clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs

Bor-Yiing Su
University of California, Berkeley
EECS Department
subrian@eecs.berkeley.edu

Kurt Keutzer
University of California, Berkeley
EECS Department
keutzer@eecs.berkeley.edu

ABSTRACT

Sparse matrix vector multiplication (SpMV) kernel is a key computation in linear algebra. Most iterative methods are composed of SpMV operations with BLAS1 updates. Therefore, researchers make extensive efforts to optimize the SpMV kernel in sparse linear algebra. With the appearance of OpenCL, a programming language that standardizes parallel programming across a wide variety of heterogeneous platforms, we are able to optimize the SpMV kernel on many different platforms. In this paper, we propose a new sparse matrix format, the *Cocktail Format*, to take advantage of the strengths of many different sparse matrix formats. Based on the *Cocktail Format*, we develop the *clSpMV* framework that is able to analyze all kinds of sparse matrices at runtime, and recommend the best representations of the given sparse matrices on different platforms. Although solutions that are portable across diverse platforms generally provide lower performance when compared to solutions that are specialized to particular platforms, our experimental results show that *clSpMV* can find the best representations of the input sparse matrices on both Nvidia and AMD platforms, and deliver 83% higher performance compared to the vendor optimized CUDA implementation of the proposed hybrid sparse format in [3], and 63.6% higher performance compared to the CUDA implementations of all sparse formats in [3].

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Parallel programming; C.1.2 [**Processor Architectures**]: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms

Performance

Keywords

clSpMV, OpenCL, GPU, SpMV, Sparse Matrix Format, Autotuner, Cocktail Format

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

1. INTRODUCTION

In scientific computation, operations research, image processing, data mining, structural mechanics, and other fields, the system matrices are naturally sparse, and sparse matrix algorithms are required for analysis. Iterative methods are widely used to solve linear systems and find eigen decompositions. Many iterative methods are composed of sparse matrix vector multiplication (SpMV) operations with BLAS1 updates, such as the conjugate gradient method and the Krylov subspace methods [22]. Since the matrix size is orders of magnitude larger than the vector, the SpMV operations dominate the execution time of these iterative methods. In order to accelerate these iterative methods it is essential to optimize the SpMV kernel.

The SpMV kernel is notorious for its extremely low arithmetic intensity (the upper bound of the flop:byte ratio is 0.25, two flops for eight bytes on single precision floating point data type), and irregular memory access patterns [20]. The SpMV kernel is a pure memory bounded problem as shown in [21]. Although the peak floating point operations per second (FLOPS) of modern microprocessors are increasing rapidly, the maximum memory bandwidth is not improving at a similar pace. Therefore, the SpMV kernel usually performs poorly, achieving only 10% of the peak performance on single core cache based microprocessors [18]. Studies to improve performance of the SpMV kernel can be categorized into two directions: applying architecture specific optimizations, and applying new sparse matrix formats.

Interest in SpMV has increased with the advent of more powerful multi-core CPUs and many-core GPUs. Williams et al. [20] evaluates different optimization strategies on AMD Opteron X2, Intel Clovertown, Sun Niagara2, and STI Cell SPE. Bell and Garland [3] optimizes different SpMV kernels with different sparse matrix formats on Nvidia GPUs. Bordawekar and Baskaran [4] further optimizes the SpMV kernel with the Compressed Sparse Row (CSR) sparse matrix format on Nvidia GPUs. Choi et al. [6] implements Blocked Compress Sparse Row (BCSR) and Sliced Blocked ELLPACK (SBELL) formats on Nvidia GPUs.

Researchers have also proposed various sparse matrix formats with the goal of minimizing the memory footprint, and enforcing some regularity on the access pattern. Buluc et al. [5] uses the symmetric Compressed Sparse Block (CSB) and the bitmasked register block data structures to minimize the storage requirement of blocked sparse matrices. Monakov et al. [13] proposes the Sliced ELLPACK (SELL) format as an intermediate format between the CSR and the ELL format. Vázquez et al. [17] suggests the ELLPACK-R

format that can preserve the data alignment requirement on Nvidia GPUs.

Different sparse matrices have different characteristics, and different microprocessors have different strengths. In order to achieve the best SpMV performance for a specific sparse matrix on a specific microprocessor, an autotuner is required to adjust the sparse matrix parameters and the platform parameters. The Optimized Sparse Kernel Interface (OSKI) library [18] is the state-of-the-art collection of sparse matrix operation primitives on single core cache based microprocessors. It relies on the SPARSITY framework [11] to tune the SpMV kernel. The major optimization strategy includes register blocking and cache blocking. Autotuning is used in [6, 13] to find the best block sizes and the slice sizes of the given sparse matrices on Nvidia GPUs. Guo and Wang [10] also autotune the implementation parameters of the CSR SpMV implementation on Nvidia GPUs. Grewe and Lokhmotov [8] develop a code generator to generate CUDA and OpenCL code for SpMV kernels that can facilitate the autotuning process. However, the paper focuses on the generated CUDA code. For OpenCL code, only CSR SpMV results are presented. It is unclear how it will perform on other sparse matrix formats.

The micro-architectures of parallel microprocessors are increasing in their diversity. As a result, different hardware vendors develop their own languages to exploit parallelism in their architectures, such as SSE [15] and AVX [12] for x86, CUDA [14] for Nvidia GPUs, and Stream [1] for AMD GPUs. Fortunately, the leaders of the parallel computing industry have standardized parallel computations with OpenCL [16]. The goal of OpenCL is to make parallel code portable to heterogeneous platforms. With OpenCL, we can expect to develop an autotuner that can tune the SpMV performance on every existing parallel platform. This is the ultimate goal of the *clSpMV* project. However, the parallelization strategies on different platforms are different. In this paper, we show how the *clSpMV* framework can be used to tune the performance of SpMV on GPU platforms.

There are three major contributions of this work:

1. This is the first SpMV OpenCL work that covers a wide spectrum of sparse matrix formats (9 formats in total).
2. We propose a new sparse matrix format, the *Cocktail Format*, that takes advantage of the strengths of many different sparse matrix formats.
3. We have developed the *clSpMV* framework, the first framework that is able to analyze the input sparse matrix at runtime, and recommend the best representation of the given sparse matrix. It is also the first framework that can optimize the SpMV kernel across different GPU platforms.

The remainder of this paper is organized as follows. Section 2 introduces the new proposed *Cocktail Format*. Section 3 introduces the *clSpMV* framework in detail. Section 4 explains our platforms of choice, the supported 9 different sparse matrix formats, and the parallelization strategies on the target platforms. The experimental results are summarized in Section 5. Finally, the conclusions are given in Section 6.

2. THE COCKTAIL FORMAT

As stated in Section 1, many SpMV studies have developed novel sparse matrix formats. However, there is no one-size-fits-all solution. Every sparse matrix representation has its own strengths and weaknesses as explained in Section 4. The symmetric CSB, the bitmasked register block data structures in [5], the BCSR, and the SBELL data structures in [6] all assume dense blocks in the sparse matrix. The performance of the SELL format in [13] and the ELLPACK-R format in [17] relies on the variation of the numbers of non-zero per row in the sparse matrix. The experimental results in [3] also shows that the best SpMV results are heavily dependent on the choice of the sparse matrix format.

Based on the observation that different sparse matrix formats are good at different sparse matrix structures, we have developed the *Cocktail Format* to take advantages of different matrix formats. The *Cocktail Format* is a combination of many different sparse matrix formats. The *Cocktail Format* partitions a given matrix into several submatrices, each specialized for a given matrix structure. The trivial case finds a single best format for a given sparse matrix. The most complicated case is to partition the sparse matrix into many submatrices, and represent different submatrices using different formats. The list of sparse matrix formats in the *Cocktail Format* can be arbitrary. In *clSpMV*, we support 9 sparse matrix formats in 3 categories. The 3 categories and the 9 matrix formats are summarized as following, and will be further explained in Section 4:

- Diagonal based category: formats that store dense diagonals.
  - DIA: stores dense diagonals.
  - BDIA: stores a band of diagonals together.
- Flat category: formats that need a column index for every non-zero data point on the sparse matrix.
  - ELL: packs non-zeros into a dense matrix.
  - SELL: cuts the matrix into slices, and use different ELL settings for each slice.
  - CSR: the common compressed sparse row format.
  - COO: the common coordinate format.
- Block based category: formats that store dense blocks.
  - BELL: the blocked variant of the ELL format.
  - SBELL: the blocked variant of the SELL format.
  - BCSR: the blocked variant of the CSR format.

