CS 515 Programming Languages and Compilers I

Project 1: Sparse Matrix Vector Multiplication on GPU

In this project, you will parallelize sparse matrix vector multiplication (*spmv*) on GPU. SPMV is arguably the most important operation in sparse linear algebra [1]. You will exploit fine-grained parallelism in spmv and implement your algorithm in CUDA.

Your submission includes both code and report: the code comprises of 60% project 1 grade and the report comprises 40% of project 1 grade. Your code needs to compile and execute on a given set of *ilab* machines (listed in the second half of the project description). You will receive **0 credit** if we cannot compile or run your code on the given set of *ilab* machines. You report needs to include the required experiment data presented in tables and figures, as well as any other interesting findings you made, for instance, the pros and cons of a particular implementation method.

1 Sequential Sparse Matrix Vector Multiplication

Let's review the sequential *spmv* algorithm first. Assume the input matrix is $A[]$, the input vector is $\vec{x}$, and the output vector is $\vec{y}$, $\vec{y} = A * \vec{x}$. We will use the coordinate format (COO) [1] for sparse matrix storage. The non-zero values are stored compactly in the **A_compact** array. The **coordinate** array stores the row coordinate and column coordinate for every non-zero element in the **A_compact** array. Let the number of non-zeros in the sparse matrix A be K, the number of rows of A be m. Listing 1 shows the sequential implementation of *spmv* in COO format.

Listing 1: Sequential SPMV

```
1 int i = 0;
2 memset(y, m*sizeof(y[i]), 0);
3 for (i = 0; i < K; i++) {
4     x_coordinate = coordinate[i].x;
5     y_coordinate = coordinate[i].y;
6     y[x_coordinate] += A_compact[K] * x[y_coordinate];
7 }
```

2 Use Segment-Scan in Parallel SPMV Implementation

You will need to implement a parallel version of SPMV based on the segment scan algorithm [7] we discussed in class. In this project, you are required to implement segment scan within a thread warp (the thread warp size is 32).

The segment scan algorithm is based on placing predicate guard on a prefix sum algorithm. The prefix sum operation is a *scan* with the binary operator as addition. Given a sequence of numbers of X = { x_0, x_1, x_2, ..., x_n}, the prefix sum of X is a sequence numbers of Y = {y_0, y_1, y_2, ... y_n} such that: $y_0 = x_0$, $y_1 = x_0 + x_1$, $y_2 = x_0 + x_1 + x_2$, and so on. The prefix sum operation can be performed within logarithmic **parallel** time steps [2]. The segment scan operation applies predicate guard on the scan operation such that the entire input sequence is divided into a set of segments and we perform prefix sum on each segment. For instance, assuming every x_i value is associated with a key value k_i. Prefix sum is performed only for the consecutive x_i values associated with the same key value. Assume the key values are $k_0 = 0$, $k_1 = 0$, $k_2 = 0$, $k_3 = 1$, $k_4 = 1$, $k_5 = 2$, $k_6 = 2$, $k_7 = 2$, the segmented scan result for the X sequence is Y = { $y_0 = x_0$, $y_1 = x_0 + x_1$, $y_2 = x_0 + x_1 + x_2$, $y_3 = x_3$, $y_4 = x_3 + x_4$, ..., $y_7 = x_5 + x_6 + x_7$ }.

In the context of sparse matrix vector multiplication, the predicate guard is the row coordinate for every non-zero element in the matrix. Only the multiplication value $A[i][j] * x[j]$ ($j = 0..n$) corresponding to the i-th row is added to $y[i]$.

Listing 2: Segment Scan Based SPMV Parallelization

```
1  __device__ void
2  segmented_scan(const int lane, const int * rows, float * vals)
3  {
4    // segmented scan in shared memory, assuming corresponding A values
5    // are loaded into the shared memory array vals, the row indices loaded
6    // into rows[] array in shared memory
7    // lane is the thread offset in the thread warp
8
9          if ( lane >= 1 && rows[threadIdx.x] == rows[threadIdx.x - 1] )
10                vals[threadIdx.x] += vals[threadIdx.x - 1];
11         if ( lane >= 2 && rows[threadIdx.x] == rows[threadIdx.x - 2] )
12                vals[threadIdx.x] += vals[threadIdx.x - 2];
13         if ( lane >= 4 && rows[threadIdx.x] == rows[threadIdx.x - 4] )
14                vals[threadIdx.x] += vals[threadIdx.x - 4];
15         if ( lane >= 8 && rows[threadIdx.x] == rows[threadIdx.x - 8] )
16                vals[threadIdx.x] += vals[threadIdx.x - 8];
17         if ( lane >= 16 && rows[threadIdx.x] == rows[threadIdx.x - 16] )
18                vals[threadIdx.x] += vals[threadIdx.x - 16];
19 }
```

