FPGA and GPU Implementation of Large Scale SpMV

Yi SHAN[1], Tianji WU[1], Yu WANG[1], Bo WANG[1], Zilong WANG[1], Ningyi XU[2], Huazhong YANG[1]

[1]Tsinghua National Laboratory for Information Science and Technology
Department of Electronic Engineering, Tsinghua University, Beijing 100084, China
[2]Hardware Computing Group, Microsoft Research Asia
[1]{shany08,wutj06,wangb06}@mails.tsinghua.edu.cn, {yu-wang, yanghz}@tsinghua.edu.cn
[2]xu.ningyi@microsoft.com

Abstract—**Sparse matrix-vector multiplication (SpMV) is a fundamental operation for many applications. Many studies have been done to implement the SpMV on different platforms, while few work focused on the very large scale datasets with millions of dimensions. This paper addresses the challenges of implementing large scale SpMV with FPGA and GPU in the application of web link graph analysis. In the FPGA implementation, we designed the task partition and memory hierarchy according to the analysis of datasets scale and their access pattern. In the GPU implementation, we designed a fast and scalable SpMV routine with three passes, using a modified Compressed Sparse Row format. Results show that FPGA and GPU implementation achieves about 29x and 30x speedup on a StratixII EP2S180 FPGA and Radeon 5870 Graphic Card respectively compared with a Phenom 9550 CPU.**

Keywords-component; SpMV; FPGA; AMD GPU; memory hierachy

I. Introduction

Sparse matrix-vector multiplication (SpMV) is an important subroutine in numerical linear algebra. The sparse matrix means a large matrix mostly composed of zero entities. As the data scales, it brings many challenges to the storage and computation of the SpMV for large scale applications. For example, the current Internet has hundreds of billions of web pages, the ranking of which becomes a valuable and challenging problem. The most widely used method is the PageRank [1], which iteratively perform sparse matrix-vector multiplication (SpMV) over all the web pages' linkages. A high performance, cost-effective SpMV computation solution will benefit a lot of applications in research and commercial computing.

As the physical constraints are preventing frequency scaling of CPUs and power consumption is becoming a critical problem recent years, the parallel computing becomes the dominant paradigm for large scale computing applications. There are already many studies which use the parallel computing platforms to accelerate the SpMV algorithm, such as the multi-cores [14], clusters, GPUs [6-8] and FPGAs [4, 5]. In the multi-cores and clusters implementation, the programming is easy and flexible, because designers do not need to care about the memory access and computation sequence. However, the multi-threads' memory access will cause many performance penalties and the power consumption is still a crucial problem for the CPU based platforms.

As one of the alternative computing platforms, FPGAs has been widely explored in various high performance computing applications in recent years [2], because i) FPGA is reconfigurable and easy to change functionalities without changing the platform; ii) logic elements in FPGA work in a naturally fine-grained parallel way with high flexibility; and iii) FPGA is one of the best hardware devices that can follow the Moore's Law persistently [3]. Although FPGA researchers have implemented SpMV through many methods [4, 5], the matrix they used is either too small or not so sparse. In that scenario, the vectors' random accessing problem can be solved by storing them on-chip and the processor scheduling cost between tasks is really small. In the FPGA implementation of this paper, we analyzed different scale of datasets, and then proposed different memory hierarchies for them. For the computation, we design a stack based accumulator which can support the continuous data stream processing and a static processor scheduling scheme to cover the scheduling cost.

Another important computing platform, the graphic processing units (GPUs) have become popular during recent years. GPUs are massively parallel devices with high computation performance and high power efficiency. Many previous studies focused on accelerating SpMV with GPU [6, 7, 8]. However, general purpose SpMV routines do not perform well for large scale matrix operations used in PageRank application. In the GPU implementation of this paper, we analyzed the characteristic of the sparse matrices used in PageRank, and introduced a fast SpMV 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). Our SpMV subroutine is highly optimized for the AMD GPU architecture using AMD Compute Abstraction Layer (CAL) and Intermediate Language (IL).