There exists works that partition a sparse matrix into many submatrices. However, the partitions are very restricted. Vuduc [19] partitions the sparse matrix into 2 to 4 submatrices with different dense block sizes. However, it only focuses on the BCSR format. If the number of dense blocks per row is regular, the BELL format will be a better choice. [3] partitions the sparse matrix into the ELLPACK portion and the COO portion. However, it does not take advantage of dense blocks in the matrices. The *Cocktail Format* is the first proposal that partitions the matrix into many different specialized regions.

3. THE CLSPMV FRAMEWORK

Based on the *Cocktail Format*, every sparse matrix can be partitioned into many submatrices. However, it is very challenging to find the best partitioning scheme of the given sparse matrix. Moreover, each sparse format can have many different parallelization strategies. Assuming the *Cocktail Format* is composed of k sparse matrix formats $f_1, f_2, \ldots, f_k$. $\bigcup_{i=1}^{k} f_i = F$. For a matrix format f_i, assuming there exists b_i implementations p_{i1}, p_{i2}, ..., p_{ib_i}. $\bigcup_{j=1}^{b_i} p_{ij} = P_i$. Let $t(A, f_i, p_{ij})$ be the execution time of the SpMV kernel using

format f_i and implementation p_{ij} on matrix A. The matrix partitioning problem can be formulated as following:

- Problem *CMP* (*Cocktail Matrix Partitioning*): Given sparse matrix A, the k sparse formats in the *Cocktail Format*, the b_i implementations of format f_i, find k submatrices $A_1, A_2, \ldots, A_k$, and k implementations $L_1, L_2, \ldots, L_k$ such that $\sum_{i=1}^{k} A_i = A$, $L_1 \in P_1, L_2 \in P_2, \ldots, L_k \in P_k$, and the value of $\sum_{i=1}^{k} t(A_i, f_i, L_i)$ is minimized.

The CMP problem is a NP-complete problem. For a sparse matrix with n non-zeros, and the *Cocktail Format* with k formats, the size of the sample space is $O(k^n \times \max_{1 \le i \le k} b_i)$. If we allow single non-zero being covered by multiple formats, the sample space is even larger. Moreover, function $t(A, f_i, p_{ij})$ is nonlinear. The actual execution time will depend on the thread scheduler, system load, cache behavior, and many other factors.

3.1 Overall Structure of the clSpMV Framework

In addition to the CMP problem, we also need to compute the $t(A, f_i, p_{ij})$ function. When multiple implementations of a single sparse matrix format are available, most autotuners execute all implementations exhaustively to find the best implementation [8, 10, 13]. This strategy will give us the exact $t(A, f_i, p_{ij})$ value, but is very time consuming. For the *Cocktail Format*, the brute force search strategy will involve expensive reformatting the submatrices, because the submatrices may need to be adjusted frequently. The overhead is unacceptable. Some autotuners develop models of some specific architectures, and predict performance based on the models [6]. This strategy is applicable, but requires significant effort. For each platform we support, we need to develop its performance model. Then we need to apply the performance model on every implementation. Once a new platform is released, we need to go through the entire procedure again. For portability concerns, this is not the best strategy.

Following the philosophy of OSKI [18], the *clSpMV* framework is composed of two stages. The offline benchmarking stage and the online decision making stage. The goal of the offline benchmarking stage is to sample some performance data with different sparse matrix settings, and to provide a way of estimating the value of $t(A, f_i, p_{ij})$. The online decision making stage then solves the CMP problem.

3.2 The Offline Benchmarking Stage

The purpose of the offline benchmarking stage is to solve the performance approximation problem. Given a sparse matrix format f_i, and a corresponding implementation p_{ij}, the offline benchmarking stage will sample the execution time on different sparse matrix settings. The sparse matrix settings include matrix dimensions, total number of non-zeros, average number of non-zeros per row, variations of number of non-zeros per row, and so forth. The sample density controls the trade-offs between approximation accuracy and the offline benchmarking time. More data points with wider matrix settings will yield better approximation results, but it requires more offline benchmarking time. Given an arbitrary sparse matrix, the execution time can be approximated by interpolating nearby data points.

In our current implementation, we only consider matrix dimensions and average number of non-zeros per row, and benchmark on dense banded matrices. The performance benchmarking results for Nvidia GTX 480 and AMD Radeon 6970 are summarized in Figure 10 and Figure 11, and will be discussed in Section 5. When sampling on the matrix dimensions, we want to have representative data for the case that the processors are under-utilized most of the time, and the case that all processors are saturated. Therefore, we choose to use an exponential scale, ranging from 2^{10} to 2^{21}. When sampling on the number of non-zeros per row, we need to cover the case that the matrix is extremely sparse, to the case that every row has enough work to saturate all processors for a while. As will be discussed in Section 4, the parallelization strategies for different formats are different, so the sampling density of different formats are different. If the parallelization strategy is based on having different work items working on independent rows, having the number of non-zeros ranging from 1 to 64 should be enough. On the other hand, if the parallelization strategy is based on having multiple work items working on the same row, we will need hundreds to thousands non-zeros per row to saturate the processors.

For the 9 sparse matrix formats, we have 75 implementations in total, and we need to collect hundreds of sample points on each implementation. Including generating the sparse matrices with different dimensions and number of non-zeros per row, the offline benchmarking stages on both the Nvidia platform and the AMD platform take about half a day. However, it is a one-time cost. We only need to benchmark on a platform once, and the benchmarking results can be used to tune the performance of SpMV on as many sparse matrices as we want.

3.3 The Online Decision Making Stage

This stage solves the CMP problem by analyzing the input sparse matrix. We achieve this goal by collecting matrix features and enforcing partition policies.

Transforming a matrix from a format to another is very time and memory consuming. The *clSpMV* framework tries to explore the design space of 30 different formats (block based formats with different block dimensions are considered as different formats here), so it is infeasible to analyze the structures of all the matrix formats by converting to the formats. Instead we only collect statistical features that are representative of different matrix formats. When collecting features for diagonal formats, we maintain a histogram to count the number of non-zeros per diagonal. When collecting features for blocked formats, we maintain two histograms for each block dimension. One histogram counts the number of blocks per superrow. A superrow is defined on a blocked matrix as a submatrix of h rows, where h is the height of the block. The other histogram only counts the number of dense blocks per superrow. The definition of a dense block is given in the partition policies. The first histogram is used to estimate the execution time if we store the entire matrix using the blocked formats. The second histogram is used to estimate the execution time if we only store the dense blocks of the matrix using the blocked formats. When collecting features for flat formats, we maintain a histogram to count the number of non-zeros per row. The feature histograms are used to capture the characteristics of a matrix under different formats. In the partition policies, we also

make our partition decisions based on the collected feature histograms.

