Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining

Xintian Yang
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210
yangxin@cse.ohio-state.edu

Srinivasan Parthasarathy
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210
srini@cse.ohio-state.edu

P. Sadayappan
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210
saday@cse.ohio-state.edu

ABSTRACT

Scaling up the sparse matrix-vector multiplication kernel on modern Graphics Processing Units (GPU) has been at the heart of numerous studies in both academia and industry. In this article we present a novel approach to data representation for computing this kernel, particularly targeting sparse matrices representing power-law graphs. Using real data, we show how our representation scheme, coupled with a novel tiling algorithm, can yield significant benefits over the current state of the art GPU and CPU efforts on a number of core data mining algorithms such as PageRank, HITS and Random Walk with Restart.

1. INTRODUCTION

Over the last decade we have witnessed a revolutionary change in the way commodity processor architectures are being designed and implemented. CPUs with superscalar out-of-order execution, vector processing capabilities, and simultaneous multithreading, chip multiprocessing (CMP), and high end graphics processor units (GPU) have all entered the mainstream commodity market. Data mining algorithms often require significant computational resources, and thus stand to benefit significantly from such innovations if appropriately leveraged.

In this article we develop a novel approach to facilitate the efficient processing of key graph-based data mining algorithms such as PageRank [7, 25], HITS [15] and Random Walk with Restart[26] on modern GPUs. A common feature of these algorithms is that they rely on a core sparse matrix vector multiplication kernel (SpMV). Implementations of this kernel on GPUs has received much attention recently from the broader scientific and high performance computing communities [8, 2, 9] including an industrial strengthened effort from NVIDIA research [3, 4] – arguably the leader in the development of GPU technology.

The key difference between past work and ours is that here we are interested in the processing of sparse matrices that represent large graphs – typically with power-law [20] characteristics. This difference is also central to the specific architecture-conscious approach we propose for processing the SpMV kernel. We transform and represent the matrix in such a way so as to facilitate tiling – a key strategy used to enhance temporal locality. Additionally we rely on a composite storage algorithm that leverages the skew in the degree distribution afforded by the fact that these matrices represent power-law graphs. Architectural features of the GPU such as the texture cache are also effectively leveraged in the processing of the kernel.

We present a comprehensive empirical evaluation of the proposed approach on three data mining algorithms and the base SpMV kernel on a range of real datasets including several moderately large graph datasets. We also provide a detailed comparison with other leading efforts – both academic and industrial – on this problem. On the core SpMV kernel we find that our approach is typically a *factor of 2* faster than the next best GPU competitor on matrices representing graphs with power law characteristics. This is a significant result especially in light of the industrial strength efforts spent on this problem. On the data mining kernels – HITS, PageRank and Random Walk with Restart - operating on datasets drawn from Flickr, LiveJournal, Wikipedia and Youtube we find that our approach is anywhere from 1.5 to 2 times faster than the next best GPU competitor. To put these results in context, our best results are 13 to 37 times faster than similarly structured and optimized implementations on state of the art CPUs.

2. BACKGROUND

GPUs were fixed-function devices for accelerating graphics rendering pipelines before NVIDIA introduced the CUDA general purpose parallel computing architecture [17, 21]. CUDA enables developers to program the devices for computation and data intensive applications. In this section, we discuss the hardware architecture and programming model of CUDA GPUs. Figure 1 illustrates the organization of computing hardwares and memory hierarchy in CUDA GPUs.

A CUDA device consists of a set of *streaming multiprocessors* (SMs), each one equipped with one instruction unit and a cluster of 8 *streaming processors* (SPs). The paral-

lel region of a CUDA program is partitioned into a *grid* of thread *blocks* that run *logically* in parallel. The programmer can decide the dimensions of grid and block. Thread blocks are distributed evenly on multiprocessors. A *warp* is a group of 32 threads that actually run concurrently on a multiprocessor. The execution of the threads follows a *single instruction multiple threads* (SIMT) model [17, 21]. The instruction unit on a multiprocessor issues one instruction for all the threads in a warp at each time [23]. The streaming processors executes this instruction for all the threads in a warp. Different warps within a block are time-shared on the actual hardware resources. A *kernel* is the actual code in the parallel region to be executed by each thread. Conditional instructions cause divergence in the execution if threads in a warp take different conditional paths. The threads are serialized in this situation.

There are various memory units on a CUDA device. The device memory, also called global memory is a large memory which is visible to all threads on the device [23]. The access latency of global memory is high. Memory requests of a half warp (16 threads) are served together at a time. When accessing a 4- or 8-byte word, the global memory is organized into 128-byte segments [23]. The number of memory transactions executed for a half warp is the number of memory segments requested by this half warp.[1] The requests from threads in a half warp are *coalesced* into one memory transaction if they are accessing addresses in the same segment. When the addresses accessed by a half warp are all in one segment, we call this request *fully coalesced*. Each multiprocessor is equipped with an on-chip scratchpad memory [23], called shared memory. The shared memory has very low access latency. It is only visible to threads within one block and has the same lifetime as the block [23]. The shared memory is organized into banks. If multiple addresses in the same bank are accessed at the same time, it leads to bank conflicts and the accesses are serialized. There are also a set of registers shared by threads in a block. The constant and texture memories are read-only regions in the global memory space with on-chip caches. The programmer can bind a region of global memory to constant or texture memory before the kernel starts.

2.1 Related Work

The Sparse Matrix-Vector Multiplication (SpMV) kernel computes a vector y as the product of a n by m sparse matrix A and a dense vector x. Since the SpMV kernel is widely used in scientific computing, several existing efforts have targeted optimizing this kernel for the GPU [3, 4, 2, 9]. However, none of the above take into account the skew of the non-zero distribution present in matrices representing power-law graphs.

Bell and Garland [3, 4] propose several representations of sparse matrices on the CUDA platform for SpMV kernels in NVIDIA's SpMV library. The *compressed sparse row* (CSR) format is widely used to represent sparse matrices. In the CSR format, the non-zeros in the same row are stored contiguously in memory, and all rows are stored in one data array, with another array holding the column indices of the non-zeros. A third array of row pointers marks the boundary of each row. The corresponding CSR kernel assigns the computation of each row to a thread. With power-law graphs, it is possible to balance the workload among thread blocks, but it is hard to balance among threads within one block. So all the threads in one block will wait for the thread which is assigned to the longest row in this block. To optimize this method, Bell and Garland [3, 4] developd the CSR-vector format, in which a warp of 32 threads are assigned to work on each row. This strategy only helps the rows with more than 32 non-zeros, but most of the nodes in power-law graphs have degree lower than 32. The computation resources of the warps assigned to such rows will therefore be wasted. Baskaran and Bordawekar [2] further optimized the CSR-vector format by using a half warp for each row to improve global memory accesses, and also a padding technique is used to ensure the memory requests are fully coalesced.

