CS 516/415 Programming Languages and Compilers II

Project 1: Parallelizing SPMV on GPU

In this project, you will implement three parallel versions of sparse matrix vector multiplication (spmv) on GPUs. SPMV is arguably the most important operation in sparse matrix computations [1]. You will exploit fine-grained parallelism in SPMV and write a GPU program with the parallel programming interface called CUDA.

You need to submit both code and report. The code comprises 60% and the report comprises 40% of project 1 grade. In your report, you will need to present the evaluation results of three different implementations as well as the strengths and weaknesses of each implementation. You can also report any new findings that you have made and that can potentially further improve sparse linear algebra application in many-core GPU architecture. We will provide 15 sparse matrices that are obtained from real applications. You need to evaluate your implementations on each sparse matrix and report the results. NOTE: You will receive 0 credit if we cannot compile/run your code or your code fails correctness tests. Grading will be done on *ilab machines*. Please make sure your code can compile and run on ilab machines.

The interface of the implemented code needs to conform to the requirements defined in Section 4. We have provided sample code for you to start with. The following three sections describes the three different versions of spmv you need to implement. Have fun!

1 Simple Atomics Based SPMV

The first spmv version you need to implement is the *simple atomics* based approach. In this approach, different threads can process the same matrix row and communicate by atomic *read-modify-write* instructions. In modern computer architecture, typically a load/store instruction is atomic, that is, a load or store instruction can be executed without being interrupted from another load or store instruction.

However, to ensure a three-step operation – *read-modify-write* atomic, that is, execute this three-step operation without being interrupted by any other load/store to the same memory location, you will need to use special hardware instruction called *atomic instruction*.

Listing 1: Sequential SPMV

```
1  int i, j;
2  for (i = 0; i < M; i++) {
3      y[i] = 0;
4      for (j = rowBeg[i]; j <= rowEnd[i]; j++) {
5          int tmp = col[j];
6          y[i] += A_val[j] * x[tmp];
7      }
8  }
```

To illustrate the atomic based approach, let's review the sequential SPMV code in Listing 1. The input matrix is A, the input vector is x, and the output vector is y such that $y = A * x$. Every loop iteration process a row, assuming it is the i-th row, we perform dot product of row i and the input vector x, and store it into $y[i]$. In above code, A_val is the list of non-zero values in matrix A. Since addition is both commutative and associative, the order on how the multiplication results are added to the output vector $y[i]$ does not matter.

We can let multiple threads process the same row i, and add the multiplication result to the memory location $y[i]$ using *atomicAdd* operations. The three steps: the read to $y[i]$, the modification to $y[i]$, and the write to $y[i]$ happen in one atomic step.

Listing 2: Atomic Based SPMV Parallelization

```
1  __global__ void spmv_atomic_kernel
2                      (const int nnz,
3                       const int * coord_row,
4                       const int * coord_col,
5                       const float * A,
6                       const float * x,
7                             float * y)
8  {
9          int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
10         int thread_num = blockDim.x * gridDim.x;
11         int iter = nnz % thread_num ? nnz/thread_num + 1: nnz/thread_num;
12
13         for ( int i = 0; i < iter; i++ )
14         {
15                 int dataid = thread_id + i * thread_num;
16                 if ( dataid < nnz ) {
17                         float data = A[dataid];
18                         int row = coord_row[dataid];
19                         int col = coord_col[dataid];
20                         float temp = data * x[col];
21                         atomicAdd(&y[row], temp);
22                 }
23         }
24 }
```

We show the pseudo code in Listing 2. Every thread is assigned more or less

the same number of multiplication operations and the communication between different threads is accomplished using the atomic instructions.

There will be an ordering between different threads' *read-modify-write* operations. Though the ordering might be non-deterministic from run to run, no thread will interrupt another thread's $read - modify - write$ process.

You need to add an command line option as the thread block size so that users can try different thread block size to test your program. You also need to add a command line option for the total thread block number. We assume every thread block is one-dimensional. More details about the command line option format can be found in Section 4.

When you test your program, you can set the total number of threads based on the total number of non-zeros, or the maximum number of active threads the GPU can support. To obtain the maximum number of active threads for a given GPU kernel on a given architecture, you can use the CUDA occupancy calculator:

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Make sure the total thread block number and the thread block size do not exceed the maximum number of threads the CUDA device allows. The details can be found at CUDA programming guide [5].

2 Segment Scan Based SPMV

The second version of spmv you need to implement is the segment scan approach [11], which is based on parallel prefix sum [2]. It performs segment scan within a thread warp (thread warp size is 32). Threads within a warp run in single instruction multiple data (SIMD) manner.

The (inclusive) prefix sum is an operation that scans a sequence numbers of x_0, x_1, x_2, ..., x_n, and obtain a second sequence numbers of y_0, y_1, y_2, ... y_n such that: $y_0 = x_0$, $y_1 = x_0 + x_1$, $y_2 = x_0 + x_1 + x_2$. Similar to the summation reduction operation, the prefix sum operation can be performed within logarithmic time steps [2].

