# Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors

Kaixi Hou, Wu-chun Feng
*Department of Computer Science*
*Virginia Tech, Blacksburg, USA*
{*kaixihou, wfeng*}*@vt.edu*

Shuai Che
*AMD Research*
*Bellevue, USA*
*Shuai.Che@amd.com*

***Abstract*—Because sparse matrix-vector multiplication (SpMV) is an important and widely used computational kernel in many real-world applications, it behooves us to accelerate SpMV on modern multi- and many-core architectures. While many storage formats have been developed to facilitate SpMV operations, the compressed sparse row (CSR) format is still the most popular and general storage format. However, parallelizing CSR-based SpMV on multi- and many-core processors (e.g., CPUs, APUs, GPUs) remains a challenging problem, including dealing with uncoalesced memory access, balancing workload, and identifying the most appropriate parallelizing strategy.**

**In the paper, we propose a novel auto-tuning framework that automatically finds the most efficient parallelizing strategy to achieve high-performance SpMV. Our framework can determine the right binning schemes to group similar workloads into bins (e.g., buckets) with negligible overhead. Then, for each bin, the most suitable kernel is selected to process the rows within. Our framework is input-aware and based on a machine-learning method. The results show that our auto-tuned SpMV performs significantly better than the default SpMV. The speedups on 16 representative matrices range from 1.2x to 52.0x. Compared to the state-of-the-art SpMV kernel, our work yields better performance in most cases, achieving up to a 1.9x speedup.**

## I. Introduction

Sparse matrix-vector multiplication (SpMV) is an important computational kernel in sparse linear system solvers, and more broadly, many real-world applications [1], [2], [3]. Hence, there exists a need to accelerate SpMV kernels on modern multi-/many-core platforms, including many library efforts designed to take advantage of the underlying parallel computing resources [4], [5]. Due to the trade-off between performance and space in sparse storage formats, many formats have been proposed to handle one or more types of matrices on various dedicated platforms (e.g., BCCOO [6], BRC [7], and CSR5 [8]). Among them, however, compressed sparse row (CSR) is one of the most widely adopted, general-purpose formats, especially with respect to CPU implementations [9], [10], [11], [12], [13], [14]. Moreover, the transformation between different formats is non-negligible in terms of performance, and the situation is worse when one is working across devices in a heterogeneous computing environment (e.g., APUs). Thus, we focus on the CSR format in this paper.

Throughput-oriented processors, such as graphics processing units (GPU), are well-known for efficiently solving regular and compute-intensive problems. However, dealing with sparse matrices in the CSR format is non-trivial and brings several challenges. First, irregular access to the non-zero elements (in memory) are usually non-contiguous, leading to an uncoalesced access pattern, which is not amenable for GPU execution. Second, load imbalance occurs due to having independent rows of different lengths. Specifically, resource underutilization occurs when some SIMD lanes become inactive when executing branches. Third, though we can group similar workloads together (i.e., binning) and then conduct the computation (i.e., kernel execution) for the SpMV, it is non-trivial to figure out the appropriate grouping policies, computational kernels, and their interactive relationships as they may depend on several factors, including the shape of the input sparse matrices and the underlying architectures.

This paper proposes a novel auto-tuning framework to automatically find the most efficient parallelizing strategies for computations with sparse data structures; different strategies are applied to solving subproblems (e.g., different rows or regions of rows). With respect to different sparse matrices, our framework can determine the optimal binning schemes to place similar workloads into different bins with negligible overhead, and associated with each bin, it can find the most suitable kernel to process the workloads within. This paper uses SpMV as a case study, and the work can be directly applied to other kernels with different potential implementations for different inputs, such as some general computational kernels [15], [16], [17] and domain-specific applications [18], [19], [20], [21].

Our framework is based on a machine-learning model (i.e., C5.0 decision tree [22]), where the off-line training process extracts sparsity feature parameters from the given sparse matrices. Subsequently, the model evaluates and identifies the best combinations of binning schemes and kernels from our candidate pools, which include one coarse-grained binning method with various binning granularities and nine

different SpMV kernels with the same functionality.

Our auto-tuning framework differs from previous auto-tuning work for SpMV, which has focused on searching for the best storage format based on the input sparse matrices and underlying architecture [10], [23]. In contrast, we build our framework using the widely adopted format of CSR, and the auto-tuning process occurs at the algorithmic level, along with traditional parameter tuning. In addition, we make the following two major contributions: (1) the extraction of reusable patterns over different sparse matrices, binning schemes and optimized kernels, which will later assist in the selection of parallelization strategy; and (2) the dynamic scheduling of binning schemes and kernels at run time, which in turn, automatically delivers better speedups.

To empirically demonstrate our framework, we conduct experiments on an AMD APU with HSA (Heterogeneous System Architecture) and SNACK (Structured No API Compiled Kernels) [24] and show that our approach performs significantly better than the single-kernel SpMV. The speedups on 16 representative matrices range from 1.2x to 52.0x. Compared to the state-of-the-art SpMV kernel [11], our work also yields better performance in most cases, achieving up to a 1.9x additional speedup.

## II. Background

In this section, we provide background on the Heterogeneous System Architecture (HSA), programming models, and the compressed sparse row (CSR) format.

### A. Heterogeneous System Architecture

The Heterogeneous System Architecture (HSA) [25] defines a system architecture with the goal to make heterogeneous computing efficient and programming easier. It targets systems that contain multiple processing units, such as CPUs, GPUs, and other accelerators. HSA allows any accelerator to operate as a peer to CPU rather than the traditional offload model. HSA specifies an intermediate language called HSAIL, which is an intermediate instruction set for parallel processing with a memory model that is compatible with C++11 and a runtime for dispatching tasks to hardware queues. The current OpenCL™2.0 [26] standard supports many features of HSA.

An important feature of both HSA and OpenCL 2.0 is that each defines a unified virtual address space for compute devices. CPUs and GPUs traditionally have disjoint memory spaces. In contrast, HSA allows both devices to access memory through the same virtual address. The system is further simplified in that it only needs to manage one set of page tables. This shared virtual memory (SVM) feature is partly enabled by the I/O memory-management unit (IOMMU). SVM allows users to directly use the same pointers and share data structures on both the host and the device. Programmers can allocate memory regions with memory allocators such as `malloc` and use them on both the CPU and the GPU (without the need of copying data).