We provide the pseudocode for warp-level parallel segment scan in Listing 2. Note that the above code is only for 32 consecutive non-zero elements. In this example, each thread is processing one non-zero value, thus in Listing 2, 32 values are processed by all threads in one warp. Note that there is implicit synchronization between threads within a single warp: the threads within a warp run in single instruction multiple data (SIMD) manner, that is, all 32 threads within the same warp run in lock-step manner – executing one instruction at one time, a thread cannot move on to the next instruction if other threads haven't completed current

instruction. If you want to perform segment scan at a larger thread granularity, for example – a thread block, you will need to use explicit barrier instructions, i.e., *__synchthreads()*.

Work Distribution The first question that comes to parallel programming is how to divide the workload and assign them to each thread. In this project, you will need to divide the list of non-zero elements from sparse matrix A evenly into every thread warp. Assume we have K non-zero elements and W thread warps. Each thread warp is assigned $\lceil K/W \rceil$ non-zeros (except the last warp, since K may not be a multiple of W). The range for the indices of the non-zeros for the i-th thread warp is $[\lceil K/W \rceil * i$, min($\lceil K/W \rceil * (i+1)$, K)). Note that, one warp process 32 elements at one time, thus, you will need to write a loop to process all the non-zero elements one thread warp is assigned to.

Segment Boundary Our code let every thread warp handle 32 elements at one time. It is possible that one row has more than 32 non-zero cells. Therefore, it is possible for one matrix row be handled by multiple thread warps. To ensure the correctness of the program, we use atomic operations. In modern computer architecture, a load/store instruction is typically atomic, that is, a load or store instruction can be executed without being interrupted. However, to ensure a three-instruction operation – *read-modify-write* atomic, that is, executing this three-instruction operation without being interrupted by another thread's access to the same memory location, we will need to use special hardware supported *atomic instructions*.

To write the partial sum result (processed by one thread warp) back to a particular location in the output vector $\vec{y}$, you will need to use atomicAdd instruction in CUDA. Further more, within a thread warp, not all the 32 threads need to write the partial sum result back to $\vec{y}$. To accomplish this, you can let every thread (with thread id tid) check its immediate neighbour thread (with thread id $tid + 1$ if the thread is not the last thread in the current warp). If the immediate neighbour thread maps to a different row coordinate, then the current thread is processing an element at the end of a segment, it can write the segment scan element it is in charge of back to the corresponding location in the vector $\vec{y}$. The last thread in the warp always need to write back to a location in the vector $\vec{y}$.

Preprocessing The segment scan method requires that the non-zero elements be divided into segments and each segment correspond to a row. Therefore, the non-zero elements that correspond to the same row need to be placed consecutively in the one-dimensional array **A_compact** in memory. In your code, you will need to write a procedure that preprocess the list of non-zero elements from the input

matrix and place them into the format that segment scan requires. This procedure can be implemented in CPU. You can also implement it in GPU, but it is optional.

Shared Memory You are required to use shared memory for segment scan. Both **vals** and **rows** in Listing 2 should be shared memory arrays.

3 Optimize SPMV Algorithm (Optional)

This part of the project involves solving an open-ended research problem. If you are interested and also have more time to explore, you can try to complete the optimization component. This component requires a lot of thinking, coding, and reading. It may take non-trivial amount of time. It is completely okay if you do not finish this part. Finally, I strongly encourage you to read through this section whether you choose to implement it or not.