In this paper, we use both FPGA and AMD GPU to implement the SpMV algorithm for the web linkage datasets. The purpose is to provide efficient computation solutions in these two widely available acceleration platforms. For each platform, the optimized methods are analyzed according to the hardware architecture. Processing parallelism, processor scheduling, and memory hierarchy are discussed for both FPGA and GPU implementations. The main contributions of this paper are:

978-1-4244-7954-2/10/$26.00 ©2010 IEEE

1. A reasonable mapping SpMV to platform according to the architecture and the datasets' feature;
2. Specifying several types of task partition and memory hierarchy methods on FPGA design according to different scales of dataset;
3. Giving an optimized design method on AMD GPU based on the analysis of web linkage matrices, and discussing the scalability of the GPU implementation.

The remainder of this paper is organized as follows. Section 2 gives some background knowledge and a review on related works. Section 3 introduces the FPGA implementation of SpMV. Section 4 gives a brief introduction to the AMD RV870 GPU. The optimized GPU implementation of SpMV is discussed in Section 5. The experimental results are presented and analyzed in Section 6. Section 7 concludes this work.

II. Background Knowledge and Related Works

In this part, the SpMV algorithm and a typical representation method of the sparse matrix (CSR) will be introduced. Then the implementations of SpMV algorithm will be surveyed and their feature will be analyzed.

A. The CSR Format

In order to analyze the SpMV algorithm, the CSR format will be introduced firstly for the very sparse matrix representation. 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 which records the start and end position of each row in A_v. An example of the CSR format is given as follows.

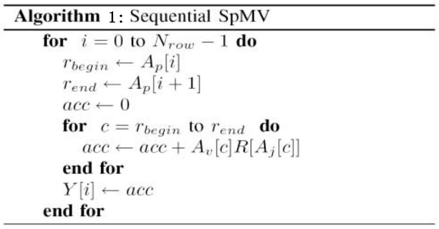
$$A = \begin{matrix} 2 & 3 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{matrix}$$