The solution space of the CMP problem is enormous, so we use greedy policies to partition the sparse matrix. The policies are based on our analysis of the strengths and weaknesses of the formats as will discussed in Section 4. According to the 9 supported sparse matrix formats, we use the following policies:

- Priority of the categories: the priority of the 3 categories (the diagonal based category, the flat category, and the block based category) are decided by the maximum estimated performance of that category according to the current matrix settings explored in the offline benchmarking stage.
- Dense Diagonals: let g_d be the maximum GFLOPS the diagonal category can achieve under the current matrix settings. Let g_f be the maximum GFLOPS the flat category can achieve under the current matrix settings. A diagonal with dimension n_d is considered to be dense if its non-zero number e_d satisfies the following formula:

$$e_d > n_d \times \frac{g_f}{g_d}$$

- BDIA vs DIA: choose the maximum achievable GFLOPS in the following 3 cases: only using BDIA, only using DIA, using BDIA to represent thick bands and DIA to represent disjoint diagonals or thin bands.
- Dense Blocks: let g_d be the maximum GFLOPS the block category can achieve under the current matrix settings and the given block size. A block with size n_b is considered to be dense if its non-zero number e_b satisfies the following formula:

$$e_b > n_b \times \frac{g_f}{g_b}$$

- SBELL vs BELL vs BCSR: choose the maximum achievable GFLOPS in the following 3 cases: only using SBELL, only using BELL, only using BCSR.
- ELL and SELL vs CSR and COO: let the maximum achievable GFLOPS of ELL, SELL, CSR, and COO are g_{ELL}, g_{SELL}, g_{CSR}, and g_{COO}, respectively. Use CSR and COO if $m_c = \max(g_{CSR}, g_{COO}) > m_e = \max(g_{ELL}, g_{SELL})$. Otherwise, extract the ELL and SELL portion first, then represent the remaining nonzeros using CSR or COO.
- Extract ELL: let w be the ELL width (the definition of ELL width is given in Section 4), let c be the number of columns of the matrix, let $z(w)$ be the zero paddings when the ELL width is w, and let $e(w)$ be the nonzeros covered by the ELL format with width w. w is decided by solving the following problem:

$$\begin{array}{ll} \max & w \\ s.t. & (z(w)+e(w))/g_{ELL} < e(w)/m_c \\ & w \le c \\ & w \in N \end{array}$$

  The possible values of w is bounded by c, so we use brute force method to solve this problem.
- Extract SELL: the idea is the same as extracting ELL. The only difference is to consider the ELL width of each slice independently.
- ELL vs SELL: choose the one that has higher achievable GFLOPS value.
- CSR vs COO: the decision is based on the load balancing issue of CSR. Assuming there are u work groups in the CSR implementation. Let $nnz(i)$ be the number of non-zeros computed by work group i. For a matrix with n non-zeros, use the CSR format if the following rule is satisfied:

$$\frac{u \times \max\limits_{1 \le i \le u} nnz(i)}{g_{CSR}} < \frac{n}{g_{COO}}$$

- Merge small submatrices: merge a submatrix into another existing submatrix if such behavior results in better estimated performance.

The *Cocktail Format* is a superset over many single sparse matrix representations. Theoretically, the SpMV performance of the *Cocktail Format* should be at least the same as the SpMV performance of every format it covers. In practice, the performance depends on the policies of the *clSpMV* framework, and the accuracy of the estimated execution time (the value of the $t(A, f_i, p_{ij})$ function). This is a trade-off between analysis time and the SpMV performance. If we use more complicated policies and more accurate execution time estimates, we can find better matrix partitions, and achieve higher SpMV performance. However, it requires more analysis time. The analysis overhead of OSKI is about 40 SpMVs. *clSpMV* takes 1 SpMV for diagonal analysis, 20 SpMVs for block analysis of one block size, and 4 SpMVs for flat analysis.

Because the matrix analysis overhead is not trivial, *clSpMV* will be ideal for iterative methods that perform SpMV on a single sparse matrix for hundreds or thousands of times. Moreover, if the framework user is dealing with matrices with similar structures, one can perform full analysis on some exemplar matrices, and use the knowledge one gained from the experiments to speed up the analysis of future matrices. For example, if the exemplar matrices do not have any dense blocks, one can advise *clSpMV* to skip the analysis of the block based formats. To further reduce the analysis overhead, we can also follow the strategy used in [19]. Instead of analyzing the entire matrix, we can randomly sample the non-zeros of the matrix and make decision based on the samples. Since most of the analysis is based on counting (counting the number of non-zeros in a diagonal/in a block), we can also parallelize the analysis procedure to further reduce the overhead.

4. SPARSE MATRIX FORMATS AND PARALLELIZATION STRATEGIES ON GPU PLATFORMS

As introduced in Section 2 and Section 3, the *clSpMV* framework is an autotuner of the SpMV kernel across all platforms supporting OpenCL. It uses the *Cocktail Format* that combines many sparse matrix formats together. It also dynamically decides the representation of a given sparse matrix at runtime. The idea of the *Cocktail Format* and the idea of the *clSpMV* framework are platform independent. However, the optimized implementation of the SpMV kernel of every single format is platform dependent. In this section, we will discuss our platform of choice, and the optimized implementations of the 9 supported formats on the target platforms.

4.1 Target Platforms

The concept of the *Cocktail Format* is to take advantage of the strengths of a set of different sparse matrix formats. The idea of the *clSpMV* framework is to plug in many implementations of many SpMV kernels, performing analysis on the matrices, and to decide the best partition of the matrices. No matter what is the final decomposition of the matrices, the SpMV performance fully depends on the implementations of all supported formats. Because different platforms have different characteristics, there is no one-size-fits-all solution. To get the best performance, the implementations of the SpMV kernels should be platform dependent. Although many different platforms support OpenCL, they favor different parallelization strategies. For example, CPU based platforms favor coarse-grained parallelism, while GPU based platforms favor fine-grained parallelism. Because SpMV is a memory bounded problem [21], and modern GPU platforms have more memory bandwidth than DRAM, we decide to start by plugging in GPU-optimized SpMV kernels in our *clSpMV* framework. In the future, the *clSpMV* framework can support more platforms through more platform dependent implementations. Since we have decided to target GPU platforms, we will discuss the matrix formats and their corresponding parallelization strategies on GPU platforms in the following sections.

4.2 Diagonal Based Formats

$$B = \begin{bmatrix} 3 & 7 & 0 & 0 \\ 0 & 4 & 8 & 0 \\ 1 & 0 & 5 & 9 \\ 0 & 2 & 0 & 6 \end{bmatrix} \quad (1)$$

The diagonal based formats are the formats that capture dense diagonals. We will use matrix B in Equation (1) to explain the data structure of the formats in this category.

4.2.1 DIA Format

```
Offsets  =  [-2  0  1]
Data     =  [0  0  1  2,  3  4  5  6,  7  8  9  0]
```

Figure 1: The DIA format of matrix B.

As explained in [3], the diagonal (DIA) format is composed of two arrays, the `Offsets` array that stores the offsets of each diagonal, and the `Data` array that stores the dense diagonals. Figure 1 shows the DIA format of matrix B. The parallelization strategy of the DIA kernel is similar to [3]. Each work item is responsible for one row of the matrix. Because AMD platforms favor explicit vectorization by using the *float4* data type [2], we also implement the kernel that each work item being responsible for four rows of the matrix. Moreover, we implement kernels that use texture memory and kernels that do not use texture memory to store the multiplied vector. In the following we summarize the pros and cons of the DIA format:

- Pros: It does not need explicit column indices for each non-zero data. It has single-stride and aligned access on the matrix data. It also has single-stride access on the multiplied vector.
- Cons: It needs zero paddings in the `Data` array to ensure the lengths of all diagonals are the same. On sparse diagonals, the zero padding overhead might be significant.

4.2.2 BDIA Format

```
Offsets  =  [-2  0]
BandPtr  =  [0  1  3]
Data     =  [0  0  1  2,  3  4  5  6,  7  8  9  0]
```

Figure 2: The BDIA format of matrix B.

The banded diagonal (BDIA) format is a variation of the DIA format. Instead of storing disjoint diagonals, it stores a band as a whole. This format is composed of three arrays. The `Offsets` array stores the offset of the first diagonal in each band. The `BandPtr` array stores the position of the first element of each band. In other words, the elements of band i are stored between `BandPtr`$[i]$ and `BandPtr`$[i+1]$. The `Data` array is exactly the same as the `Data` array in the DIA format. Figure 2 shows the BDIA format of matrix B.