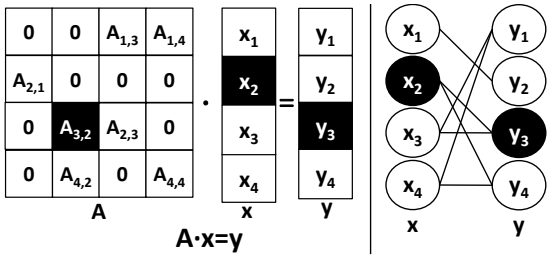

Figure 1: An Example of Data Reuse in SPMV

The implementation in Section 2 only balances the workload, but not minimizes data reuse. To understand data reuse, we show an example in Figure 1. In the bipartite graph on the right side of Figure 1, every node represents an input vector element or a output vector element. There is an edge between x[i] and y[j] if there exists an non-zero entry A[j,i] in sparse matrix A such that A[j,i] is multiplied by x[i] and the result is added to y[j]. For instance, $A_{3,2}$ is associated with x_2 ($A_{3,2} * x_2$) and y_3 += $A_{3,2} * x_2$. From the bipartite graph, we can see that x_2 is used twice, while x_1 is used once. Therefore, if we load x_2 from global memory to shared memory, we can reuse it in shared memory during program execution. Since shared memory is typically orders of magnitude faster than global memory, loading it from global memroy once and using it twice in shared memory can significantly improve data access performance.

Since we need to ensure load balancing – every thread block gets the same number of non-zero elements to process, when we try to optimize data reuse within one thread block (shared memory is visible only to the threads in the same thread

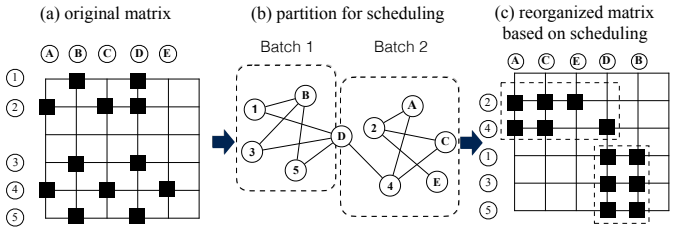

Figure 2: Load Balancing and Maximizing Data Reuse

block), we have the constraint of load balancing. We show an example in Figure 2, in which case we divide the 12 non-zero elements into two thread blocks (assuming each thread block processes 6 non-zero elements), the partition strategy shown in Figure 2 (b) gives the optimal solution, only x_D needs to appear in two different thread blocks' shared memory, every other data nodes can be reused completely within its own thread block. This partition strategy divides the workload evenly and in the meantime ensures data reuse is maximized within its own thread block and minimized across different thread blocks.

Before we start running a program, we can preprocess the list of non-zeros and determine the best work partitioning strategy for data reuse or other performance/energy goals. During program execution, we apply the work partition result. This approach falls into an important class of program optimization techniques – the inspector-executor method. The inspector phase determines the best computation reorganization schedule and/or data reorganization schedule. The executor is a piece of transformed code that takes the computation/data reorganization schedule and applies it during program execution.

In our parallel *spmv*, the inspector and executor is defined as follows: first we divide the list of m non-zeros from the sparse matrix into K partitions, such that each partition has m/K non-zeros. The partition algorithm should also try to minimize data reuse as much as possible. Next, we place the non-zeros that are in the same partition consecutively in logical memory. We will have the logical memory layout that looks like this:{all non-zeros for 1st partition, all non-zeros for 2nd partition, ...}. Within every partition, we reorder the non-zeros based on their row indices. The row coordinate and column coordiante arrays need to be reorganized with respect to the reorganization of the non-zero elements. The set of all partitions are then evenly distributed to all thread blocks. One thread block process one partition at one time. Therefore, the partition size (the number of non-zeros) need to be at least the same as or larger than the thread block size.

In the executor code, we need to preload the data that can be reused into shared

memory first. In *spmv*, the reusable data objects are the elements from the input vector and the output vector. Since we reorganize non-zeros within each partition by rows and we use the segment scan method (which already reuse the data object from the output vector in shared memory well) described in Section 2, we will only need to place the input vector items into shared memory. We can pre-load all the the input vector items of interest for all threads in the same thread block into shared memory before the computation starts. You will need to maintain a mapping between the global memory address and the shared memory address, such that every thread block can operate on the right pre-loaded data object in shared memory during *spmv* computation phase. The simplest approach is to maintain an array of the size as the number of non-zeros, and every element of this array corresponds to the shared memory location of x[j] assuming the non-zero element is A[i,j]. Thus when reading the list of non-zero elements from global memory in a coalesced way, the location of the input vector element in shared memory can be read in a coalesced way. You will also need to maintain a mapping between every partition and its list of used input vector elements. All of this can be done in CPU. For the *spmv* computation, you can use the same algorithm as described in Section 2, except that you will use input vector element from shared memory. The pseudocode for the executor is listed in Listing 3.