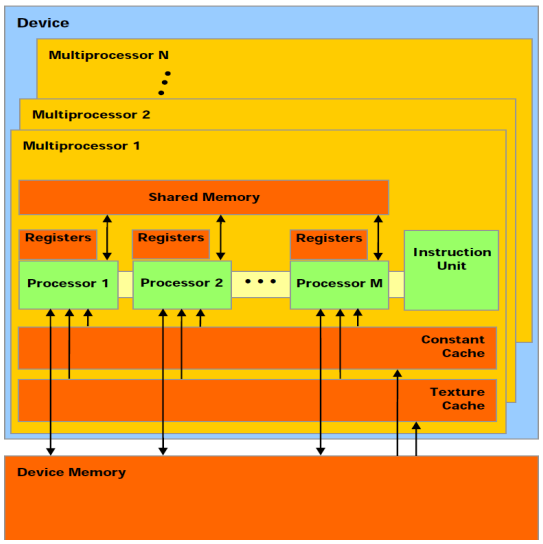


Figure 1: Hardware Organization and Memory Hierarchy of a CUDA Device [23]

Besides the CSR format, the *coordinate* (COO) and *ell-pack* (ELL) formats are also used in Bell's SpMV kernel. In COO format, all the non-zeros in matrix A are combined in a long vector grouped by row index, and the kernel first computes the multiplication of each non-zeros with the corresponding elements of vector x in the first pass; then the segmented reduction of the rows is done on this long vector by thread warps. In the reduction phase, because the length of each row is not necessary a multiple of warp size, synchronization points are heavily used and warp thread divergence is frequent. However, the COO kernel is the most insensitive to variable row length in the matrix according to a previous study [3]. The ELL format requires the number of non-zeros on each row is bounded by some small number k, so that the matrix A can be represented by a dense n by k matrix M, in which only non-zeros in A are stored, and the corresponding column indices of these non-zeros are also stored in a separate matrix. In the ELL kernel, M is stored in column major, and the thread assigned to each row can access global memory very efficiently. In M, zeros are added to rows with fewer than k non-zeros, so k cannot be large, otherwise it will

[1]Devices with Compute Capability lower than 1.2 have stricter requirements.

introduce large overhead to access these zeros. The ELL format cannot be directly applied to graph mining algorithms, where the node degree in the graph cannot be bounded by a small number k. However, ELL and COO format can be mixed together to represent a matrix, where the first k non-zeros of each row are stored in ELL format and the others are stored in COO format. This is the *hybrid* (HYB) kernel of NVIDIA's SpMV Library [3, 4]. The *diagonal* (DIA) format is only applicable to matrices in which all non-zeros fall into a band around the diagonal. The *packet* (PKT) format first uses Metis [14] to cluster non-zeros into dense sub-blocks, then a sub-block is loaded into shared memory and processed by a thread block as a dense sub-matrix. Choi *et al* [9] propose *blocked ell-pack* format in which the non-zeros are stored in fixed size blocks first and the blocks are indexed with similar method in ELL format. Blocking techniques [27, 28] will gain locality when accessing vector x and reduce loop overhead when computing matrix indices; but it will also introduce memory overhead if the small blocks cannot be filled with enough non-zero elements.

Because of the importance of the SpMV kernel, researchers have put substantial efforts on optimization techniques over various architectures and platforms. Vuduc *et al* [31, 30, 16] study optimizations and performance auto-tuning in single core CPUs over the Sparsity framework [12]; Nishtala [22] provides detailed research about how blocking can benefit SpMV kernel over CPU. Williams [32] compares SpMV kernels on emerging multicore platforms, including multicore CPUs and the Cell Broadband platform. Blelloch [5, 6] studies sparse matrix computation on vector machines. Sengupta [29] develops efficient segmented scan primitives for GPUs, which can be used in the reduction step of COO kernel. NVIDIA's SpMV library implements a more efficient segmented reduction than segmented scan.

A large class of graph mining algorithms leverage the SpMV kernel iteratively to perform computation until the algorithms converge, e.g. PageRank [7, 25], HITS [15] and Random Walk with Restart [26]. These algorithms first transform the adjacency matrix of a graph and then operate on the transformed matrix. The graph dataset used by these algorithms usually have strong power-law properties, hence the number of non-zeros on each row or column of the corresponding matrix will follow a power-law distribution. The skewness of the distribution leads to poor load balancing and low memory access efficiency in GPU. Recent work by Kang et. al. [13] employs the MapReduce [11] framework to implement iterative SpMV kernel based on Hadoop, an open source version of MapReduce, and performs large-scale graph mining over this platform.

3. METHODS

In this section we present our optimization techniques for SpMV kernel on matrices representing power-law graphs. Our optimizations are based on a serie of observations from benchmarking results which reveal the limitations of previous work on matrices representing power-law graphs. We propose corresponding solutions which target these limitations and improve the performance.

The SpMV kernel is a bandwidth limited problem since the floating point operations per memory access is low. When computing a vector y as the product of a sparse matrix A and a vector x, the global memory accesses to matrix A has been optimized to be fully coalesced in NVIDIA's SpMV library. But the accesses to vector x have never been addressed in previous work. Also the only reuse possibility in SpMV problem comes from the reuse of vector x.

Observation 1: Each row accesses random elements in vector x.

In the adjacency matrix of a power-law graph, the column indices of the non-zeros on each row are not continuous, and are relatively random. This leads to non-coalesced memory addresses when accessing x. Previous work [3, 4, 2] binds the entire vector x to the texture memory and utilize the cache of texture memory to improve the locality when accessing x. But the dimension of matrix A is much larger than the size of the texture cache. When a cache miss happens, the pre-fetched cache content will be kicked out of the cache. A cahce miss will reduce the memory bandwidth utilization due to the long latency of non-coalesced global memory accesses.

Solution 1: Tiling matrix A and vector x with texture cache.

Tiling is a cache-based optimization for matrix and vector multiplications [12]. Suppose we divide matrix A into fixed width tiles by column index and segment vector x correspondingly, so that each tile of A only needs to access one segment of x. If one segment of x can fit in the texture cache, once the elements are fetched into cache, none of them will be kicked out until the computation of this tile finishes. Therefore, we can get maximum reuse of x.

