2010 39th International Conference on Parallel Processing

Efficient PageRank and SpMV Computation on AMD GPUs

Tianji WU*, Bo WANG*, Yi SHAN*, Feng YAN†, Yu WANG* and Ningyi XU‡
*Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Email: {wutj06,wangb06,shany08}@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn
†Department of Computer Science, Purdue University, Email: yan12@purdue.edu
‡Hardware Computing Group, Microsoft Research Asia, Email: xu.ningyi@microsoft.com

***Abstract*—Google's famous PageRank algorithm is widely used to determine the importance of web pages in search engines. Given the large number of web pages on the World Wide Web, efficient computation of PageRank becomes a challenging problem. We accelerated the power method for computing PageRank on AMD GPUs. The core component of the power method is the Sparse Matrix-Vector Multiplication (SpMV). Its performance is largely determined by the characteristics of the sparse matrix, such as sparseness and distribution of non-zero values. Based on careful analysis on the web linkage matrices, we design a fast and scalable SpMV routine with three passes, using a modified Compressed Sparse Row format. Our PageRank computation achieves 15x speedup on a Radeon 5870 Graphic Card compared with a PhenomII 965 CPU at 3.4GHz. Our method can easily adapt to large scale data sets. We also compare the performance of the same method on the OpenCL platform with our low-level implementation.**

***Keywords*-PageRank; SpMV; GPU; OpenCL**

I. Introduction

Nowadays, search engines play a central role in extracting useful information from the World Wide Web. When a search engine returns a list of relevant web pages for a given query, it is crucial to determine the qualities of these pages. The PageRank algorithm is an effective algorithm to address this problem. It analyzes the linkage information between web pages to obtain the PageRank values of web pages, which indicate the page importance.

The current Internet has hundreds of billions of web pages. Efficient PageRank computation for all pages is a challenging problem. Mathematically, the PageRank computation is a numerical linear algebra problem that computes the eigenvector corresponding to the largest eigenvalue of the linkage matrix. Many methods have been proposed in the past decades. The most widely used method is the power method [1], which iteratively performs sparse matrix-vector multiplication (SpMV), an important subroutine in numerical linear algebra.

In recent years, the graphic processing units (GPUs) have become popular for general purpose computing. GPUs are massively parallel devices with high computation performance and high power efficiency. A lot of research has been focused on accelerating SpMV with GPU previously. However, we find that general purpose SpMV routines do not perform well in PageRank.

In this paper, we analyze the characteristic of the sparse matrices used in PageRank, and introduce a fast SpMV implementation using a modified Compressed Sparse Row (CSR) format. The linkage matrices used in PageRank are always very sparse, with highly uneven row sizes (number of non-zero values in a row of a matrix). Based on the row sizes, we sort the rows into three bins, and assign different number of threads to process the rows in different bins. Thus improves load balance and hardware resource utilization.

Our SpMV subroutine is implemented on the AMD GPU architecture using AMD Compute Abstraction Layer (CAL) and Intermediate Language (IL) as well as the newly released OpenCL, which is a promising standard for heterogenous parallel computing. Our experiment of PageRank using a few datasets shows that our method brings up to 15x speedup compared to CPU and outperforms some other GPU methods. We also compare the performance between IL and OpenCL based implementations. The results show that the gap between OpenCL and IL varies with the algorithms. Kernels generated by OpenCL compiler is not quite efficient compared to hand-written IL kernels. Generally, data intensive algorithms have better performance with

This work was supported by National Science and Technology Major Project, 2010ZX01030-001-001-04, NSFC (No.60870001), 863 program of China (No. 2009AA01Z130), TNList Cross-discipline Foundation, and AMD/MSRA UR project.

0190-3918/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPP.2010.17

OpenCL, since memory access can cover the drawback of not-well-optimized ALU instructions. In the final part of this paper, we discuss the scalability of our method. Since the throughput (up to 17GB/s) exceeds the bandwidth of PCIe, the kernel execution time can not effectively cover the time to transfer data, if the whole dataset can not stay in the graphic memory. A solution is to add more graphic cards or computer nodes, and to partition the matrix. However, from the view of a single card, the problem is similar to that of a smaller data set. Therefore, our method is applicable.

The remainder of this paper is organized as follows. Section II gives a review on related works. Section III briefly introduces the power method of PageRank algorithm and SpMV computation. Section IV gives a brief introduction to the AMD RV870 GPU. The optimized implementation of PageRank is discussed in Section V. The experiment results are presented and discussed in Section VI. Section VII concludes this work.