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

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

B. SpMV

Algorithm 1 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 1: 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
```

C. FPGA and GPU Implementation

There are many implementations of SpMV on FPGA. deLorimier and DeHon implemented a SpMV algorithm on FPGA and achieved 2.24GFLOPS on a Virtax-II FPGA[4]. Yan Zhang presented a FPGA and GPU comparison for SpMV, but the FPGA architecture requires that each row has a minimum of 8 non-zero data, otherwise, the performance will be highly reduced[5]. There are also other implementations, but nearly all of them have a limitation that Matrix's dimension is less than 100,000, thus the vector value can be stored on-chip. However, the random accessing pattern of vector value make this memory concerned about latency. So when the matrix's scale becomes large, on-chip memory cannot support this kind of access. In our implementation, we will use multiple off-chip SRAM and efficient partition scheme to handle this problem.

The implementation of SpMV on GPU platform received great attention. SpMV was firstly accelerated using a one-thread-onerow (1T1R) method [6]. The 1T1R method suffers from the various numbers of non-zeros each row, so that its performance improvement is not notable. A scan-based method used in [6] was also explored to accelerate the SpMV operation. A technical report from IBM Research [7] demonstrated how to optimize SpMV using Compile-time and Run-time strategies. The acceleration of SpMV was also designed and tested with different sparse matrix format [8]. In [8], a one-warp one-row (1W1R) method was introduced to reduce the divergences between different rows. The 1W1R method has shown great performance gain in the experiment. Besides GPUs, this operation was also accelerated on Cell platform [9]. All of these methods are not specifically designed to target the web linkage matrix, which is very large in size and very sparse in density. Our experiments show that both 1T1R and 1W1R methods suffer from performance drop when processing web linkage matrices, due to underutilization of hardware capacities. In our implementation, we address the characteristics of web linkage matrices to achieve better performance.

III. FPGA Implementation of SpMV

In FPGA implementation of SpMV, the memory hierarchy needs to be carefully designed, different from CPU and GPU implementations which have a fixed memory hierarchy. At the same time, the parallelism should also be concerned in order to improve the performance.

A. Overview of the Architecture

The SpMV design architecture on FPGA is shown in figure 1. Firstly, there are several PEs (processing elements) which implement the functions of the algorithm in a parallel way. Secondly, there are 4 memories which contain matrix's non-zero value A_v, column position A_j, row/row_pointer A_p and vector value respectively. The memory type depends on the storage column and processing/accessing pattern. Thirdly, there are processor scheduler and data manager to scheduling the work status and data accessing. SpMV is not a control intensive application but a data and computation intensive one. The most important thing is how to efficiently fulfill the memory bandwidth and instantiate as many PEs as the FPGA can support.

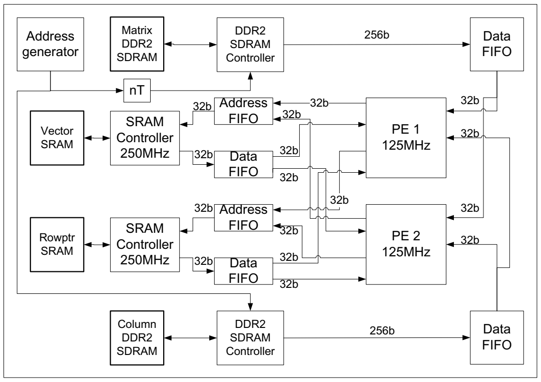


Figure 1. SpMV architecture on FPGA

For the datasets in Table 1, 8 PEs which run at 125MHz are instantiated to accelerate the SpMV implementation. Two DDR2 SDRAMs are used for Matrix and column storage. Each DDR2 has 256 bits data width in local side, which can serve 8 PEs at the same time. Two synchronized SRAMs are used for vector and rowpointer storage. Both of them have a 32 bits data width and run at 250MHz by the ZBT SRAM controller. So, one SRAM can provide data for 2 PEs. In total, to implement the SpMV algorithm for the Table 1 datasets, 8 PEs, 2 DDR2 SDRAMs and 8 SRAMs are needed.

B. Parallelsim and Task Scheduling

