area versus throughput per watt for various SpMV implementations operating on the rail507 sparse matrix. All metrics data is plotted in Figure 5.3 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	28.8	208
Quadro K620 Cusparse CsrMv	22.4	161.6
Quadro K620 Cusparse HybMv	20.8	153
i7-3720QM Merge-based	24	252.8
BigMemSnake	105	3067
BigMemParaOne	113.6	2200
BigMemParaTwo	143	2340
BigMemParaTwoNnz	125	2340
BigMemParaFour	**208**	3190
BigMemParaFourPad	**208**	**3289**
BigMemSubParaFour	91.1	1029
BigMemSubParaFourTable	91.1	1042

5.6 Area Efficiency Comparison with other Work

The eight proposed SpMV implementations were simulated on 10 sparse real matrices chosen from Table 2.2 on page 17 using 1 to 23 processors on the many-core platform, which can improve throughput per area versus using the methods on the GPU or general-purpose processor. The implementations on the many-core platform increase throughput per area by up to 165.4× versus the general-purpose processor implementations, and by up to 102.4× versus the GPU implementations.

BigMemParaTwo is usually the most area efficient implementation for most simulated matrices, but as the matrix size (NNZ) changes, other implementations provide a higher throughput per area for some specific matrices.

5.6.1 Small and Sparse Matrix

For a relatively small and sparse matrix, Figure 5.4 shows throughput per watt versus throughput per area for all implementations on different platforms when performing SpMV using the GD01_a sparse matrix. The minimum number of cores for the storage of vector x is first used for the *BigMemSnake* implementation, and then increased to create additional implementations.

Since the processing core consumes relatively high power and the matrix size is small, increasing the number of cores of *BigMemSnake* beyond the number of cores required to store the vector x reduces the throughput per watt. The optimal *BigMemSnake* implementation uses the least amount of cores to store the vector x.

Modify the *BigMemSnake* implementation to use *BigMemParaOne* with an extra NNZ counter core to simplify the logic control overhead, thereby increasing throughput per area while reducing throughput per watt. The main reason is that the processing core typically consumes more power than the shared memory, which has three times the chip size of the processing core.

The relatively small size and NNZ of GD01_a reduce the motivation to find alternative implementations for SpMV. The results on Figure 5.4 show that the *BigMemPara* and *BigMemSubPara* implementations attempt to achieve higher throughput per area by increasing the number of arrays processed in parallel come to failure. Comparing to the optimal *BigMemParaOne*, as the implementation becomes more complex and uses more cores, throughput per watt and throughput per watt will decrease. The most area efficient implementations tend to use as few shared memory modules as possible while adding extra NNZ counter core to simplify the control and flow overhead, namely *BigMemParaOne*. The least efficient many-core implementations tend to use more cores for the sorting network, including the *BigMemSubPara* implementations.

The least efficient implementation is the Cusparse HybMv algorithm in the NVIDIA Quadro K620. The GPU is unable to perform the SpMV operations efficiently because the column size of the matrix is the smallest of all the simulated matrices, and it is difficult to balance the workload of each thread. Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per area and throughput per watt as the least efficient one.

All the implementations on many-core platform for GD01_a provide greater throughput per area than the general-purpose processor-based and GPU-based implementations. As shown in Table 5.5, the implementations on the many-core platform provide 16.15-165.38× and 10-102.38× higher throughput per area than the general-purpose processor and GPU designs.