The texture cache size is a key factor in determining the width of a tile. We conduct a benchmarking experiment to estimate the texture cache size (since this is not provided to us by the manufacturer) on our Tesla C1060 GPU. We use a large sparse matrix and multiply it with a vector. The column indices are *mod*-ed by tile width, so all accesses to vector x are mapped into one tile. We change the tile width from $100,000$ to 1000 and run this multiplication with NVIDIA's SpMV library. The performance improves most significantly when tile width $= 64,000$, corresponding to 250 KB of cache size. So our tile width is fixed to $64,000$ columns.

The performance of tiling the entire matrix A and vector x is still low. The reason is if we divide all the columns of matrix A into tiles, there could be too many tiles when the matrix is large. Each tile needs to add its partial result to the final result y. Rows in each tile may become empty after tiling partition, this leads to non-coalesced memory access to y because we only need to write the result of non-empty rows back. To make things worse, the write-back result of one tile has to be visible to the next tile before it can start, otherwise memory read-after-write conflicts could happen. To avoid memory conflicts, we restart a CUDA kernel for each tile. This also causes an overhead.

Observation 2: Column lengths are power-law distribution.

Based on this observation, we want to tile the matrix more efficiently and effectively. Suppose a matrix is the adjacency matrix of a power-law graph, the number of non-zeros in columns of the matrix will follow a power-law distribution. So there are large number of columns with few non-zeros in them. In tiles containing such columns, we cannot get much reuse of vector x, but we still need to restart large number of kernels to compute them and afford the overheads.

Solution 2: Reorder columns by column lengths and partially tile A.

Our idea is to first reorder the matrix columns by decreasing order of number of non-zeros in each column. We can divide the reordered matrix into two sub-matrices by setting a threshold of column length. One denser sub-matrix formed by long columns contains more non-zero elements and fewer columns; the other sparser sub-matrix formed by short columns contains fewer non-zero elements and more columns. According to Amdahl's law [1], the overall performance of the SpMV kernel will be improved if the computation in the denser matrix can be finished efficiently. Now we can tile the denser sub-matrix with texture cache. The non-zero elements are concentrated in small number of tiles so that we can still gain the benefits from x vector caching as well as avoid the overhead of initializing too many tiles.

Figure 2 illustrates the above transformation procedure on a small sparse matrix. Figure 2(a) is the original matrix; Figure 2(b) reorders the columns of the matrix in decreasing order of column length. In this example, we set the column length threshold to 2. Columns with more than or equal to 2 non-zero elements will be placed in the denser sub-matrix; the other columns with only 1 non-zero elements will be placed together in the sparser sub-matrix. Suppose the texture cache can only hold 2 floating point numbers in this small example, the denser sub-matrix with 4 columns will be partitioned into 2 tiles as shown in Figure 2(c).

Amongst all the kernels in NVIDIA's SpMV library, HYB and COO perform best on matrix with power-law property. The computation in the sparser matrix is run under the HYB kernel in NVIDIA's SpMV library, because HYB achieves best performance. The computation within each tile of the denser matrix will be performed using NVIDIA's COO kernel. The resulting vector y from the denser and sparser sub-matrix will be combined to compute the final result.

Observation 3: Within each tile, performance of COO kernel is limited by thread divergence and serialization.

When computing each tile, the COO kernel cannot utilize the massive thread level parallelism in CUDA efficiently although it is more efficient than the CSR-vector and ELL kernel on such data. In the COO kernel, the inputs are three arrays storing the row indices, column indices and values of non-zero elements in the matrix. These three arrays are all divided into equal length intervals. Each interval is assigned to one warp. Note that this partition only equally distributes workload to warps, it does not consider that a row may cross the boundary between two warps. A warp of threads iterate over an interval in a strided fashion. Here stride size equals warp size, a thread within a warp only works on one non-zero element in one stride. A thread first fetches the value in the x vector based on the column index, and then multiplies the x value with the non-zero element in matrix A and stores the result in a shared memory space reserved for it. The next step is the *sum reduction* of the multiplication results within one stride. A binary reduction operation is performed within a thread warp. But one stride can contain non-zeros from more than one rows. When the reduction operation tries to add two operands, it has to first check whether the two operands are from the same row in the original matrix. If not, this warp of threads will be serialized due to thread divergence. This leads to low thread level parallelism in the COO kernel.

Observation 4: Within each tile, performances of CSR-verctor and ELL kernel are limited by imbalanced workload.

The CSR-vector kernel performs best when the rows of a matrix are long and with similar length. Non-zeros are stored in row major in CSR format. CSR-vector kernel assigns one thread warp on one row. The thread warp iterates on the row with stride size the same as warp size, and performs multiplication and summation operations. After the last iteration on this row, the threads in a warp perform a binary reduction to obtain the final result of this row. In all the summation and reduction operations in CSR-vector kernel, the threads within a warp do not need to check whether two operands are from the same row. However, CSR-vector kernel is the most efficient only when rows have more than a warp of non-zeros and the number of non-zeros is an integer multiple of warp size.

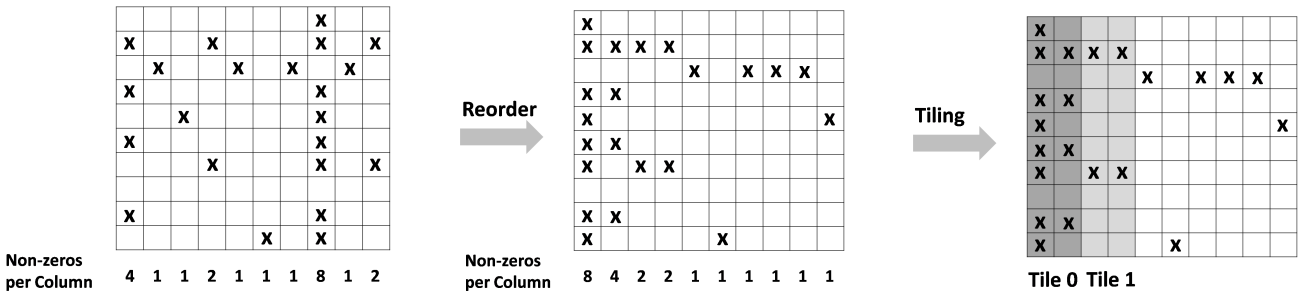
The ELL kernel achieves best performance if there are large number of rows in the sparse matrix and they have similar length. In the ELL format, all rows have the same length and 0 are padded to rows shorter than this length. The non-zeros are stored in column major. A warp of threads are assigned to work on 32 consecutive rows, each thread works on the multiplication and reduction of one row. The threads within one warp iterate over the columns efficiently with hardware synchronization.