II. Related Works

Previous research on PageRank acceleration focuses on parallel computer clusters and computing algorithms. How to minimize communication overhead and achieve faster convergence speed is usually the focus. For example, Zhu et al. [2] used iterative aggregation-disaggregation method to increase the convergence speed. Wicks and Greenwald utilized domain information to rearrange the linkage matrix, and reduced the number of sparse matrix-vector multiplication, resulting reduced computation workload and communication overhead [3]. Wang and DeWitt considered very large-scale PageRank computation [4]. They proposed programming framework to support PageRank computation on a large number of connected low-cost computers. To the best of our knowledge, there is no literature on GPU acceleration of PageRank computation.

The most time consuming part of PageRank is SpMV computation which has already been accelerated by different methods, for instance, the one-thread-one-row (1T1R) method on NVIDIA GPU[5]. However, the running time of a thread group depends on the slowest thread in it, which makes the computation less efficient. [5] also proposed a scan-based method, and the primitive operation of which, segmented scan, is realized in CUDA Data Parallel Primitives Library (CUDPP). In this method, multiplications and additions are separated, so do gathering and accumulation. It is good in load balancing, but requires extra spaces for the intermediate data. And the space requirement is in the same order of non-zero values. Another method is the one-warp-one-row (1W1R) on, which is proposed in [6], [7], and has been implemented on NVIDIA GPU and CELL platform. The experimental results show great performance gain, however, this method potentially wastes a large number of threads when the matrix is very sparse, like those in PageRank. On ATI Stream platform, there is an example of SpMV provided with AMD Stream SDK. It uses ITPACK format [8] to store the matrices, which we believe to be not suitable for large-scale web linkage data sets. The space requirement of the ITPACK is the largest number of non-zero values in all rows of the linkage matrix multiplied by the number of rows. If the numbers of non-zero values in rows are highly uneven, the ITPACK format will result in high usage of device memory. Moreover, our experiments show that **neither** 1T1R **nor** 1W1R achieves very notable performance for PageRank on AMD GPUs. It is because these general SpMV methods do not address the special characteristics of web linkage matrices.

III. The Power Method of PageRank Algorithm

In this section, we give a concise description of the PageRank algorithm [1] used in our implementation. Suppose we need to rank N_{row} web pages. Let R be a N_{row}-dimension column vector, and $||R|| = 1$. We want to solve R_i, which is the importance value (PageRank value) of the i^{th} page. The PageRank algorithm solves the problem $R = cAR$ to obtain R, where c is the dominant eigenvalue of A. And the sparse matrix A is defined by

$$A_{ij} = \begin{cases} \frac{1}{C_j}, & \text{if page } j \text{ has a link to page } i \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where C_j is the number of out linkage from page j. Usually, A is a very sparse matrix. In Practice, we always mix a escape vector E to represent the possibility of the web surfer jumping randomly to another page, not following the link. In computation, the introduction of E increases numerical stability. In our implementation, we use $E_i = \mathbf{1}$. Hence, the PageRank problem is to solve

$$R = c' \left[qA + (1-q)E^T \times \mathbf{1} \right] R \quad (2)$$

where q is the possibility of the surfer following the link. $q = 0.85$ is usually used in practice. c' is the dominant eigenvalue of $A' = qA + (1-q)E^T \times \mathbf{1}$.

A. The power method

A popular method to compute PageRank is the power method, described in Algorithm 1. In this algorithm, ϵ is a stopping threshold value. We can see that, the majority of the computation in PageRank is sparse matrix-vector multiplication $Y = AR_k$. In our experiments, $92\% - 95\%$ running time is dedicated to SpMV computation. In practice, N_{row} can be very large, so it is desired that an efficient PageRank implementation has a scalable and fast SpMV subroutine.

Algorithm 1: Power method of PageRank

Initialize R randomly to be R_0, and let $k = 0$
repeat
  Compute $Y = qAR_k$
  $d = ||R_k||_1 - ||Y||_1$
  $R_{k+1} = Y + dE$
  $k \leftarrow k + 1$
until $||R_{k+1} - R_k||_1 < \epsilon$

B. The CSR Format[9]

The CSR format is a compact data structure of sparse matrix. The Space requirement of the CSR format scales linearly with the number of non-zero values in the matrix. In the CSR format, N_{nz} non-zero values in the sparse matrix A are stored in a row-major vector A_v, and another vector A_j records the column position of each non-zero value in A_v. There is also a vector A_p record the start and end position of each row in A_v. An example of the CSR format is given as follows.

$$A = \begin{pmatrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$A_v = [2\ 3\ 1\ 1], \quad A_j = [0\ 1\ 1\ 0], \quad A_p = [0\ 2\ 3\ 4]$$

We shall implement our SpMV based on a modification of this representation of sparse matrices.