The parallelization strategy of the BDIA is very similar to the DIA format. We have implementations where each work item is responsible for one matrix row, and have implementations where each work item is responsible for 4 matrix rows using the *float4* data type. The major difference between the DIA format and the BDIA format comes from the fact that the diagonals in a band are consecutive, so we can predict the vector sections that each work item is accessing. For example, assuming that the size of a work group is 128. Assuming that work item r to work item $r+127$ in this work group are responsible for row r to row $r+127$, respectively. Considering a band with d diagonals, and the offset of the first diagonal being o, the work item i will access vector elements $o+i, o+i+1, \ldots, o+i+d-1$. The entire work group will access vector elements $o+r, o+r+1, \ldots, o+r+127+d-1$. The consecutive vector section can be cached into the shared local memory. We have implemented kernels that use this caching strategy, and kernels that do not use this caching strategy. In the following we summarize the pros and cons of the BDIA format:

- Pros: It does not need explicit column indices for each non-zero data. It has single-stride and aligned access on the matrix data. It also has single-stride access on the multiplied vector. It can use shared local memory to cache the multiplied vector.
- Cons: It needs zero paddings in the data array to ensure the lengths of all diagonals are the same. On sparse diagonals, the zero padding overhead might be significant. Compared to the DIA format, it requires an additional `BandPtr` array.

4.3 Flat Formats

The flat formats are the formats that need explicit storage of the column indices of all non-zeros. We will use matrix B to explain the data structure of the formats in this category.

4.3.1 ELL Format

The idea of the ELLPACK (ELL) format [9] is to pack all non-zeros towards left, and store the packed dense matrix. Assuming the packed dense matrix has dimension $m \times n$. The ELL width of the matrix will be n, the width of the packed dense matrix. The ELL format is composed of two

arrays, the `Col` array stores the column indices of all elements in the dense $m \times n$ matrix. The `Data` array stores the values of the dense $m \times n$ matrix. Figure 3 shows the ELL format of matrix B. The parallelization strategy is similar to [3]. Each work item is responsible for one row of the matrix. Considering platforms that favor explicit vectorization such as the AMD GPUs, we also have the implementation that each work item is responsible for 4 rows using the *float4* data type. Again, kernels using texture memory and kernels not using texture memory to cache the multiplied vector are all included. In the following we summarize the pros and cons of the ELL format:

```
Col  = [0 1 0 1, 1 2 2 3, 0 0 3 0]
Data = [3 4 1 2, 7 8 5 6, 0 0 9 0]
```

Figure 3: The ELL format of matrix B.

- Pros: The access pattern of the `Col` and the `Data` arrays is single-stride and aligned.
- Cons: Assuming the packed dense matrix has dimension $m \times n$, the ELL format needs zero paddings on every row that has number of non-zeros less than n. The zero paddings might introduce significant overhead. It requires random access on the multiplied vector.

4.3.2 SELL Format

```
SlicePtr = [0 4 10]
Col      = [0 1, 1 2; 0 2, 1 3, 3 0]
Data     = [3 4, 7 8; 1 2, 5 6, 9 0]
```

Figure 4: The SELL format of matrix B.

The sliced ELLPACK (SELL) format is proposed in [13]. The idea is to cut the original matrix into slices, and pack the slices into dense matrices with different dimensions. The SELL format is composed of three arrays. The `SlicePtr` array stores the beginning position of each slice. In other words, the elements of slice i are stored between `SlicePtr`$[i]$ and `SlicePtr`$[i+1]$. The `Col` array and the `Data` array are similar to the ELL format, storing the column indices and values of each element in the slices. Figure 4 shows the SELL format of matrix B with slice height 2. The semicolon in the array is used to separate different slices. Monakov et al. [13] uses autotuning to find the best slice height, reorders the matrix to further reduce the necessary zero paddings, and proposes the variable-height slice format. According to the experimental results in [13], the matrix reordering technique and the variable-height format only result in marginal improvements. Since these strategies might increase the complexity of the policies in the online decision making stage, current *clSpMV* does not include these approaches. Regarding the slice height, we develop kernels with slice heights equal to multiples of GPU alignment requirement (128 bytes). The parallelization strategy is the same as that in the ELL format. The only difference is that we need to cache the `SlicePtr` array in the local shared memory. In the following we summarize the pros and cons of the SELL format:

- Pros: The access pattern of the `Col` and the `Data` arrays is single-stride and aligned. It requires less zero paddings compared to the ELL format.
- Cons: It still needs zero paddings for each slice. The zero paddings might introduce significant overhead. It requires random access on the multiplied vector. It needs an additional `SlicePtr` array to store the slice positions.

4.3.3 CSR Format

```
RowPtr = [0 2 4 7 9]
Col    = [0 1, 1 2, 0 2 3, 1 3]
Data   = [3 7, 4 8, 1 5 9, 2 6]
```

Figure 5: The CSR format of matrix B.

The compressed sparse row (CSR) format is the most common sparse matrix format. It is composed of three arrays. The `RowPtr` array stores the beginning position of each row. In other words, the elements of row i are stored between `RowPtr`$[i]$ and `RowPtr`$[i+1]$. The `Col` array and the `Data` array are used to store the column indices and values of each non-zero. Figure 5 shows the CSR format of matrix B. [3] proposes two parallelization strategies for the CSR format. The scalar strategy will let one work item working on one row of the matrix. The vector strategy will let one warp of work items working on one row of the matrix. According to [3], the scalar strategy only outperforms the vector strategy when the number of non-zeros per row is small. However, when the number of non-zeros per row is small, the ELL format will be a better candidate. Therefore, we only have the vector strategy implemented in *clSpMV*. Again, implementations using texture memory and not using texture memory are both implemented. In the following we summarize the pros and cons of the CSR format:

- Pros: Need very few zero paddings.
- Cons: It might have unaligned access on both the `Col` array and the `Data` array. The access pattern on the multiplied vector is random. It might have load balancing problem if the number of non-zeros per row varies significantly.

4.3.4 COO Format

```
Row  = [0 0, 1 1, 2 2 2, 3 3]
Col  = [0 1, 1 2, 0 2 3, 1 3]
Data = [3 7, 4 8, 1 5 9, 2 6]
```

Figure 6: The COO format of matrix B.

The coordinate (COO) format explicitly stores the row indices. It is composed of three arrays. The `Row` array, the `Col` array, and the `Data` array store the row indices, the column indices, and the values of all non-zeros in the matrix, respectively. Figure 6 shows the COO format of matrix B. The parallelization strategy is the same as [3]. We are performing segmented reduction computation on the three arrays. However, the implementation in [3] requires three kernel launches. By padding zeros at the end of three arrays to match the work group size, we only need two kernel launches in our OpenCL implementation. In the following we summarize the pros and cons of the COO format:

- Pros: Need very few zero paddings. There is no load balancing problem. As shown in [3], it can deliver

consistent performance regardless of the structure of the matrix.

- Cons: Has the worst memory footprint. It requires explicit indexing on both row and column indices. It needs random access on the multiplied vector.

4.4 Block Based Formats

$$C = \begin{bmatrix} 0 & 1 & 2 & 3 & g & h & i & j \\ 4 & 5 & 6 & 7 & k & l & m & n \\ 8 & 9 & a & b & o & p & q & r \\ c & d & e & f & s & t & u & v \end{bmatrix} \quad (2)$$

The block based formats are the variations of the flat formats. Instead of storing each non-zero independently, we store a block contiguously. We are going to use matrix C in Equation (2) to show exemplars of block based formats. For block sizes, because AMD platforms always prefers the *float4* data type [2], while Nvidia platforms achieve similar performance on both *float* and *float4* data types, we decide to use block sizes that are multiples of 4. Moreover, when using texture memory to cache the multiplied vector, the OpenCL API always returns a *float4* value. If we do not use all the 4 elements in the returned value, memory bandwidth is wasted. Therefore, we choose block sizes with widths being multiples of 4. The block sizes supported by *clSpMV* are 1×4, 2×4, 4×4, 8×4, 1×8, 2×8, 4×8, and 8×8.