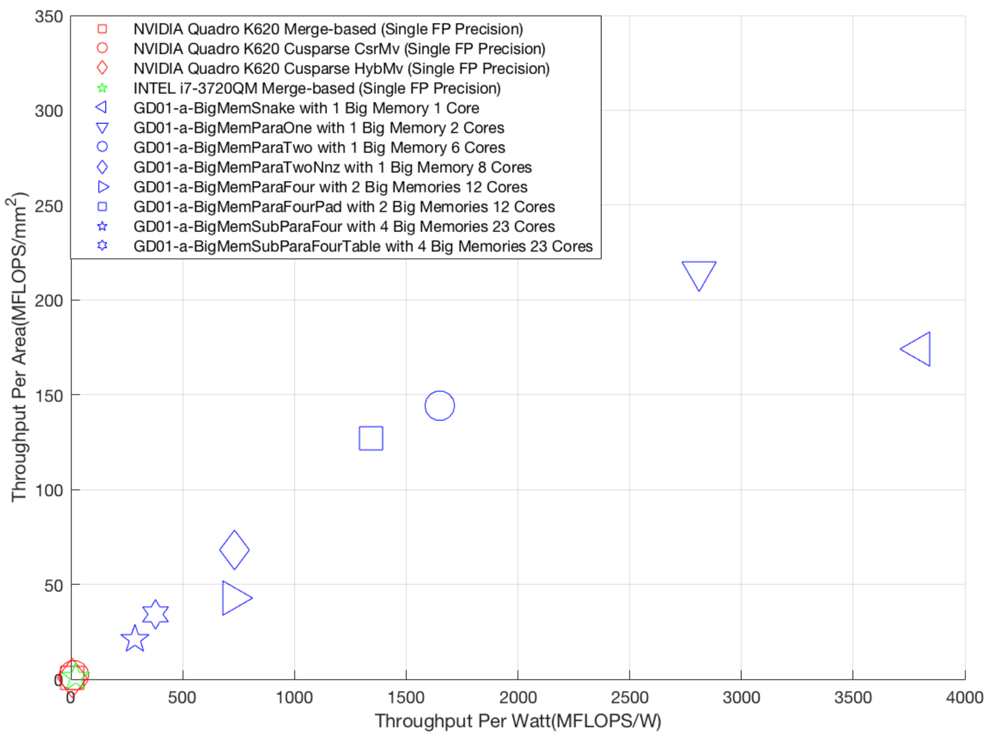


Figure 5.4: Throughput per area versus throughput per watt for various SpMV implementations operating on the GD01_a sparse matrix. All eight implementations on many-core platform for GD01_a are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.5. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 16.15-165.38× and 10-102.38× higher throughput per area than the general-purpose processor and GPU designs.

5.6.2 Medium Size Matrix

Implementations with multiple processing rows generally become more area efficient as the size and density of the sparse matrix increases.

For a relatively large and dense matrix, Figure 5.5 displays throughput per watt versus throughput per area for all implementations on different platforms when performing SpMV using the sparse matrix complex. The minimum number of cores to store the vector x is first used for the *BigMemSnake* implementation, and then increased to create additional implementations.

Table 5.5: Throughput per area versus throughput per watt for various SpMV implementations operating on the GD01_a sparse matrix. All metrics data is plotted in Figure 5.4 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	0.9	6.4
Quadro K620 Cusparse CsrMv	2.1	14.1
Quadro K620 Cusparse HybMv	0.6	4.3
i7-3720QM Merge-based	1.3	20
BigMemSnake	174	**3800**
BigMemParaOne	**215**	2810
BigMemParaTwo	144	1651
BigMemParaTwoNnz	68	732
BigMemParaFour	43	724
BigMemParaFourPad	127	1344
BigMemSubParaFour	21	285
BigMemSubParaFourTable	34	380

The relatively large size and NNZ of complex motivate finding alternative implementations for SpMV. Modify the *BigMemSnake* implementation to use *BigMemParaOne* with extra NNZ counter core to simplify logic control overhead, thereby reducing throughput per watt and throughput per area. Due to the medium size and density of matrix complex, only one processing core and shared memory will be needed for *BigMemSnake* so that extra NNZ counter core will hurt throughput per area versus throughput per watt. The results on Figure 5.5 show that the *BigMemPara* implementations attempt to achieve higher throughput per area by increasing the number of arrays processed in parallel succeed. The most efficient implementation is *BigMemParaTwo*, which is the simplest implementation in which big shared memory is connected to two processing cores on neighbour arrays, making full use of arbitrary access of shared memory to save area consumption. The throughput per area decreases as implementation uses more processing arrays since the relative low density of this matrix limits doubling the throughput while the number of processing core doubles, as shown in Figure 5.5.

The least efficient implementation is the merge-based algorithm in the NVIDIA Quadro K620. GPU cannot perform SpMV operations efficiently because the column size of matrix complex is just 1408, even if a merge-based algorithm is used to balance the workload between all threads.