C. Naive SpMV Implementation

Algorithm 2 is the pseudo code of a basic serial algorithm of CSR format based SpMV, i.e. $Y = AR$. The outer loop enumerates all rows by performing a sparse vector - dense vector dot production. To calculate the doc production, firstly the elements in the vector R are gathered by the indices of the sparse vector, then multiplied with the corresponding values in the sparse vector, and finally accumulated to the results.

Algorithm 2: Sequential SpMV

for $i = 0$ to $N_{row} - 1$ **do**
  $r_{begin} \leftarrow A_p[i]$
  $r_{end} \leftarrow A_p[i + 1]$
  $acc \leftarrow 0$
  for $c = r_{begin}$ to r_{end} **do**
    $acc \leftarrow acc + A_v[c]R[A_j[c]]$
  end for
  $Y[i] \leftarrow acc$
end for

IV. AMD RV870 Architecture

AMD has a general purpose programming platform for its GPUs, named ATI Stream. In this section, we briefly discuss the hardware functionality and programming model of AMD RV870 GPUs, which belong to the latest family of AMD GPUs. Please refer to [10] for detailed information.

A. Hardware architecture

Figure 1 illustrates the hardware architecture of RV870. The GPU's main computation power comes from 320 5-way VLIW processors, called thread processors. Each thread processor contains 5 shader cores, including 4 normal cores that can perform 32-bit integer or floating point arithmetic, and 1 transcendental core that can do transcendental functions such as trigonometric or exponential functions. Each thread processor also has its private register file, with a size of 128-bit by 256 entries. Each general purpose register (GPR) is a 4-element short vector, which is usually named `float4`, `int4`, etc. Each element in a short vector is staticly addressed using x, y, z or w.

The 320 thread processors are organized in 20 SIMD engines. Along with the 16 thread processors, each SIMD engine also has a 32kB dedicated scratch memory, called the local data share (LDS). The LDS is organized as 32-bit by 256 entries by 32 banks. It supports random access. However, the access must be serialized when multiple threads try to access a same bank. The LDS also supports several atomic integer arithmetic.

All thread processors in a certain SIMD engine perform instructions in lockstep. Different SIMD engines can perform different instructions. In the programmer's view, the width of SIMD engines is 64, due to hardware switching of threads. This means each 64 threads are bundled logically, and perform instructions in lockstep. These bundles of threads are called wavefronts.

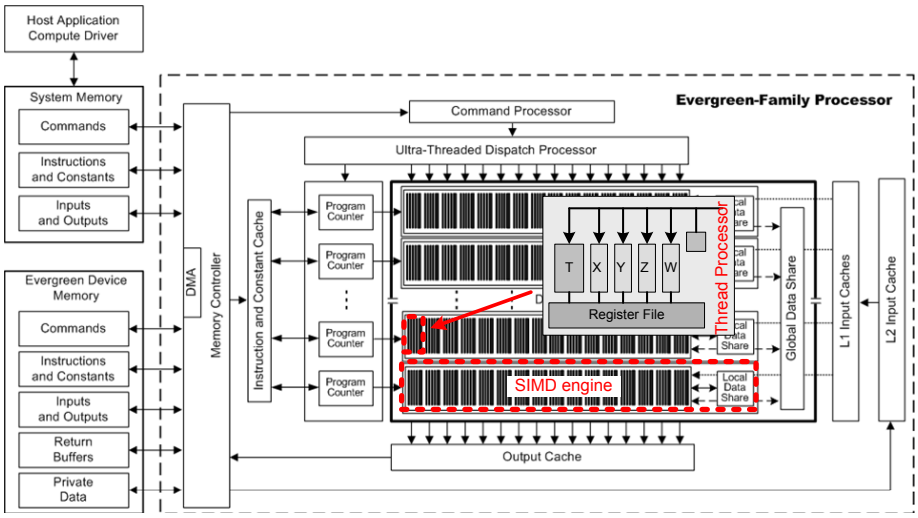

Figure 1. RV870 Architecture [10]

Each thread processor can access its GPRs, the LDS, and the off-chip graphic memory, while the memory latency differs quite a lot. The GPRs are the fastest, which can be accessed at full speed of the processor cores. Accessing the graphic memory, however, may take as long as several hundreds of clocks. To cover the huge latency, the hardware scheduler suspends the current wavefront and switch to another data-ready wavefront. Wavefront switching is applicable as long as there are enough resources, such as free GPRs and LDS space.