4.4.1 BELL Format

```
Col  = [ 0  0,  4  4  ]
Data = [ 0  1   2  3,  8  9  a  b,
         4  5   6  7,  c  d  e  f;
         g  h   i  j,  o  p  q  r,
         k  l   m  n,  s  t  u  v  ]
```

Figure 7: The BELL format of matrix C. The block size is 2×4

The blocked ELLPACK (BELL) format is a variation of the ELL format. Instead of storing singular non-zeros, it stores a block of consecutive non-zeros. Each block only needs one column index, so the memory footprint is reduced. If the height of the block is larger than 1, the same data read from the multiplied vector can be reused across all the rows in the block. The BELL format is composed of two arrays. The `Col` array stores the column indices of the first elements from all blocks. The `Data` array stores the values of all blocks. Moreover, we need special arrangement to enforce single-strided memory access on the `Data` array. Because the 1×4 block is the smallest unit of all block sizes we support, the `Data` array is managed in a 2D fashion. The first dimension corresponds to the data of a 1×4 block. The second dimension corresponds to the number of 1×4 units in the block dimension. Figure 7 shows the BELL format of matrix C. The block size is 2×4, so there are two 1×4 units in the block. The `Data` array can be viewed as a 2×16 array. We store the first 1×4 unit of each block, and then store the next 1×4 unit of each block.

The parallelization strategy is similar to the ELL format. However, instead of letting one work item working on a row, we let one work item work on a superrow. Because the smallest unit of all block sizes is 1×4, we use the *float4* data type in our implementation. In the following we summarize the pros and cons of the BELL format:

- Pros: We have single-stride data access on the `Data` array. The required memory storage of the column indices is reduced. If the block has height larger than 1, the segment of the multiplied vector can be cached in registers, and used across multiple block rows.
- Cons: It needs zero fillings in the blocks. It also needs zero paddings to make sure that the number of blocks per row are all the same. The fillings and paddings might introduce overhead.

4.4.2 SBELL Format

```
SlicePtr = [ 0  4  8  ]
Col      = [ 0  0,  4  4;  0  0,  4  4  ]
Data     = [ 0  1   2  3,  4  5   6  7,
             g  h   i  j,  k  l   m  n;
             8  9   a  b,  c  d   e  f,
             o  p   q  r,  s  t   u  v  ]
```

Figure 8: The SBELL representation of matrix C. The block size is 1×4. The slice height is 2.

The sliced blocked ELLPACK (SBELL) format is proposed in [6]. Although the data arrangement in *clSpMV* is different from [6], the idea is similar. In *clSpMV*, the SBELL format is composed of three arrays. The `SlicePtr` array stores the beginning position of each slice. In other words, the elements of slice i are stored between `SlicePtr`$[i]$ and `SlicePtr`$[i+1]$. The `Col` array stores the column indices of the first elements from all blocks. The `Data` array stores the values of all blocks. The data of a slice are stored consecutively. Like the case in the BELL format, in a slice, the data of a 1×4 unit are stored consecutively, and the data of multiple 1×4 units will be stacked into a large array. Figure 8 shows the SBELL format of matrix C, with block size 1×4 and slice height 2. Choi el al. [6] also rearranges the matrix rows to reduce the paddings of the SBELL format. Because we are using the SBELL format to represent only the portion of the matrix that is best for SBELL, we believe the remaining singular non-zeros will be taken care of by the flat formats. Therefore, we did not try to reorder matrix in our implementation. The parallelization strategy is similar to the BELL format. Each work item is responsible for a superrow. In the following we summarize the pros and cons of the SBELL format:

- Pros: We have single-stride data access on the `Data` array. The required memory storage of the column indices is reduced. If the block has height larger than 1, the segment of the multiplied vector can be cached in registers, and used across multiple block rows. It requires less zero paddings compared to the BELL format.
- Cons: It needs zero fillings in the blocks. It also needs zero paddings to make sure that the number of blocks per row in a slice are all the same. The fillings and paddings might introduce overhead. It also requires an additional `SlicePtr` array.

4.4.3 BCSR Format

The blocked compressed sparse row (BCSR) format is also discussed in [6]. The data arrangement in *clSpMV* is differ-

```
RowPtr  =  [  0  2  4  ]
Col     =  [  0  4,  0  4  ]
Data    =  [  0  1  2  3,  g  h  i  j,
              8  9  a  b,  o  p  q  r;
              4  5  6  7,  k  l  m  n,
              c  d  e  f,  s  t  u  v  ]
```

Figure 9: The BCSR format of matrix C. The block size is 2×4

ent, but the idea is similar. The BCSR format is composed of three arrays. The `RowPtr` array stores the beginning position of each superrow. In other words, the elements of superrow i are stored between `RowPtr`$[i]$ and `RowPtr`$[i+1]$. The `Col` array stores the column indices of the first elements from all blocks. The `Data` array stores the values of all blocks. Like the case in the BELL format, the data of a 1×4 unit are stored consecutively, and the data of multiple 1×4 units will be stacked into a large array. Figure 9 shows the BCSR format of matrix C, with block size 2×4. The parallelization strategy is similar to the vector CSR strategy used in [3]. A warp of work items is responsible for a superrow. In the following we summarize the pros and cons of the BCSR format:

- Pros: The required memory storage of the column indices is reduced. If the block has height larger than 1, the segment of the multiplied vector can be cached in registers, and used across multiple block rows. It does not need to pad zero blocks at the end of each superrow.
- Cons: It needs zero fillings in the blocks. It might have unaligned access on the `Data` array. It might have load balancing problem.

5. EXPERIMENTAL RESULTS

We can evaluate the performance of *clSpMV* on different platforms, given the cross-platform capabilities of OpenCL. Nvidia and AMD are the major GPU vendors, so we evaluate the framework's performance on these two different platforms. Since both platforms achieve higher performance on single precision floating point data type, we use such data type in our experiments. In this section, we will first introduce the matrices we used for benchmarking, and then discuss the performance of *clSpMV* on Nvidia GTX 480 and AMD Radeon 6970. The code from our implementation is freely available at `http://www.eecs.berkeley.edu/~subrian/clSpMV.html`.

5.1 The Benchmarking Matrices

We use the 14 matrices in [20] as our benchmarking matrices. The same set of matrices is also used in [3,6,13]. The statistics of the matrices are summarized in Table 1. The # rows column, the # cols column, the # nnzs column, and the nnz/row column summarize the number of rows, the number of columns, the number of non-zeros, and the average number of non-zeros per row, respectively. Unfortunately, this set contains mostly regular matrices that are well-suited for single-format representation. Although the *clSpMV* is able to find the best single format to represent the matrices, it is hard to see how the *Cocktail Format* will further improve the performance. Therefore, we add 6 additional matrices from [7] in our benchmarking suite. We choose matrices that has balanced portions of diagonals, dense blocks, and random singular non-zeros, matrices that has highly irregular distributions of the non-zeros, and matrices that have enough large number of non-zeros such that the overhead of launching multiple kernels will not be significant. The statistics of the 6 additional matrices are also summarized in Table 1.

Table 1: Overview of the sparse matrix benchmark.