The segment scan algorithm is based on prefix sum algorithm. It adds predicate guard to determine which values should be added at every parallel step of prefix sum. For instance, assuming every x_i value is associated with a key value k_i. Prefix sum is performed only for the x_i values associated with the same key value. For example, assume the key value is 0,0,0,1,1,2,2,2 for x_i, $i = 0..7$, the segmented scan result for the above example would be $y_0 = x_0$, $y_1 = x_0 + x_1$, $y_2 = x_0 + x_1 + x_2$, $y_3 = x_3$, $y_4 = x_3 + x_4$, ..., $y_7 = x_5 + x_6 + x_7$.

In the context of sparse matrix vector multiplication, the predicate guard is the row index for every non-zero value in the matrix. Only the multiplication value

corresponding to the row i need to be added to $y[i]$.

Listing 3: 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 parallel segment scan in Listing 3. Note that the above code is for one iteration of segment scan within a thread warp. In this example, each thread corresponds to one data value, thus in Listing 3, 32 values are processed. Also note that there is implicit synchronization between threads within a single warp, this is due to the fact that 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()*.

It is possible that one row has more than 32 non-zero cells. You may need to do segment scan multiple times. You can use a loop, and be sure to carry the partial summation value to the next iteration if a row is processed by multiple iterations of the same warp.

You need to write the summation result back to $y[i]$. To accomplish this in parallel, you can check the immediate neighbour thread (the one that has a thread id as current thread's id + 1). If the immediate neighbour thread maps to a different row index, then the current thread can write the segment scan value to $y[i]$ in memory, otherwise this thread is idle. You need to consider the boundary case too, i.e., the last thread in the warp does not have an immediate neighbour thread in the same warp such that the neighbour's thread index is current_threadId + 1.

In the segment scan version, every thread warp will be assigned more or less the same number of non-zeros to process. In class, we give an example of how to partition the workload to different warps and how to handle boundary cases.

NOTE: You are required to use shared memory for segment scan. Both vals[ ] and rows[ ] in Listing 3 should be shared memory arrays.

On GPUs, there is software cache, which you can allocate and manage explicitly. It is named as *shared memory* [5]. You can allocate shared memory dynamically or statically. Below are two examples for dynamic allocation and static allocation.

Dynamic shared memory allocation: .

```
1 Kernel<<< gridDim, blockDim, smem_size>>>(parameter_list)
2 __global__ void Kernel(parameter_list)
3 {
4         extern __shared__ int a[ ];
5   ......
6 }
```

The *smem_size* variable specifies the amount of shared memory in bytes for every thread block. "a" points the allocated shared memory region. Currently all thread blocks can only be assigned the same shared memory size, even if some thread blocks do not use all the shared memory.

Static shared memory allocation:

```
1 __global__ void Kernel(parameter_list)
2 {
3         __shared__ int a[100];
4 }
```

In the above case, 100*4 bytes are the shared memory size allocated for every thread block. In static allocation, the size of shared memory needs to be known at compile time, that is, before the program gets executed. However, dynamic allocation is not restricted, the size of shared memory to be allocated can be determined at runtime.

3 Your Own SPMV Design

In the third version of spmv, you will implement your own SPMV. It can be a new algorithm, an optimization upon the two aforementioned approaches, or an performance tuning approach, for example, input-adaptive algorithm-selection (which approach to choose for a given input). We will test your own design version for a class-wide performance competition.

We provide a potential idea below. It may or may not be the best design. You don't have to implement exactly the same idea. This is an idea to get you started and think more.

Control-Divergence Minimization In a GPU program, threads in the same warp execute the same instruction at one time. If there is control flow statement, for

instance, a "if (condition) ... " statement in the code, assume only 10% of the threads has evaluated the "condition" to be true, the thread warp needs to execute this branch, with 10% threads (processing cores) active and 90% threads (processing cores) idle, wasting 90% of the computation power. This is called *thread divergence* [13] as discussed in class.

In the *segment scan* approach, the divergence mainly comes from the number of logarithmic steps every thread has to execute. In Listing 3, depending on the number of non-zeros of each row in the 32 values being processed, a thread may execute 1 to 4 steps at lines 9 to 18. For instance, in the 32 values to be processed by one warp, if 15 values come from 15 different rows in matrix A (one in each row), and other 17 values come from another one row, 15 threads only execute up to statement at line 10 (although condition checking is necessary for all four steps), while other 17 threads need to execute up to statement at line 18.

To minimize thread divergence, you can reorder the list of non-zero elements to be assigned to different threads, so that the rows that are processed by the same thread warp at one iteration have more or less the same number of non-zero elements.