B. Programming model

The program run on GPU is called kernel. Normally, we run thousands of threads on the GPU. In this case, each thread is an instance of the kernel, which means threads are running exactly the same program with its unique ID.

There are two types of kernel, pixel shader (PS) and compute shader (CS). The CS mode is usually used for general purpose computing. In CS mode, threads are organized in thread groups. Threads in a same group are guaranteed to be run on a same SIMD engine, so that they can exchange data via the LDS. Note that the size of thread group is set by the programmer, although limited by hardware resources, while the size of wavefronts is determined by hardware. A thread group can have several full wavefronts and a trailing partial wavefront.

Unlike multi-threaded CPU programs, where each thread runs different program and the total number of threads usually do not exceed the number of CPU cores, GPU program usually initiates tens of thousands or even more threads, in order to keep all thread processors busy. The number of threads usually exceeds the number of thread processors by orders of magnitude. The power of a single thread on GPU is much weaker than that of a CPU thread, however, it is the massive parallel processing ability that makes GPU powerful.

V. GPU Acceleration of PageRank

Figure 2 shows the workflow of our PageRank implementation. The target web linkage matrix is first preprocessed by CPU, then transferred on to the graphic memory. The power loop of PageRank consists of an SpMV and several other vector operation, including examination of the end condition. The power loop finally outputs the PageRank vector.

As we have discussed, our experiments show that SpMV takes more than 92% of running time. Therefore, the focus of accelerating PageRank is accelerating SpMV, with specific consideration of the web linkage matrices. In this section, we discuss the characteristics of typical web linkage matrices, then propose our implementation and optimization of PageRank on AMD GPUs.

A. Characteristics of Web Linkage Matrices

The width and height of a linkage matrix N_{row} is the total number of pages being ranked, which easily reaches millions or billions. The number of pages on the web is exploding all the time, the average number of links in a page, however, is small and remains relatively stable. Hence, the number of non-zero elements N_{nz} in a web linkage matrix is proportional to N_{row}.

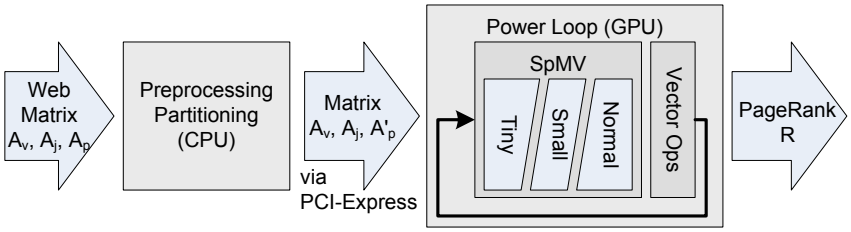

Figure 2. PageRank workflow

Table I shows the statistics of data sets we used in our work. They are retrieved from [11]. In a web linkage matrix A, $A(i,j) > 0$ if page j has links to page i. Hence, the number of backlinks (incoming links) of a page equals to size (i.e. the number of non-zero elements) of the corresponding row. The statistics show that from November 2005 to November 2006, the number of pages on wikipedia nearly doubled, while the average number of links on each page remains around 12. The average number of links of pages in the edu domain is even lower.

Figure 3 shows the cumulative distribution of row sizes of two data sets. Both data sets have over 50% of rows with less than 2 non-zero elements; and over 95% of rows have less than 64 non-zero elements. However, there are quite a few rows that have very big sizes. These characteristics will affect our SpMV implementation.

B. SpMV Implementation

The basic SpMV routine described in Algorithm 2 can be directly ported to multi-threaded platforms by mapping each dot production between a row and the vector to a thread. We call this the one-thread-one-row

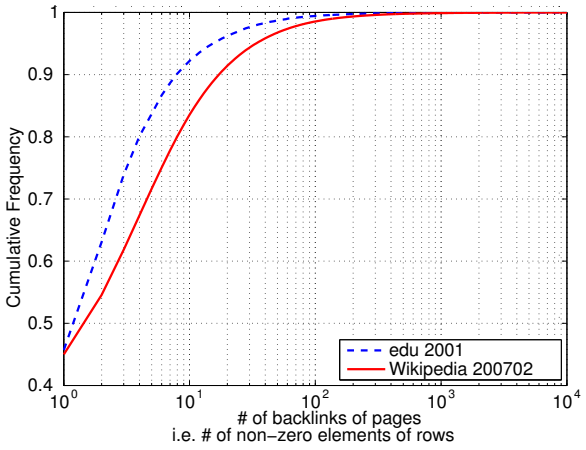

Figure 3. Cumulative distribution of row sizes

(1T1R) method. This method has a drawback when processing matrices with highly uneven row sizes, like web linkage matrices. The difference of sizes of rows results in unbalanced workload among nearby threads. On AMD GPUs, workload balance should be considered on a scale of wavefront. Since the time consumption for any wavefront is determined by the slowest thread in it, those threads with less workload finish early and have to "wait" for other threads to catch up.