Name	# rows	# cols	# nnzs	nnz/row
Dense	2K	2K	4M	2000
Protein	36K	36K	4.3M	119
FEM/Spheres	83K	83K	6M	72
FEM/Cantilever	62K	62K	4M	65
Wind Tunnel	218K	218K	11.6M	53
FEM/Harbor	47K	47K	2.37M	50
QCD	49K	49K	1.9M	39
FEM/Ship	141K	141K	3.98M	28
Economics	207K	207K	1.27M	6
Epidemiology	526K	526K	2.1M	4
FEM/Accelerator	121K	121K	2.62M	22
Circuit	171K	171K	959K	6
Webbase	1M	1M	3.1M	3
LP	4K	1.1M	11.3M	2825
circuit5M	5.56M	5.56M	59.5M	11
eu-2005	863K	863K	19M	22
Ga41As41H72	268k	268k	18M	67
in-2004	1.38M	1.38M	17M	12
mip1	66K	66K	10M	152
Si41Ge41H72	186k	186k	15M	81

5.2 Experimental Results on Nvidia GTX 480

We summarize the performance benchmarking results on the Nvidia GTX 480 platform in Figure 10. The x axis is the dimension of the matrix. On row 1 and 3, the y axis is the number of non-zeros per row. On row 2, the y axis is the number of dense blocks per superrow. The unit of the color-bar is in GFLOPS. Some performance numbers at the top-right corners are missing because the matrix storage size is larger than a pre-defined upperbound. Some performance numbers at the top-left corners are missing because the matrix is too dense to be considered as a sparse matrix. The performance of the diagonal based formats are benchmarked on dense diagonal matrices. Although each diagonal format has multiple implementations, the heat-map only shows the best achievable performance among all implementations. As expected, the performance increases with the increase in matrix dimension and number of non-zeros per row. The peak performance of the BDIA format is larger than that of the DIA format. When the number of non-zeros per row is very small, the DIA format will slightly outperform the BDIA format. The performance of the block based formats are benchmarked on dense diagonal blocked matrices. Due to the space limitations, only the performances of the 1×4 blocked matrices are included in the figure. Blocked matrices with other block dimensions follow the similar pattern. For the BELL and the SBELL formats, each thread is working on a superrow, we can get close to peak performance when there are 20 to 30 dense blocks per superrow. However, for the BCSR format, because a warp of work items is responsible for a superrow, we need more than 200 dense blocks per superrow to saturate the processors. The performance of the flat formats are benchmarked on dense diagonal matrices. The performance patters of the flat formats are very close to their blocked variations, but their peak achievable

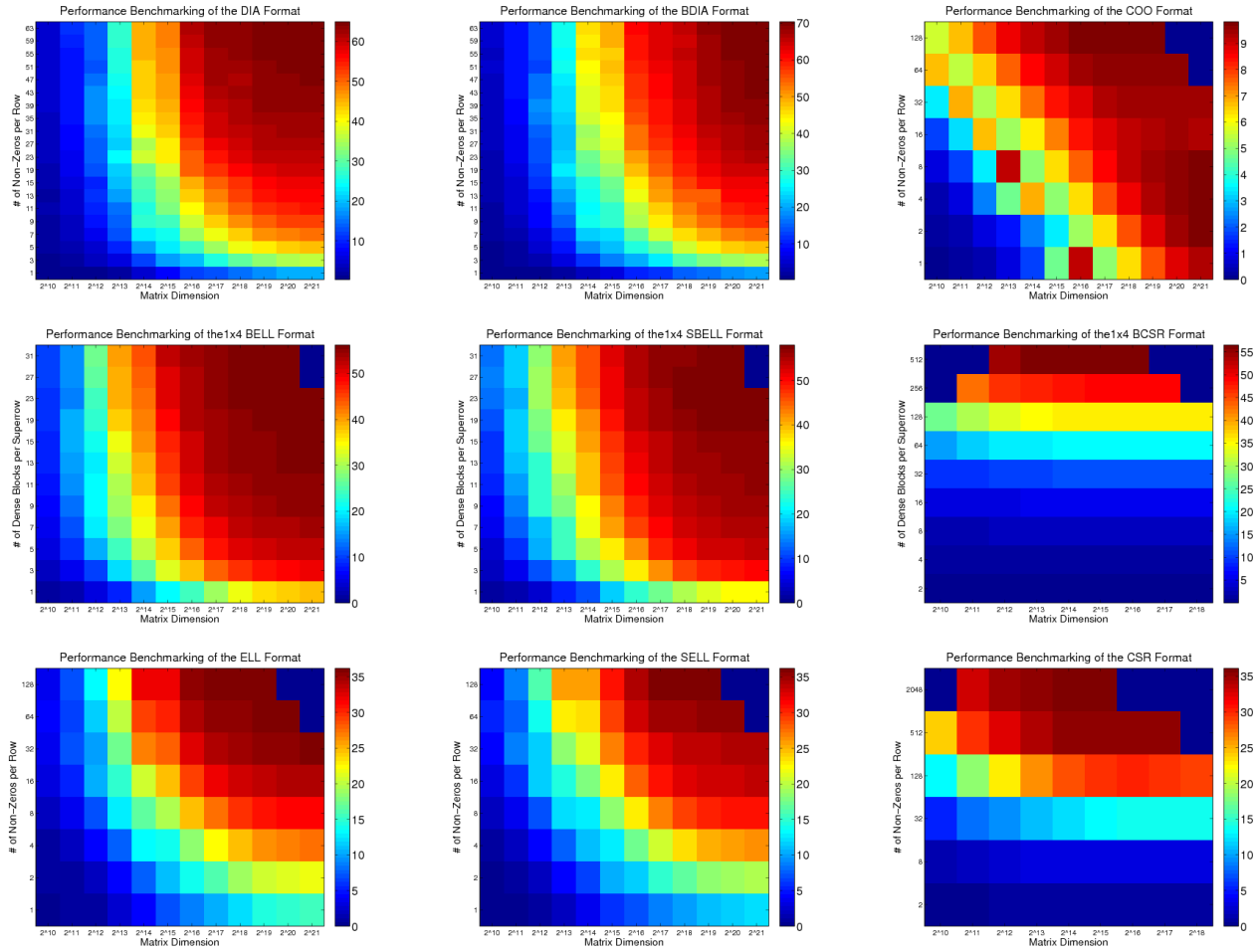

Figure 10: The performance benchmarking on the Nvidia GTX 480 platform. The x axis is the dimension of the matrix. On row 1 and 3, the y axis is the number of non-zeros per row. On row 2, the y axis is the number of dense blocks per superrow. The unit of the color-bar is in GFLOPS.

performances are significantly reduced. The COO performance is very stable while the dimension of the matrix is large enough. However, The peak achievable performance is the lowest among the 9 formats.

To evaluate the performance of *clSpMV*, we compare the performance with other implementations on the 20 benchmarking matrices. We first compare the performance with [3]. The released code of [3] is based on CUDA, and has SpMV kernels of the DIA, ELL, CSR, COO, and the HYB format. The HYB format in [3] is composed of ELL format and COO format. Therefore, the HYB format is a subset of our *Cocktail Format*. Although we also want to compare the performance of *clSpMV* with the SELL format in [13] and the blocked formats in [6], they did not release their code. As a result, we compare to our own OpenCL implementations of the SELL, BELL, SBELL, and the BCSR formats instead.

The experimental results are summarized in Table 2. The performance is measured by $\frac{2\times nnz}{ExecutionTime}$ (GFLOPS). Sometimes using texture memory to cache the multiplied vector will result in higher performance, sometimes not. We evaluate both cases and only report the highest number in the table. The NV HYB column lists the performance achieved by the CUDA code of the HYB format in [3]. The Best NV column lists the highest performance achieved among all the 5 implementations supported by [3], including DIA, ELL, CSR, COO, and HYB. The Best NV Format column lists the format that achieves the highest performance. The Best Single column lists the highest performance achieved among all single formats. Among all single format performances, the DIA, ELL, CSR, and COO performance is measured using the CUDA implementations from [3]; the BDIA, SELL, BELL, SBELL, and BCSR performance is measured using our own OpenCL implementations. Because we have multiple implementations of every single format as introduced in Section 4, such as different block sizes of the block based formats, only the highest performance numbers among all implementations are reported. The Best Single Format column summarizes the format that achieves the highest performance. The *clSpMV* column lists the performance achieved by *clSpMV* framework. The *clSpMV* Format column lists the decision made by *clSpMV*. The percentage numbers following the formats refer to the portions of non-zeros covered by the formats.

Table 2: The *clSpMV* performance on the selected 20 matrices, compared to implementations in [3], and all the single formats supported by *clSpMV* on Nvidia GTX 480. The highest achieved performance for each matrix is in bold.