Observation 5: Row lengths in each tile follow power-law.

Due to the scale-free property of power-law graphs, we observe that after tiling, the row length within a tile also follows a power-law distribution. We propose to leverage this fact via a novel storage format of matrix A within one tile to further improve the efficiency of SpMV kernel.

Solution 3: Composite tile storage scheme.

Our composite row and column storage scheme combines the CSR and ELL packing scheme as follows. Our algorithm starts with the ranking of the row length from high to low. A workload size is defined as the total number of non-zeros in the longest row or several long rows at the top of the ranking, depending on the dataset. Then rows in a tile will be partitioned into approximately balanced workload. This can be implemented by traversing the row length ranking



(a) The original sparse matrix and column length. X – non-zero element.

(b) Reorder columns by decreasing order of column length

(c) Tiling the first 4 columns.

Figure 2: Illustrative example of tiling

from top to bottom. A new row is packed into a workload until it exceeds the workload size, then a new workload is initialized. Each workload can be viewed as a rectangle area in the tile, where the width w is defined as the length of the first row (the longest row in this workload) in the rectangle and the height h is defined as the number of rows in this workload. If $w \geq h$, this workload will be stored in row major in global memory and computed by CSR-vector kernel; otherwise, it will be stored in column major and computed by ELL kernel. Note that if a workload is stored in row major, all rows will be padded to the same length as w with 0; and 0 will also be padded to ensure that w (or h) is an integer multiple of warp size when a workload is stored in row (or column) major. After the above partition and transformation of storage format, each workload is assigned to a warp of threads and computed with the most suitable kernel.

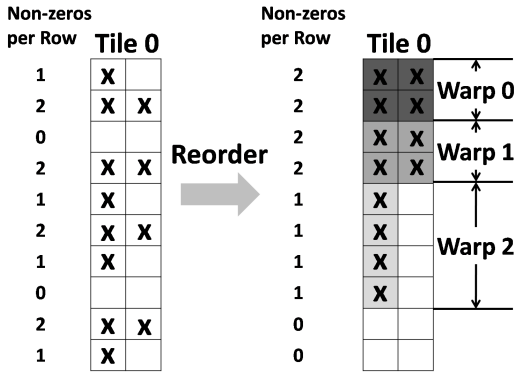


Figure 3: Composite storage of one matrix tile

Figure 3 illustrates how tile 0 from Figure 2(c) is transformed in our composite storage scheme on a fictitious architecture with two threads per warp. The rows in tile 0 are first reordered by row length. Suppose we set the workload size to be 4. The first two rows are packed into the first workload, stored in row major and assigned to warp 0 for computation. The two threads in warp 0 first do multiplication and reduction on row 0 using CSR-vector kernel and move to row 1 together. The next two rows are packed together, stored and computed in the same way by warp 1. The remaining four single element rows are stored in column major and computed by warp 2. The two threads in warp 2 start from the first two rows vertically using ELL kernel and then move to the last two rows.

Sorting Cost: In our tiling and composite storage optimizations, sorting is used to re-construct the original matrix to improve memory access locality and kernel efficiency. The cost of sorting columns, and the cost of sorting rows within a tile is relatively cheap when the corresponding distributions follow a power law. Because of the power law distribution, large number of the row and column lengths can be bounded by some small number k. They correspond to the long tail of the power law distribution. These rows or columns can be sorted by counting sort in linear time [10]. The number of remaining rows or columns is very small. They can be sorted very quickly. We only need to perform sorting once as data preprocessing. In power method where the SpMV kernel is called iteratively until the result converges, the cost of sorting can be amortized by the iterations.

4. EXPERIMENTS

In this section we begin by describing the datasets used in our evaluation and the configuration of our hardware platform.

Datasets: In our experiments, we use four web-based graph datasets.

The four graph datasets are user link relationship graphs from Flickr, LiveJournal and Youtube and a webpage link relationship graph from Wikipedia [19, 18]. All graphs exhibit power law properties. In addition to the graph datasets we also include results on six popular unstructured matrix data, representing various scientific kernels, used in previous studies [3, 4]. Among these, one is a 2000 by 2000 dense matrix, which while not sparse, is a useful benchmark to show the maximum bandwidth that each kernel can achieve. Details of these graphs (represented in an adjacency matrix) and matrices are shown in Table 1. In the four graph datasets, the number of non-zeros (NNZ) is the number of directed links and the number of rows (or columns) is the number of nodes in the graphs.

Hardware configuration: All CPU results are reported on a single processor of a dual core Opteron X2 2218 system running at 2.6 GHz and with 8 GB of 667 MHz DDR2 main memory. The machine is also equipped with two NVIDIA Tesla C1060 GPUs. Each GPU has 30 multiprocessors with

Matrix	Rows	Columns	NNZ	NNZ/Row	Power-law?
Dense	2,000	2,000	4,000,000	2000.0	No
Circuit	170,998	170,998	958,936	5.6	No
FEM/Harbor	46,835	46,835	2,374,001	50.6	No
LP	4,284	1,092,610	11,279,748	2632.9	No
Protein	36,417	36,417	4,334,765	119.3	No
Webbase	1,000,005	1,000,005	3,105,536	3.1	No
Flickr	1,715,255	1,715,255	22,613,981	13.2	Yes
LiveJournal	5,204,176	5,204,176	77,402,652	14.9	Yes
Wikipedia	1,870,709	1,870,709	39,953,145	21.4	Yes
Youtube	1,138,499	1,138,499	4,945,382	4.3	Yes

Table 1: Matrix and Graph Datasets

240 processing cores and 4 GB of global memory. All GPU experiments are reported on a single GPU. The host code is complied with the gcc compiler version 4.1.2. The device code is compiled with CUDA version 2.3.

4.1 Sparse Matrix and Vector Multiplication kernel

In this section, we compare a CPU-based implementation of the CSR kernel, all six kernels from NVIDIA's SpMV library, Baskaran and Bordawekar's optimized CSR kernel(BSK & BDW CSR) and our two optimized kernels (TILE-COO and TILE-COMPOSITE) on the matrix datasets in Table 1. We report the speed of execution in GFLOPS determined by dividing the number of arithmetic operations, which is twice the number of non-zeros in the matrix, by the running time. The running time is averaged over 500 iterations. Since the SpMV kernel is a bandwidth limited problem, we also report the *effective* bandwidth utilization of each kernel in GB/s, which is the total number of bytes read and written by the kernel divided by the running time. Note that different storage formats have their own auxiliary indices and data structures. These data structures are counted into the effective memory accesses. All kernels are run in single precision (32-bit). Binding the entire vector x to texture cache performs consistently better than not binding in all NVIDIA's SpMV kernels [3, 4] and Baskaran and Bordawekar's CSR kernel [2]. So we only report the performance of these kernels with texture cache binding. We use 256 threads per thread block. This setting is default in NVIDIA's SpMV library. Under this setting, there are enough number of warps in each thread block to hide the memory latency and the multiprocessors can be fully utilized by thread blocks. In our tiling method, we had to decide a threshold to decide how many columns are placed in the denser sub-matrix to be tiled (the width of each tile is predetermined by the size of the texture cache and corresponded to 64K columns). We typically found for the larger graphs 9 or 10 tiles were beneficial while for the smaller unstructured matrix and for the Youtube dataset 1 or 2 tiles were optimal.