SNACK (Structured No API Compiled Kernels) is a programming API that enables easy programming of HSA. It hides the complexity of dealing with the low-level HSA API. SNACK also comes with a CL Offline Compiler (CLOC) [24] that allows OpenCL kernels to be programmed directly and compiled to HSAIL.

In SNACK, programmers can directly call a GPU kernel on the host similar to a C function call. The only requirement is that programmers use `SNK_INIT_LPARM` to specify kernel launch parameters in a structure (e.g., global and local sizes) and pass it to a kernel as a function argument (i.e., the last *arg*). GPU kernels in SNACK can be written in OpenCL. Programmers use SNACK to compile an OpenCL kernel with $-c$ option, and an object file including a wrapper function with HSA API calls and HSA brig kernel is generated. This object file can be linked with other object files and the HSA runtime library to generate the final executable. The GPU kernel is finalized (i.e., to machine ISA) the first time it is launched.

In this paper, we use SNACK with kernels written in OpenCL on the AMD accelerated processing unit (APU), which is an HSA platform that consists of an x86 multicore CPU and an AMD Graphics Core Next (GCN) GPU [27]. Each compute unit (CU) on the GPU contains one scalar unit and four vector units. Each vector unit contains an array of 16 processing elements (PEs). Each PE consists of one ALU. The four vector units use SIMD execution of a scalar instruction. Each CU contains a data cache for the scalar unit, a L1 data cache and a local data share (LDS) (i.e., software-managed scratchpad). The CPU and GPU share the same DRAM controller.

### B. CSR-based Matrix-Vector Multiplication

Sparse matrices appear in many real-world applications, which comprise different computational patterns. Accordingly, different storage formats have been proposed to facilitate the efficient storage and retrieval of meaningful data in matrices, such as COO, ELL, and DIA. The paper focuses on the compressed sparse row (CSR) format for three major reasons. First, it is a widely adopted, general-purpose storage format that represents sparse matrices without any assumption of the sparsity structure and is favored in many application domains [10], [11]. Second, because of the popularity of the CSR format, one usually needs to transform CSR matrices to another format if it could give superior performance at the price of significant overhead [11]. Third, in the HSA environment, such space and time overhead of transformation can be avoided, leading to potential benefits.

Here we focus our study on the CSR-based sparse matrices. Figure 1 shows an example of the CSR storage format on a 4x4 matrix with eight non-zero elements. To represent the matrix in CSR format, we use three arrays to represent

the matrix: (1) *rowPtr* maintains the offsets of the each row's first non-zero in *colIdx* and *val*; (2) *colIdx* stores all the column indices of the non-zeros in the row-major order; and (3) *val* keeps the corresponding values of the non-zeros.

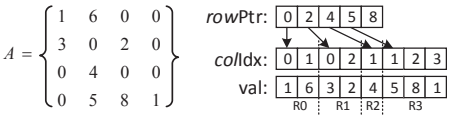



Figure 1: The CSR format to store an exemplar 4x4 sparse matrix

In order to perform the CSR-based sparse matrix-vector (SpMV) multiplication, we compute the products of each non-zero element of one row from the input matrix *A* and the corresponding value from the input vector *v*. Then, we accumulate the products into one value and store it in the resultant vector *u*. Algorithm 1 shows the basic sequential algorithm to conduct the SpMV over the sparse matrix *A* and dense vector *v*. Because the calculation of one row is independent from that of another line (see line 4), the most straightforward parallelization might be to make each thread work on one row and write the result to an exclusive position in the vector *u*. Unfortunately, however, this scheme results in poor performance due to the resulting unbalanced workload and uncoalesced memory access patterns (see line 9).

**Algorithm 1:** The basic CSR-based SpMV algorithm

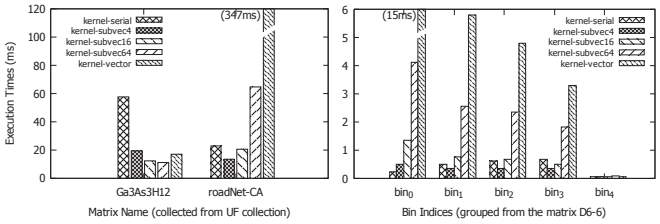
```
   /* rowPtr, colIdx, val: CSR data structures of matrix A */
1  /* m: number of rows of matrix A                        */
2  /* v: input dense vector                                */
3  /* u: output dense vector                               */
4  for Int i = 0 to m − 1 do
5      Int rowStart ← rowPtr[i];
6      Int rowEnd ← rowPtr[i + 1];
7      u[i] ← 0;
8      for Int j = rowStart to rowEnd − 1 do
9          u[i] ← u[i]+val[j]*v[colIdx[j]];
10     end
11 end
```

### C. Motivation

In the parallelization of sparse matrix algorithms, we often encounter the problems of load imbalance as a result of the unpredictable distribution of non-zero elements in a given sparse matrix. To mitigate the negative effects of load imbalance, one can group similar rows together and then do the corresponding computation. This grouping process is known as *binning*. In the context of load balancing, there are two binning schemes: (1) *inter-bin* load balancing [11], where each bin contains a variable number of adjacent rows so that the workloads (e.g., the total number of non-zeros of these rows in SpMV) between every two bins can approximate one another; and (2) *intra-bin* load balancing [12], [15], where we collect the rows with similar workloads together into bins and the rows are *not* required to be neighbors. Generally, the first scheme only needs to store the first row index of the grouped rows due to their adjacency, while the second scheme gathers all the row indices in each bin, leading to higher space and time overhead.

The differences found in the rows make some computing kernels better than others to get more efficient performance results. This, in turn, requires different kernels for each bin. However, if we take into account the features of matrices (e.g., the dimensions, the distribution of non-zeros, and the sparsity shape), the computational patterns (e.g., SpMV and SpGeMM), and the underlying platforms (e.g., CPU, GPU, and APU), it is non-trivial to determine the appropriate mapping relationship between bins and kernels.

Figure 2a shows an example of performance results for five SpMV kernels (when we have two distinct input matrices and for each matrix we put all the rows into a single bin for simplicity). While the kernels offer the same functionality, the choice of kernels significantly affects efficiency. The binning schemes also add more complexity to the decision-making process. Figure 2b provides another example where we distribute all the rows into four bins. For each bin, we examine the performance results of the five SpMV kernels. The results indicate that different best kernels are selected among bins, even for the same input.