FPGA is a parallel platform and contains lots of logic elements that work concurrently. For SpMV application, the parallelism comes from the multiplications for different rows in the matrix. In the former situation, a fixed number of PEs will compute the multiplications for one row. If the row contains less non-zero values than the PE number, the dummy data must be inserted. The more PEs are instantiated, the more dummy data are inserted. So, in most cases, the matrix is partitioned by rows and each PE accounts for several sequential rows. In order to assure the load balance, the tasks assigned to different PEs should have similar non-zero data.

Another way to assign tasks is to partition the matrix by column. The n*n square matrix is partitioned into several n*n' (n' is n/m) matrix and the vector are partitioned in the similar way. In this case, each PE is responsible for a partial SpMV and another reduction operation will be needed to merge the partial results into the final result which is the disadvantage of this partition method. The advantage of this method is that each PE only has to store a part of the vector value which is randomly accessed.

For the datasets in Table 1, the matrix is partitioned by rows and the partitioned tasks are assigned to the 8 PEs in a load balance way.

C. Memory Hierachy and Data Scheduling

For the matrix's non-zero data storing, if the matrix is not that sparse and has more than one value in one row, the A_v storage is the largest among all the parts of input data. And the access pattern for this part is sequential, so it can be stored in on-chip memory, off-chip SRAM or DDRx SDRAM according to its scale. Column position A_j has the same number as A_v, so it can be stored in the same way.

For the row position, if the original format of sparse matrix is used, the row position storage is the same as the non-zero data. If the CSR format is adopted, row_pointer A_p must be stored separately for different PEs.

Individual vector data has to be prepared for different PEs due to the random access. At the same time, the data stream requires the vector access must be finished in no-wait cycle. Because of this, only the SRAM can be used here. If the vector data is less than 512kbit, it can be stored on-chip using the MRAM in Altera's FPGAs. Because the MRAM can support dual port accessing, one MRAM can serve 2 PEs. Take Stratix II EP2S180 as example; there are 9 MRAMs which can support 18 PEs only by using MRAM resource.

If the vector data is much larger than 512kbits, it has to be stored in off-chip SRAM. The popular synchronized SRAM can support 36bit data width at 250MHz and 72Mbits in size. For the PE logic runs at 125MHz, one SRAM can store vector for 2 PEs. Figure 1 use this type of vector memory for PEs. However, if vector is much larger than 72Mbits, the solution is to use more SRAMs or just let the processing discontinuous.

The solutions above are all based on the row partition method. If we partition the matrix by column, the vector can be partitioned at the same time. In this scenario, the vector size can be 72Mbits* N_{PE}. But the disadvantage is that we have to store the intermediate results and do one more reduction operation.

Matrix dimension	Sparse degree	Partition method	Vector storage	Matrix storage	Represent method
~512k/8	very	By rows	MRAM	MRAM	CSR
~512k/8	common	By rows	MRAM	SRAM/ DDRx	CSR
~72M/8	very	By rows	SRAM	SRAM/ DDRx	CSR
~72M/8	common	By rows	SRAM	DDRx	CSR
>>72M/8		By columns	SRAM	DDRx	absolute coordinate

For the datasets in Table 1, two DDR2 SDRAMs are used for Matrix and column storage and eight SRAMs are used for vector and rowpointer storage according to their size and access pattern.

D. Processing Elements

The processing elements for SpMV is MAC (multiply-accumulate) module. The PE will firstly fetch the rowpointer value A_p which suggests how many non-zero value is there in this row. Then the PE will do the one matrix row multiplying vector operation until all the non-zero values in this row have been processed. These intermediate data will be accumulated to build the final result. In order to improve the performance, the key problem is how to build an efficient MAC module which can serve streaming input data processing.

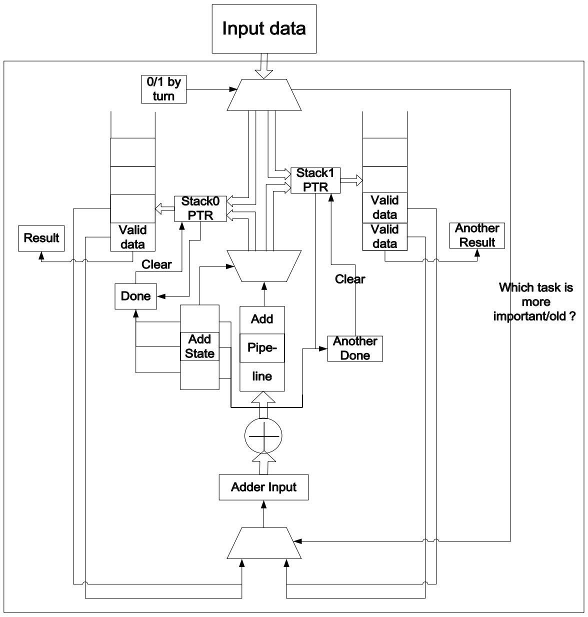


Figure 2. The architecture of Processing Elements