Listing 3: Executor Code for Parallel SPMV

```
1  void spmv_executor(parameter list)
2  {
3      local_block_work = get_nonzeros_list(blockid, all_non_zeros);
4      for all partitions p in local_block_work) {
5        input_vector_element(p) -> shared_memory;
6        __synchthreads();
7        local_warp_work = get_nonzeros_list(warpid, p);
8        performSPMV(local_warp_work);
9      }
10
11 }
```

Extra-Credit Components: You will get extra-credit for achieving the following goals.

- Implement only the inspector phase on CPU (regardless of the overhead) and improve the *data reuse* metric by 10% for at least one real-world sparse matrix with at least 1 million non-zeros. (5% Extra-Credit)

  The metric is formally defined as follows.

  Definition 1. *Assume we have a graph $D = (V, E)$ with the set of vertices V and the set of edges $E \subset \binom{V}{2}$. Let n represent the number of vertices, m represent the number of edges, and k represent the number of partitions.*

Let L_i denote the number of edges in partition i. Assume x is a valid k-way balanced edge partition for D. $\forall v \in V$, assume v's incident edges are placed into p_v distinct partitions in x. As if the vertex v was copied into p_v partitions, we define the vertex copy cost for v as $p_v - 1$. We optimize the total vertex copy cost $C^{EP}(x)$:

$$\begin{array}{rl} \min & C^{EP}(x) = \sum_{v \in V}(p_v - 1) \\ \textit{s.t.} & \forall i \in [1...k] \;\; L_i(x) = \frac{m}{k} \\ & x \; is \; a \; valid \; k - way \; edge \; partitioning \end{array} \quad (1)$$

- Implement both the inspector and executor phases. Improve the GPU performance for at least one real-world matrix for at least 1.1X speedup. You will need to identify which sparse matrix it is and where we can download it for our verification purpose. (10% Extra-Credit)

Ideas for implementing the task partitioner: To partition sparse matrix for best data reuse, there are two major types of approaches. One is tiling, as illustrated in Figure 1 (c). You can partition the sparse matrix into multiple tiles (a tile is a rectangular or space region in the 2-dimensional matrix space). The tiling approach is a common approach used for dense matrix multiplication. In dense matrices, the tile size is typically fixed. However, in sparse matrix, the tile size does not have to be fixed since the number of non-zero elements is not necessarily the same in every equal-size tile. The sparse tiling approaches have been studied in different ways [8, 5, 4]. The second type of approach is the graph edge partition approach, as illustrated in Figure 1 (b). Recent graph edge partition approaches either adopt a weighted-vertex heuristic [3] or a split-and-connect heuristic [6]. You can implement a method that appeared in a previous paper (either the tiling or the graph partition method). However, your method does not have to be restricted to the work that has appeared in literature. You can investigate the previous approaches, understand their pros and cons, and develop your method.

4 Compilation and Execution

Compilation: There is a Makefile in the sample code package. However, if you want to add new files. Please make sure your Makefile is updated. Please do not change the final executable name – spmv. The grader will compile your code with the following commands:

- make spmv should compile everything into a single executable called **spmv**. The usage will be described in execution interface.

- make clean should remove all generated executable and intermediate files.

You need to add command line option for different input paremeters. For instance, the total number of thread blocks and the size of each thread block are specified as command line parameters. We assume every thread block is one-dimensional. The detailed command line options are listed below.

- -mat [matrixfile], input matrix file name.
- -ivec [vectorfile], input vector file name.
- -alg [approachOptions], this specifies which approach to be used for running spmv code. It can be one of the following:

  -alg segment, use the required segment scan implementation

  -alg design, use the version of spmv designed by you
- -blksize [threadBlockSize], this specifies the thread block size to be used.
- -blknum [threadBlockNum], this specifies the number of thread blocks to be used.

We provide a sample code package for you to start with. It is placed here: */ilab/users/zz124/cs515_2017/projects/proj1/code_sample*. The sample code package provided to you include some implementation for parsing and evaluating the command options. Feel free to modify them.