(a) Different input matrices (b) Different bins

Figure 2: Examples of comparing different kernels given (a) different inputs and (b) grouping policies

Therefore, this paper focuses on tackling two major problems: (1) determining the optimal binning scheme and kernel selection for each bin given different sparse matrices; and (2) generating a framework that automatically makes decisions based on a data-mining method at runtime. The paper places particular emphasis on the SpMV algorithm and the AMD APU platform.

## III. Methodology

In this section, we first provide a brief overview of our proposed framework to automatically find out the most efficient binning schemes and associated kernels while performing parallel SpMV. Second, we detail the candidate binning schemes and kernels within the framework. Third, we describe a machine-learning model that is based on a C5.0 decision tree, including the extracted attributes, the classifier, and configuration setups.

### A. Framework Overview

Sparse matrices present different properties for different domains of applications. These properties include the basic

information of matrices, such as the dimensions and number of non-zeros (NNZs), and statistics of non-zeros, such as the average and variance of the non-zeros. These factors affect the decision-making process and resultant choices of binning schemes and, in turn, affect the kernel selection for each bin. For example, suppose we have a sparse matrix $A$ that contains rows with no more than ten non-zeros. It is reasonable to assign each thread to process each row when we are parallelizing the SpMV over $A$. In contrast, if we have another matrix $B$, each of whose rows contains hundreds of non-zeros, assigning multiple threads to deal with one row simultaneously is more suitable. In reality, there are various lengths of rows existing in the same sparse matrix. Because non-zeros are distributed in the matrix in an unpredictable manner, the static and fixed binning scheme and kernel selection are not universally applicable to all the situations. Thus, our framework is designed to automatically search for the most efficient combinations of binning schemes and kernels by using the data mining method. The framework is shown in Figure 3.

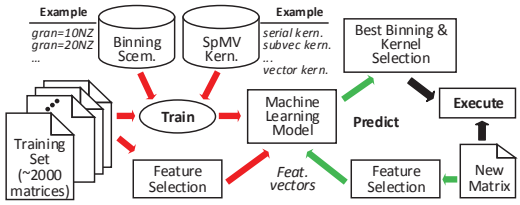



Figure 3: The overview of our data mining framework of SpMV

In the framework, we use over 2000 sparse matrices from the UF collection [28] as the training set. These matrices are collected from various scientific and engineering domains and present many specific features in sparse matrix algorithms. We use some extracted attributes to represent the feature of matrices (i.e., the feature selection module). The candidate binning schemes contain the grouping policies in the framework. The grouping policy defines how to gather similar rows of non-rows into the same bins. The candidate SpMV kernels, on the other hand, are the kernels with the same functionality, but use different parallelization strategies (see algorithms in Section III-B) to deal with the rows. For every sparse matrix from the training set, we find out the best combination of binning scheme and right kernels for each bin and store it as one record. Then, these records serve as inputs to the training process, which is illustrated using red arrows. This step is done only once and the decision result is stored in a database. When a new incoming sparse matrix is entered during runtime, the framework can predict the most appropriate binning scheme and kernels based on the extracted feature of the matrix and training results. The process is shown in green arrows. Finally, the SpMV is performed over the matrix using the predicted solution, which is denoted by the black arrows.

### B. Binning Schemes and Kernel Choices

The SpMV method in the framework is shown in Figure 4, which contains three major steps: (1) we collect the information about the number of non-zeros per row in the given matrix; (2) we group the rows according to the selected binning schemes; and (3) we conduct selected kernels on the rows in each bin. The step 2 and step 3 are reconfigurable, as they can be reprogrammed to change the binning schemes or the kernels based on the corresponding candidate pools.

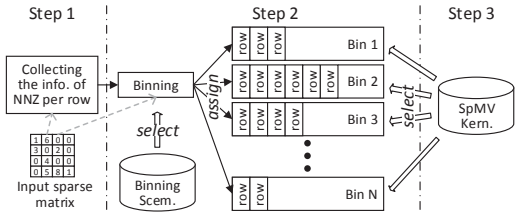



Figure 4: The reconfigurable SpMV method in the framework

*First step*: In SpMV, the rows of the input sparse matrix can be processed simultaneously. However, the workloads per row vary, thereby leading to the load imbalance issue. Since the workloads are determined by the number of non-zeros per row (line 8 in Algorithm 1), we need to first collect the information regarding the distribution of non-zeros in the matrix. Because the fine-grained binning based on every row will require higher space and time costs (in Section II-C), the framework treats every $U$ neighboring rows as a single "virtual" row in the binning step. The $U$ is controlled by the selected binning scheme. Algorithm 2 contains the pseudocode for step 1.

*Second step*: The similar rows are placed together not only for load balance, but also for the customized kernel operations. In the framework, we adopt a coarse-grained way to group the similar rows into different bins. Since we treat multiple neighboring rows as a single row, we can use the first row index to represent them and this will save memory space for the bins. Furthermore, we have a set of granularity units (denoted as $U$), which determine the number of neighboring rows in the "virtual" row, to adjust the binning results. Therefore, the bin index $binId$ indicates that the amount of workloads of the "virtual" row is between $U * binID$ and $U * (binID + 1)$. For example, suppose the granularity $U$ is 10. Every row index $k$ in bin 1 means its following 9 rows in addition to itself contain the workloads in the range of $[10, 20)$. In the experiments, $U$ is preset to be 10, 20, 50, 100, $\cdots$, $10^6$ and there are up to 100 bins. For the extremely long rows, whose number of non-zeros exceeds any bin's capacity, we will place them into the last bin.

This binning method is different from [11], although both work on the adjacent rows. For example, suppose there is a sparse matrix with 10 rows, where the first 5 rows each have 1 non-zero (representing short rows), while the last 5 have 9 non-zeros (representing medium rows). The large bin

workload limit (e.g., 50) in the method [11] puts all 10 rows into the same bin and thus an approach, somewhere in the middle of solving short and medium rows, would be used, failing to efficiently process any types of rows. The short bin workload limit (e.g., 5) puts the first 5 rows together and uses approaches for short rows, but the last 5 rows will appear in 5 separate bins because they each exceed the bin workload limit. That way, the last 5 bins would probably use approaches for long rows based on the fact that there is only a small number of rows (e.g., 1) in the bin. In contrast, we set the number of concatenated neighboring rows as an adjustable $U$. Therefore, in this particular case, our framework is able to find the optimal $U$ as 5 and put the first 5 rows as a "virtual" row into a small bin and the next "virtual" row of last 5 rows into a medium bin.