Benchmark Name	NV CUDA			Single All		*clSpMV*	
	NV HYB (GFLOPS)	Best NV (GFLOPS)	Best NV Format	Best Single (GFLOPS)	Best Single Format	*clSpMV* (GFLOPS)	*clSpMV* Format
Dense	8.38	32.63	CSR	**54.08**	BCSR	53.05	BCSR
Protein	15	23.17	CSR	35.84	SBELL	**35.86**	SBELL
FEM/Spheres	25.11	25.11	HYB	34.44	SBELL	**34.52**	SBELL
FEM/Cantilever	19.06	34.9	DIA	35.03	SBELL	**35.10**	SBELL
Wind Tunnel	25.07	25.07	HYB	**42.94**	SBELL	**42.94**	SBELL
FEM/Harbor	11.67	13.83	CSR	27.17	SBELL	**27.21**	SBELL
QCD	25.05	25.09	ELL	**30.93**	SELL	29.88	ELL
FEM/Ship	19.11	19.11	HYB	40.59	SBELL	**40.73**	SBELL
Economics	7.61	7.61	HYB	7.32	SELL	**10.59**	ELL(81%)COO(19%)
Epidemiology	24.02	24.02	ELL	25.36	SELL	**26.55**	ELL
FEM/Accelerator	9.35	9.35	HYB	**16.29**	SBELL	15.25	SELL
Circuit	7.35	7.35	HYB	7.72	SELL	**11.40**	ELL(84%)COO(16%)
Webbase	9.74	9.74	HYB	7.30	COO	**12.77**	ELL(64%)COO(36%)
LP	8.89	12.78	CSR	**12.99**	BCSR	12.98	BCSR
circuit5M	12.81	12.81	HYB	9.02	COO	**17.07**	DIA(9%)SELL(73%)COO(18%)
eu-2005	12.14	12.14	HYB	11.84	SBELL	**16.03**	SELL(85%)COO(15%)
Ga41As41H72	13.28	16.11	CSR	16.11	CSR	**16.80**	BDIA(18%)ELL(32%)CSR(50%)
in-2004	10.53	10.53	HYB	12.04	SBELL	**16.87**	SELL(79%)COO(21%)
mip1	10.8	18.92	CSR	18.92	CSR	**19.00**	SBELL(80%)SELL(17%)COO(3%)
Si41Ge41H72	12	17.68	CSR	17.68	CSR	**18.77**	BDIA(15%)ELL(27%)CSR(58%)

Table 3 summarizes the improvement ratios of *clSpMV* compared to all other implementations based on the performance numbers in Table 2. On average, *clSpMV* is 83% faster than the CUDA implementation of the proposed HYB format in [3]; 63.6% faster than all CUDA implementations in [3]; and 16.8% faster than all single formats supported by *clSpMV*.

For the 14 matrices from [20], most of them have regular structures, and the total number of non-zeros are small. Therefore, they favor single format representation. As shown in Table 2, most of the time *clSpMV* can successfully find the best single representations that match the results in the Best Single Format column. Even if the chosen format is not the same, the performance difference is very small. There are three matrices that the *clSpMV* matches the HYB format in [3]. The reason that *clSpMV* outperforms the CUDA implementation of the HYB format is due to three factors. First, the HYB format in [3] assumes that ELL format is 3 times faster than COO format. In contrast, *clSpMV* uses the more accurate offline benchmarking numbers. Second, the COO implementation in [3] needs three kernel launches, but *clSpMV* only needs two. Third, the number of work groups (or thread blocks) used by the COO implementation in [3] is fixed. However, *clSpMV* chooses the best work group size based on the offline benchmarking information.

For the 6 additional matrices from [7], *clSpMV* partitions them into many submatrices. *clSpMV* achieves significant improvements on three matrices ($40\% - 90\%$ better performance), but small improvements on the other three matrices ($0.4\% - 6\%$). This is due to texture memory. Texture memory boosts CSR performance from 10 GFLOPS to $16 - 18$ GFLOPS. Therefore, the data access pattern of CSR has very high hit rate on the texture cache. Though CSR performance is good, *clSpMV* achieves even greater performance. Theoretically, the *Cocktail Format* should outperform every single format. In practice, *clSpMV* uses good policies to find reasonable matrix partitions, represents them using the *Cocktail Format*, and achieves better performance compared to all other single formats.

Table 3: The improvement of *clSpMV* compared to the hybrid format in [3], the best implementations in [3], and the best single format implementations supported by *clSpMV*.

Benchmark Name	*clSpMV* Improvement		
	NV HYB	Best NV	Best Single
Dense	533.1%	62.6%	-1.9%
Protein	139.1%	54.8%	0.1%
FEM/Spheres	37.5%	37.5%	0.2%
FEM/Cantilever	84.2%	0.6%	0.2%
Wind Tunnel	71.3%	212.5%	0.0%
FEM/Harbor	133.1%	96.7%	0.1%
QCD	19.3%	19.1%	-3.4%
FEM/Ship	113.1%	123.4%	0.3%
Economics	39.2%	85.2%	44.7%
Epidemiology	10.5%	10.5%	4.7%
FEM/Accelerator	63.1%	97.5%	-6.4%
Circuit	55.1%	124.9%	47.6%
Webbase	31.1%	74.9%	74.9%
LP	46.0%	1.5%	-0.1%
circuit5M	33.3%	89.2%	89.2%
eu-2005	32.1%	82.4%	35.5%
Ga41As41H72	26.5%	4.3%	4.3%
in-2004	60.2%	86.8%	40.1%
mip1	75.9%	0.4%	0.4%
Si41Ge41H72	56.4%	6.2%	6.2%
Average	83.0%	63.6%	16.8%

5.3 Experimental Results on ATI Radeon 6970

The experimental settings on the AMD platform are very similar to that on the Nvidia platform. The performance benchmarking results are summarized in Figure 11. More performance numbers at the top-right corners are missing because the required matrix storage sizes of these sample points exceed 256 MB, the largest consecutive memory size allowed by the AMD OpenCL runtime. We are not aware of any SpMV project that targets AMD platforms, so we only compare *clSpMV* with the single format implementations supported by *clSpMV*. The results are shown in Table 4. On average, the performance of *clSpMV* is 43.3% higher than all the single format implementations. On the dense