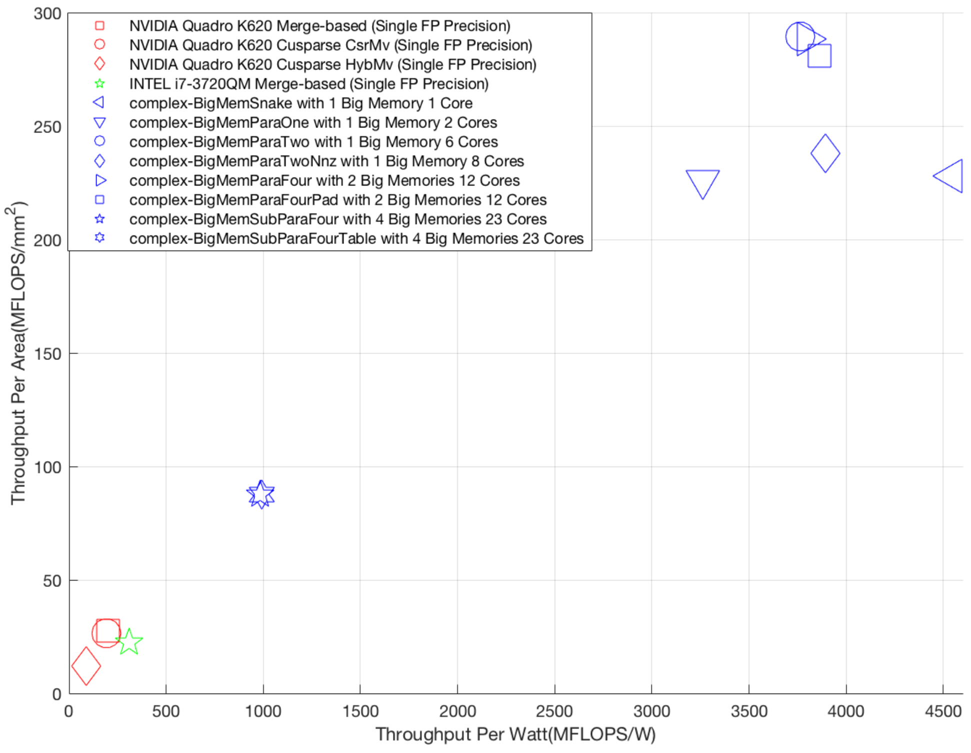


Figure 5.5: Throughput per area versus throughput per watt for various SpMV implementations operating on the complex sparse matrix. All eight implementations on many-core platform for complex are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.6. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 3.91-12.92× and 3.16-10.45× higher throughput per area than the general-purpose processor and GPU designs.

Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per area and throughput per watt, compared to the least efficient one.

The most area efficient many-core implementation tends to achieve a balance between increasing number of processing rows and simplifying control logic between processing cores, namely the *BigMemParaTwo* design. The least efficient many-core implementations tend to use more cores for a sorting network, including the *BigMemSubPara* implementations.

All implementations on many-core platform for complex provide greater throughput per

Table 5.6: Throughput per area versus throughput per watt for various SpMV implementations operating on the complex sparse matrix. All metrics data is plotted in Figure 5.5 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	27.7	199.8
Quadro K620 Cusparse HybMv	26.6	192
Quadro K620 Cusparse CsrMv	12.2	88
i7-3720QM Merge-based	22.4	308.8
BigMemSnake	228	**4545**
BigMemParaOne	226	3263
BigMemParaTwo	**289.5**	3763
BigMemParaTwoNnz	238	3892
BigMemParaFour	288.5	3800
BigMemParaFourPad	281	3861
BigMemSubParaFour	87.5	982
BigMemSubParaFourTable	87.5	994

watt than the general-purpose processor-based and GPU-based implementations. Compared with the general-purpose processor and GPU designs, the throughput per area on the many-core platform is increased by 3.91-12.92× and 3.16-10.45× respectively, as shown in Table 5.6.

5.6.3 Large and Dense Matrix

For large and dense matrices, Figure 5.6 shows throughput per watt versus throughput per area for all implementations on different platforms when performing SpMV using the sparse matrix rail582. The minimum number of cores for the storage of vector x is first used for the *BigMemSnake* implementation, and then increased to create additional implementations.

The large size and NNZ of rail582 motivate finding alternative implementations for SpMV. Since the column size for this matrix is 56097, which means at least two big shared memory modules are needed for storing the vector x. By increasing the number of processing rows for the *BigMemParaOne* to handle multiplication in parallel, it provides nearly 2× and 4× throughput for *BigMemParaTwo* and *BigMemParaFour* implementations, and is relatively small in run size because big shared memory module can be shared by processing cores from nearby rows.