The pseudocode for step 2 is also shown in Algorithm 2. The bin index can be calculated based on the workloads (in line 7). Then, the representative rows are stored into the target bins (line 8 to 11). Besides of the binning schemes explained above, there are other alternatives, including, for example, a fine-grained method which stores every single row index in the bins and a hybrid method which uses fine-grained binning over short rows, but a coarse-grained one over long rows. Our framework can be easily extended to include other binning schemes by modifying the definition and distribution of workloads in Algorithm 2.

**Algorithm 2:** The step 1 and 2 of our SpMV framework

```
   /* Step 1: collect workloads, i.e. NNZ (wl)                    */
1  /* U is a configurable granularity unit number                 */
2  for Int i = 0 to ⌈m/U⌉ − 1 do
3  |   wl[i] ← rowPtr[min((i + 1) · U, m)]-rowPtr[i · U];
4  end
5  /* Step 2: group rows into bins                                */
6  for Int i = 0 to ⌈m/U⌉ − 1 do
7  |   Int binId ← ⌊wl[i]/U⌋;
8  |   if binId < MAX_BIN_ID then
9  |   |   bin[binId].insert(i);
10 |   else
11 |   |   bin[MAX_BIN_ID].insert(i);
12 |   end
13 end
```

*Third step*: Since the bins present different features of the number and distribution of non-zeros, each bin will need to determine the most suitable kernel to process these non-zeros. In the framework, we have nine SpMV kernels with the same functionality, but using different thread organizations. These kernels can also be categorized into three groups according to thread usage. For clarity, we treat one work-group of threads in OpenCL as a vector, and subvector means a subgroup of threads to provide finer grained control of threading.

(1) Kernel-Serial: The kernel is the most straightforward one to process rows in parallel for each bin. Algorithm 3 shows the pseudocode of the kernel. We assign one row to each thread (line 2) and then the thread will sequentially calculate the products of the non-zeros in the row with the corresponding values in the input vector $v$ (line 7). Finally, the thread accumulates the products into $sum$ and stores it into the output vector $u$ (line 9). Although the kernel is simple and easy to implement, it is quite powerful to handle small bins, where all the rows are very short. Given the bin, the kernel will be launched with $\lceil bin.size()/256 \rceil$ work-groups and each work-group has a fixed 256 threads. Though, the workgroup size is a tunable parameter too.

**Algorithm 3:** Kernel-Serial

```
   /* Every thread handles one row                                */
1  Int tid ← get_global_id();
2  Int rid ← bin[binID]+tid;
3  Int rowStart ← rowPtr[rid];
4  Int rowEnd ← rowPtr[rid + 1];
5  Float sum ← 0.0;
6  for Int i ← rowStart to rowEnd − 1 do
7  |   sum ← sum+val[i]*v[colIdx[i]];
8  end
9  u[rid] ← sum;
```

(2) Kernel-SubvectorX: The kernel is designed to process one row using $X$ threads simultaneously. $X$ represents a subgroup of 256 threads in the work-group in OpenCL and sets to be 2, 4, 16, 32, 64, 128 in our experiments. Algorithm 4 provides the pseudocode of the kernel. In the kernel, we first assign every $X$ threads to one row index (line 3). Second, we load the target data into a local memory buffer *localMem* (line 10 to 15). Additionally, we set the size of local memory to be $factor$ times of the work-group size. That way, we can more efficiently utilize the high bandwidth of the local memory. Third, we conduct a segmented parallel reduction over *localMem*, where every $X$ threads reduce their buffered partial products and store the results in their first position (line 16). After that, the first threads in the subgroups will put the results in a register $sum$ and another round of computation will occur until the end of the rows. Finally, the accumulated results will be stored in the destination of vector $u$ (line 24). The variants of the kernel are designed to handle the vast majority of middle-sized rows with less than 100 non-zeros (in Figure 5). In the experiments, each work-group still contains 256 threads, and we will launch the kernel using $\lceil bin.size() * X/256 \rceil$ work-groups.

(3) Kernel-Vector: In this kernel, the entire work-group of threads are used to tackle the assigned row. Algorithm 5 provides the pseudocode of this kernel. The procedure is similar to that of the "Kernel-SubvectorX", but assigns all the threads in the work-group to one row (line 3) and reduces the products in the buffer using these threads in parallel (line 16). For the large bins with long rows, this kernel might be preferred. We launch the kernel with $bin.size()$ work-groups of 256 threads.

It is worth pointing out that our framework includes no kernels that use multiple work-groups to process one single row for two main reasons. (1) The overhead of using multiple work-groups simultaneously is relatively high, since it involves the global synchronization or atomic operations, which are not preferable in the GPU implementa-

**Algorithm 4:** Kernel-SubvectorX

```
   /* X threads handle one row                                    */
1  Int tid ← get_global_id()%X;
2  Int bid ← get_global_id()/X;
3  Int rid ← bin[binID]+bid;
4  Int rowStart ← rowPtr[rid];
5  Int rowEnd ← rowPtr[rid + 1];
6  Int factor ← 4;
7  Float sum ← 0.0;
8  for Int i ← rowStart + tid to rowEnd − 1 do
9  |   Int t ← 0;
10 |   for Int j ← i to min(i + factor∗X, rowEnd − 1) do
11 |   |   localMem[get_local_id()+t∗get_local_size()] ←
   |   |     val[j]*v[colIdx[j]];
12 |   |   j ← j+X;
13 |   |   t ← t + 1;
14 |   end
15 |   barrier(CLK_LOCAL_MEM_FENCE);
16 |   seg_parallel_red(localMem, factor, X);
17 |   if tid == 0 then
18 |   |   sum ← sum+localMem[get_local_id()];
19 |   end
20 |   i ← i + factor∗get_local_size();
21 |   barrier(CLK_LOCAL_MEM_FENCE);
22 end
23 if tid == 0 then
24 |   u[rid] ← sum;
25 end
```