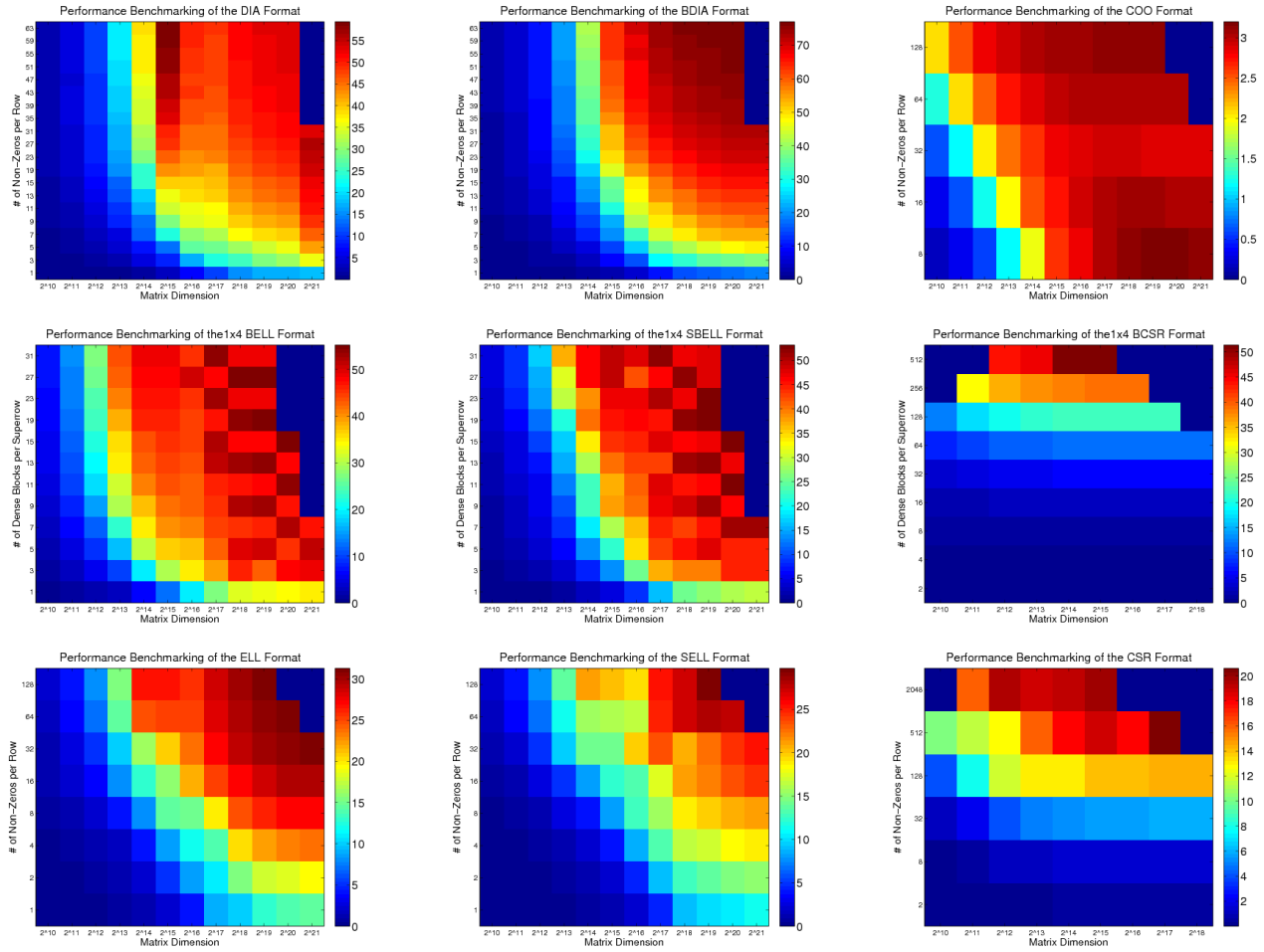

Figure 11: The performance benchmarking on the ATI Radeon 6970 platform. The x axis is the dimension of the matrix. On row 1 and 3, the y axis is the number of non-zeros per row. On row 2, the y axis is the number of dense blocks per superrow. The unit of the color-bar is in GFLOPS.

matrix and the LP matrix, *clSpMV* chooses the right single format, but the chosen block size is not optimal, and the performance is worse than the best single format. An offline benchmarking procedure with wider and denser sample points can give better execution time estimates, and enable *clSpMV* to find the best block size.

When comparing Table 2, Table 4, Figure 10, and Figure 11, we see that *clSpMV* makes decisions based on platform strengths. Since the BDIA format achieves significant higher performance than all other formats on the AMD platform, it favors BDIA format whenever possible. Moreover, the ELL performance on the AMD platform is significantly better than the COO performance, so the *clSpMV* increases the ratio of the ELL portion on the AMD platform.

6. CONCLUSION

In this paper, we have proposed a new sparse matrix format, the *Cocktail Format*, and the *clSpMV* framework, an OpenCL SpMV framework on GPU platforms. Theoretically, the *Cocktail Format* is a superset over all single sparse matrix formats, so its performance should be better than, or at least equal to all single formats. In practice, with the help of the *clSpMV* framework, we have achieved 16.8% better performance than any single formats on the Nvidia GTX 480 platform, and 43.3% better performance on the AMD Radeon 6970 platform. Although solutions that are portable across diverse platforms generally provide lower performance when compared to solutions that are specialized to particular platforms, we achieved 83% better performance compared to the CUDA implementation of the proposed HYB format in [3]; and 63.6% better performance compared to all CUDA implementations in [3]. In conclusion, the *Cocktail Format* delivers better SpMV performance both theoretically and practically; *clSpMV* is a cross-platform framework that is able to choose the best representation of any given matrices, and deliver very high performance SpMV kernels.

7. ACKNOWLEDGMENTS

Thanks to Nadathur Satish for reviewing and commenting the clSpMV framework. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding, and by matching funding from U.C. Discovery (Award $\#DIG07-10227$).

Table 4: The *clSpMV* performance on the selected 20 matrices, compared to all the single formats supported by *clSpMV* on AMD Radeon 6970. The highest achieved performance for each matrix is in bold.

Benchmark Name	Single All		*clSpMV*		Improvement
	Best Single (GFLOPS)	Best Single Format	*clSpMV* (GFLOPS)	*clSpMV* Format	
Dense	**46.85**	BCSR	41.85	BCSR	-10.7%
Protein	29.91	SBELL	**30.99**	BDIA(43%)SBELL(57%)	3.6%
FEM/Spheres	**31.85**	SBELL	31.44	SBELL	-1.3%
FEM/Cantilever	33.72	DIA	**35.93**	BDIA(90%)ELL(10%)	6.5%
Wind Tunnel	**35.23**	SBELL	34.51	SBELL	-2.0%
FEM/Harbor	**22.29**	SBELL	22.20	SBELL	-0.4%
QCD	24.84	SELL	**25.01**	BELL	0.7%
FEM/Ship	33.75	SBELL	**34.43**	SBELL	2.0%
Economics	4.87	SELL	**9.04**	ELL(88%)COO(12%)	85.9%
Epidemiology	22.51	ELL	**22.58**	ELL	0.3%
FEM/Accelerator	**15.83**	SELL	15.51	SELL	-2.0%
Circuit	3.06	COO	**8.40**	ELL(88%)COO(12%)	174.7%
Webbase	3.26	COO	**6.42**	ELL(70%)COO(30%)	97.0%
LP	**10.03**	BCSR	9.50	BCSR	-5.3%
circuit5M	3.21	COO	**8.06**	SELL(82%)COO(18%)	150.7%
eu-2005	3.01	COO	**8.19**	ELL(83%)COO(17%)	172.1%
Ga41As41H72	4.70	CSR	**6.93**	BDIA(18%)ELL(32%)CSR(50%)	47.5%
in-2004	3.04	COO	**7.42**	SBELL(28%)ELL(53%)COO(19%)	144.2%
mip1	8.27	BCSR	**8.28**	BDIA(20%)SBELL(62%)SELL(14%)COO(4%)	0.2%
Si41Ge41H72	10.81	SBELL	**11.10**	BDIA(15%)SBELL(85%)	2.7%
Average					43.3%

8. REFERENCES

[1] AMD. ATI Stream Computing User Guide, 2008.

[2] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, 2011. `http://developer.amd.com/zones/OpenCLZone`.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 18:1–18:11, New York, USA, 2009.

[4] R. Bordawekar and M. M. Baskaran. Optimizing sparse matrix-vector multiplication on gpus. *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[5] A. Buluc, and, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 721–733, may 2011.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 115–126, New York, USA, 2010.

[7] T. A. Davis and Y. Hu. University of florida sparse matrix collection. 38(1), 2011. `http://www.cise.ufl.edu/research/sparse/matrices`.

[8] D. Grewe and A. Lokhmotov. Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 12:1–12:8, New York, USA, 2011.

[9] R. G. Grimes, D. R. Kincaid, and D. M. Young. Itpack 2.0 user's guide. Technical Report CNA-150, University of Texas, Austin, TX, USA, August 1979.

[10] P. Guo and L. Wang. Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus. In *International Conference on Computational and Information Sciences (ICCIS)*, pages 1154–1157, 2010.

[11] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, pages 18:135–18:158, February 2004.

[12] Intel. Intel Advanced Vector Extensions Programming Reference. 2009. `http://software.intel.com/en-us/avx`.

[13] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *High Performance Embedded Architectures and Compilers*, pages 111–125, 2010.

[14] Nvidia. Nvidia cuda, 2007. `http://nvidia.com/cuda`.

[15] S. Thakkur and T. Huff. Internet streaming simd extensions. *Intel Technology Journal Q2*, 32(12):26–34, dec 1999.

[16] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems, 2011. `http://www.khronos.org/opencl`.

[17] F. Vázquez, G. Ortega, J. Fernández, and E. Garzón. Improving the performance of the sparse matrix vector product with gpus. In *IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 1146–1151, 2010.

[18] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, June 2005.

[19] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[20] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 38:1–38:12, New York, USA, 2007.

[21] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[22] S. Yousef. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.