CS 314 Principles of Programming Languages

Project 4: Parallelizing SPMV on GPU

Due Date: 1/2/2017, Monday, 5:55pm EST

No late submission will be accepted!

In this project, you will implement five parallel versions of sparse matrix vector multiplication (spmv) on GPUs. SPMV is arguably the most important operation in sparse matrix computations [1]. In project 3, you are already familiar with the sparse matrix vector multiplication algorithm and you have parallelized it on CPU with OpenMP pragmas. In this project, you will exploit fine-grained parallelism and learn how to program GPUs using a shared-memory threading model called CUDA.

The following three sections describes the five different versions of spmv and potential optimization techniques in CUDA. You will also need to write a report that describe the algorithms you implemented, compare them, and discuss any interesting findings you have made in this project.

You need to submit both code and report. The code comprises 70% of the overall grade, and the report comprises 30% of the overall grade. You may receive additional 50% credit if your code runs the fastest among all submission or runs better than the current main-stream spmv libraries cusp and cusparse or has made a new finding that shed light on future spmv optimization in many-core architecture.

The implemented code needs to conform to the API interface defined Section 4 in this project description. You will receive zero credit if we cannot compile or run your code on ilab machines. Since the grader has less than one day to grade your project, this policy will be strictly enforced.

Have fun!

1 Simple Segment Scan and Atomics Based Approach

1.1 Simple Atomics Based Approach

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 a computer architecture, typically a read or write operation is atomic, that is, a read instruction or write instruction can be executed without interference from another read or write instruction.

However, to ensure these three-step operation – *read-modify-write* be executed atomically, that is, without being interrupted by any other read/write to the same memory location, you will need to use special hardware supported instruction called *atomic instructions*.

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 a reduction type operation, which is both commutative and associative, the order on how the multiplication results are added to the output vector $y[i]$ does not affect he correctness of the program output.

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

```
__global__ void spmv_atomic_kernel
                    (const int nnz,
                     const int * coord_row,
                     const int * coord_col,
                     const float * A,
                     const float * x,
                           float * y)
{
        int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
        int thread_num = blockDim.x * gridDim.x;
        int iter = nnz % thread_num ? nnz/thread_num + 1: nnz/thread_num;

        for ( int i = 0; i < iter; i++ )
        {
                int dataid = thread_id + i * thread_num;
                if ( dataid < nnz ) {
                        float data = A[dataid];
                        int row = coord_row[dataid];
                        int col = coord_col[dataid];
                        float temp = data * x[col];
                        atomicAdd(&y[row], temp);
                }
        }
}
```

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 atoimc 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. The format of the input option is "-blocksize 128" if the user needs to try thread block size 128. More details about the input parameter format can be found in Section 4.

You can set the total number of threads for every given matrix. It can be based on the total number of non-zeros in the input matrix, 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

You can also set the total thread number to any other number that you deem reasonable. Make sure it does not exceed the maximum number of threads the CUDA device can support. The details can be found at CUDA programming guide [5].

1.2 Simple Segment Scan Based Approach

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

The 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 used for prefix sum. For instance, assuming every x_i number 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 condition guard is the row index for every non-zero value in the matrix. Only the multiplication result 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 segment scan within a thread warp, and each thread corresponds to one data value, thus in the 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()*.

You may need to segment scan more than 32 values in a single thread warp. In this case you need to call this parallel segment scan described in Listing 3 multiple times using a loop, and be sure to carry the partial summation value to the next iteration if a row is processed across multiple iterations by the same warp.

Finally you need to write the summation result back to $y[i]$. To accomplish this in parallel, you can check the neighbour thread (the one that has a thread id that is current thread id + 1). If the neighbour thread maps to a different row index, then the thread of interest can write the multiplication value to $y[i]$ in memory, otherwise this thread does nothing. You need to consider the corner case too, i.e., the last thread in the warp does not have a 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 rows to process. The workload assignment is in the unit of thread warps, rather than individual threads.

1.3 Optimization for Control Divergence

In the third version of spmv you will combine segment scan and atomic communication approach. The simple atomic version may not yield the best performance result, however, it relaxes the constraint that one matrix row can only be processed by one thread or one thread warp. If combined with the segment scan approach, it may yield better performance than just using segment scan itself [6].

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 the "condition" as 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].

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 need to reorder the list of non-zero elements to be processed, 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. That is, the coordinate list needs to be reordered.