**Algorithm 5:** Kernel-Vector

```
   /* All threads in work-group handle one row                    */
1  Int tid ← get_local_id();
2  Int bid ← get_group_id();
3  Int rid ← bin[binID]+bid;
4  Int rowStart ← rowPtr[rid];
5  Int rowEnd ← rowPtr[rid + 1];
6  Int factor ← 4;
7  Float sum ← 0.0;
8  for Int i ← rowStart + tid to rowEnd − 1 do
9  |   Int t ← 0;
10 |   for Int j ← i to min(i + factor∗get_local_size(), rowEnd − 1) do
11 |   |   localMem[tid+t∗get_local_size()] ← val[j]*v[colIdx[j]];
12 |   |   j ← j+get_local_size();
13 |   |   t ← t + 1;
14 |   end
15 |   barrier(CLK_LOCAL_MEM_FENCE);
16 |   parallel_red(localMem, factor);
17 |   if tid == 0 then
18 |   |   sum ← sum+localMem[get_local_id()];
19 |   end
20 |   barrier(CLK_LOCAL_MEM_FENCE);
21 end
22 if tid == 0 then
23 |   u[rid] ← sum;
24 end
```

tion. Though there is some research about using dynamic parallelism techniques to deal with the long rows of non-zeros [12], the efficiency will depend on the specific feature and implementation of the underlying hardware. (2) The majority of rows in the sparse matrices in the real-world applications are short rows. Figure 5 shows the histogram of non-zeros in the rows of the sparse matrices, which are collected from the UF collection and totals 2760 matrices in all. As shown in the figure, about $98.7\%$ of the rows have only 100 non-zeros or less. Therefore, in the paper we focus on the aforementioned kernels to handle the common sparse matrices; our framework is also flexible and can be extended to support other kernels for long rows in the latest research [12], [29], but we leave it to future work.

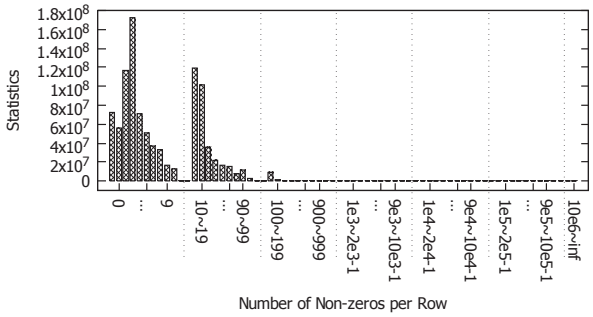



Figure 5: The histogram of non-zeros in rows of sparse matrices in UF collection

### C. Machine Learning Model

In the framework, we use a machine learning method to automatically find out the binning schemes and kernels for any given sparse matrix. The sparse matrices are abstracted to a set of parameters. On top of that, we define the attribute collection for our data mining method. Finally, we will discuss our data mining tool and its setups.

*Matrix Feature Parameters*: Since our framework is input matrix adaptive, we need to extract enough feature parameters to represent the given sparse matrix. We borrow the general parameters from [10] and add *Min_NNZ* to describe the minimum number of non-zeros among the rows. The parameters can be categorized into two groups: basic information and non-zero distribution information. The full list is provided in Table I.

Table I: Extracted parameters to represent given sparse matrices

| Module Name | Description |
|---|---|
| *Basic Matrix Info.* | |
| M | The number of rows |
| N | The number of columns |
| NNZ | The overall number of non-zeros |
| *Non-zero Distribution Info.* | |
| Var_NNZ | The variation of non-zeros per row |
| Avg_NNZ | The average of non-zeros per row |
| Min_NNZ | The minimal of non-zeros per row |
| Max_NNZ | The maximal of non-zeros per row |

*Attribute Vectors*: Our machine learning model adopts a two-stage training process. First, we focus on finding out the best binning scheme for the given sparse matrix. As a result, the first attribute vector is in the form of {M, N, NNZ, Var_NNZ, Avg_NNZ, Min_NNZ, Max_NNZ, $U$} and the target attribute is the binning scheme $U$. Second, we change the attribute vector to the form of {M, N, NNZ, Var_NNZ, Avg_NNZ, Min_NNZ, Max_NNZ, U, binID, $kernelID$} and focus on finding out the most efficient kernel $kernelID$ for each bin of the given sparse matrix under the binning scheme $U$. The two-stage training process enables us to figure out the mapping relationship between binning schemes and kernels for each bin.

*Data Mining Model*: In the framework, we use the C5.0 data mining tool [22] to extract the informative relationship between binning schemes and kernels, because it has a

user-friendly design and only knowledge of target features is required during the generation of the model. After the training process (i.e., discovering patterns and assembling them into classifiers), the C5.0 can offer a rule-set, which is a set of if-then statements. In our experiments, we use over 2000 sparse matrices from the UF collection [28] as the training set, where 75% of them are used for training and the rest are used for testing. We observe the error rate for the first stage of learning is around 5%, while the second stage error rate is up to 15%. After training, we have two rule-sets: one is for selecting binning schemes; another is for selecting kernels under the selected binning scheme. Thus, we are able to make the decision of the parallelization strategies for any new incoming matrix in the predict process (shown in Figure 3).

## IV. Evaluation

In the section, we will first introduce the experimental platform followed by the selected representative matrices. Then, we show the performance results of SpMV, where the parallelization strategies are selected by using our data mining method. Finally, we will analyze and discuss the binning overhead and some limitations of our framework.

### A. Experimental Setup

*Platform*: Our experimental results are measured on real HSA hardware using an AMD A10-7850K APU. The AMD A10-7850K features four 3.7 GHz CPU cores and eight 720MHz GPU compute units. Our system is equipped with 16 GB memory. We use AMD Heterogeneous System Architecture (HSA) - Linux amdkfd v1.4 release, and CL Offline Compiler CLOC V0.9.5 (HSA 1.0F) with SNACK support.

*Input sparse matrices*: In the off-line training process, we use over 2000 matrices from the UF collection, 75% of which are for training and 25% are for testing. To evaluate the selected parallelization strategies, we use 16 representative sparse matrices from different application domains. These matrices are summarized by [10] and Table II gives the list of them. Note, the 16 representative matrices are exclusive from the matrices in the training set.