The results on Figure 5.6 show that the *BigMemParaFourPad* achieves the highest

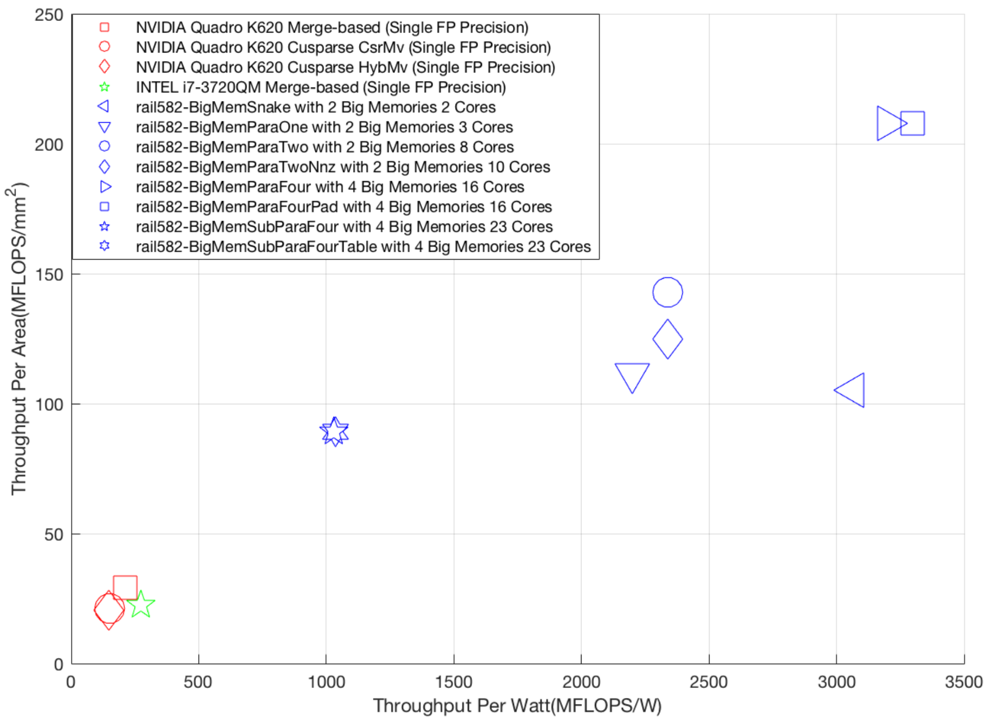


Figure 5.6: Throughput per area versus throughput per watt for various SpMV implementations operating on the rail582 sparse matrix. All eight implementations on many-core platform for rail582 are compared against general-purpose processor and GPU implementations. The evaluation precision of all benchmarks is single-precision floating point (32 bit). Throughput per watt and throughput per area of all implementations are shown in Table 5.7. The optimal design has the largest throughput per watt and throughput per area. The implementations on many-core platform provide 3.97-9.29× and 3.04-7.09× higher throughput per area than the general-purpose processor and GPU designs.

throughput per area because the average NNZ per row for this matrix is 691, which is the second densest one of all simulated matrices. The second optimal *BigMemParaFour* implementation uses the same architectural mapping as *BigMemParaFourPad*, which uses a padding NNZ distribution kernel to provide higher throughput per watt while still delivering the same throughput per area.

The least efficient implementation is the Cusparse HybMv algorithm in the NVIDIA Quadro K620. Since the column size and density of the matrix are the second highest of all simulated matrices, the merge based algorithm can effectively perform SpMV operations over the other two

Table 5.7: Throughput per area versus throughput per watt for various SpMV implementations operating on the rail582 sparse matrix. All metrics data is plotted in Figure 5.6 and the highest throughput per area versus throughput per watt of various implementations are in bold.