To balance the workload in wavefronts, we further parallelize the inner loop into N_{wave} threads, where N_{wave} is the size of a wavefront (and equals to 64 in our case). The pseudo code is presented in Algorithm 3. A wavefront-wide reduction is needed, using the LDS memory. We call this the one-wavefront-one-row (1W1R) method. This method is similar to a CUDA SpMV implementation proposed in [12]. However, this method do not work quite well with web matrices. As shown in the previous discussion, most rows have quite few non-zero elements compared to $N_{wave} = 64$, which means a large number of threads are idle while only a few busy ones are doing the actual work. For example, when a wavefront is processing a row with only one non-zero element, only the first thread performs the actual work, while all other threads in the wavefront have to join the reduction process.

To solve this problem, we can use a hybrid method. The basic idea is to sort the sizes of rows in an roughly ascendent order, so that we can use different schemes to process different parts of the data set. Before sorting, we need to generate and save some additional metadata for each row: 1) the size of the row i, denoted by $A'_p[i].size$, and 2) the original index of the row, denoted by $A'_p[i].index$. $A_p[i]$ is now denoted by $A'_p[i].begin$. Both of the additional fields are implicit in ordinary CSR row array A_p. After adding additional metadata, we can reorder A'_p depending on sizes of rows.

The preprocessing only changes the representation of the original matrix by replacing the row index

Algorithm 3: Parallel SpMV : 1 row 1 wavefront

for each thread in parallel **do**
  Obtain thread ID in group Tid and wavefront ID Gid
  $r_{begin} \leftarrow A_p[Gid]$
  $r_{end} \leftarrow A_p[Gid+1]$
  $acc \leftarrow 0$
  for $c = r_{begin}$ + Tid to r_{end} stride N_{wave} **do**
    $acc \leftarrow acc + A_v[c]R[A_j[c]]$
  end for
  $LDS[Tid] \leftarrow acc$
  Parallel reduce using LDS, result in LDS[63]
  if Tid is the last thread in group **then**
    $Y[Gid] \leftarrow LDS[Tid]$
  end if
end for

array A_p in basic CSR into A'_p. Both A_v and A_j remain unchanged. The pre-processing is done on CPU before loading the matrix to graphic memory, which takes $O(N_{row})$ time and is only executed once. The preprocessing also expands the size of row array to 3 times. However, compared to the size of A_v and A_j, both of which at $O(N_{nz})$ order, both A_p and A'_p's requirement is $O(N_{row})$.

In practice, we classify the rows into three classes: 1) Tiny Problems: each row with no more than t_{r1} non-zero elements is processed by one thread, using the direct mapping scheme (1T1R) we have discussed. Due to the classification, successive rows now have relatively little differences in size. Hence, there is not big workload unbalance within a wavefront. 2) Small Problems: each row with more than t_{r1} and no more than t_{r2} non-zero elements is processed by quarter wavefront, i.e. $N_{wave}/4 = 16$ threads. The 16T1R algorithm is similar to Algorithm 3. Finally, 3) Normal Problems: rows with more than t_{r2} non-zero elements are processed with 1W1R method.

On AMD GPU, we choose $t_{r1} = 6$ and $t_{r2} = 96$. These parameters ensure that 1) in both tiny and small problems, the loop do not exceed 6 iterations, so that the possible uneven workload do not lead to great waste of computation power; 2) in normal problems, all threads are doing useful work. In section VI, the experiments show that these parameters can bring the best performance improvements compared to pure 1T1R or 1W1R implementations.

C. Optimized GPU Kernels

Here we present the pseudo code of kernels for tiny, small and normal problems in Algorithm 4, 5 and 6 respectively. Note that here A_p represents the modified CSR row index array. The three kernels should be run in the order of tiny-small-normal to perform one SpMV. For small and normal problems, the kernel must also know how many rows have already been processed by preceding kernels, i.e. the $Offset$ parameters.