Table II: The list of representative matrices for evaluation, which is summarized by [10]

| Graph | Name | #Row | #Col | #NZ | Kind |
|---|---|---|---|---|---|
|  | apache1 | 81k | 81k | 542k | Structural problem |
|  | bfly | 49k | 49k | 197k | Undirected graph sequence |
|  | ch7-9-b3 | 106k | 18k | 423k | Combinatorial problem |
|  | crankseg_2 | 64k | 64k | 14m | Structural problem |
|  | cryg10000 | 10k | 10k | 50k | Materials problem |
|  | D6-6 | 120k | 24k | 147k | Combinatorial problem |
|  | denormal | 89k | 89k | 1m | Counter-example problem |
|  | dictionary28 | 53k | 53k | 178k | Undirected graph |
|  | europe_osm | 51m | 51m | 108m | Undirected graph |
|  | Ga3As3H12 | 61k | 61k | 6m | Theoretical/quantum chemistry problem |
|  | HV15R | 2m | 2m | 283m | CFD problem |
|  | pcrystk02 | 14k | 14k | 969k | Duplicate materials problem |
|  | pkustk14 | 152k | 152k | 15m | Structural problem |
|  | roadNet-CA | 2m | 2m | 6m | Undirected graph |
|  | shar_te2-b2 | 200k | 17k | 601k | Combinatorial problem |
|  | whitaker3_dual | 19k | 19k | 57k | 2D/3D problem |

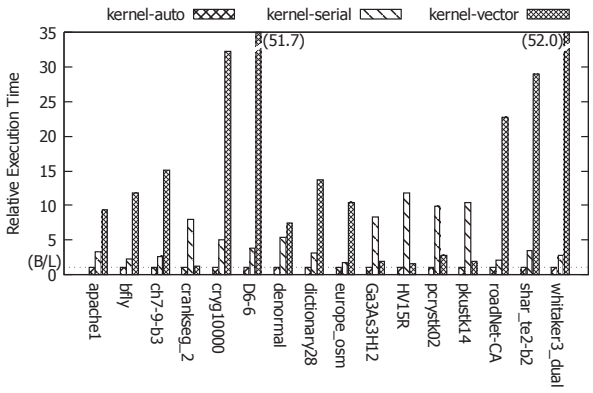



Figure 6: Performance comparison of SpMV kernels over different matrices. Execution time: normalized to our "kernel-auto".

### B. Speedups from Our Framework

Our SpMV is able to automatically select a binning scheme and equip every bin with appropriate kernels. We first compare our optimized SpMV (kernel-auto) with the default SpMV using only one single kernel. The kernels selected for the default one are the "kernel-serial" and "kernel-vector" respectively, which represent two ends of threading granularity. As we can see from Figure 6, "kernel-auto" can yield better performance than its counterparts for all the 16 representative matrices. Compared to "kernel-serial", we can achieve 1.7x to 11.9x speedups, while compared to "kernel-vector", the speedups are changed to 1.2x to 52.0x. In most cases, the "kernel-serial" can provide superior performance over "kernel-vector" in that most sparse matrices contain very short rows on average, which is consistent with the statistics shown in Figure 5. However, there are 5 matrices making "kernel-vector" perform better due to their relatively long rows. In contrast, our SpMV takes into consideration of the lengths and distribution of the non-zeros in the matrix. Hence, it outperforms the others.

Then, we compare our SpMV with the prior state-of-the-art GPU SpMV "CSR-Adaptive" [11], which outperforms some widely-used tools from clSpMV [30], ViennaCL [31] and format ELLPACK. It has been adopted in current ViennaCL. We tested against a SNACK version of CSR-Adaptive, though the performance may be lower than the

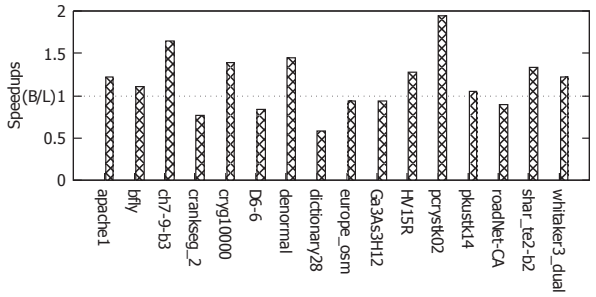



Figure 7: Performance comparison over the state-of-the-art SpMV kernel [11]

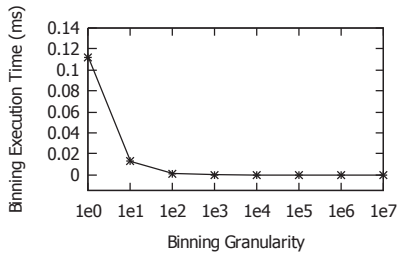



Figure 8: Binning overhead of the changing binning granularity for a sample sparse matrix ($10^7$ rows and each has only one non-zero). Each row contains only one non-zero.

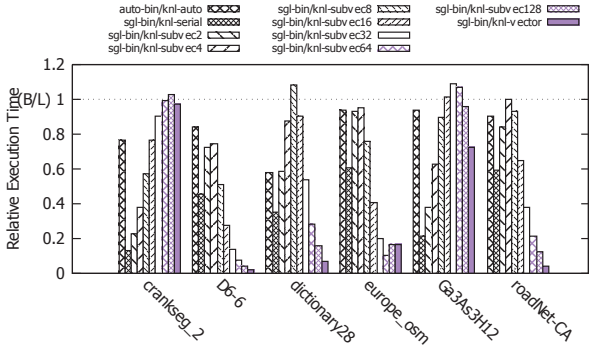



Figure 9: The single-bin strategy with various kernel selections. Execution time: normalized to the state-of-the-art SpMV kernel [11]

latest available in clSPARSE. The "CSR-Adaptive" uses the binning method to achieve inter-bin load balance and it is able to automatically select the right kernel while processing different rows within a single bin. The choice of the strategy is defined (with hard-coded parameters) and may not perform well on all the platforms. In contrast, our binning method focuses on intra-bin load balance and possible different choices of kernels occur across bins. Figure 7 shows our speedups over the "CSR-Adaptive" SpMV. Our SpMV can yield better performance over 10 out of 16 sparse matrices and achieve up to 1.9x speedups. However, for the matrices of *crankseg_2*, *D6-6*, *dictionary28*, *europe_osm*, *Ga3As3H12*, and *roadNet-CA*, "CSR-Adaptive" outperforms ours. This might be caused by the lack of other efficient binning schemes and the error of prediction. We will discuss them in the next section.