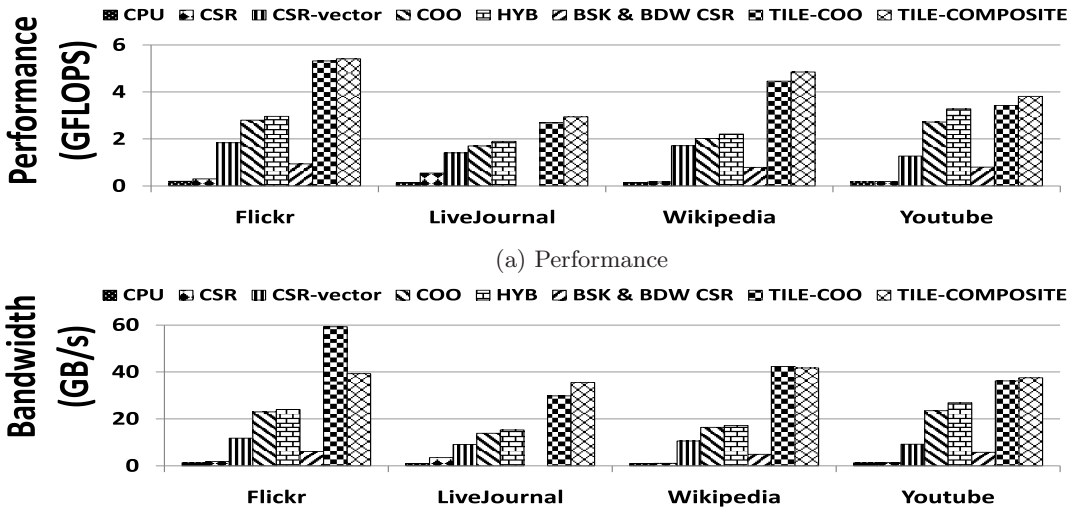
The performance of the SpMV kernels on matrices representing power-law graphs are shown in Figure 4. The results on the other matrices are presented in Figure 5.

Performance on power-law matrices: We do not report performance on the PKT kernel on these datasets since the partition step within this kernel does not produce balanced enough *packets* and leads to kernel failure. Our tiling and tiling with composite storage methods clearly dominate the other kernels on the Flickr, LiveJournal, Wikipedia datasets. Our tiling with composite kernel has an average 1.75x speedup over NVIDIA's best kernel – the HYB kernel on these datasets. On the Youtube dataset, the smallest of our graph datasets, NVIDIA's COO and HYB kernel perform close to our optimizations, tiling with composite storage runs marginally (4.5%) faster than HYB kernel. From Table 1, we can see the numbers of rows and columns are low in the Youtube matrices, and also the numbers of non-zeros per row and column are low. These properties of the Youtube matrix hide the advantages of our optimizations for the following reasons. First, there is little reuse of vector x if non-zeros per column is low. This leads to low benefit from our tiling optimization. Second, when the number of columns is small, COO and HYB kernel have better probability of cache hits when they bind the entire vector x to texture cache. Third, the total number of non-zeros in a tile is low so our composite storage scheme will pad more zeros and cause memory access overhead.

Performance on Unstructured Matrix Data: The PKT kernel for similar reasons noted above cannot run over the LP and Webbase matrices, so that performance number is not included. The speed and bandwidth performance of different kernels on these datasets are shown in Figure 5. We immediately observe that our methods while comparing favorably on some of the kernels do not always perform as strongly as the best. In fact on these datasets, interestingly, there is no single kernel outperforms all others.

Our tiling with composite storage kernel performs the best on the 2000 by 2000 dense matrix with 17.57 GFLOPS speed and 105.5 GB/s bandwidth. This bandwidth utilization is higher than the peak bandwidth of 102 GB/s in the official hardware specification from NVIDIA website. This somewhat surprising result is due to the effect of texture binding of vector x allowing for elements in x to be directly fetched from the cache. Our tiling with composite storage kernel runs 30% faster than CSR-vector kernel on the dense matrix. This is because we pad the storage of the matrix in global memory to ensure that all global memory accesses are fully coalesced. The CSR-vector format concatenates all rows together. If one row is not padded to an integer multiple of the warp size, all global memory accesses after this row will not be fully coalesced resulting in a loss in performance.

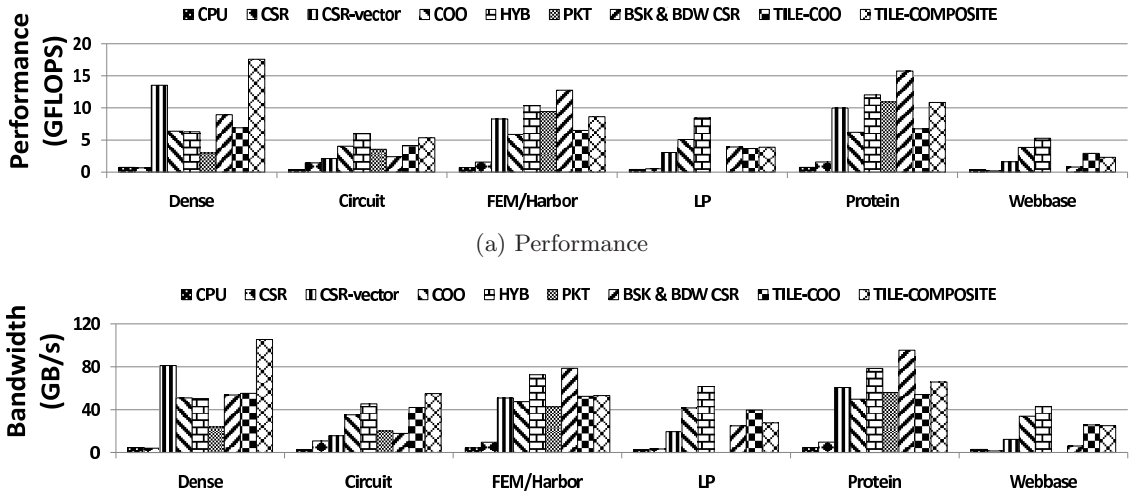
Comparison with CPU SpMV: We also implemented the SpMV kernel with CSR format on the CPU. CSR format is the most efficient on CPU among different sparse matrix formats. We ran experiments with the CPU kernel



(a) Performance

(b) Bandwidth

Figure 4: SpMV kernels comparison on matrices representing power-law graphs.