Implementation/Benchmark (Single FP Precision)	Throughput Per Area (MFLOPS/mm^2)	Throughput Per Watt (MFLOPS/W)
Quadro K620 Merge-based	29.3	211.2
Quadro K620 Cusparse CsrMv	21.1	152
Quadro K620 Cusparse HybMv	20.5	148.3
i7-3720QM Merge-based	22.4	272
BigMemSnake	105	3067
BigMemParaOne	111.6	2200
BigMemParaTwo	143	2340
BigMemParaTwoNnz	125	2340
BigMemParaFour	**208**	3200
BigMemParaFourPad	**208**	**3299**
BigMemSubParaFour	89	1030
BigMemSubParaFourTable	89	1036

algorithms. Other GPU and general-purpose processor implementations provide the same order of magnitude throughput per area and throughput per watt, compared to the least efficient one. The most area efficient implementation tends to increase the number of processing rows while reducing control and flow overhead, namely the *BigMemParaFourPad* design. The least efficient many-core implementation tends to use more cores for the sorting network, including the *BigMemSubPara* implementations.

All implementations on many-core platform for rail582 provide greater throughput per area than the general-purpose processor-based and GPU-based implementations. Compared with the core general-purpose processor and GPU designs, the throughput per area on the many-core platform is increased by 3.97-9.29× and 3.04-7.09× respectively, as shown in Table 5.7.

5.7 Performance Comparison Summary

In addition to the six sample matrices analyzed in this chapter, the throughput per area versus throughput per watt for various SpMV implementations operating on other four real sparse matrices from Table 2.2 on page 17 were also simulated, as shown in Table 5.8.

Table 5.8: Performance Comparison of AsAP3 with general-purpose processor and GPU implementations. The evaluation precision of all implementations is single-precision 32-bit IEEE-754 format. AsAP3 metrics are normalized against the comparison device, and are scaled to the same technology using data from Holt [68].

Matrix Name	Throughput/Area (i7 3720QM)	Throughput/Watt (i7 3720QM)	Throughput/Area (Quadro 620)	Throughput/Watt (Quadro 620)
GD96_a	88.3	107.1	53	125
Franz9	7	9.5	6.1	7.7
as-caida	16.4	45.5	11.3	30.3
GD97-c	78.6	112.9	88.2	214.7
GD01-a	**165.4**	**190**	**102.4**	**269.5**
GD01-Acap	75	64.6	56	107.3
GD00-c	83.7	119	58.7	128
rail507	8.7	13	7.2	15.8
rail582	7.1	12.1	9.3	15.6
complex	12.9	14.7	10.4	22.8

Table 5.8 shows that implementing SpMV on the many-core platform increases throughput per watt by 68.8× on average, and as much as 190× versus the general-purpose processor implementations, and by 93.7× on average, up to 269.5× versus the GPU implementations. Throughput per area is increased by 54.3× on average, and up to 165.4× versus the general-purpose processor implementations, and by 40.3× on average, as much as 102.4× versus the GPU implementations.

Chapter 6

Thesis Summary and Future Work

6.1 Thesis Summary

This thesis summarizes the current research and challenges in many-core SpMV algorithms, and why the topic is relevant. The background information about the matrix database and the different sparse matrix compression formats to be explored are given. The large 2D mesh architecture used throughput the thesis, AsAP3 (KiloCore), is explained.

It continues to demonstrate and explore the SpMV implementations on a many-core platform (AsAP3). Three main scenarios: *BigMemSnake*, *BigMemPara*, *BigMemSubPara*, which consist of one or more of the three main phases of SpMV: NNZ distribution network, Sorting network and Processing array are explored. Based on these three main scenarios and phases, a total of eight functionally equivalent sparse matrix dense vector multiplication implementationns are created.

Two main metrics for various SpMV implementations: throughput per watt and throughput per area are used to measure the area and power efficiency of various designs on different platforms.

To measure against SpMV implementations on the GPU and general-purpose processor, the classic sparse matrix APIs [33,34] and the most advanced benchmarks [7] are used for implementing SpMV. The evaluation precision of all benchmarks is single-precision 32-bit IEEE-754 format. The throughput, power, and area consumption of all implementations are measured for performing SpMV on unstructured sparse matrices from distinct scientific workloads of varying sizes. The many-core implementations increase power efficiency by 68.8× versus the general-purpose processor SpMV on average, and by 93.7× versus the GPU SpMV on average. Simultaneously, they provide 54.3×

improvement in area efficiency versus the general-purpose processor SpMV on average, and 40.3× improvement versus the GPU SpMV on average.

This summary is followed by a list of the author's proposed future work.