In our floating point accumulator, several cycles' computation resource of the pipeline adder is wasted. Most of the tasks are short, the starting and ending phased will make the efficiency of the adder below 50%. In our implementation, two buffers are instantiated to execute two tasks at the same time. When the first task isn't occupying the adder, the accumulator sends two numbers of the second task (stored in another buffer) into the adder. Adder State is the register that is used to distinguish the current user of the adder. Simply to say, the adder will execute the tasks in turn to make full use of the adder. Though the results will be out of order, the adder can be more efficient. In this way， the bottleneck is changed to bandwidth of memory, instead of PE's computation power.

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 [11] for detailed information.

A. Memory Hierarchy

In RV870, several memory resources can be used, each with different access constraints and speed. General purpose registers (GPRs) are the fastest memories. Each thread has access to up to 127 GPRs in float4 type, which is a short vector with 4 single precision floating point elements, named x, y, z and w.

Each SIMD engine has a 32KB dedicated Local data sharing memory (LDS) which enables low latency data sharing between threads in the same SIMD. On RV870, the LDS is organized in 32-bank*256-row structure. Each memory entry is 32-bit width. The LDS supports random access.

Off-chip graphic memory is the largest and slowest memory resource. It supports several access models. In compute mode, input-only resources can be allocated as image resources, which support texture sampling. Each kernel can bind multiple image resources. Read-write resources can be allocated as Uniform Access Views (UAVs) or global buffer. On RV870, multiple UAVs are supported; however, only one global buffer is supported. UAVs and the global buffer support several atomic operations.

B. Computation Units and Programming Model

The stream cores or ALUs are organized as 5-way VLIW processors, called thread processors. Each thread processor contains four normal cores that can perform 32-bit integer or floating point arithmetic, and one transcendental core that can perform transcendental functions such as trigonometric or exponential functions.

In RV870, 16 thread processors are grouped into a SIMD engine and there are 20 SIMD engines altogether. All thread processors in a SIMD engine performs the same instruction at any time, but on different datasets; different SIMD engines can perform different instructions. In a programmer's view, the width of SIMD engines is 64, due to hardware switching of threads. The bundle of 64 threads that simultaneously run on a SIMD is called a wavefront. The total run time of a wavefront is determined by the slowest thread in it.

There are two types of kernels, pixel shader (PS) and compute shader (CS). CS is usually used in general purpose computing. In this mode, threads are organized in groups. Each group consists of several wavefronts. These wavefronts are guaranteed to be run on the same SIMD engine, and thus can share data through the LDS. Threads in different groups cannot share data through the LDS.

V. GPU Acceleration of SpMV

In this section, we briefly discuss the characteristics of typical web linkage matrices and our implementation of SpMV on AMD GPUs.

A. Characteristics of Web Linkage Matrices

The width and height of a web linkage matrix, N_{row} is usually very large, which easily reaches millions or billions. However, the average number of links in a page is small and remains relatively stable. Table 1 shows the statistics of datasets we used in our work. Although the size of Wikipedia linkage matrix grows greatly, the average number of links on each page remains around 12. On the other hand, the average number of links of pages in the edu domain is even lower. Hence, web linkage matrices are extremely sparse. The larger a web linkage matrix is, the lower the density is.

Table 1. Statistics of the datasets.

Dataset	#rows, N_{row}	#non-zero values, N_{nz}	#avg non zero values/row
Wikipedia-20051105	1,634,989	19,753,078	12.08
Wikipedia-20060925	2,983,494	37,269,096	12.49
Wikipedia-20061104	3,148,440	39,383,235	12.50
Wikipedia-20070206	3,566,907	45,030,389	12.62
Edu-2001	9,845,725	57,156,537	5.81

Moreover, the non-zero values of different rows are highly uneven. Over 40% of rows have less than or equal to 1 non-zero elements; over 95% of rows have less than 64 non-zero elements.