Algorithm 4: Optimized Parallel SpMV: 1 row 1 thread

for each thread in parallel **do**
  Obtain absolute thread ID $aTid$
  $r_{begin} \leftarrow A_p[aTid].begin$
  $r_{end} \leftarrow r_{begin} + A_p[aTid].size$
  $acc \leftarrow 0$
  for $c = r_{begin}$ to r_{end} **do**
    $acc \leftarrow acc + A_v[c]R[A_j[c]]$
  end for
  $Y[A_p[aTid].index] \leftarrow acc$
end for

Algorithm 5: Optimized Parallel SpMV: 1 row 16 threads (Group size=64)

for each thread in parallel **do**
  Obtain absolute thread ID $aTid$ and thread ID in group Tid
  $row \leftarrow aTid >> 4 + Offset_{small}$
  $r_{begin} \leftarrow A_p[row].begin$
  $r_{end} \leftarrow r_{begin} + A_p[row].size$
  $acc \leftarrow 0$
  for $c = r_{begin}$+(Tid & 0x03) to r_{end} stride 16 **do**
    $acc \leftarrow acc + A_v[c]R[A_j[c]]$
  end for
  $LDS[Tid] \leftarrow acc$
  Parallel reduce using LDS, result in LDS[15, 31, 47 and 63]
  if $Tid \in 15, 31, 47, 63$ **then**
    $Y[A_p[row].index] \leftarrow LDS[Tid]$
  end if
end for

VI. Experiments and Analysis

We use 7 data sets retrieved from the University of Florida sparse matrix collection[11], Kamvar group and Gleich group. Data sets named su and su+ucb are web linkage of Stanford wab and Stanford and Berkeley web, respectively. There are four data sets extracted from Wikipedia, and one is extracted from

Table I
STATISTICS OF THE DATA SETS.

Data set	# of rows	# of non-zeroes	avg.# of non-zeroes per row	CSR size (MB)	After preproc. (MB)
su	281,903	2,312,497	8.20	18.7	20.9
su+ucb	683,446	7,583,376	11.10	60.5	65.7
wikipedia-20051105	1,634,989	19,753,078	12.08	156.9	169.4
wikipedia-20060925	2,983,494	37,269,096	12.49	295.7	318.5
wikipedia-20061104	3,148,440	39,383,235	12.50	312.5	336.5
wikipedia-20070206	3,566,907	45,030,389	12.62	357.2	384.4
edu-2001	9,845,725	57,156,537	5.81	473.6	548.7

Algorithm 6: Optimized Parallel SpMV: 1 row 64 threads (Group size=64)

for each thread in parallel **do**
  Obtain thread ID in group Tid and group ID Gid
  $row \leftarrow Gid + Offset_{normal}$
  $r_{begin} \leftarrow A_p[row].begin$
  $r_{end} \leftarrow r_{begin} + A_p[row].size$
  $acc \leftarrow 0$
  for $c = r_{begin} + Tid$ to r_{end} stride 64 **do**
    $acc \leftarrow acc + A_v[c]R[A_j[c]]$
  end for
  $LDS[Tid] \leftarrow acc$
  Parallel reduce using LDS, result in the LDS[63]
  if $Tid = 63$ **then**
    $Y[A_p[row].index] \leftarrow LDS[Tid]$
  end if
end for

.edu domain. We summarized the statistics of these data sets in Table I.

A. Experimental Setup

We perform experiments on the Radeon 5870(RV870) GPU at 850MHz. The counterpart CPU is PhenomII 965 at 3.4GHz. We run PageRank on the data sets to measure the detailed breakdown of elapsed time. Specifically, we measure the preprocessing time, the average time per round for SpMV and vector operations.

The algorithms are implemented on both Cal/IL and OpenCL. The OpenCL framework is a novel one that can take advantage of the parallel processing power of both CPUs and GPUs. It is a promising technology, especially addressing the issues of programming heterogenous parallel systems. The logic of algorithms we used in OpenCL implementation is exactly identical to our CAL/IL implementation. We use ATI Stream SDK v2.0.1 with OpenCL support.

B. Results on CAL/IL

The experiment result of PageRank with CAL/IL acceleration is summarized in Table II. It shows the average time for each round of computation on CPU and GPU, the time devoted to SpMV, and the preprocessing time. We can see that up to 12x speedup can be obtained for 500 rounds, and the speedup depends on the data set.

Figure 4 shows the SpMV performance comparison in GFlops of 5 different methods. The 5 methods are serial implementation on CPU, 1T1R, 16T1R, 1W1R and our hybrid method on RV870. The total Flops is estimated by $2N_{nz} - N_{row}$. Since the data sets of wikipedia on 20060925, 20061104 and 20070206 have similar dimensions, we only show the experimental results of the largest data set of these three in the chart. From the chart we can see that our method outperforms other three GPU implementation. Besides, 16T1R is more suitable for web matrices SpMV.