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.

By doing this, it also improves *memory coalescing*. In modern computer architecture, one memory transaction loads a consecutive memory segment, 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 utilization can be significantly improved. By changing A_val[ ] to A_val_prime[ ], consecutive threads access A_prime[ ] elements consecutively, that is, thread 0 access A_prime[0], thread 1 access A_prime[1], and so on. These two transformations are called *job swapping* and *data layout transformation* [13].

There are different ways to reduce thread divergence. You can try to design your own algorithm to minimize thread divergence. Below is a suggested 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 parts, one part has a multiple of 32 nonzeros, the other part has less than 32 elements. The part has a multiple of 32 elements can be placed into the empty thread warp sets from the first warp one by one. Repeat this for all all rows.

2. For all rows that have an non-empty second part, divide the remaining non-zero elements into no more than two parts, one part has 16 non-zero elements (if this part exists), the other part has less than 16 non-zero elements. Now we place the 16-element non-zero groups into any thread warp set that is not full yet. Again, we repeat this step for all rows.

3. We repeat similar step as Step 2, for every row, we divide its remaining non-zero elements into two parts, one part has a multiple of 8 non-zero elements. The other part has less than 8 non-zero elements.

4. The above process goes on and every time the dividing group size is reduced by half until we reach a preset threshold T. At this point, for all the remaining non-zeros, we place them into the rest thread warp sets that are not full, and apply the simple atomics based approach.

According to the work [6], the atomic based approach is more beneficial when the amount of reduction parallelism in segment scan is limited. In the above suggested approach, *step 4* is applied at a cutoff point where the benefits of segment scan cannot outweigh the overhead of segment scan (more binary instructions). A careful selection of T is necessary. In your implementation, please explore different T values and report their impact for different input matrices. Note that it might vary with respect to the input matrix in your report.

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. Assuming you do not have to worry about the overhead, check the best performance improvement you can gain.

In your report, please discuss the pros and cons of each approach. Describe what you have learned from this set of experiments and report the best parameter you have found, for instance, the best thread block size, the cutoff point described in the combined approach.

We provide a list of 20 sparse matrices, but you are welcome to use other sparse matrices to make your point, as long as you report results for these 20 required sparse matrices.

2 Cache Sharing Aware Approach

2.1 Locality Enhancement

In the four version of spmv, you will implement an approach that improves data locality for sparse matrix vector multiplication.

You will need to partition the non-zero elements into a number of different work lists. Each work list is assigned to one thread block. One thread block might be mapped to more than one work list. The non-zero entries within a work list are processed at the same time or within a short time period. Recall that in Lecture 26, if data reuse happens soon enough, it yields locality, that is, the data can still

be found in cache, and we don't have to load it from memory again. Cache access latency is usually a few cycles while DRAM access latency is hundreds of if not thousands of cycles.

In this locality enhancement version of *spmv* code, you will implement an approach to maximize data reuse within a work list and minimize data reuse across different work lists (for the work lists that are assigned to different thread blocks). The cache is fast but is typically much smaller than DRAM. Thus you need to carefully design the work list such that the amount of data to be reused within one work list fits into the cache.

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.

Within a thread block, you should use the segment scan based approach you implemented in Section 1, unless you find the atomic based approach is more beneficial in some cases. If you use segment scan based approach, you need to sort every single work list with the respect to the row indices, since the segment scan based approach relies on the fact that non-zero values on the same row are placed contiguously in memory for best computation efficiency.

If one row is split across different work lists in different results, you will need to use atomic addition to add the partial summation result (if you use segment scan based approach) to y[i] if the row id is i.

2.2 Inspector-Executor Optimization Model

To implement the fourth approach, first you need to implement the GPU kernel that takes the work lists and perform the corresponding multiplication and addition operations within every thread block. Recall that one work list is assigned to at most one thread block. However, one thread block can be mapped to multiple thread blocks. You will need to account for the case that multiple work lists be assigned to the same thread block.

You might need to maintain two arrays, one specifies the non-zero entries for every work list, from the first work list to the last list. You can store the global index (within the matrix A in compressed format) of the first non-zero entry in every work list, assuming that non-zero entries in the same work list are placed consecutively. In the other array, you need to specify, which work lists are assigned to which thread block, from the first thread block to the last thread block. Similarly, you can store the global index of the first work list in every thread block, again assuming the work lists in the same thread block are placed consecutively.