6.2 Future Work

This thesis explores implementing SpMV on a many-core platform (AsAP3). However, due to time limitation, this thesis has not tested huge matrix with column size or NNZ larger than millions. The most complex but least efficient kernel *BigMemSubPara* for relatively small matrix may help to explore larger matrix since it will use fewer processing cores than the other two main kernels:*BigMemSnake* and *BigMemPara*, which means achieving more power and area efficiency. From the perspective of architecture, increasing the amount of on-chip shared memory and streaming instructions from a shared memory to a neighboring processor will be helpful, especially for designing more complex kernels for huge matrices. The exploration of relatively large matrix for other SpMV kernels is left as a future research endeavor.

Furthermore, in addition to sparse matrix vector multiplication, there are many other common scientific kernels that could be implemented on a many-core platform to explore trade-offs and efficiency, including general sparse matrix-matrix multiplication, 2D stencil computations, and lattice quantum chromodynamics (QCD).

Glossary

API Application Programming Interface. APIs are a set of programming routines, and tools for building software.

AsAP3 The third generation Asyncronous Array of simple Processors (AsAP) chip. AsAP3 is a fine-grained many-core system with 1000 independently clocked homogeneous programmable processors.

BCCOO Blocking Coordinate format. BCCOO divides the matrix into blocks to reduce the access to column index, and store in COO format.

CMOS Complementary Metal Oxide Semiconductor. CMOS circuits are based on field-effect transistors and use both n-channel and p-channel transistors. Most modern chips use CMOS technology.

COO Coordinate format. In this storage format, three arrays are used to store the matrix. The arrays have the same number of non zeros as the original matrix. One of the arrays holds the matrix entries, and the other two arrays hold the column and row index of the matrix entries from the original matrix.

CSR Compressed sparse row. A format represents a matrix A by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices.

CUDA Compute Unified Device Architecture. An application programming interface (API) by NVIDIA to enable general purpose processing on a GPU.

cuSPARSE The NVIDIA CUDA Sparse Matrix library. It provides a collection of basic linear algebra subroutines used for sparse matrices that delivers up to 8x faster performance than the latest MKL.

ELL Ellpack-Itpack generalized diagonal format. In this storage format, two rectangular arrays are used to store the matrix. The arrays have the same number of rows as the original matrix, but only have as many columns as the maximum number of nonzeros on a row of the original matrix. One of the arrays holds the matrix entries, and the other array holds the column numbers from the original matrix.

FIFO First in, first out. A method for organizing and manipulating a data buffer, where data are sent out in the same order in which they were received.

flops floating point operations per second. Flops is a measure of computer performance, useful in fields of scientific computations that require floating-point calculations.

GPGPU general-purpose GPU. GPGPU is a graphics processing unit (GPU) that performs non-specialized calculations that would typically be conducted by the central processing unit (CPU). Ordinarily, the GPU is dedicated to graphics rendering.

HYB Hybrid format. The purpose of the HYB format is to store the typical number of nonzeros per row in the ELL data structure and the remaining entries of exceptional rows in the COO format.

IEEE-754 A techincal standard for FP arithmetic and data representation. The standard specifies a set of formats, operations, rounding rules, flags, and the handling of exceptions.

Intel MKL Intel Math Kernel Library. A library of math processing routines.

LSB Least Significant Bit. In this thesis, LSB refers to the right-most bit of a binary value.

MIMD Multiple Instruction, Multiple Data. MIMD is a type of parallel architecture where different processors may be executing different instructions on different pieces of data.

NNZ Number of non-zeros.

OpenMP Open Multi-Processing. It is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

PD-SOI Partially Depleted Silicon On Insulator. A fabrication technology where partially depleted layered silicon is placed on an insulator, which is then placed on a silicon substrate. SOI lowers parasitic capacitance by isolating the silicon junction using an insulator.

QCD Quantum chromodynamics. It is the theory of the strong interaction between quarks and gluons, the fundamental particles that make up composite hadrons such as the proton, neutron and pion.

SpMV Sparse Matrix-Vector Multiplication. A common scientific kernel involving the multiplication of a sparse matrix with a dense vector.

TDP Thermal design power. It is the maximum amount of heat generated by the CPU that the cooling system in a computer is required to dissipate in typical operation.

TLB Translation lookaside buffer. It is a cache that memory management hardware uses to improve virtual address translation speed.

Xeon Phi Xeon Phi. Xeon Phi is a series of x86 manycore processors designed and made entirely by Intel. They are intended for use in supercomputers, servers, and high-end workstations. Its architecture allows use of standard programming languages and APIs such as OpenMP.