(a) Performance

(b) Bandwidth

Figure 5: SpMV kernels comparison on unstructured matrices from NVIDIA's SpMV Library [3, 4].

on all datasets in Table 1. The GPU kernels significantly outperform the CPU kernel in almost all settings. GPU CSR kernel is the slowest kernel on GPU. It is slower than CPU kernel on the Webbase dataset due to poor load balancing and in the Dense matrix data because the clock rate of one GPU processor is lower than CPU. The GPU kernels perform dominantly faster than CPU kernel in all the other formats with speedups ranging from 2.05x to 37.31x.

We also compared the performance of SpMV kernels from Bell and Garland [3, 4], Baskaran and Bordawekar [2] and our methods on all matrices used in their paper and our matrices representing power law graphs. The details of these datasets and the performance numbers are presented in the Appendix.

4.2 Data Mining Applications on Graph Datasets

In this section, we describe the three data mining algorithms which can be written in the form of matrix-vector multiplication. These algorithms essentially compute the power method for different matrices related to the link structure of the graphs. Within one iteration of the power method, the running time is dominated by the time required to compute the matrix-vector product. These algorithms usually operate on large power-law graphs. Hence they can be sped up using our sparse matrix representation and computed by our SpMV kernels.

In the following sections, we show how we can use SpMV kernel as a key subroutine to implement three important graph mining algorithms: PageRank, Random Walk with Restart (RWR) and HITS. We implement these algorithms using 4 GPU SpMV kernels: COO, HYB, TILE-COO, and TILE-Composite kernels. These 4 kernels are generally the top four kernels from the experimental results in previous section. At the end of each iteration in PageRank, RWR and HITS, a convergence criterion needs to be checked. CUDA SDK [24] provides a convenient parallel reduction primitive to perform this task. We choose the fastest parallel reduction kernel from CUDA SDK in our implementation of all the algorithms.

PageRank: The PageRank algorithm models the link structure of web graphs by the random walk behavior of a *random surfer* [7, 25]. The web graph can be represented by a directed graph $G = (V, E)$, where V is a set of n vertices and E is the set of directed edges. The adjacency matrix A is defined as $A(u, v) = 1$ if edge $(u, v) \in E$; otherwise, $A(u, v) = 0$. Matrix W denotes the row normalized matrix of A. The PageRank vector p is computed iterative using the following equation until it converges:

$$p = (cW^T + (1 - c)U)p \quad (1)$$

where c is a damping factor (set to 0.85 in our experiment), U is a n by n matrix with all elements set to $1/|V|$. We preprocess each graph dataset in Table 1 to a matrix $M = cW^T + (1 - c)U$, and initialize the PageRank vector p with all elements equal to $1/|V|$. We run M times p iteratively with the corresponding SpMV kernel and check whether p converges at the end of each iteration. The speed and bandwidth performance of PageRank implementations based on the four kernels are shown in Figure 6(a) and Figure 7(a). The total running time on each graph is shown in Table 2

Graph	CPU	COO	HYB	TILE-COO	TILE-Comp
Flickr	239.94	16.65	15.99	9.05	8.30
LiveJournal	822.89	61.94	55.69	37.46	34.42
Wikipedia	5211.91	299.86	283.40	176.08	163.41
Youtube	11.81	0.72	0.66	0.68	0.65

Table 2: Total running time of PageRank (in seconds)

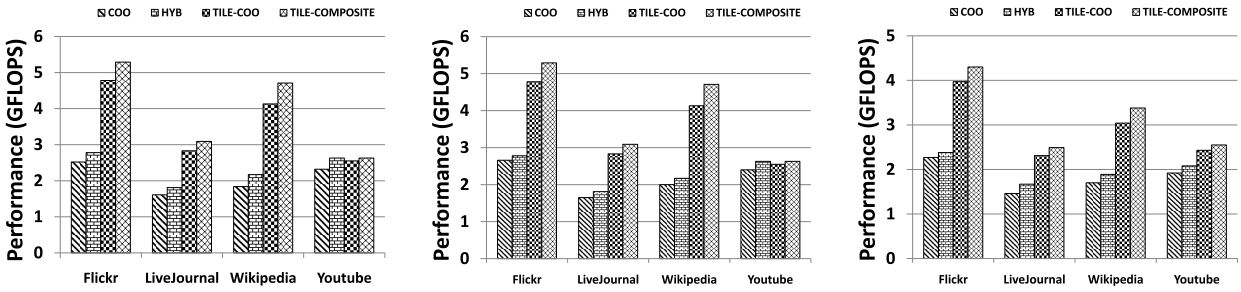
in comparison to a corresponding CPU implementation of PageRank. Our optimized TILE-COO and TILE-Composite kernel achieves about 2x speedup over COO and HYB kernel on Flickr, LiveJournal and Wikipedia graphs. The four kernels perform roughly the same on Youtube graph (reasons noted earlier). Compared with the CPU PageRank, all GPU implementations achieve between 13x and 31x speedup. The performance improvement comes from two parts: the fast SpMV kernel and the fast reduction operation for checking convergence.