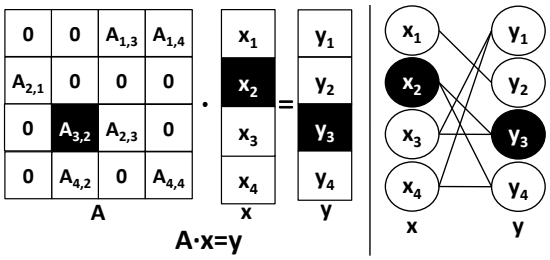

Figure 1: Data Reuse Example

In the GPU kernel, you will need to pre-load the data that will be reused. None of the data entries in the matrix A[ ] is reused, however the vector x[ ] and y[ ] are reused. An example is shown in Figure 2.2. For $x[\ ]$, x[2], x[3], x[4] are reuse while x[1] is not.

Before the computation in the kernel starts, you need to preload the reusable data for every thread block into shared memory, and you need to maintain the index of the pre-loaded data in shared memory so that if you use them later, you will use their shared memory index, rather than their global memory index.

The shared memory index and global memory index mapping can be done a

preprocessing step on CPU before the GPU kernel starts (you need to copy the mapping from CPU to GPU once it is done).

Since the original coordinate array maintains the global memory index for the input vector and output vector, you can build a new coordinate array coord_prime[ ] similar to coord[ ] and use it in the GPU kernel execution. You will still need to know the global memory index for every preloaded data in the preloading phase and for writing the result back at the end of each kernel.

Note that you only need to preload the input vector x[ ] if you use the segment scan approach within a thread block, since the output vector y[ ] is implicitly stored in shared memory using the segment scan method in Listing 3, in which the rows[ ] and vals[ ] arrays are stored in shared memory for fast processing. When estimating shared memory usage per block, you need to take the shared memory used in segment scan into consideration as well.

Once the computation is done, you need to store the results back to global memory, thus you need the original coord[ ] array as well. Recall that in the segment scan method, you can use one thread for one segment to store the partial prefix sum result back to the right location in y[ ] array using atomic instruction.

2.3 Design a Good Inspector Algorithm

Once the GPU kernel is ready for the locality enhancement approach, you will need to determine the best work list schedule. The transformed kernel is called executor [10] [12]. The code to determine the best work list schedule is called inspector [10] [12].

There are different approaches to partition the set of non-zero entries. The approach you need to implement is the graph partition approach. You can model the problem as a graph edge partition problem [8] [3]. You need to implement either the approach in [8] or the [3], which are currently two best balanced edge partition approaches we are aware of.

Every node in the graph represents a input vector entry or an output vector entry. 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]. An example is shown in Figure 2.2. Partitioning the edges is equivalent to partitioning the workload. The graph edge partition problem aims to minimize the number of nodes that have to appear in more than one partition, which is to minimize the data reuse across different work lists in our case.

You can perform a balanced edge partition such that every work list has exactly the same number of edges, also a multiple of the thread block size so that every thread gets the same amount of workload. However, sometimes load imbalance might yield better performance if memory system is the major bottleneck. An ex-

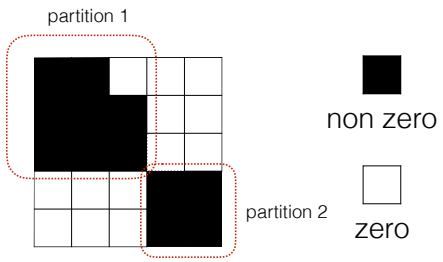

Figure 2: Load Imbalance Example

ample is shown in Figure 2, in which every data can be maximally reused within every work list if the work in different thread blocks is slightly imbalanced. However, load imbalancing might cause under-utilization of certain processing cores.

To overcome this problem, if there are enough work lists, we can schedule multiple work lists into the same thread block, such that eventually every thread block is assigned the same amount of workload.

To control constraints on workload (im)balancing, a typical graph partitioning tool such as METIS [7] would allow you to indicate a load imbalancing factor ϵ, i.e., every partition's workload is no more than (1+ϵ)*(TotalWorkload/#Partition) while the partition cost is minimized. You can explore different imbalancing factors ϵ for different sparse matrices.

Note that if you use any external library on ilab (or the one you installed by yourself on ilab), please make sure to include the full path in your Makefile. When test your program on ilab under a different account, your code should compile as well.