### C. Analysis and Discussion

*Binning overhead*: In our framework, we choose not to set the binning granularity $U$ to small numbers, such as 1 or 2, because the small granularity will cause very high overhead. Considering the low ratio of the computation to the memory access in SpMV, the overhead cannot be compensated by the later computational operations. If we adopt the fine-grained binning scheme in SpMV, not only the number of bins will increase significantly, but also more time will be consumed. Figure 8 provides an example of the overhead of binning a sample matrix with $10^7$ rows and each row has only one non-zero. As it is shown, the binning with $U$ of 1 consumes much more time than the other granularities. When the granularity rises to 100, the overhead becomes negligible. Therefore, our framework prefers to use coarser-grained binning schemes with larger granularity numbers so that binning overhead is minimized while in the meantime it is sufficient for optimizing for different kernels. For large-scaled SpMV, we can conduct segmented analysis to hide the binning overhead [32] or overlapped binning and computation [33].

*Grouping to Single Bin*: In previous section, our SpMV cannot provide better performance over the baseline "CSR-Adaptive" for the matrices of *crankseg_2*, *D6-6*, *dictionary28*, *europe_osm*, *Ga3As3H12*, and *roadNet-CA*. After taking a close look at the matrices, we find there are other better binning schemes. For example, we can simply put all the rows into a single bin and then look for the best kernel. Figure 9 shows the performance of the single-bin strategy. In this case, we manually change the selected kernel and test the corresponding performance. The horizontal dashed line indicates the performance of "CSR-Adaptive". Although two matrices still generate inferior performance, the other four are able to outperform or become equal to the baseline. This observation shows that for some matrices, though the variance of the number of non-zeros per row exists, the strategy of grouping the rows into different bins might not be able to compete with simply using one kernel to process all the rows. However, we still need to automatically figure out which kernel is the most suitable one. Currently, our framework does not accommodate the case for single-bin strategy effectively which we plan to improve in future work. Also, we found that most of these matrices have either very low non-zeros per row (e.g., *D6-6*, *europe_osm*) or very high non-zeros per row (e.g., *crankseg_2*, *Ga3As3H12*). Our current framework addresses irregular matrices with varying non-zeros more effectively.

*Parameters*: In the framework, we currently use some general feature parameters (in Table I) to represent sparse matrices. Although these parameters can approximately reflect the shape of the matrices and distribution of the non-zeros, how to represent a sparse matrix with a set of key parameters is still an open research area. For example,

metrics are needed to capture the ratio and adjacency of the long, medium, and short rows. In the future, we plan to further update our machine learning model to decrease the error rate of learning and improve accuracy of prediction by using the parameters, such as the histogram of rows of non-zeros.

## V. Related Work

Since the SpMV is widely used in the scientific and engineering applications, there is a great deal of literature with a focus on optimizing and tuning SpMV on the modern accelerators (e.g., multi-core CPUs and GPU), or finding the most preferable storage formats to facilitate the SpMV. Williams *et al.* [34] examine a set of optimization methods of SpMV kernels on the modern processors including homogeneous and heterogeneous parallel platforms. Their methods include thread-level parallelism, blocking for better cache and local storage access, and register blocking. Bell *et al.* [35] implement SpMV kernels on the GPU, where they explore how different formats (e.g., COO, CSR, and ELLPACK) affect the performance of SpMV and they propose a hybrid format of the SIMD-friendly ELLPACK and non-zero distribution invariant COO in order to take advantage of both sides. The tools, such as clSPARSE [36], cuSPARSE [4], clSpMV [30], take into account different sparse storage formats and provide corresponding interfaces. However, for a given sparse format, the necessity and overhead of explicitly converting to a more SIMD-friendly one is not negligible on GPUs. In contrast, on CPUs, we can use directives/pragmas to directly restructure loops to utilize vector processing resources [37], [38].

Ashari *et al.* [12] organize rows into different bins according to the number of non-zeros within and these bins are statically mapped to different kernels. Their binning method is akin to the fine-grained binning scheme. Moreover, they take advantage of the *dynamic parallelism* (DP) of the latest NVIDIA GPUs. Although dynamic parallelism provides runtime launch of kernels to process long rows in sparse matrices (e.g., graphs), the naive use of this feature oftentimes deteriorates the performance [39], [40]. Merriall and Garland [41] propose a merge-based SpMV kernel and achieve superior performance on both CPU and GPU. Actually, both of the DP-based and merge-based kernels can be integrated into our framework as kernel candidates, which will be covered as part of our future work. Liu *et al.* [15] focus on the SpGeMM kernels and uses another hybrid binning method, where for the short rows the fine-grained binning is used, while for the long rows the coarse-grained binning is used. In contrast to the "intra-bin" balanced methods, Greathouse and Daga [11] devise an alternate binning solution to achieve "inter-bin" load balance. Their approach puts neighboring rows together so that each bin has similar workloads (the workload limits are set in advance). Then, the number of rows within each bin will determine the suitable methods to process each row. Our work is different from the aforementioned SpMV for two major reasons. (1) Our binning method treats multiple adjacent rows as one "virtual" row (the number of adjacent rows is determined by $U$ in Section III-B) and then only stores the first row's index in the bins. By doing so, we can take advantage of the benefits from both fine-grained (finer kernel assignment) and "inter-bin" binning methods (adjacent data access, less time and space overhead of binning). (2) The binning scheme and kernels are not fixed to any criteria in our work. We can adjust the strategies automatically according to the sparsity features of the inputs.

Considering the diversity of sparsity features of the target matrices in addition to the parallel platforms, it is promising to find a reliable auto-tuning and prediction model to resolve the performance portability issue. On the other hand, there are different sparse storage formats to handle different types of matrices. DIA is proved to be the right format for the non-zeros distributed along the diagonal [35]. ELLPACK and its variants are SIMD-friendly and might be an effective formats for the modern accelerators [35], [42]. Other formats (e.g., BCCOO [6], BRC [7], and CSR5 [8]) are proposed to further accelerate the parallel SpMV. Since different formats are usually adopted to resolve one or more types of sparse matrices, some research focuses on using auto-tuning and prediction to automate the process of decision making for which format to use. SMAT [10] and Sedaghati *et al.* [23] propose machine learning models to select the most efficient storage format to facilitate the SpMV operations. Vázquez *et al.* [43] propose an auto-tuning method to find optimal parameters for the ELLR-T algorithm to compute SpMV on GPUs. In contrast to the previous work, we focus on the most popular CSR format and our auto-tuning and prediction model is in the level of selecting optimal parallel strategies of binning schemes and kernels.