Random Walk with Restart: Random Walk with Restart (RWR) is a algorithm that tries to measure the relevance between two nodes in a undirected graph [26]. Given a query node i in the graph, the relevance score from all other nodes to node i forms a vector $\overrightarrow{r_i}$. In RWR, vector $\overrightarrow{r_i}$ is computed by the following equation:

$$\overrightarrow{r_i} = cW\overrightarrow{r_i} + (1 - c)\overrightarrow{e_i} \quad (2)$$

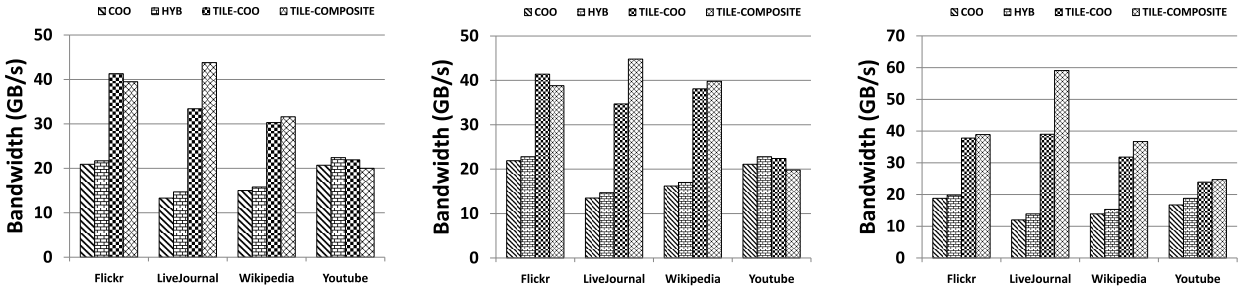
where c is a restart probability parameter (set to 0.9 in our experiment), W is the column normalized adjacency matrix and $\overrightarrow{e_i}$ is a vector whose i^{th} element is 1 and all the other elements are 0. Vector $\overrightarrow{r_i}$ can be computed using the power method. In each iteration, there is a matrix-vector multiplication followed by vector addition and convergence checking operations. In our implementation, we use the GPU SpMV kernels for matrix-vector multiplication, and GPU parallel reduction for checking convergence in the same way as PageRank. An efficient vector addition kernel is also implemented by assigning one GPU thread to compute one element in the resulting vector. Note that RWR is an interactive application, we randomly select 25 query nodes in our experiment and the performance is reported by averaging (arithmetic mean) the result of each query. Since RWR operates on undirected graphs, we treat each link in our directed graph datasets as an undirected link in our experiments. The speed and bandwidth performance of RWR implementations on four graph datasets based on four GPU SpMV kernels are shown in Figure 6(b) and Figure 7(b). The total running time is listed in Table 3. We observe similar performance results as in the case of PageRank. Our optimized TILE-COO and Tile-Composite kernels are 1.5x to 2.0x as fast as COO and HYB kernels on Flickr, LiveJournal and Wikipedia graphs. The four kernels perform about the same on Youtube graph. All GPU implementations are 13x to 37x faster than CPU implementation. The best speedup is achieved by our TILE-Composite kernel on Wikipedia graph.

HITS: HITS is a link analysis algorithm of web pages [15]. It gives each web page two attributes: authority and hub. It rates web pages by assigning authority score and hub score to each web page. Let matrix A be the adjacency matrix of a directed graph $G = (V, E)$ or G may be a query specific



(a) PageRank (b) Random Walk with Restart (c) HITS

Figure 6: Performance of Data Mining kernels on graph datasets.



(a) PageRank (b) Random Walk with Restart (c) HITS

Figure 7: Bandwidth of Data Mining kernels on graph datasets.

subgraph of the whole web graph. Then the authority score vector $\overrightarrow{a}$ and hub score vector $\overrightarrow{h}$ are recursively defined as

$$\overrightarrow{a} = A^T \overrightarrow{h} \qquad \overrightarrow{h} = A \overrightarrow{a} \tag{3}$$

This recursive definition with two matrix-vector products can be rewritten as one matrix and vector multiplication by

$$\begin{bmatrix} \overrightarrow{a} \\ \overrightarrow{h} \end{bmatrix} = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \overrightarrow{a} \\ \overrightarrow{h} \end{bmatrix} \tag{4}$$

The power method can be used to solve this eigen vector problem. Elements in $\overrightarrow{a}$ and $\overrightarrow{h}$ vectors are all initialized to $1/|V|$. In each iteration, a $2|V|$ by $2|V|$ matrix in equation 4 is multiplied by a vector combined with $\overrightarrow{a}$ and $\overrightarrow{h}$. Then the first and second half of the resulting vector are normalized to sum to 1 separately. Each normalization requires a reduction operation on the vector and a division of the vector by a constant. A convergence check is also needed at the end of each iteration. Each iteration of our HITS implementation involves one SpMV kernel, three parallel reduction kernels (two for normalization and one for convergence check) and two vector division by constant kernels. The vector division by constant kernel can be implemented very efficiently in the same way as vector addition. On our implementation of the HITS algorithm we compare the performance of our four GPU SpMV kernels on the four graph datasets. The speed and bandwidth performance are shown in Figure 6(c) and Figure 7(c). Our TILE-COO and TILE-Composite kernels perform better than COO and HYB kernels in all four datasets. On Flickr, LiveJournal and Wikipedia, the speedups are similar to those observed in PageRank and RWR algorithms. On Youtube, our optimizations are actually a bit faster when compared to the NVIDIA kernels inspite of the relatively small size of the dataset. Combining the two matrices into one in the HITS algorithm results in a larger and sparser matrix making it more amenable to our optimizations. The total running time compared with CPU implementation is listed in Table 4. We observe a 11x to 28x speedup of the GPU implementations over the corresponding CPU implementation.

5. CONCLUSIONS

In this paper, we proposed architecture conscious optimizations for the sparse matrix-vector multiply kernel on GPUs and studied the implications of this effort for graph mining algorithms. Our optimizations take into account both the architecture features of GPUs and the characteristic of graph mining applications. Our tiling approach utilizes the texture cache on GPUs in a more efficient way than previous work and provides much better memory locality. Our tiling with composite representation leverages the power-law characteristics of large graphs in graph mining problems. We have obtained significant performance improvement over the state-of-the-art on such graph based datasets. We also present empirical evaluations of applying our optimizations to PageRank, Random Walk with Restart and HITS algorithms.

We plan to propose auto-tuning method to provide optimal performance based on the distribution of non-zeros in the dataset. We also intend to leverage blocking and loop unrolling techniques to improve the performance further. Our approaches can also be extended to distributed systems where GPUs are installed on multiple machines.

Graph	CPU	COO	HYB	TILE-COO	TILE-Comp
Flickr	8.25	0.59	0.56	0.33	0.29
LiveJournal	36.99	2.85	2.60	1.73	1.52
Wikipedia	23.23	1.46	1.35	0.71	0.62
Youtube	2.32	0.14	0.13	0.14	0.13

Table 3: Average running time of Random Walk with Restart (in seconds) on 25 random query nodes

Graph	CPU	COO	HYB	TILE-COO	TILE-Comp
Flickr	4.97	0.40	0.38	0.23	0.21
LiveJournal	44.88	3.82	3.33	2.41	2.24
Wikipedia	39.36	2.73	2.45	1.52	1.37
Youtube	4.35	0.33	0.30	0.26	0.25

Table 4: Total running time of HITS (in seconds)

6. REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)*, pages 483–485, 1967.

[2] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. Technical report, IBM RC24704, 2008.