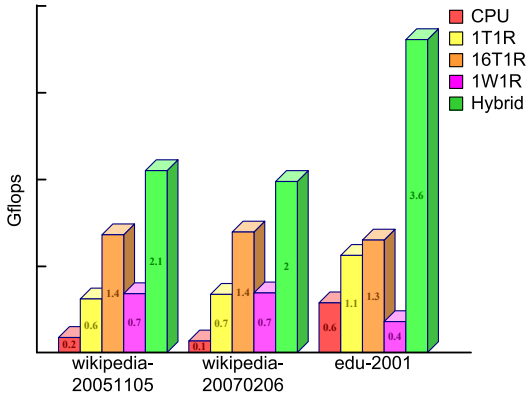

Figure 4. SpMV performance comparison, in GFlops. From left to right: CPU, 1T1R, 16T1R, 1W1R, our hybrid method

C. Results on OpenCL

There are a few constraints related to memory allocation and control in current OpenCL support. Currently, the memory space that can be used is relatively small compared to CAL/IL on the same graphic card. And the maximum size of a single buffer is no more

Table II
PERFORMANCE OF PAGERANK (TIME IN MS)

Data set	CPU	GPU			speedup for 500 rounds
	time/round	preprocessing	SpMV/round	overall/round	
su	16.67	11.51	2.74	2.81	5.9
su+ucb	25.54	31.00	4.74	5.18	4.9
wikipedia-20051105	234.31	72.65	18.08	18.66	12.5
wikipedia-20060925	541.85	154.01	35.98	37.25	14.4
wikipedia-20061104	558.11	146.35	38.13	39.42	14.1
wikipedia-20070206	701.54	156.22	43.91	45.44	15.3
edu-2001	289.92	467.92	28.99	32.31	8.7

than a quarter of the total graphic memory. Meanwhile, remote memory is not supported.

Due to these constraints, our OpenCL implementation of SpMV and PageRank can only work on relatively small datasets. Hence, we do not perform SpMV experiments using all the data sets discussed above. We also compared the performance of OpenCL and CAL/IL implementation of vector addition, L1 norm and L1 distance calculation, which are also elements of PageRank algorithm. Although these vector operations are not the bottleneck of PageRank calculation, in term of comparison between OpenCL and IL, we think the experiments are relevant.

Table III
PERFORMANCE COMPARISON BETWEEN CAL/IL AND OPENCL

Operation	Performance (GFlops)			IL/OCL ratio
	CPU	IL	OCL	
Vec-addition-1M	0.405	6.329	2.577	2.46x
Vec-addition-10M	0.408	9.852	6.329	1.56x
Vec-norm-1M	0.165	9.259	1.368	6.77x
Vec-norm-10M	0.165	26.455	6.734	3.93x
Vec-dist-1M	0.317	10.870	2.356	4.61x
Vec-dist-10M	0.320	29.283	9.556	3.06x
SpMV-wp-20051105	0.173	2.095	1.898	1.10x
SpMV-wp-20060925	0.138	1.989	1.893	1.05x
SpMV-wp-20061104	0.137	1.983	1.890	1.05x

The experiments are also done with a Radeon 5870 (RV870) card and a PhenomII 965 CPU. The results are reported in Table III.

The experimental results suggest that OpenCL GPU implementation is generally a few times slower than CAL/IL implementation. This is comprehensible since OpenCL is a high-level language which actually generates the lower-level IL code eventually for execution. Moreover, the performance gap between OpenCL and IL is varied for different algorithms. The gap for SpMV is the smallest that the performance of OpenCL implementation is quite satisfactory. The gap for vector norm is the largest, which is generally unacceptable.

Taking into account the computation density of these algorithms, which is defined as the ratio of global memory access to ALU operations, we find that algorithms with higher computation density tend to have poorer performance in OpenCL, compared with IL. For example, vector norm calculation has the highest computation density in our tested algorithms. It requires coalesced read of the vector only slightly more than once, since most of the reduction is done in the LDS memory. Vector distance calculation has twice read compared to norm calculation. Vector addition has almost the same global memory access to distance calculation, however, it has much less ALU operations. Finally, SpMV needs random (i.e. not coalesced) read from the vector, which is a main bottleneck of the GPU implementation. So the computation density is the lowest among our tested algorithms.