B. *SpMV Implementation*

The basic SpMV routine can be directly ported to multi-threaded platforms by mapping each row-vector dot production to a GPU thread, i.e. the 1T1R method.

In this method, each thread loops through all non-zero values of one row, fetches the vector data according to the column indices of these non-zero values, and accumulates the products of corresponding vector and matrix values.

However, due to the differences with non-zero sizes of rows, the workloads of all threads are not well balanced in this method. On GPUs, threads are scheduled in bundles, which are called warps or wavefronts. Unbalanced workload within a wavefront may lead to hardware underutilization. To better balance the workload of threads within a wavefront, we rearrange the order of matrix rows being processed according to their sizes. Hence, rows with similar sizes are likely to be processed in a same wavefront.

To rearrange the rows, we pre-process the CSR matrix and sort the sizes of rows in an ascendant order. Before sorting, we need to save some additional fields for each row, besides the index of the first value of the row. These additional fields are 1) the size of the row *i*, denoted by $A_{0p}[i]$*:size*, and 2) the position of the row, denoted by $A_{0p}[i]$*:index*. $A_p[i]$ is now denoted by $A_{0p}[i]$*:begin*. After adding additional fields, we can sort A_{0p} by the size field.

After rearranging the rows, we have 3 different methods for rows in different scales, illustrated in Figure 4. Rows with less than t_{r1} non-zero values are processed with one thread. The results of each thread are directly written to the output vector. Rows with more than t_{r1} and no more than t_{r2} non-zero values are processed by 16 threads. The results of each thread are merged with 4 steps in the LDS memory. Rows with more than t_{r2} non-zeros are processed with entire wavefronts (64 threads on AMD GPU). Then a binary reduction with 6 steps is needed. These three classes of problems are denoted to tiny, small and normal problems respectively.

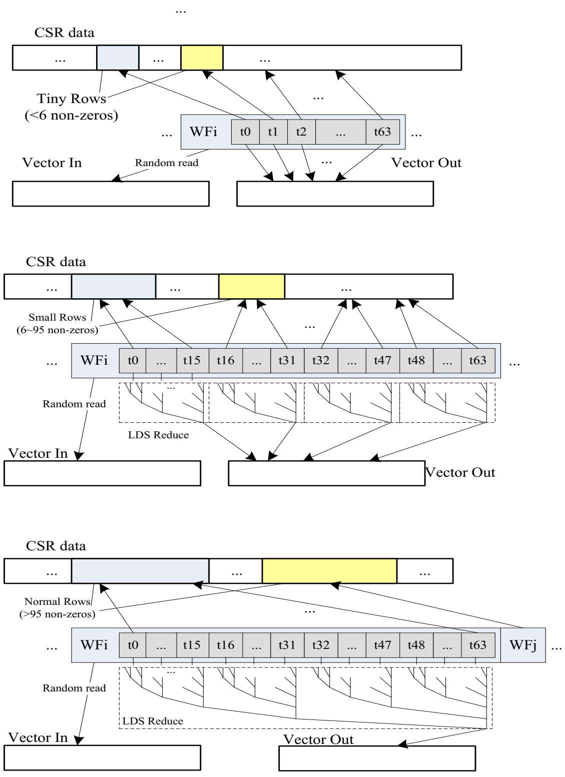
On AMD GPU, we choose t_{r1} = 6 and t_{r2} = 96. These parameters ensure that in both tiny and small problems, the loop will not exceed 6 iterations, so that the possible uneven workload will not lead to great waste of computation power.

VI. Experiments and Analysis

We use five datasets retrieved from the University of Florida sparse matrix collection [13]. The first four datasets are extracted from Wikipedia, and they have different data scale but similar sparse grade. The last one is the largest and sparsest that is extracted from .edu domain. The statistics of these datasets are shown in Table 1, and the characteristics of them are discussed in Section V.

A. *Experimental Results*

We perform experiments on the Altera StratixII EP2S180 FPGA at 125MHz and Radeon 5870(RV870) GPU at 850MHz. The comparison CPU is Phenom 9550 at 2.2GHz. The resource occupation of FPGA implementation is presented in Table 3.



a) Each thread processes a tiny row with <6 non-zero elements. Each thread output the result. b)16 threads process a small row with 6~95 non-zero elements. After local reduction, 4 results are written. c) 64 threads process a normal row with >95 non-zero elements. 1 result is written after the reduction.

Figure 4. Three types of kernel/thread allocation and reduction

Table 2 is the summarized experiment results of SpMV. It shows the average time for SpMV computation. We can see that up to 29x and 30x speedup is obtained on FPGA and GPU respectively. For FPGA implementation, since the computation is pipelined, the time cost equals to the data transferring cost. But for the empty rows, one cycle has to be wasted to do the judgment. When several PEs are instantiated, the overall computation time will be determined by the slowest PE. Hence, it is important to balance the workload before the computation.

Table 2. Performance of SpMV

Dataset	Time(ms)			Speedup	
	CPU	FPGA	GPU	FPGA	GPU
wikipedia-20051105	462.3	20.21	18.08	22.9	25.6
wikipedia-20060925	985.3	38.14	35.98	25.8	27.4
wikipedia-20061104	1048.4	40.32	38.13	26.0	27.5
wikipedia-20070206	1354.1	46.11	43.91	29.4	30.8
edu-2001	282.1	57.87	28.99	4.87	9.7