3 Your Own SPMV Approach

In the fifth version of spmv, you will implement your own approach. It can be a new algorithm based on your experience on building the aforementioned four versions. Or it can be a combination of the approaches you implemented above, i.e., choosing the best of all for a given input matrix. We will test this version for performance competition unless you specify otherwise. Note that we might also evaluate other aspects besides competition, for instance, overhead analysis, cost-effectiveness, or any other findings you made during your exploration.

Note that in your report, you will need to compare your own approach with CUSP (http://cusplibrary.github.io/) or CUSPARSE (already installed in ilab, or can be found at https://developer.nvidia.com/cusparse) on the 20 provided sparse

matrices and report the running time.

You are encouraged to explore as many different algorithms you have in your mind. Here are a few possible ideas for you to start with.

1. You can study the approaches that you have already implemented and determine which one works best for each input matrix. It's possible that no single approach is the best for all matrices. You can potentially use a machine learning approach to determine which method to use for a given input matrix [9].

2. You can try tuning parameters for a spmv algorithm and determine which execution parameter will be the best for any given sparse matrix. The execution parameters include but are not limited to the following: thread block size, work list size, load imbalancing factor. Even tuning these factors may be non-trivial task for a massively parallel program. Sub-optimal parameters cannot take advantage of GPU's massive parallelism, and may even make the GPU program run slower than the CPU algorithm. How to automatically find the best parameter is an important research question as well [4].

3. You can reduce the preprocessing overhead of the fourth version you have to implement. Although we will mainly test the implemented GPU kernel performance and you may try to achieve the best running time without considering the overhead, it will be potentially very useful to have a low-overhead inspector for the cache sharing aware approach to be broadly applicable. Currently the graph partition algorithm might take the same amount of time as a GPU kernel time or a few hundred iterations of the GPU kernel time. It is still be beneficial since a lot of applications that rely on sparse matrix computation might invoke the GPU spmv kernel thousands of times, i.e., an iterative conjugate gradient solver [8]. You can try to design a new partition algorithm that has similar quality but much lower overhead.

4. Parallelize the preprocessing steps in GPU as well, for instance, the graph partition inspector in the fourth approach, or the divergence remover in the third approach. They may significantly reduce the optimization overhead.

4 Compilation and Execution

Compilation: Please write a Makefile such that the grader can compile and manage your code by typing the following commands:

- Make spmv will compile into a single spmv executable called **spmv**. The usage will be described in execution interface.
- Make clean will remove all generated executable and intermediate files.

Note that we might compile your program on different GPU architectures. Please take this into consideration as well. You can check the Makefile in cuda samples in "/usr/local/cuda/samples" directory.

Running 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's an optional parameter. If it's not provided by the user, you will run your own approach by default. Otherwise, it can be one of the following:

  -alg atom, use the simple atomics version

  -alg scan, use the segment scan version

  -alg optscan, use the third approach that combines scan and atom

  -alg cache, use the cache sharing aware approach

  -alg design, use the approach you designed
- -blocksize [threadBlockSize], this specifies the thread block size to be used. It's an optional parameter. If not specified, you need to have a default thread block size.

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".

Additionally the program needs to output a vector file called "output.txt" which lists the y vector values from the first one to the last one. Note that if you performed data layout transformation on x and y arrays for memory coalescing optimization, when you output the y vector, you need to output in the original order before transformation. 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

There are a lot of good sample examples in /usr/local/cuda/samples. You might want to browse a few of them before you start programming in GPU.

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 20 sparse matrices will be used to test your five 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.

You can use the sparse matrix I/O functions in project 3. We will only test these matrices in matrix market format. Please make sure your program accepts matrix market format.

- pwtk
- mac_econ_fwd500
- pdb1HYS
- scircuit
- rma10
- rail4284
- webbase-1M
- shipsec1
- mc2depi
- qcd5_4
- cop20k_A
- cant

- consph
- watson_2
- turon_m
- asia_osm
- circuit5M_dc
- FullChip
- in-2004
- cit-Patents

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. As mentioned earlier, if you use an external library (i.e. a graph partition library), please be sure to include the full path of the library in your Makefile so that the grader can compile it as well. You also need to set the file read permission correctly so that the grader can access them.

Please do not include any sparse matrix files in your submission. If you use any other matrix not in the 20 specified sparse matrices, please specify the source (i.e., a web link), we will download the sparse matrices from the 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.