Bibliography

[1] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, April 2017.

[2] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[3] Josef Stoer and Roland Bulirsch. Introduction to numerical analysis. 12, 2013.

[4] Kadir Akbudak, Enver Kayaaslan, and Cevdet Aykanat. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM Journal on Scientific Computing*, 35(3):C237–C262, 2013.

[5] Sam Williams, Nathan Bell, Jee Whan Choi, Michael Garland, Leonid Oliker, and Richard Vuduc. Sparse matrix-vector multiplication on multicore and accelerators. *Scientific Computing with Multicore and Accelerators*, pages 83–109, 2010.

[6] Satoshi Ohshima, Takahiro Katagiri, and Masaharu Matsumoto. Performance optimization of spmv using crs format by considering openmp scheduling on cpus and mic. In *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on*, pages 253–260. IEEE, 2014.

[7] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. IEEE Press, 2016.

[8] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. Accelerating spmv on fpgas by compressing nonzero values. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 64–67. IEEE, 2015.

[9] Yan Zhang, Yasser H Shalabi, Rishabh Jain, Krishna K Nagar, and Jason D Bakos. Fpga vs. gpu for sparse matrix vector multiply. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 255–262. Citeseer, 2009.

[10] Yun Liang, Wai Teng Tang, Ruizhe Zhao, Mian Lu, Huynh Phung Huynh, and Rick Siow Mong Goh. Scale-free sparse matrix-vector multiplication on many-core architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12):2106–2119, 2017.

[11] Athena Elafrou, Georgios Goumas, and Nectarios 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.

[12] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc*, volume 2005, pages 445–448, 2005.

[13] Shizhao Chen, Jianbin Fang, Donglin Chen, Chuanfu Xu, and Zheng Wang. Optimizing sparse matrix-vector multiplication on emerging many-core architectures. *arXiv preprint arXiv:1805.11938*, 2018.

[14] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep 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.

[15] Robert J Gove, Keith Balmer, Nicholas K Ing-Simmons, and Karl M Guttag. Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation, May 18 1993. US Patent 5,212,777.

[16] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598. IEEE, 2008.

[17] Nathan Bell and Michael 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.

[18] Yongchao Liu and Bertil Schmidt. Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 82–89. IEEE, 2015.

[19] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[20] Javed Razzaq, Rudolf Berrendorf, Soenke Hack, Max Weierstall, and Florian Manuss. Fixed and variable sized block techniques for sparse matrix vector multiplication with general matrix structures. 2016.

[21] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaspmv: yet another spmv framework on gpus. In *Acm Sigplan Notices*, volume 49, pages 107–118. ACM, 2014.

[22] Wai Teng Tang, Wen Jun Tan, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu. *IEEE Transactions on Parallel and Distributed Systems*, 26(9):2373–2385, 2015.

[23] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 26. ACM, 2013.

[24] Ankit Jain. *pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures.* PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

[25] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[26] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 30. ACM, 1999.

[27] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 307–316. ACM, 2006.

[28] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of research and development*, 41(6):711–725, 1997.

[29] George Chrysos. Intel® xeon phi^{TM} coprocessor-the architecture. *Intel Whitepaper*, 176, 2014.

[30] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan 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.

[31] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[32] Jee W Choi, Amik Singh, and Richard W Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *ACM sigplan notices*, volume 45, pages 115–126. ACM, 2010.

[33] MKL Intel. Intel math kernel library. 2007.

[34] CUsparse Toolkit Documentation. v7. 5, nvidia corporation, sep 2015.

[35] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James 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.

[36] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):30, 2017.

[37] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A fine-grained 1,000-processor array for task parallel applications. *IEEE Micro*, 37(2):63–69, March 2017.

[38] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[39] Aaron Stillmaker, Brent Bohnenstiehl, and Bevan Baas. The design of the kilocore chip. In *ACM/IEEE Design Automation Conference*, Austin, TX, Jun. 2017.

[40] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array. In *Symposium on VLSI Circuits*, June 2016.

[41] Tinoosh Mohsenin and Bevan M. Baas. Split-row: A reduced complexity, high throughput LDPC decoder architecture. In *IEEE International Conference on Computer Design (ICCD)*, October 2006.

[42] Z. Yu and B. M. Baas. High performance, energy efficiency, and scalability with GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):66–79, January 2009.