[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[5] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *PPOPP '93*, pages 102–111, 1993.

[6] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical report, Pittsburgh, PA, USA, 1993.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[8] L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009.

[9] J. W. Choi, A. Singh, and R. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPOPP*, pages 37–48, 2010.

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, pages 10–10, 2004.

[12] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[13] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM '09*, pages 229–238, 2009.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[15] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[16] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP '04*, pages 169–176, 2004.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[18] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. In *WOSN'08*, August 2008.

[19] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC'07*, October 2007.

[20] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, December 2005.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[22] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18(3):297–311, 2007.

[23] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[24] NVIDIA. *NVIDIA CUDA SDK Code Samples*. 2008.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999.

[26] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. In *KDD '04*, pages 653–658, 2004.

[27] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[28] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[29] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH '07*, pages 97–106, 2007.

[30] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing '02*, pages 1–35.

[31] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, 2003.

[32] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35(3):178–194, 2009.

APPENDIX

Matrix	Rows	Columns	NNZ	NNZ/Row	Power-law?
Dense	2,000	2,000	4,000,000	2000.0	No
Circuit	170,998	170,998	958,936	5.6	No
Economics	206,500	206,500	2,633,278	12.8	No
Epidemiology	525,825	525,825	3,674,625	7.0	No
FEM/Accelerator	99,843	99,843	2,624,331	26.3	No
FEM/Cantilever	62,451	62,451	4,007,383	64.2	No
FEM/Harbor	46,835	46,835	2,374,001	50.6	No
FEM/Ship	140,874	140,874	7,813,404	55.5	No
FEM/Spheres	83,334	83,334	6,010,480	72.1	No
LP	4,284	1,092,610	11,279,748	2632.9	No
Protein	36,417	36,417	4,334,765	119.3	No
QCD	49,152	49,152	2,113,536	43.0	No
Webbase	1,000,005	1,000,005	3,105,536	3.1	No
Wind Tunnel	217,918	217,918	11,634,424	53.4	No
Flickr	1,715,255	1,715,255	22,613,981	13.2	Yes
LiveJournal	5,204,176	5,204,176	77,402,652	14.9	Yes
Wikipedia	1,870,709	1,870,709	39,953,145	21.4	Yes
Youtube	1,138,499	1,138,499	4,945,382	4.3	Yes

Table 5: Full Datasets

Matrix	CPU	CSR	CSR-vec	COO	HYB	PKT	BSK&BDW	TILE-COO	TILE-COMP
Dense	0.74	0.67	13.52	6.37	6.28	2.99	8.95	6.9	17.57
Circuit	0.35	1.45	2.12	4.04	5.98	3.55	2.40	4.14	5.35
Economics	0.51	1.74	4.68	4.07	7.75	8.7	5.83	4.49	3.08
Epidemiology	0.62	2.76	2.68	4.15	16.1	10.13	3.57	5.31	2.24
FEM/Accelerator	0.47	1.58	5.51	5.11	8.61	10.46	8.02	3.29	4.58
FEM/Cantilever	0.73	1.73	9.12	6.16	13.46	9.66	14.6	6.73	7.53
FEM/Harbor	0.71	1.56	8.27	5.88	10.34	9.42	12.76	6.49	8.63
FEM/Ship	0.71	1.74	9.15	6.04	15.27	13.31	14.11	6.11	5.74
FEM/Spheres	0.72	1.79	9.21	6.13	18.01	10.88	14.32	5.42	8.20
LP	0.41	0.52	3.06	5.09	8.45	n/a	3.91	3.68	3.89
Protein	0.75	1.59	10.00	6.20	12.04	10.93	15.74	6.74	10.85
QCD	0.69	1.90	8.13	6.01	18.60	8.88	13.06	6.57	5.85
Webbase	0.39	0.19	1.62	3.85	5.27	n/a	0.80	2.93	2.31
Wind Tunnel	0.58	1.90	8.69	5.88	18.35	15.03	14.39	6.21	6.89
Flickr	0.20	0.30	1.84	2.80	2.96	n/a	0.94	5.32	5.41
LiveJournal	0.13	0.55	1.41	1.70	1.90	n/a	n/a	2.70	2.94
Wikipedia	0.13	0.17	1.71	2.02	2.21	n/a	0.79	4.46	4.85
Youtube	0.17	0.19	1.27	2.72	3.28	n/a	0.80	3.43	3.81

Table 6: Performance results on full datasets (in GFLOPS)

Matrix	CPU	CSR	CSR-vec	COO	HYB	PKT	BSK&BDW	TILE-COO	TILE-COMP
Dense	4.5	4.0	81.2	51.0	50.2	23.9	53.7	55.2	105.5
Circuit	2.6	10.8	15.7	35.2	45.4	20.1	17.8	42.0	55.0
Economics	3.4	11.5	31.0	33.8	56.2	46.4	38.7	41.5	24.3
Epidemiology	4.4	19.7	19.2	35.5	105.8	55.0	25.5	51.6	21.1
FEM/Accelerator	2.9	10.0	34.8	41.7	57.1	53.6	50.5	27.3	30.3
FEM/Cantilever	4.5	10.6	55.8	49.6	86.5	48.0	89.3	54.2	46.1
FEM/Harbor	4.4	9.6	50.9	47.5	72.6	42.7	78.6	52.4	53.2
FEM/Ship	4.4	10.7	56.2	48.7	96.3	65.7	86.7	50.2	36.9
FEM/Spheres	4.4	10.9	56.3	49.4	113.5	60.0	87.5	44.0	51.0
LP	2.6	3.3	19.5	41.6	61.6	n/a	24.9	39.6	27.7
Protein	4.6	9.7	60.7	49.8	78.5	56.0	95.5	54.1	65.8
QCD	4.3	11.7	50.3	48.6	113.4	60.6	80.8	53.2	36.2
Webbase	2.9	1.5	12.3	33.9	43.0	n/a	6.1	25.8	25.0
Wind Tunnel	3.5	11.7	55.1	47.5	112.3	71.9	88.5	51.5	45.4
Flickr	1.3	1.9	11.8	23.0	24.0	n/a	6.1	59.3	39.3
LiveJournal	0.8	3.5	9.1	13.9	15.3	n/a	n/a	29.8	35.5
Wikipedia	0.8	1.1	10.6	16.4	17.2	n/a	4.9	42.3	41.7
Youtube	1.3	1.4	9.2	23.5	26.9	n/a	5.8	36.2	37.5

Table 7: Bandwidth utilization on full datasets (in GB/s)