The arrays, coord_row[ ] coord_col[ ] in Listing 2 implicitly denote the sequence of coordinate pairs to be consecutively assigned to threads in ascending thread index order, for instance the non-zero element at co-ordinate – coord_row[0] (row id), coord_col[0] (column id) is assigned to thread 0. If we change coord_row[ ] and coord_col[ ] to coord_row_prime[ ] and coord_col_primte[ ] and use the same GPU code to process the permuted list of non-zero elements, we can change how tasks are assigned to threads in the same warp. In the meantime, you will need to change A_val[ ] to A_val_prime[ ] according to the permuted new non-zero elements order so that the segment scan code remains the same.

Transforming A_val[ ] to A_val_prime[ ] improves *memory coalescing*. In modern computer architecture, one memory transaction loads a consecutive memory chunk, usually in the unit of 128 bytes. If the data used by the same thread warp fall into the same segment or as few memory segments as possible, the memory bandwidth can we well utilized. By changing A_val[ ] to A_val_prime[ ], consecutive threads still access A_prime[ ] elements consecutively, that is, thread 0 access A_prime[0], thread 1 access A_prime[1], and so on.

Below is a detailed algorithm for permuting the coordinates to minimize thread divergence. Assuming every thread warp corresponds to a set that is empty at the beginning

Coordinate List Permutation:

1. For every row's non-zero elements, divide them into no more than two com-

ponents, one component has a multiple of 32 nonzeros, the other component has less than 32 elements – we name it as remainder component. The component with a multiple of 32 elements for all rows can be placed first in the work list, row by row.

2. For all rows that have a remainder component, further divide the remainder into no more than two components: one has a multiple of 16 non-zero elements, the other component has less than 16 non-zero elements. Now place the 16-element non-zero components into the work list (row by row).

3. For remainders of Step 2, we further split every reminder into two components, one has a multiple of 8 non-zero elements, the other has less than 8 non-zero elements. Then we place the 8-element components into the work list.

4. Repeat the above process until a remainder cannot be decomposed any more. You can also stop at an earlier threshold, for instance, when remainder size is less than 8. You can try different threshold and see what gives best performance. In the end, we place the remainders into the work list one by one.

The transformation of coord_row to coord_row_prime and from coord_col to coord_col_prime, as well as from A_val to A_val_prime can be done on CPU. You can copy the reordered arrays to GPU once the transformation is done. And the GPU kernel uses the new coordinate and A values.

Finally, in this implementation, assume you do not have to worry about the overhead, see what is the best performance improvement you can obtain. You only need to report the total kernel running time.

4 Compilation and Execution

We provide a sample code package for you to start with. It is placed here: */ilab/users/zz124/cs516_2017/proj1/code_sample*.

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.

Run Your Code: Your code will need to take different number of command line options defined as follows:

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

  -alg atom, use the simple atomics version

  -alg scan, use the segment scan version

  -alg design, use the approach you designed
- -blockSize [threadBlockSize], this specifies the thread block size to be used.
- -blockNum [threadBlockNum], this specifies the number of thread blocks to be used.

The sample code package provided to you have some code for parsing and evaluating the command options. Feel free to reuse them or modify them.

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 use any other type of timing function that you consider as accurate.

The program also needs to output a vector file called "output.txt" which lists the $y = A * x$ vector values. We will use this vector to test the correctness of your implementation.

Machine to Use: Currently only the ilab machines in Hill 120 have CUDA compatible GPUs. The GPUs are GT 630, with 2GB Memory, 192 CUDA cores, CUDA capability 3.0, CUDA driver version: 7.5. 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

These 15 sparse matrices will be used to test your spmv implementations. They can be found either in matrix market (http://math.nist.gov/MatrixMarket/) or Florida matrix collection (http://www.cise.ufl.edu/research/sparse/matrices/). If you search with the keyword of the matrix name and these two collection names on a web search engine, you should be directed to these matrices. 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 test these 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.

The matrices are as follows: *cant, circuit5M_dc, consph, FullChip, mac_econ_fwd500, mc2depi, pdb1HYS, pwtk, rail4284, rma10, scircuit, shipsec1, turon_m, watson_2, webbase−1M*

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 report called spmv_report.pdf that describes the pros and cons of each method. The report needs to have at least six 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.

Please do not include any sparse matrix files in your submission. If you use any other matrix not in the 15 specified sparse matrices and described it in your report, please specify the source (i.e., a web link).

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] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[5] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[6] Ian J. Egielski, Jesse Huang, and Eddy Z. Zhang. Massive atomics for massive parallelism on gpus. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 93–103, New York, NY, USA, 2014. ACM.

[7] George Karypis and Vipin Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[8] Lingda Li, Ari B. Hayes, Stephen A. Hackler, Eddy Z. Zhang, Mario Szegedy, and Shuaiwen Leon Song. A graph-based model for GPU caching problems. *CoRR*, abs/1605.02043, 2016.

[9] Yixun Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, May 2009.

[10] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 140–152, New York, NY, USA, 1988. ACM.

[11] 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.

[12] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '15, pages 521–532, New York, NY, USA, 2015. ACM.

[13] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA, 2011. ACM.