Computation intensive algorithms tend to perform poorer in OpenCL, compared with IL. This is because the ALU instructions generated by OpenCL compiler is not quite optimized. The time of memory access can cover the drawback if the algorithm is data intensive. After all, the results suggest that there is a big gap between the performance of OpenCL and IL which developers have to fight to attenuate.

D. Scaling to Larger Data Sets

The data sets we used in our experiments are large but still able to fit in the graphic memory. In some real applications, however, the data sets may be much larger, for example, the web linkage data of the whole www. Our method can easily scale to larger data sets.

A frequently used technique for GPU application is streaming, i.e. pipelining data uploading, kernel execution and data downloading, so that they run concurrently. The prerequisite for this method is that the data throughput of kernel does not exceed the bandwidth of data uploading and downloading, so that kernel execution time covers data transfer time. However, a quick math shows this is not the case for PageRank.

From table II and table I we can see that the kernel throughput exceeds $384.4MB/45.44ms = 8.46GB/s$

for wikipedia-20070206 and $548.7MB/32.31ms = 16.98GB/s$ for edu. The PCI-Express x16 bandwidth is $5GB/s$ for uploading, however, a long way from satisfying the kernels' appetite. Hence, the kernel execution time can not cover the data transfer time unless some kind of intensive data compression is applied.

To scale to larger data set, we can add more graphic card to the system, or add more computer node to form a cluster. We can horizontally split a web matrix into several tiles. Then, transfer each tile to a computation chip's local memory. The current vector is broadcasted to all nodes, and multiplies each tile either on CPU or on GPU. The preprocessing only takes place on GPU tiles, and different tiles can be preprocessed independently.

The requirement for this method is that the vector can be wholly stored on all computation chips' local memory. For a system using graphic cards with 1GB RAM, we can process data sets with up to 100M pages. If this is not the case, we can further divide each tile of matrix vertically. The matrix need to be properly divided to maintain workload balance of all nodes. Awareness of heterogeneity may also be needed. However, from the view of a single node, the problem go back to SpMV and vector operation, where our method can be utilized.

VII. Conclusion

In this paper, we accelerate the PageRank computation on AMD GPUs using the intermediate language and OpenCL. As a primitive, we implement a sparse matrix-vector multiplication routine. We carefully classify the problem into three sub-classes and design efficient kernels. In the experiments, up to 15x speedup is obtained. The major bottleneck of our current implementation is the gathering operation in SpMV, which requires random accesses to the device memory, hence the cache hit rate is generally low. In the future, we shall find methods to increase the cache hit rate, which further increases the overall performance.

We also compare the performance of AMD OpenCL platform based implementation with that of low-level IL based implementation, using the same algorithm and method. The results show that currently OpenCL to IL compiler is not quite optimized. Data intensive algorithms may perform more satisfactorily than computation intensive algorithms on AMD OpenCL.

The scalability to larger datasets is also analyzed in our work. Based on the comparison of kernel throughput and PCIe bandwidth, we find that the kernel execution time can not effectively cover that of the data transfer, if the data can not stay in the graphic memory. Hence, we suggest to add more graphic cards or computer nodes for scaling to larger problems.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," 1999.

[2] Y. Zhu, S. Ye, and X. Li, "Distributed pagerank computation based on iterative aggregation-disaggregation methods," in *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*. New York, NY, USA: ACM, 2005, pp. 578–585.

[3] J. R. Wicks and A. Greenwald, "More efficient parallel computation of pagerank," in *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. New York, NY, USA: ACM, 2007, pp. 861–862.

[4] Y. Wang and D. J. DeWitt, "Computing pagerank in a distributed internet search system," in *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 2004, pp. 420–431.

[5] M. Garland, "Sparse matrix computations on manycore gpu's," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*. New York, NY, USA: ACM, 2008, pp. 2–6.

[6] M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies," IBM Technical Report, Tech. Rep., 2008.

[7] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," NVIDIA Technical Report NVR-2008-004, Tech. Rep., 2008.

[8] D. R. Kincaid, T. C. O. Oppe, and D. M. Young, *ITPACKV 2D User's Guide*, Report CNA-232, The University of Texas at Austin, May 1989.

[9] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM Publication, 2000, ch. Common Issues: Sparse Matrix Storage Formats, pp. 403–404.

[10] *ATI Stream Computing Programming Guide - Compute Abstraction Layer (CAL)*, Advanced Micro Devices, Inc., Mar 2010.

[11] T. Davis, "University of florida sparse matrix collection," http://www.cise.ufl.edu/research/sparse/matrices.

[12] M. G. Nathan Bell, "Efficient sparse matrix-vector multiplication on cuda," NVIDIA, Tech. Rep., Dec 2008.