Input Vector: You need to generate your own input vectors for testing purpose. The input vector file should be a text file starting with the size of the input vector and followed by a list of values. The number of values should be the same as the input vector size. Check the file mmio.c and look for "read_vector_file" procedure for more details. We have provided a vector generator script for you in the sample code package: vector_generator.sh . It takes the size of vector as input and generates a random input vector of the requested size. It prints the vector values to std out and you need to redirect it to a file.

Here is an example:

vector_generator.sh 5 > vecOfLength5.txt.

Program Output: The program output at standard out needs to report the kernel execution time in the following format.

"The total kernel running time on GPU [gpuDeviceName] is xxxx milli-seconds".

We provided sample timing function in the code package. Feel free to modify it and use any other type of timing function (as long as it is accurate).

The program also needs to output a vector file called "output.txt" which gives the $y = A * x$ vector value in plain text file – y[0], y[1], y[2], and so on. We will use this vector to test the correctness of your implementation.

Correctness Test : Your code need to generate correct sparse matrix vector multiplication result. You can compare the output of your GPU program with the output of a CPU program that also implements sparse matrix vector multiplication. We have provided a sample CPU *spmv* implementation and a tester in the code package. Feel free to adapt it to print out more detailed debugging information.

Machine to Use: Currently the ilab machines have two types of CUDA compatible GPUs. One type of GPU is GT 630, with 2GB Memory, 192 CUDA cores, CUDA capability 3.0, the other of GPU is GT 730, with 2GB Memory, 384 CUDA cores, CUDA capability 3.5. Please find the list of machines here: http://report.cs.rutgers.edu/mrtg/systems/ilab.html

We will compile and test your code on these ilab machines only.

Performance Debugging: You can use *nvprof* to check various performance metrics for a given CUDA program. You can run it from command line, i.e., "nvprof –metrics l2_l1_read_hit_rate a.out [paramlist]" will give you the l2 hit rate for the program a.out running on the input parameters [paramlist]. More information about nvprof is here: docs.nvidia.com/cuda/profiler-users-guide/

5 Input Matrix List

We provide more than 10 sparse matrices for you to test your code. For grading purpose, we will use 5 to 10 additional hidden test cases. Make sure your code is well thought and well designed. These matrices can be found either in matrix market (http://math.nist.gov/MatrixMarket/) or Florida matrix collection (http://www.cise.ufl.edu/research/sparse/matrices/). These matrices and their links are provided in a text file in the project code package.

You can use other sparse matrices from these two collections to test your code and reason about the pros and cons of your implementation.

We will only use input matrices in matrix market format. Please make sure your program accepts matrix market format. We provide matrix I/O functions in the sample code package.

6 What to Submit?

You will need to submit a package called spmv_gpu.tgz including all the source files, the Makefile, as well as a README for any special instructions on how to compile and run your program. You will also need to submit a separate report called spmv_report.pdf that describes the experiment data and performs the data analysis for your implemented SPMV algorithms. The report needs to have at least two pages, not including references. Please use the latex or word template from here: https://www.usenix.org/conferences/author-resources/paper-templates. Please do not modify the template format.

There are two deadlines: one for the code, and the other for the report (after the code package is submitted). Please check Sakai page for the deadline.

Report: Your report needs to include at least the following tables and figures that summarize the performance of your *spmv* implementation.

- Two tables (one for each type of GPU) that lists the performance for each sparse matrix on the following block number and block size combination: (10, 192), (8, 256), (5, 384), (4, 512), (2, 768), (2, 1024), as well as the pre-processing time for reorganizing the non-zero elements to achieve a format which the segment scan method accept.

- For each sparse matrix, generate two figures (one for each GPU architecture on ilab) for the performance over different thread block number. The block number should start from 1 and end at the maximum possible thread block number the hardware allows (or the maximum possible thread block number with respect to the maximum thread number the hardware allows). Use thread block size 256.

References

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[2] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[3] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465, New York, NY, USA, 2014. ACM.

[4] Peng Di and Jingling Xue. Model-driven tile size selection for doacross loops on gpus. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 401–412, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, July 2006.

[6] Lingda Li, Robel Geda, Ari B. Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario Szegedy. A simple yet effective balanced edge partition model for parallel computing. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):14:1–14:21, June 2017.

[7] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[8] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, January 2011.