## VI. Conclusion & Future Work

In this paper, we propose a SpMV framework using the machine learning model to automatically find the optimal parallel strategies (i.e., the binning schemes and kernels for each bin). Our framework is based on the CSR format, which is a widely-used storage format for sparse matrices. According to the different sparsity features of the given matrix, our SpMV framework can assist in choosing the appropriate grouping policy and organizing independent rows into different bins. Then, each bin will look for the suitable kernels to process the rows within. Both searching processes are based on the results of previous training steps. The results show that our SpMV kernels are able to significantly outperform the SpMV kernels using single kernel selection. Moreover, for 10 out of 16 representative matrices, our SpMV can achieve better performance over the state-of-the-art SpMV kernels. This approach is also generic to other sparse matrix applications (e.g., SpGeMM, SpElementWise,

etc.) and other applications with different implementation variants, which would be useful for researchers and developers.

For future work, we plan to extend our work to fully utilize the underlying heterogeneous architectures. Since the bins grouped by our binning schemes are optimized for "inter-bin" workload balancing, it would be promising to schedule the execution of the small sized but high volume bins onto the throughput-oriented processors and the large sized but low volume bins onto the latency-oriented processors, such as [44].

## VII. Acknowledgment

We thank the anonymous reviewers for their helpful feedback. We also thank Joseph Greathouse and Mayank Daga for their help on the CSR-Adaptive algorithm and suggestions for improving the paper. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

## References

[1] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," *Comput. Netw. ISDN Syst.*, 1998.

[2] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley *et al.*, "PETSc Users Manual Revision 3.5," Tech. Rep., 2014.

[3] X. Yu, H. Wang, W. C. Feng, H. Gong, and G. Cao, "cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs," in *Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, 2016.

[4] NVIDIA, "CUSPARSE Library," 2016. [Online]. Available: https://developer.nvidia.com/cusparse

[5] J. L. Greathouse, K. Knox, J. Pola, K. Varaganti, and M. Daga, "clSPARSE: A Vendor-Optimized Open-Source Sparse BLAS Library," in *Int'l Workshop on OpenCL (IWOCL)*. ACM, 2016.

[6] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet Another SpMV Framework on GPUs," in *SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2014.

[7] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2014.

[8] W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2015.

[9] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," Ph.D. dissertation, 2003, aAI3121741.

[10] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication," in *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2013.

[11] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2014.

[12] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2014.

[13] W. Liu and B. Vinter, "A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors," *J. Parallel Distrib. Comput. (JPDC)*, 2015.

[14] W. Liu and B. Vinter, "Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors," *Parallel Comput. (ParCo)*, 2015.

[15] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *Int'l Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2014.

[16] K. Hou, H. Wang, and W.-c. Feng, "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2015.

[17] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel Transposition of Sparse Data Structures," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2016.

[18] K. Hou, H. Wang, and W. C. Feng, "AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors," in *Int'l Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[19] C. Chu, J. Zhang, and Y. Wu, "GINDEL: Accurate Genotype Calling of Insertions and Deletions from Low Coverage Population Sequence Reads," *PloS one*, 2014.

[20] X. Yu and M. Becchi, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," in *Int'l Conf. on Computing Frontiers (CF)*. ACM, 2013.

[21] J. Zhang, H. Wang, H. Lin, and W. C. Feng, "cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU," in *Int'l Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2014.

[22] C5.0: A System that Extracts Informative Patterns from Data. https://www.rulequest.com/see5-unix.html.

[23] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic Selection of Sparse Matrix Representation on GPUs," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2015.

[24] CL Offline Compiler and SNACK. https://github.com/HSAFoundation/CLOC.

[25] HSA: Heterogeneous System Architecture. http://hsafoundation.com/.

[26] K. O. W. Group. The OpenCL Specification. https://www.khronos.org/opencl/.

[27] GCN: Graphics Core Next. http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx.

[28] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, 2011.

[29] M. Daga and J. L. Greathouse, "Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices," in *Int'l Conf. on High Performance Computing (HiPC)*. IEEE, 2015.

[30] B.-Y. Su and K. Keutzer, "clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2012.

[31] ViennaCL. http://viennacl.sourceforge.net/.

[32] J. Liu, S. Byna, and Y. Chen, "Segmented Analysis for Reducing Data Movement," in *Int'l Conference on Big Data (BigData)*. IEEE, 2013.

[33] X. Cui, T. R. W. Scogland, B. R. d. Supinski, and W. C. Feng, "Directive-Based Pipelining Extension for OpenMP," in *Int'l Conf. on Cluster Computing (CLUSTER)*. IEEE, 2016.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms," in *Conf. on Supercomputing (SC)*. IEEE/ACM, 2007.

[35] N. Bell and M. Garland, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2009.

[36] AMD and Vratis, "clSPARSE Library." [Online]. Available: https://github.com/clMathLibraries/clSPARSE

[37] K. Hou, H. Wang, and W. c. Feng, "Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study," in *Int'l Conf. on Parallel Processing Workshops (ICPPW)*. IEEE, 2014.

[38] D. Zhang, H. Wang, K. Hou, J. Zhang, and W. chun Feng, "pDindel: Accelerating indel detection on a multicore CPU architecture with SIMD," in *Int'l Conf. on Computational Advances in Bio and Medical Sciences (ICCABS)*. IEEE, 2015.

[39] H. Wu, D. Li, and M. Becchi, "Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU," in *Int'l Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[40] D. Li, H. Wu, and M. Becchi, "Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations," in *Int'l Conference on Parallel Processing (ICPP)*, 2015.

[41] D. Merrill and M. Garland, "Merge-based Parallel Sparse Matrix-vector Multiplication," in *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2016.

[42] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors," in *Int'l Conf. on Supercomputing (ICS)*. ACM, 2013.

[43] F. Vázquez, J. J. Fernández, and E. M. Garzón, "Automatic Tuning of the Sparse Matrix Vector Product on GPUs Based on the ELLR-T Approach," *Parallel Comput. (ParCo)*, 2012.

[44] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "FinePar: Irregularity-aware Fine-grained Workload Partitioning on Integrated Architectures," in *Int'l Symp. on Code Generation and Optimization (CGO)*. IEEE, 2017.