[43] Z. Yu and B. M. Baas. A low-area multi-link interconnect architecture for GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(5):750–762, May 2010.

[44] Anh T. Tran and Bevan M. Baas. Achieving high-performance on-chip networks with shared-buffer routers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(6):1391–1403, June 2014. Date of publication July 3, 2013.

[45] A.T. Tran and B.M. Baas. DLABS: a dual-lane buffer-sharing router architecture for networks on chip. In *Signal Processing Systems, 2010. SiPS 2010. IEEE Workshop on*, pages 331–336, October 2010.

[46] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(10):1125–1134, October 2007.

[47] Bin Liu and Bevan M. Baas. Parallel AES encryption engines for many-core processor arrays. *Computers, IEEE Transactions on*, 62(3):536–547, march 2013.

[48] Tinoosh Mohsenin and Bevan M. Baas. High-throughput LDPC decoders using a multiple split-row method. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 2007.

[49] Aaron Stillmaker, Lucas Stillmaker, and Bevan Baas. Fine-grained energy-efficient sorting on a many-core processor array. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 652–659, December 2012.

[50] Jon J. Pimentel and Bevan M. Baas. Hybrid floating-point modules with low area overhead on a fine-grained processing core. In *IEEE Asilomar Conference on Signals, Systems and Computers*, November 2014.

[51] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An asynchronous array of simple processors for DSP applications. In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume 49, pages 428–429, 663, February 2006.

[52] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Dean Truong, Tinoosh Mohsenin, and Bevan Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits (JSSC)*, 43(3):695–705, March 2008.

[53] Z. Yu and B. M. Baas. Implementing tile-based chip multiprocessors with GALS clocking styles. In *IEEE International Conference on Computer Design (ICCD)*, October 2006.

[54] Z. Yu and B. Baas. Performance and power analysis of globally asynchronous locally synchronous multi-processor systems. In *IEEE Computer Society Annual Symposium on VLSI*, March 2006.

[55] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A fine-grained many-core platform for DSP applications. *IEEE Micro*, 27(2):34–45, March 2007.

[56] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, A. T. Tran, Z. Xiao, E. W. Work, J. W. Webb, P. Mejia, and B. M. Baas. A 167-processor computational platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 44(4):1130–1144, April 2009.

[57] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor computational array for highly-efficient DSP and embedded application processing. In *IEEE HotChips Symposium on High-Performance Chips*, August 2008.

[58] A. T. Tran, D. N. Truong, and B. M. Baas. A reconfigurable source-synchronous on-chip network for GALS many-core platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):897–910, June 2010.

[59] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *Symposium on VLSI Circuits*, pages 22–23, June 2008.

[60] Bin Liu, Mohammad H. Foroozannejad, Soheil Ghiasi, and Bevan M. Baas. Optimizing power of many-core systems by exploiting dynamic voltage, frequency and core scaling. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2015.

[61] A.T. Tran, D.N. Truong, and B.M. Baas. A GALS many-core heterogeneous DSP platform with source-synchronous on-chip interconnection network. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 214–223, May 2009.

[62] Anh Tran, Dean Truong, and Bevan Baas. A complete full-rate 802.11a baseband reciever implemented on an array of programmable processors. In *Asilomar Conference on Signals, Systems and Computers*, October 2008.

[63] Anh Tran and Bevan Baas. Design of an energy-efficient 32-bit adder operating at subthreshold voltages in 45-nm CMOS. In *International Conference on Communications and Electronics*, August 2010.

[64] Paul Husted and Bevan Baas. Method and apparatus for transient frequency distortion compensation, March 2008. US Patent 7,340,265.

[65] Jon Pimentel. *Methods for Reducing Floating-Point Computation Overhead*. PhD thesis, University of California, Davis, CA, USA, August 2017. http://vcl.ece.ucdavis.edu/pubs/theses/2017-2.pimentel/.

[66] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 283–292. IEEE, 2008.

[67] Stephen A Jarvis, Steven A Wright, and Simon D Hammond. *High performance computing systems. Performance modeling, benchmarking and simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*, volume 8551. Springer, 2014.

[68] William M Holt. 1.1 moore's law: A path going forward. In *Solid-State Circuits Conference (ISSCC), 2016 IEEE International*, pages 8–13. IEEE, 2016.