Table 3. Resource occupation of FPGA implementation

Resource	PE*8	ddr2 ctrl*2	total
Percentage	29%	3%	36%

B. *Discussion*

In this section, we will discuss three aspects. The first is our current implementation does not fully utilize the hardware's capacity both for FPGA and GPU. For FPGA, we can instantiate more PEs; and for GPU, the 4 way vector arithmetic is currently underutilized. The second is the scalability and usability of our method for larger datasets, such as the whole internet web linkage data. We will also discuss the design efforts of the two platforms when adopting MapReduce framework and OpenCL respectively.

1) More PEs for FPGA Implementation

Altera StratixII EP2S180 offers about 140K logic elements, but we just use 36% of them to achieve the similar performance as GPU. Theoretically, the performance can be 3 times higher with more PEs and more memory instances. Another hard limit is that the FPGA has a fixed number of I/O pins which can be used for more memory interfaces. The matrix data access can utilize the high bandwidth of the DDRx SDRAM for continuous access, but the demand for random access for the vector pushes us to duplicate many SRAMs in order to retain low access latency. Moreover, the SRAMs are expensive.

One way to ease this problem is to partition the matrix vertically as introduced in section 3. In this way, each PE only need a part of the vector value, the SRAM capacity requirement can be reduced by N_{PE} times. However, at the same time, we need additional computation resources to handle the reduction operation in this method.

2) AMD GPU Short Vector Utilization

AMD GPUs are optimized for 128-bit memory access and vector arithmetic. In order to evaluate whether we should explicitly utilize this feature, we compare our original method with the 1x4 blocked CSR format based method. Table 4 shows the performance and memory consumption while using the float4 short vector. From the results, we find that the performance is actually decreased when using the short vector. Due to the extremely low density of the matrix, it is very rare to have a float4 vector containing more than 1 non-zero elements. Hence, the times of memory fetch do not decrease notably, even if float4 short vector type is utilized. However, the total amount of data scale increased significantly.

Table 4. CSR and 1x4 BCSR SpMV Comparisons

Dataset	Time(ms)		A_V size(MB)	
	CSR	BCSR	CSR	BCSR
wiki-20051105	18.08	18.89	79	300
wiki-20060925	35.98	37.35	149	577

3) Scaling to Larger Datasets on both FPGA and GPU

For FPGA, when scaling to larger datasets, the vertical partition method can be used to ease the memory limitation. But the scale can be only tens of times than the current one if SRAMs are not added to the board. Another way is to use DDRx SDRAM to store the vector values when the data keep increasing. In this case, the system will cost several cycles for the random access latency. The optimized memory hierarchy for this scenario is to use cache architecture as the CPU and GPU do.

For GPU, when the data scales too large for the graphic memory to store, we can only store it in the system memory. One method is to split it into several tiles. The vector multiplies each tile either on CPU or on GPU. The preprocessing only takes place on GPU tiles and different tiles can be pre-processed independently. Moreover, the scheduling system should be carefully designed so that the overhead of transferring matrix tiles can be hidden by the computation. This is achieved by the concurrent running of DMA data transfer and GPU kernel.

4) Discussion about the Programming Model

For the FPGA implementation, design effort is really a big problem. Nowadays, there are several frameworks for FPGA design those can ease the effort [15, 16]. We have proposed FPMR framework in [15], which is the MapReduce framework on FPGA. MapReduce framework on FPGA is a promising framework that can hide the synchronization and communication cost by dynamic processor scheduling. For SpMV design, the multiply operation can be thought as the Map operation, and the accumulate operation can be thought as the Reduce operation. Under this framework, it is easy to build a quick version of SpMV implementation. However, for SpMV, each task is really small. The dynamic processor scheduling will be the large portion of the cost. So, maybe the MapReduce framework fits for the matrix-vector multiply that is not that sparse.

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 heterogeneous parallel systems. The current AMD OpenCL support has a few constraints related to memory allocation and control. For example, the memory space that can be used is relatively small compared to CAL/IL in the same graphic card. And the maximum size of a single buffer is no more than a quarter of the total graphic memory. Meanwhile, remote memory is not supported, which brings the difficulty in using typical Ping-Pong buffers. Despite these constraints, OpenCL greatly ease the programming effort compared to CAL/IL, and the drawback in performance is affordable.

FPGAs and GPUs have different implementation features for SpMV application. For FPGA, more flexibility in the hardware makes the upgrade easy by adding more memory and duplicating more PEs. Of course, the memory hierarchy and processor scheduling scheme should be tuned according to the larger task. GPU has more powerful floating point computation ability, but the fixed hardware architecture makes the upgrade can be done only by program mapping. For the design efforts, GPU is easier to program than FPGA. Although many frameworks have been proposed, to be compliant with different platforms, the performance will still rely on the designer's effort.

VII. CONCLUSION

In this paper, we accelerated the SpMV computation on FPGA and AMD GPUs. In FPGA implementation, we design a pipeline processing element and perform the task partition to memory hierarchy mapping according to the matrix's scale. In GPU implementation, we carefully classified the problem into three sub-classes and designed efficient kernels. In the experiments, up to 29x and 30x speedup is obtained respectively. The major bottleneck of our current implementation is the vector accessing/gathering operation in SpMV, which requires random accesses to the device memory. For FPGA, several SRAMs have to be used to avoid latency in the cost of expensive memory. For GPU, the fixed memory hierarchy makes the cache hit rate generally low. In the future, we shall find methods to ease the latency by only using DDRx SDRAM to store the vector value on FPGA and increase the cache hit rate on GPU, which will further increase the overall performance.

ACKNOWLEDGMENT

This work was supported by National Key Technological Program of China No. 2008ZX01035-001, NSFC No. 60870001, AMD UR project, and Tsinghua National Laboratory for Information Science and Technology Cross-discipline Foundation.

REFERENCES

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

[2] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, Doug DiSabello, Achieving High Performance with FPGA-Based Computing, Computer, Volume 40 , Issue 3 (March 2007), Pages 50-57

[3] FPGAs and Moore's Law , http://www.ciol.com/Semicon/Design-Trends/News-Reports/FPGAs-and-Moores-Law/111108112450/0/

[4] Michael deLorimier, Andre DeHon, Floating-point Sparse Matrix-Vector Multiply for FPGAs, FPGA 2005.

[5] Yan Zhang, FPGA vs. GPU for Sparse Matrix Vector Multiply, ICFPT 2009.

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

[7] M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compiletime and run-time strategies. Technical report, IBM Technical Report, 2008.

[8] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Technical Report NVR-2008-004, 2008.

[9] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. Technical report, UC Berkeley Technical Report, 2008.

[10] 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.

[11] Advanced Micro Devices, Inc. ATI Intermediate Language (IL) Specification, Dec 2009.

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

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

[14] S Williams, L Oliker, R Vuduc, J Shalf, K Yelick, J Demmel, Optimization of sparse matrix–vector multiplication on emerging multicore platforms, Parallel Computing, Volume 35, Issue 3, March 2009, Pages 178-194.

[15] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, Huazhong Yang, FPMR: MapReduce framework on FPGA, FPGA 2010.

[16] Kuen Hung Tsoi, Wayne Luk: Axel: a heterogeneous cluster with FPGAs and GPUs. FPGA 2010.